Anand Balachandran Pillai

Software Architecture with Python

Design and architect highly scalable, robust, clean, and high performance applications in Python

Packt

Software Architecture with Python

Design and architect highly scalable, robust, clean, and high performance applications in Python

Anand Balachandran Pillai

BIRMINGHAM - MUMBAI

Software Architecture with Python

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2017

Production reference: 2060619

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78646-852-9

www.packtpub.com

Credits

Author
Anand Balachandran Pillai

Reviewer
Mike Driscoll

Commissioning Editor
Aaron Lazar

Acquisition Editor
Vinay Argekar

Content Development Editor
Rohit Kumar Singh

Technical Editors
Leena Patil
Vibhuti Gawde

Copy Editor
Sonia Mathur

Project Coordinator
Vaidehi Sawant

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

About the Author

Anand Balachandran Pillai is an Engineering and Technology professional with over 18 years of experience in the software industry in Product Engineering, Software Design and Architecture and Research. He has a Bachelor's degree in Mechanical Engineering from the Indian Institute of Technology, Madras.

He has worked at companies such as Yahoo!, McAfee, and Infosys in the roles of Lead Engineer and Architect in product development teams, to build new products.

His interests lie in Software Performace Engineering, High Scalability Architectures, Security and open source communities. He often works with startups in lead technical or consulting role.

He is the founder of the Bangalore Python Users Group and a Fellow of the Python Software Foundation (PSF).

Anand is currently working as Senior Architect of Yegii Inc.

Dedicated to friends & family

About the Reviewer

Mike Driscoll has been programming in Python since 2006. He enjoys writing about Python in his blog, `http://www.blog.pythonlibrary.org/`. He has coauthored the Core Python refcard for DZone. He has also worked as a technical reviewer for *Python 3 Object Oriented Programming*, *Python 2.6 Graphics Cookbook, Tkinter GUI Application Development Hotshot*, and several other book. He recently wrote the book Python 101 and is working on his next book.

I would like to thank my beautiful wife, Evangeline, for always supporting me and my friends and family for all that they do to help me. And I would like to thank Jesus Christ for saving me.

---

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1786468522`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface ix

Chapter 1: Principles of Software Architecture 1

Defining software architecture 2
Software architecture versus design 3
Aspects of software architecture 4
Characteristics of software architecture 4
An architecture defines a structure 4
An architecture picks a core set of elements 6
An architecture captures early design decisions 6
An architecture manages stakeholder requirements 7
An architecture influences the organizational structure 8
An architecture is influenced by its environment 9
An architecture documents the system 10
An architecture often conforms to a pattern 10
Importance of software architecture 11
System versus enterprise architecture 13
Architectural quality attributes 17
Modifiability 18
Testability 21
Scalability 23
Performance 25
Availability 26
Security 27
Deployability 29
Summary 30

Chapter 2: Writing Modifiable and Readable Code 33
What is modifiability? 34
Aspects related to modifiability 34
Understanding readability 35
Python and readability 35
Readability – antipatterns 37
Techniques for readability 39
Document your code 39
Follow coding and style guidelines 47
Review and refactor code 48
Commenting the code 49
Fundamentals of modifiability – cohesion and coupling 50
Measuring cohesion and coupling 51
Measuring cohesion and coupling – string and text processing 54
Exploring strategies for modifiability 55
Providing explicit interfaces 55
Reducing two-way dependencies 56
Abstract common services 57
Using inheritance techniques 58
Using late binding techniques 62
Metrics – tools for static analysis 63
What are code smells? 64
Cyclomatic complexity – the McCabe metric 65
Testing for metrics 66
Running static checkers 69
Refactoring code 77
Refactoring code – fixing complexity 78
Refactoring code – fixing code smells 80
Refactoring code – fixing styling and coding issues 82
Summary 83
Chapter 3: Testability – Writing Testable Code 85
Understanding testability 86
Software testability and related attributes 86
Testability – architectural aspects 87
Testability – strategies 89
Reduce system complexity 89
Improving predictability 90
Control and isolate external dependencies 91
White-box testing principles 95
Unit testing 96

Unit testing in action 97
Extending our unit test case 99
Nosing around with nose2 101
Testing with py.test 102
Code coverage 105
Measuring coverage using coverage.py 105
Measuring coverage using nose2 106
Measuring coverage using pytest 107
Mocking things up 108
Tests inline in documentation – doctests 113
Integration tests 117
Test automation 120
Test automation using Selenium WebDriver 120
Test-driven development 122
TDD with palindromes 123
Summary 129

Chapter 4: Good Performance is Rewarding! 131

What is performance? 133
Software performance engineering 133
Performance testing and measurement tools 135
Performance complexity 136
Measuring performance 138
Measuring time using a context manager 139
Timing code using the timeit module 142
Measuring the performance of our code using timeit 143
Finding out time complexity – graphs 145
Measuring CPU time with timeit 151
Profiling 152
Deterministic profiling 152
Profiling with cProfile and profile 152
Prime number iterator class – performance tweaks 156
Profiling – collecting and reporting statistics 158
Third-party profilers 159
Line profiler 159
Memory profiler 161
Substring (subsequence) problem 163
Other tools 168
Objgraph 168
Pympler 170
Programming for performance – data structures 172
Mutable containers – lists, dictionaries, and sets 173
Lists 173
Dictionaries 174

Sets 174
Immutable containers – tuples 175
High performance containers – the collections module 175
deque 176
defaultdict 176
OrderedDict 178
Counter 180
ChainMap 181
namedtuple 182
Probabilistic data structures – bloom filters 184
Summary 188
Chapter 5: Writing Applications that Scale 191
Scalability and performance 193
Concurrency 196
Concurrency versus parallelism 197
Concurrency in Python – multithreading 198
Thumbnail generator 199
Thumbnail generator – producer/consumer architecture 201
Thumbnail generator – resource constraint using locks 206
Thumbnail generator – resource constraint using semaphores 211
Resource constraint – semaphore versus lock 214
Thumbnail generator – URL rate controller using conditions 215
Multithreading – Python and GIL 223
Concurrency in Python – multiprocessing 224
A primality checker 224
Sorting disk files 228
Sorting disk files – using a counter 229
Sorting disk files – using multiprocessing 232
Multithreading versus multiprocessing 235
Concurrecy in Python – Asynchronous Execution 236
Pre-emptive versus cooperative multitasking 236
The asyncio module in Python 241
Waiting for a future – async and await 244
Concurrent futures – high-level concurrent processing 247
Disk thumbnail generator 249
Concurrency options – how to choose? 252
Parallel processing libraries 253
Joblib 253
PyMP 254
Fractals – the Mandelbrot set 255
Fractals – scaling the Mandelbrot set implementation 259

Scaling for the web 262
Scaling workflows – message queues and task queues 262
Celery – a distributed task queue 264
The Mandelbrot set using Celery 264
Serving with Python on the Web – WSGI 269
uWSGI – WSGI middleware on steroids 272
Gunicorn – unicorn for WSGI 274
Gunicorn versus uWSGI 274
Scalability architectures 275
Vertical scalability architectures 275
Horizontal scalability architectures 275
Summary 279
Chapter 6: Security – Writing Secure Code 281
Information security architecture 282
Secure coding 284
Common security vulnerabilities 285
Is Python secure? 290
Reading input 291
Evaluating arbitrary input 294
Overflow errors 298
Serializing objects 300
Security issues with web applications 304
Server Side Template Injection 305
Server-Side Template Injection – Mitigation 308
Denial of Service 310
Cross-Site Scripting (XSS) 314
Mitigation – DoS and XSS 314
Strategies for security – Python 317
Secure coding strategies 325
Summary 326
Chapter 7: Design Patterns in Python 327
Design patterns – elements 328
Categories of design patterns 329
Pluggable hashing algorithms 331
Summing up pluggable hashing algorithm 334
Patterns in Python – creational 335
The Singleton pattern 335
The Singleton – do we need a Singleton? 338
State sharing – Borg versus Singleton 340
The Factory pattern 342

The Prototype pattern 345
Prototype – deep versus shallow copy 346
Prototype using metaclasses 347
Combining patterns using metaclasses 349
The Prototype factory 350
The Builder pattern 353
Patterns in Python – structural 360
The Adapter pattern 360
The Facade pattern 370
Facades in Python 371
The proxy pattern 377
An instance-counting proxy 378
Patterns in Python – behavioral 381
The Iterator pattern 382
The Observer pattern 385
The State pattern 393
Summary 400
Chapter 8: Python – Architectural Patterns 403
Introducing MVC 404
Model Template View (MTV) – Django 406
Django admin – automated model-centric views 407
Flexible Microframework – Flask 409
Event-driven programming 411
Chat server and client using I/O multiplexing with the select module 411
Event-driven programming versus concurrent programming 418
Twisted 420
Twisted – a simple web client 421
Chat server using Twisted 422
Eventlet 428
Greenlets and Gevent 430
Microservice architecture 432
Microservice frameworks in Python 433
Microservices example – restaurant reservation 435
Microservices – advantages 438
Pipe and Filter architectures 438
Pipe and filter in Python 439
Summary 445
Chapter 9: Deploying Python Applications 447
Deployability 448
Factors affecting deployability 449
Tiers of software deployment architecture 451

Software deployment in Python 452
Packaging Python code 452
PIP 453
Virtualenv 455
Virtualenv and pip 457
Relocatable virtual environments 459
PyPI 460
Packaging and submission of an application 462
The __init__.py files 463
The setup.py file 463
Installing the package 464
Submitting the package to PyPI 465
PyPA 468
Remote deployments using Fabric 468
Remote deployments using Ansible 470
Managing remote daemons using Supervisor 471
Deployment – patterns and best practices 472
Summary 474
Chapter 10: Techniques for Debugging 477
Maximum subarray problem 478
The power of "print" 479
Analysis and rewrite 481
Timing and optimizing the code 484
Simple debugging tricks and techniques 486
Word searcher program 487
Word searcher program – debugging step 1 488
Word searcher program – debugging step 2 489
Word searcher program – final code 490
Skipping blocks of code 491
Stopping execution 491
External dependencies – using wrappers 492
Replacing functions with their return value/data (mocking) 494
Saving to / loading data from files as cache 496
Saving to / loading data from memory as cache 498
Returning random/mock data 500
Logging as a debugging technique 505
Simple application logging 505
Advanced logging – logger objects 507
Advanced logging – custom formatting and loggers 508
Advanced logging – writing to syslog 512
Debugging tools – using debuggers 513
A debugging session with pdb 514

Pdb – similar tools 516
iPdb 516
Pdb++ 518
Advanced debugging – tracing 518
The trace module 519
The lptrace program 520
System call tracing using strace 521
Summary 522
Index 525

Preface

Software architecture, or creating a blueprint design for a particular software application, is not a walk in the park. The two biggest challenges in software architecture are keeping the architecture in sync, first with the requirements as they are uncovered or evolve, and next with the implementation as it gets built and evolves.

Filled with examples and use cases, this guide takes a direct approach to helping you with everything it takes to become a successful software architect. This book will help you understand the ins and outs of Python so that you can architect and design highly scalable, robust, clean, and performant applications in Python.

What this book covers

Chapter 1, *Principles of Software Architecture*, introduces the topic of software architecture, giving you a brief on architectural quality attributes and the general principles behind them. This will enable you to have strong fundamentals in software architectural principles and foundational attributes.

Chapter 2, *Writing Modifiable and Readable Code*, covers developmental architectural quality attributes, namely, modifiability and readability. It will help you gain an understanding of the architectural quality attribute of maintainability and tactics of writing code in Python to test your applications.

Chapter 3, *Testability – Writing Testable Code*, helps you understand the architectural quality attribute of testability and how to architect Python applications for testability. You will also learn about various aspects of testability and software testing and the different libraries and modules available in Python to write testable applications.

Chapter 4, *Good Performance is Rewarding!*, covers the performance aspects of writing Python code. You will be equipped with the knowledge of performance as a quality attribute in architecture and when to optimize for performance. You will learn when to optimize for performance in the SDLC.

Chapter 5, *Writing Applications that Scale*, talks about the importance of writing scalable applications. It discusses different ways to achieve of application scalability and discusses scalability techniques using Python. You will also learn about theoretical aspects of scalability and the best practices in the industry.

Chapter 6, *Security – Writing Secure Code*, discusses the security aspect of architecture and teaches you best practices and techniques of writing applications that are secure. You will understand the different security issues to watch out for and to and to architecture applications in Python that are secure from the ground up.

Chapter 7, *Design Patterns in Python*, gives you an overview of design patterns in Python from a pragmatic programmer's perspective, with brief theoretical background of each pattern. You will gain knowledge of design patterns in Python that are actually useful to pragmatic programmer.

Chapter 8, *Python Architectural Patterns*, introduces you to the modern architectural patterns in Python from a high-level perspective while giving examples of Python libraries and frameworks to realize the approaches of these patterns to solve high-level architecture problems.

Chapter 9, *Deploying Python Applications*, covers the aspect of easily deploying your code on remote environments or on the cloud using Python the right way.

Chapter 10, *Techniques for Debugging*, covers some of the debugging techniques for Python code—from the simplest, strategically placed print statement to logging and system call tracing which will be very handy to the programmer and also help the system architect to guide his team.

What you need for this book

To run most of the code samples shown in this book, you need to have Python 3 installed on your system. The other prerequisites are mentioned at the respective instances.

Who this book is for

This book is for experienced Python developers who are aspiring to become the architects of enterprise-grade applications or software architects who would like to leverage Python to create effective blueprints of applications.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
class PrototypeFactory(Borg):
    """ A Prototype factory/registry class """

    def __init__(self):
        """ Initializer """

        self._registry = {}

    def register(self, instance):
        """ Register a given instance """

        self._registry[instance.__class__] = instance

    def clone(self, klass):
        """ Return clone given class """

        instance = self._registry.get(klass)
        if instance == None:
            print('Error:',klass,'not registered')
        else:
            return instance.clone()
```

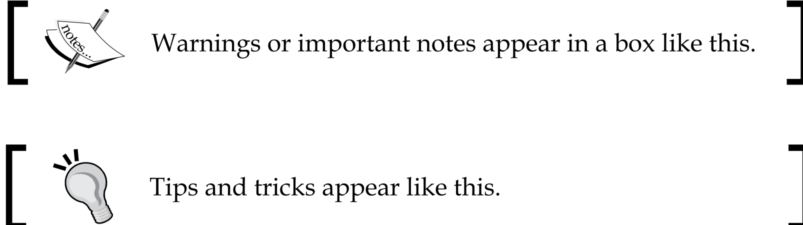
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
>>> import hash_stream
>>> hash_stream.hash_stream(open('hash_stream.py'))
'30fbc7890bc950a0be4eaa60e1fee9a1'
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Software-Architecture-with-Python`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/SoftwareArchitecturewithPython_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Principles of Software Architecture

This is a book on Python. At the same time, it is a book about software architecture and its various attributes, which are involved in a software development life cycle.

In order for you to understand and combine both aspects, which is essential to get maximum value from this book, it is important to grasp the fundamentals of software architecture, the themes and concepts related to it, and the various quality attributes of software architecture.

A number of software engineers, taking on senior roles in their organizations, often get very different interpretations of the definitions of software design and architecture, and the roles they play in building testable, maintainable, scalable, secure, and functional software.

Though there is a lot of literature in the field, which is available both in conventional book form and on the internet; very often, the practitioners among us get a confusing picture of these very important concepts. This is often due to the pressures involved in *learning the technology* rather than learning the fundamental design and architectural principles underlying the use of technology in building systems. This is a common practice in software development organizations, where the pressures of delivering working code often overpowers and eclipses everything else.

A book such as this one, strives to transcend the middle path in bridging the rather esoteric aspects of software development related to its architectural quality attributes to the mundane details of building software using programming languages, libraries, and frameworks—in this case, using Python and its developer ecosystem.

The role of this introductory chapter is to demystify these concepts, and explain them in very clear terms to the reader to prepare his/her for the path towards understanding the rest of this book. Hopefully, by the end of this book, the concepts and their practical details will represent a coherent body of knowledge to the reader.

We will now get started on this path without any further ado, roughly fitting this chapter into the following sections:

- Defining software architecture
- Software architecture versus design
- Aspects of software architecture
- Characteristics of software architecture
- Why is software architecture important?
- System versus Enterprise Architecture
- Architectural quality attributes
    - Modifiability
    - Testability
    - Scalability/performance
    - Security
    - Deployability

Defining software architecture

There are various definitions of software architecture in the literature concerning the topic. A simple definition is given as follows:

Software architecture is a description of the subsystems or components of a software system, and the relationships between them.

The following is a more formal definition, from the **Recommended Practice for Architectural Description of Software-Intensive Systems** (**IEEE**) technology:

> *"Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."*

It is possible to get umpteen such definitions of software architecture if one spends some time searching on the web. The wordings might differ, but all the definitions refer to some core, fundamental aspects underlying software architecture.

Software architecture versus design

In the experience of the author, this question of the software architecture of a system versus its design seems to pop up quite often, in both online as well as offline forums. Hence, let us take a moment to understand this aspect.

Though both terms are often used interchangeably, the rough distinction of architecture versus design can be summarized as follows:

- Architecture covers the higher level structures and interactions in a system. It is concerned with those questions that entail decision making about the *skeleton* of the system, involving not only its functional but also its organizational, technical, business, and quality attributes.
- Design is all about the organization of parts or components of the system and the subsystems involved in making the system. The problems here are typically closer to the code or modules in question, such as these:
    - Which modules to split code into? How to organize them?
    - Which classes (or modules) to assign the different functionalities to?
    - Which design pattern should I use for class "C"?
    - How do my objects interact at runtime? What are the messages passed, and how is the interaction organized?

Software architecture is about the design of the entire system, whereas, software design is mostly about the details, typically at the implementation level of the various subsystems and components that make up those subsystems.

In other words, the word *design* comes up in both contexts, however, with the distinction that the former is at a much higher abstraction and at a larger scope than the latter.

There is a rich body of knowledge available for both software architecture and design, namely, **architectural patterns and design patterns** respectively. We will discuss both these topics in later chapters of this book.

Aspects of software architecture

In both the formal IEEE definition and the rather informal definition given earlier, we find some common, recurring themes. It is important to understand them in order to take our discussion on software architecture further:

- **System**: A system is a collection of components organized in specific ways to achieve a specific functionality. A software system is a collection of such software components. A system can often be subgrouped into subsystems.
- **Structure**: A structure is a set of elements that are grouped or organized together according to a guiding rule or principle. The elements can be software or hardware systems. A software architecture can exhibit various levels of structures depending on the observer's context.
- **Environment**: The environment is the context or circumstances in which a software system is built, which has a direct influence on its architecture. Such contexts can be technical, business, professional, operational, and so on.
- **Stakeholder**: A stakeholder is a person or groups of persons, who has an interest or concern in the system and its success. Examples of stakeholders are the architect, development team, customer, project manager, marketing team, and others.

Now that you have understood some of the core aspects of software architecture, let us briefly list some of its characteristics.

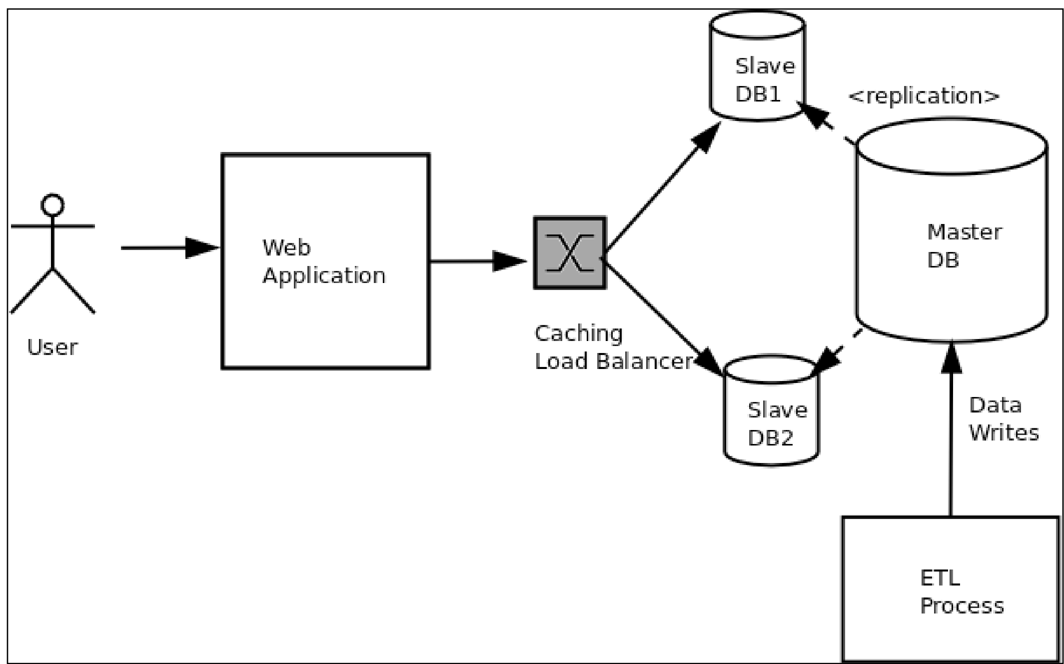
Characteristics of software architecture

All software architectures exhibit a common set of characteristics. Let us look at some of the most important ones here.

An architecture defines a structure

An architecture of a system is best represented as structural details of the system. It is a common practice for practitioners to draw the system architecture as a structural component or class diagram in order to represent the relationships between the subsystems.

For example, the following architecture diagram describes the backend of an application that reads from a tiered database system, which is loaded using an ETL process:



Example architecture diagram showing system structure

Structures provide insight into architectures, and provide a unique perspective to analyze the architecture with respect to its quality attributes.

Some examples are as follows:

- The runtime structures, in terms of the objects created at runtime, and how they interact often determine the deployment architecture. The deployment architecture is strongly connected to the quality attributes of scalability, performance, security, and interoperability.

- The module structures, in terms of how the code is broken down and organized into modules and packages for task breakdown, often has a direct bearing on the maintainability and modifiability (extensibility) of a system. This is explained as follows:
    - Code which is organized with a view to extensibility would often keep the parent classes in separate well-defined packages with proper documentation and configuration, which are then easily extensible by external modules, without the need to resolve too many dependencies.
    - Code which is dependent on external or third-party developers (libraries, frameworks, and the like) would often provide setup or deployment steps, which manually or automatically pull in these dependencies from external sources. Such code would also provide documentation (README, INSTALL, and so on) which clearly documents these steps.

An architecture picks a core set of elements

A well-defined architecture clearly captures only the core set of structural elements required to build the core functionality of the system, and which have a lasting effect on the system. It does not set out to document everything about every component of the system.

For example, an architect describing the architecture of a user interacting with a web server for browsing web pages—a typical client/server architecture—would focus mainly on two components: the user's browser (client) and the remote web server (server), which form the core elements of the system.

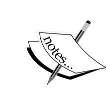
The system may have other components such as multiple caching proxies in the path from the server to the client, or a remote cache on the server which speeds up web page delivery. However, this is not the focus of the architecture description.

An architecture captures early design decisions

This is a corollary to the characteristics described previously. The decisions that help an architect to focus on some core elements of the system (and their interactions) are a result of the early design decisions about a system. Thus, these decisions play a major role in further development of the system due to their initial weight.

For example, an architect may make the following early design decisions after careful analysis of the requirements for a system:

- The system will be deployed only on Linux 64-bit servers, since this satisfies the client requirement and performance constraints
- The system will use HTTP as the protocol for implementing backend APIs
- The system will try to use HTTPS for APIs that transfer sensitive data from the backend to frontend using encryption certificates of 2,048 bits or higher
- The programming language for the system would be Python for the backend, and Python or Ruby for the frontend



The first decision freezes the deployment choices of the system to a large extent to a specific OS and system architecture. The next two decisions have a lot of weight in implementing the backend APIs. The last decision freezes the programming language choices for the system.

Early design decisions need to be arrived at after careful analysis of the requirements and matching them with the constraints—such as organizational, technical, people, and time constraints.

An architecture manages stakeholder requirements

A system is designed and built, ultimately, at the behest of its stakeholders. However, it is not possible to address each stakeholder requirement to its fullest due to an often contradictory nature of such requirements. Following are some examples:

- The marketing team is concerned with having a full-featured software application, whereas, the developer team is concerned with *feature creep* and performance issues when adding a lot of features.
- The system architect is concerned with using the latest technology to scale out his/her deployments to the cloud, while the project manager is concerned about the impact such technology deployments will have on his/her budget. The end user is concerned about correct functionality, performance, security, usability, and reliability, while the development organization (architect, development team, and managers) is concerned with delivering all these qualities while keeping the project on schedule and within budget.

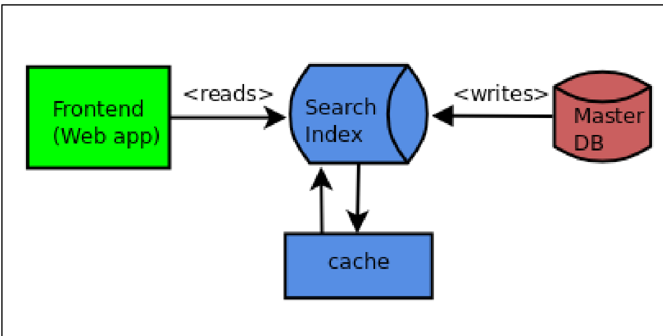
- A good architecture tries its best to balance out these requirements by making trade-offs, and delivering a system with good quality attributes while keeping the people and resource costs within limits.
- An architecture also provides a common language among the stakeholders, which allows them to communicate efficiently via expressing these constraints, and helping the architect zero-in on an architecture that best captures these requirements and their trade-offs.

An architecture influences the organizational structure

The system structures an architecture describes, quite often have a direct mapping to the structure of the teams that build those systems.

For example, an architecture may have a data access layer which describes a set of services that read and write large sets of data—it is natural that such a system gets functionally assigned to the database team, which already has the required skill sets.

Since the architecture of a system is its best description of the top-down structures, it is also often used as the basis for the task-breakdown structures. Thus, software architecture often has a direct bearing on the organizational structures that build it:



System architecture for a search web application

The following diagram shows the mapping to the team structure which would be building this application:

Frontend Team
Data Indexing Team
Database Team

An architecture is influenced by its environment

An environment imposes outside constraints or limits within which an architecture must function. In the literature, these are often called *architecture in context* [Ref: Bass, Kazman]. Some examples are as follows:

- **Quality attribute requirements**: In modern day web applications, it is very common to specify the scalability and availability requirements of the application as an early technical constraint, and capture it in the architecture. This is an example of a technical context from a business perspective.
- **Standards conformance**: In some organizations where there is often a large set of governing standards for software, especially those in the banking, insurance, and health-care domains, these get added to the early constraints of the architecture. This is an example of an external technical context.
- **Organizational constraints**: It is common to see that organizations which either have an experience with a certain architectural style or a set of teams operating with certain programming environments which impose such a style (J2EE is a good example), prefer to adopt similar architectures for future projects as a way to reduce costs and ensure productivity due to current investments in such architectures and related skills. This is an example of an internal business context.
- **Professional context**: An architect's set of choices for a system's architecture, aside from these outside contexts, is mostly shaped from his/her set of unique experiences. It is common for an architect to continue using a set of architectural choices that he/she has had the most success with in his/her past, for new projects.

Architecture choices also arise from one's own education and professional training, and also from the influence of one's professional peers.

An architecture documents the system

Every system has an architecture, whether it is officially documented or not. However, properly documented architectures can function as an effective documentation for the system. Since an architecture captures the system's initial requirements, constraints, and stakeholder trade-offs, it is a good practice to document it properly. The documentation can be used as a basis for training later on. It also helps in continued stakeholder communication, and for subsequent iterations of the architecture based on changing requirements.

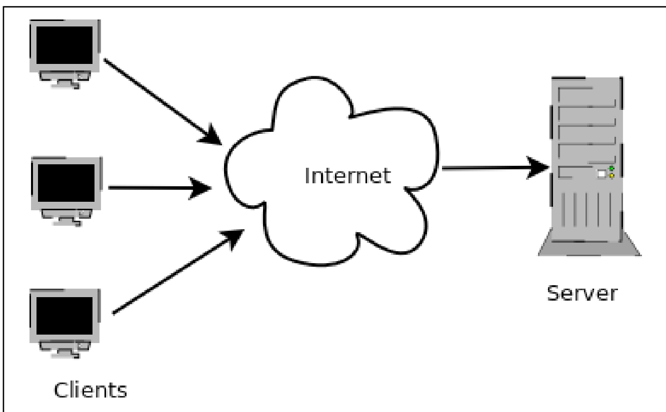
The simplest way to document an architecture is to create diagrams for the different aspects of the system and organizational architecture such as Component Architecture, Deployment Architecture, Communication Architecture, and the Team or Enterprise Architecture.

Other data that can be captured early include the system requirements, constraints, early design decisions, and rationale for those decisions.

An architecture often conforms to a pattern

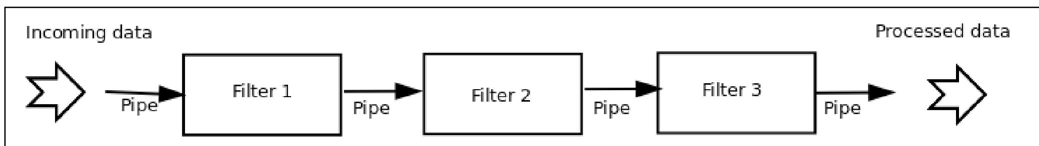
Most architectures conform to certain sets of styles which have had a lot of success in practice. These are referred to as architectural patterns. Examples of such patterns are client-server, pipes and filters, data-based architectures, and others. When an architect chooses an existing pattern, he/she gets to refer to and reuse a lot of existing use cases and examples related to such patterns. In modern day architectures, the job of the architect comes down to mixing and matching existing sets of such readily available patterns to solve the problem at hand.

For example, the following diagram shows an example of a client-server architecture:



Example of client-server architecture

The following diagram describes another common architecture pattern, namely, the pipes and filters architecture for processing streams of data:



Example of pipe and filters architecture

We will see examples of architectural patterns later in this book.

Importance of software architecture

So far, we have discussed the fundamental principles of software architecture, and also seen some of its characteristics. These sections, of course, assumed that software architecture is important, and is a critical step of the software development process.

It is time to play devil's advocate, and look back at software architecture and ask some existential questions about it as follows:

- Why software architecture?
- Why is software architecture important?
- Why not build a system without a formal software architecture?

Let us take a look at the critical insights that software architecture provides, which would otherwise be missing from an informal software development process. We are only focusing on the technical or developmental aspects of the system in the following table:

Aspect	Insight/Impact	Examples
Architecture selects quality attributes to be optimized for a system.	Aspects such as scalability, availability, modifiability, security, and so on of a system depend on early decisions and trade-offs while selecting an architecture. You often trade one attribute in favor of another.	A system that is optimized for scalability must be developed using a decentralized architecture where elements are not tightly coupled. For example: microservices, brokers.

Aspect	**Insight/Impact**	**Examples**
Architecture facilitates early prototyping.	Defining an architecture allows the development organization to try and build early prototypes, which gives valuable insights into how the system would behave without having to build the complete system top down.	Many organizations build out quick prototypes of services—typically, by building only the external APIs of these services and mocking the rest of the behavior. This allows for early integration tests and figuring out interaction issues in the architecture early on.
Architecture allows a system to be built component-wise.	Having a well-defined architecture allows the reuse and assembly of existing, readily available components to achieve the functionality without having to implement everything from scratch.	Libraries or frameworks which provide ready-to-use building blocks for services. For example: web application frameworks such as Django/RoR, and task distribution frameworks such as Celery.
Architecture helps to manage changes to the system.	An architecture allows the architect to scope out changes to the system in terms of components that are affected and those which are not. This helps to keep system changes to a minimum when implementing new features, performance fixes, and so on.	A performance fix for database reads to a system would need changes only to the DB and Data Access Layer (DAL) if the architecture is implemented correctly. It need not touch the application code at all. For example, this is how most modern web frameworks are built.

There are a number of other aspects which are related to the business context of a system, into which architecture provides valuable insights. However, since this is a book mostly on the technical aspects of software architecture, we have limited our discussion to the ones given in the preceding table.

Now, let us take on the second question:

Why not build a system without a formal software architecture?

If you've been following the arguments so far thoroughly, it is not very difficult to see the answer for it. It can, however, be summarized in the following few statements:

- Every system *has* an architecture, whether it is documented or not
- Documenting an architecture makes it formal, allowing it to be shared among stakeholders, making change management and iterative development possible
- All the other benefits and characteristics of software architecture are ready to be taken advantage of when you have a formal architecture defined and documented
- You may be still able to work and build a functional system without a formal architecture, but it would not produce a system which is extensible and modifiable, and would most likely produce a system with a set of quality attributes quite far away from the original requirements

System versus enterprise architecture

You may have heard the term *architect* used in a few contexts. The following job *roles* or *titles* are pretty common in the software industry for architects:

- The Technical architect
- The Security architect
- The Information architect
- The Infrastructure architect

You also may have heard the term *System architect*, perhaps the term *Enterprise architect*, and maybe, *Solution architect* also. The interesting question is: *What do these people do?*

Let us try and find the answer to this question.

An Enterprise architect looks at the overall business and organizational strategies for an organization, and applies architecture principles and practices to guide the organization through the business, information, process, and technology changes necessary to execute their strategies. The Enterprise architect usually has a higher strategy focus and a lower technology focus. The other architect roles take care of their own subsystems and processes. For example:

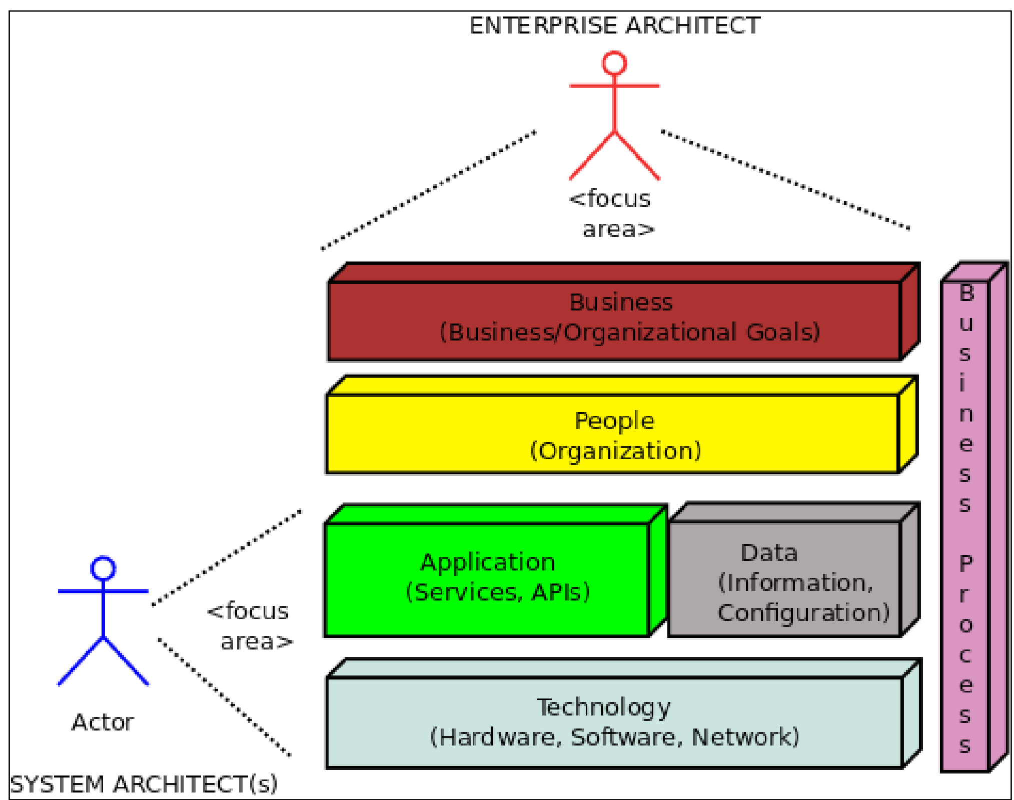
- **The Technical architect**: The Technical architect is concerned with the core technology (hardware/software/network) used in an organization. A Security architect creates or tunes the security strategy used in applications to fit the organization's information security goals. An Information architect comes up with architectural solutions to make information available to/from applications in a way that facilitates the organization's business goals.

  These specific architectural roles are all concerned with their own systems and subsystems. So, each of these roles is a System architect role.

  These architects help the Enterprise architect to understand the smaller picture of each of the business domain they are responsible for, which helps the Enterprise architect to get information that will aid him in formulating business and organizational strategies.

- **The System architect**: A System architect usually has a higher technology focus and a lower strategy focus. It is a practice in some service-oriented software organizations to have a Solution architect, who combines the different systems to create a solution for a specific client. In such cases, the different architect roles are often combined into one, depending on the size of the organization, and the specific time and cost requirements of the project.
- **The Solution architect**: A Solution architect typically straddles the middle position when it comes to strategy versus technology focus and organizational versus project scope.

The following schematic diagram depicts the different layers in an organization–**Technology**, **Application**, **Data**, **People**, **Process**, and **Business**, and makes the focus area of the architect roles very clear:



Enterprise versus System architects

Let's discuss the preceding diagram a bit to understand the picture it lays out.

The System architect is pictured on the bottom-left side of the diagram, looking at the system components of the enterprise. His/her focus is on the applications that power the enterprise, their data, and the hardware and software stack powering the applications.

The Enterprise architect, on the other hand, is pictured on the top, having a top-down view of the enterprise including the business goals and the people, and not just the underlying systems that power the organization. The vertical stack of business processes connect the technical components that power the organization with its people and business components. These processes are defined by the Enterprise architect in discussion with the other stakeholders.

Now that you have understood the picture behind Enterprise and System architecture, let us take a look at some formal definitions:

> *"Enterprise Architecture is a conceptual blueprint that defines the structure and behavior of an organization. It determines how the organization's structure, processes, personnel and flow of information is aligned to its core goals to efficiently achieve its current and future objectives."*

> *"A system architecture is the fundamental organization of a system, represented by its structural and behavioral views. The structure is determined by the components of the system and the behavior by the relationships between them and their interaction with external systems."*

An Enterprise architect is concerned with how the different elements in an organization and their interplay is tuned towards achieving the goals of the organization in an efficient manner. In this work, he/she needs the support of not just the technical architects in the organization, but also people managing the organization, such as project managers and human resource professionals.

A Systems architect, on the other hand, is worried about how the core system architecture maps to the software and hardware architecture, and the various details of human interactions with the components in the system. His/her concern never arises above the boundaries defined by the system and its interactions.

The following diagram depicts the different focus areas and scopes of the different architect roles that we've discussed so far:

Scope and focus of various architect roles in a software organization

Architectural quality attributes

Let us now focus on an aspect which forms the main topic for the rest of this book–Architectural Quality Attributes.

In a previous section, we discussed how an architecture balances and optimizes stakeholder requirements. We also saw some examples of contradicting stakeholder requirements, which an architect seeks to balance, by choosing an architecture which does the necessary trade-offs.

The term **quality attribute** has been used to loosely define some of these aspects that an architecture makes trade-offs for. It is now the time to formally define what an Architectural Quality Attribute is:

> *"A quality attribute is a measurable and testable property of a system which can be used to evaluate the performance of a system within its prescribed environment with respect to its non-functional aspects"*

There are a number of aspects that fit this general definition of an architectural quality attribute. However, for the rest of this book, we will be focusing on the following quality attributes:

- Modifiability
- Testability
- Scalability and performance
- Availability
- Security
- Deployability

Modifiability

Many studies show that about 80% of the cost of a typical software system occurs after the initial development and deployment. This shows how important modifiability is to a system's initial architecture.

Modifiability can be defined as the ease with which changes can be made to a system, and the flexibility with which the system adjusts to the changes. It is an important quality attribute, as almost every software system changes over its lifetime—to fix issues, for adding new features, for performance improvements, and so on.

From an architect's perspective, the interest in modifiability is about the following:

- **Difficulty**: The ease with which changes can be made to a system
- **Cost**: In terms of time and resources required to make the changes
- **Risks**: Any risk associated with making changes to the system

Now, what kind of changes are we talking about here? Is it changes to code, changes to deployment, or changes to the entire architecture?

The answer is: it can be at *any* level.

From an architecture perspective, these changes can generally be captured at the following three levels:

1. **Local**: A local change only affects a specific element. The element can be a piece of code such as a function, a class, a module, or a configuration element such as an XML or JSON file. The change *does not cascade* to any neighboring element or to the rest of the system. Local changes are the easiest to make, and the least risky of all. The changes can usually be quickly validated with local unit tests.
2. **Non-local**: These changes involve more than one element. The examples are as follows:
    - Modifying a database schema, which then needs to cascade into the model class representing that schema in the application code
    - Adding a new configuration parameter in a JSON file, which then needs to be processed by the parser parsing the file and/or the application(s) using the parameter

   Non-local changes are more difficult to make than local changes, require careful analysis, and wherever possible, integration tests to avoid code regressions.
3. **Global**: These changes either involve architectural changes from top down, or changes to elements at the global level, which cascade down to a significant part of the software system. The examples are as follows:
    - Changing a system's architecture from RESTful to messaging (SOAP, XML-RPC, and others) based web services
    - Changing a web application controller from Django to an Angular-js based component
    - A performance change requirement which needs all data to be preloaded at the frontend to avoid any inline model API calls for an online news application

   These changes are the riskiest, and also the costliest, in terms of resources, time and money. An architect needs to carefully vet the different scenarios that may arise from the change, and get his/her team to model them via integration tests. Mocks can be very useful in these kinds of large-scale changes.

The following table shows the relationship between **Cost** and **Risk** for the different levels of system modifiability:

Level	Cost	Risk
Local	Low	Low
Non-local	Medium	Medium
Global	High	High

Modifiability at the code level is also directly related to its readability:

> "*The more readable a code is, the more modifiable it is. Modifiability of a code goes down in proportion to its readability.*"

The modifiability aspect is also related to the maintainability of the code. A code module which has its elements very tightly coupled would yield to modification much less than a module which has a loosely coupled elements—this is the **Coupling** aspect of modifiability.

Similarly, a class or module which does not define its role and responsibilities clearly would be more difficult to modify than another one which has well-defined responsibility and functionality. This aspect is called **Cohesion** of a software module.

The following table shows the relation between **Cohesion**, **Coupling,** and **Modifiability** for an imaginary module A. Assume that the coupling is from this module to another module B:

Cohesion	Coupling	Modifiability
Low	High	Low
Low	Low	Medium
High	High	Medium
High	Low	High

It is pretty clear from the preceding table that having higher Cohesion and lower Coupling is the best scenario for the modifiability of a code module.

Other factors that affect modifiability are as follows:

- **Size of a module (number of lines of code)**: Modifiability decreases when size increases.
- **Number of team members working on a module**: Generally, a module becomes less modifiable when a larger number of team members work on the module due to the complexities in merging and maintaining a uniform code base.
- **External third-party dependencies of a module**: The larger the number of external third-party dependencies, the more difficult it is to modify the module. This can be thought of as an extension of the coupling aspect of a module.
- **Wrong use of the module API**: If there are other modules which make use of the private data of a module rather than (correctly) using its public API, it is more difficult to modify the module. It is important to ensure proper usage standards of modules in your organization to avoid such scenarios. This can be thought of as an extreme case of tight **Coupling**.

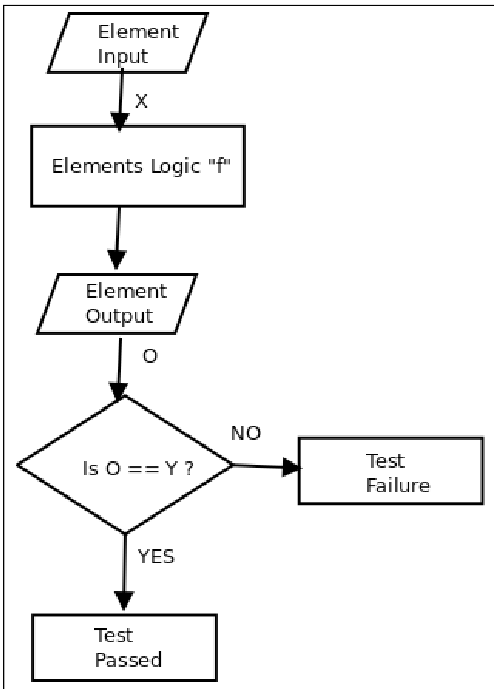
Testability

Testability refers to how much a software system is amenable to demonstrating its faults through testing. Testability can also be thought of as how much a software system *hides* its faults from end users and system integration tests—the more testable a system is, the less it is able to hide its faults.

Testability is also related to how predictable a software system's behavior is. The more predictable a system, the more it allows for repeatable tests, and for developing standard test suites based on a set of input data or criteria. Unpredictable systems are much less amenable to any kind of testing, or, in extreme case, not testable at all.

In software testing, you try to control a system's behavior by, typically, sending it a set of known inputs, and then observing the system for a set of known outputs. Both of these combine to form a testcase. A test suite or test harness, typically, consists of many such test cases.

Test assertions are the techniques that are used to fail a test case when the output of the element under the test does not match the expected output for the given input. These assertions are usually manually coded at specific steps in the test execution stage to check the data values at different steps of the testcase:



Representative flowchart of a simple unit test case for function f('X') = 'Y'

The preceding diagram shows an example of a representative flowchart for a testable function **"f"** for a sample input "**X**" with expected output "**Y**".

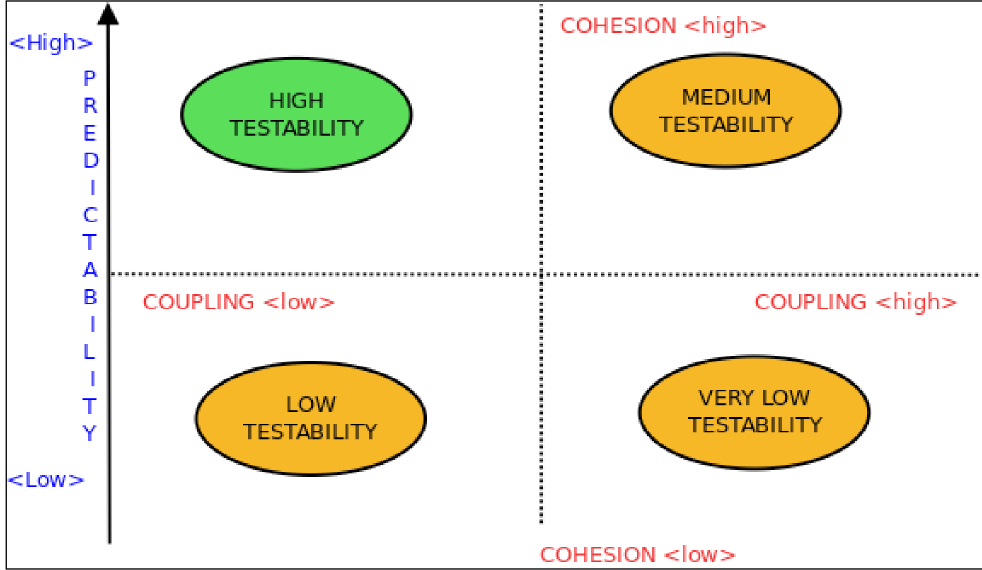
In order to recreate the session or state at the time of a failure, the *record/playback* strategy is often used. This employs specialized software (such as Selenium), which records all user actions that led to a specific fault, and saves it as a testcase. The test is reproduced by replaying the testcase using the same software which tries to simulate the same testcase; this is done by repeating the same set and order of UI actions.

Testability is also related to the complexity of code in a way very similar to modifiability. A system becomes more testable when parts of it can be isolated and made to work independent of the rest of the system. In other words, a system with low coupling is more testable than a system with high coupling.

Another aspect of testability, which is related to the predictability mentioned earlier, is to reduce non-determinism. When writing test suites, we need to isolate the elements that are to be tested from other parts of the system which have a tendency to behave unpredictably so that the tested element's behavior becomes predictable.

An example is a multi-threaded system, which responds to events raised in other parts of the system. The entire system is probably quite unpredictable, and not amenable to repeated testing. Instead, one needs to separate the events subsystem, and possibly, mock its behavior so that those inputs can be controlled, and the subsystem which receives the events becomes predictable and hence, testable.

The following schematic diagram explains the relationship between the testability and predictability of a system to the **Coupling** and **Cohesion** between its components:



Relation of testability and predictability of a system to coupling and cohesion

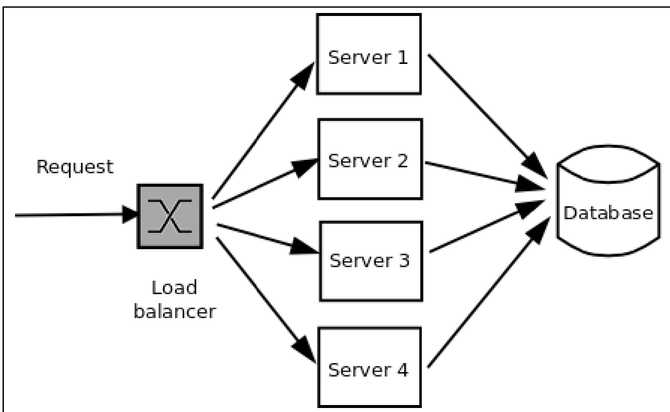
Scalability

Modern-day web applications are all about *scaling up*. If you are part of any modern-day software organization, it is very likely that you have heard about or worked on an application that is written for the cloud, which is able to scale up elastically on demand.

Scalability of a system is its capacity to accommodate increasing workload on demand while keeping its performance within acceptable limits.

Scalability in the context of a software system, typically, falls into two categories, which are as follows:

- **Horizontal scalability**: Horizontal scalability implies scaling out/in a software system by adding more computing nodes to it. Advances in cluster computing in the last decade have given rise to the advent of commercial horizontally scalable **elastic** systems as services on the web. A well-known example is Amazon Web Services. In horizontally scalable systems, typically, data and/or computation is done on units or nodes, which are, usually, virtual machines running on commodity systems known as virtual private servers (VPS). The scalability is achieved "n" times by adding n or more nodes to the system, typically fronted by a load balancer. Scaling out means expanding the scalability by adding more nodes, and scaling in means reducing the scalability by removing existing nodes:



Example deployment architecture showing horizontally scaling a web application server

- **Vertical scalability**: Vertical scalability involves adding or removing resources from a single node in a system. This is usually done by adding or removing CPUs or RAM (memory) from a single virtual server in a cluster. The former is called scaling up, and the latter, scaling down. Another kind of scaling up is increasing the capacity of an existing software process in the system—typically, by augmenting its processing power. This is usually done by increasing the number of processes or threads available to an application. Some examples are as follows:
    - Increasing the capacity of an Nginx server process by increasing its number of worker processes
    - Increasing the capacity of a PostgreSQL server by increasing its number of maximum connections

Performance

Performance of a system is related to its scalability. Performance of a system can be defined as follows:

> "*Performance of a computer system is the amount of work accomplished by a system using a given unit of computing resource. Higher the work/unit ratio, higher the performance.*"

The unit of computing resource to measure performance can be one of the following:

- **Response time**: How much time a function or any unit of execution takes to execute in terms of real time (user time) and clock time (CPU time).
- **Latency**: How much time it takes for a system to get its stimulation, and then provide a response. An example is the time it takes for the request-response loop of a web application to complete, measured from the end-user perspective.
- **Throughput**: The rate at which a system processes its information. A system which has higher performance would usually have a higher throughput, and correspondingly higher scalability. An example is the throughput of an e-commerce website measured as the number of transactions completed per minute.

Performance is closely tied to scalability, especially, vertical scalability. A system that has excellent performance with respect to its memory management would easily scale up vertically by adding more RAM.

Similarly, a system that has multi-threaded workload characteristics and is written optimally for a multicore CPU, would scale up by adding more CPU cores.

Horizontal scalability is thought of as having no direct connection to the performance of a system within its own compute node. However, if a system is written in a way that it doesn't utilize the network effectively, thereby producing network latency issues, it may have a problem scaling horizontally effectively, as the time spent on network latency would offset any gain in scalability obtained by distributing the work.

Some dynamic programming languages such as Python have built-in scalability issues when it comes to scaling up vertically. For example, the **Global Interpreter Lock** (**GIL**) of Python (CPython) prevents it from making full use of the available CPU cores for computing by multiple threads.

Availability

Availability refers to the property of readiness of a software system to carry out its operations when the need arises.

Availability of a system is closely related to its reliability. The more reliable a system is, the more available it is.

Another factor which modifies availability is the ability of a system to recover from faults. A system may be very reliable, but if the system is unable to recover either from complete or partial failures of its subsystems, then it may not be able to guarantee availability. This aspect is called **recovery**.

The availability of a system can be defined as follows:

> "*Availability of a system is the degree to which the system is in a fully operable state to carry out its functionality when it is called or invoked at random.*"

Mathematically, this can be expressed as follows:

Availability = *MTBF*/(*MTBF* + *MTTR*)

Take a look at the following terms used in the preceding formula:

- **MTBF**: Mean time between failures
- **MTTR**: Mean time to repair

This is often called the **mission capable rate** of a system.

Techniques for **Availability** are closely tied to recovery techniques. This is due to the fact that a system can never be 100% available. Instead, one needs to plan for faults and strategies to recover from faults, which directly determines the availability. These techniques can be classified as follows:

- **Fault detection**: The ability to detect faults and take action helps to avert situations where a system or parts of a system become unavailable completely. Fault detection typically involves steps such as monitoring, heartbeat, and ping/echo messages, which are sent to the nodes in a system, and the response measured to calculate if the nodes are alive, dead, or are in the process of failing.

- **Fault recovery**: Once a fault is detected, the next step is to prepare the system to recover from the fault and bring it to a state where the system can be considered available. Typical tactics used here include Hot/Warm Spares (Active/Passive redundancy), Rollback, Graceful Degradation, and Retry.
- **Fault prevention**: This approach uses active methods to anticipate and prevent faults from occurring so that the system does not have a chance to go to recovery.

Availability of a system is closely tied to the consistency of its data via the CAP theorem which places a theoretical limit on the trade-offs a system can make with respect to consistency versus availability in the event of a network partition. The CAP theorem states that a system can choose between being consistent or being available—typically leading to two broad types of systems, namely, CP (consistent and tolerant to network failures) and AP (available and tolerant to network failures).

Availability is also tied to the system's scalability tactics, performance metrics, and its security. For example, a system that is highly horizontally scalable would have a very high availability, since it allows the load balancer to determine inactive nodes and take them out of the configuration pretty quickly.

A system which, instead, tries to scale up may have to monitor its performance metrics carefully. The system may have availability issues even when the node on which the system is fully available if the software processes are squeezed for system resources, such as CPU time or memory. This is where performance measurements become critical, and the system's load factor needs to be monitored and optimized.

With the increasing popularity of web applications and distributed computing, security is also an aspect that affects availability. It is possible for a malicious hacker to launch remote denial of service attacks on your servers, and if the system is not made foolproof against such attacks, it can lead to a condition where the system becomes unavailable or only partially available.

Security

Security, in the software domain, can be defined as the degree of ability of a system to avoid damage to its data and logic from unauthenticated access, while continuing to provide services to other systems and roles that are properly authenticated.

A security crisis or attack occurs when a system is intentionally compromised with a view to gaining illegal access to it in order to compromise its services, copy, or modify its data, or deny access to its legitimate users.

In modern software systems, the users are tied to specific roles which have exclusive rights to different parts of the system. For example, a typical web application with a database may define the following roles:

- **user**: End user of the system with login and access to his/her private data
- **dbadmin**: Database administrator, who can view, modify, or delete all database data
- **reports**: Report admin, who has admin rights only to those parts of database and code that deal with report generation
- **admin**: Superuser, who has edit rights to the complete system

This way of allocating system control via user roles is called **access control**. Access control works by associating a user role with certain system privileges, thereby decoupling the actual user login from the rights granted by these privileges.

This principle is the **Authorization** technique of security.

Another aspect of security is with respect to transactions where each person must validate the actual identity of the other. Public key cryptography, message signing, and so on are common techniques used here. For example, when you sign an e-mail with your GPG or PGP key, you are validating yourself—*The sender of this message is really me, Mr. A*—to your friend Mr. B on the other side of the e-mail. This principle is the **Authentication** technique of security.

The other aspects of security are as follows:

- **Integrity**: These techniques are used to ensure that data or information is not tampered with in anyway on its way to the end user. Examples are message hashing, CRC Checksum, and others.
- **Origin**: These techniques are used to assure the end receiver that the origin of the data is exactly the same as where it is purporting to be from. Examples of this are SPF, Sender-ID (for e-mail), Public Key Certificates and Chains (for websites using SSL), and others.
- **Authenticity**: These are the techniques which combine both the Integrity and Origin of a message into one. This ensures that the author of a message cannot deny the contents of the message as well as its origin (himself/herself). This typically uses **Digital Certificate Mechanisms**.

Deployability

Deployability is one of those quality attributes which is not fundamental to the software. However, in this book, we are interested in this aspect, because it plays a critical role in many aspects of the ecosystem in the Python programming language and its usefulness to the programmer.

Deployability is the degree of ease with which software can be taken from the development to the production environment. It is more of a function of the technical environment, module structures, and programming runtime/languages used in building a system, and has nothing to do with the actual logic or code of the system.

The following are some factors that determine deployability:

- **Module structures**: If your system has its code organized into well-defined modules/projects which compartmentalize the system into easily deployable subunits, the deployment is much easier. On the other hand, if the code is organized into a monolithic project with a single setup step, it would be hard to deploy the code into a multiple node cluster.
- **Production versus development environment**: Having a production environment which is very similar to the structure of the development environment makes deployment an easy task. When the environments are similar, the same set of scripts and toolchains that are used by the developers/DevOps team can be used to deploy the system to a development server as well as a production server with minor changes—mostly in the configuration.
- **Development ecosystem support**: Having a mature tool-chain support for your system runtime, which allows configurations such as dependencies to be automatically established and satisfied, increases deployability. Programming languages such as Python are rich in this kind of support in its development ecosystem, with a rich array of tools available for the DevOps professional to take advantage of.
- **Standardized configuration**: It is a good idea to keep your configuration structures (files, database tables, and others) the same for both developer and production environments. The actual objects or filenames can be different, but if the configuration structures vary widely across both the environments, deployability decreases, as extra work is required to map the configuration of the environment to its structures.

- **Standardized infrastructure:** It is a well-known fact that keeping your deployments to a homogeneous or standardized set of infrastructure greatly aids deployability. For example, if you standardize your frontend application to run on 4 GB RAM, Debian-based 64-bit Linux VPS, then it is easy to automate deployment of such nodes—either using a script, or by using elastic compute approaches of providers such as Amazon—and to keep a standard set of scripts across both development and production environments. On the other hand, if your production deployment consists of heterogeneous infrastructure, say, a mix of Windows and Linux servers with varying capacities and resource specifications, the work typically doubles for each type of infrastructure decreasing deployability.
- **Use of containers**: The user of container software, popularized by the advent of technology such as Docker and Vagrant built on top of Linux containers, has become a recent trend in deploying software on servers. The use of containers allows you to standardize your software, and makes deployability easier by reducing the amount of overhead required to start/stop the nodes, as containers don't come with the overhead of a full virtual machine. This is an interesting trend to watch for.

Summary

In this chapter, we learned about software architecture. We saw the different aspects of software architecture, and learned that every architecture comprises a system, which has a structure working in an environment for its stakeholders. We briefly looked at how software architecture differs from software design.

We went on to look at various characteristics of software architecture such as how a software architecture defines a structure, picks a core set of elements, and connects stakeholders.

We then addressed the important question of the importance of software architecture to an organization, and why it is a good idea to have a formal software architecture defined for your software systems.

The distinction of different roles of architects in an organization was discussed next. We saw the various roles system architects play in an organization, and how an Enterprise architect's focus is different from that of the System architect. The focus of strategy and technology breadth versus technology depth was clarified with illustrations.

We then discussed the elements of the main theme of this book—Architectural Quality Attributes. We defined what a quality attribute is, and then looked, in quite some detail, at the quality attributes of Modifiability, Testability, Scalability/Performance, Security, and Deployability. While going into the details of these attributes, we discussed their definitions, techniques, and how they relate to each other.

With this chapter serving as the base, we are now ready to take on these quality attributes, and then discuss in detail the various tactics and techniques to achieve them using the Python programming language. That forms the rest of this book.

In the next chapter, we'll start with one of the very first quality attributes we discussed in this chapter, namely, Modifiability and its associated attribute, Readability.

2
Writing Modifiable and Readable Code

In the first chapter, we discussed the various aspects of software architecture and covered some definitions of the terms involved. We looked at the different aspects of software architecture that an architect should be concerned with. Toward the end of the chapter, we discussed the various architectural quality attributes that an architect should focus on when building a system. We went in some detail into each of these attributes and looked at some definitions and various concerns that should be kept in mind when architecting a system for achieving these attributes.

From this chapter onward, we will focus on each of these quality attributes one by one, and discuss them in detail, per chapter. We will delve deep into an attribute—such as its various factors, techniques to achieve it, aspects to keep in mind when programming toward it, and so on. Since our focus in this book is on Python and its ecosystem, we will also look at various code examples and third-party software support that Python provides for achieving and maintaining these quality attributes.

The focus of this chapter is on the quality attribute of modifiability.

This chapter will cover the following topics:

- What is modifiability?
- Aspects related to modifiability
- Understanding readability
- Fundamentals of modifiability—cohesion and coupling
- Exploring strategies for modifiability
- Metrics—tools for static analysis
- Refactoring code

What is modifiability?

The architectural quality attribute of modifiability can be defined as follows:

> *Modifiability is the degree of ease with which changes can be made to a system, and the flexibility with which the system adapts to such changes.*

We discussed various aspects of modifiability in the first chapter, such as **cohesion**, **coupling**, and others. We will dig a little bit deeper into these aspects in this chapter with some examples. However, before we dig deeper, it might be a good idea to take a look at the big picture of how modifiability fits in with the other quality attributes that are related to it.

Aspects related to modifiability

We have already seen some aspects of modifiability in the previous chapter. Let's discuss this a bit further and look at some of the related quality attributes that are closely related to modifiability:

- **Readability**: Readability can be defined as the ease with which a program's logic can be followed and understood. Readable software is code that has been written in a specific style, following guidelines typically adopted for the programming language used, and whose logic uses the features provided by the language in a concise, clear way.
- **Modularity**: Modularity means that the software system is written in well-encapsulated modules, which do very specific, well-documented functions. In other words, modular code provides programmer friendly APIs to the rest of the system. Modifiability is closely connected to reusability.
- **Reusability**: This measures the number of parts of a software system, including code, tools, designs, and others, that can be reused in other parts of the system with zero or very little modifications. A good design would emphasize reusability from the beginning. Reusability is embodied in the DRY principle of software development.
- **Maintainability**: Maintainability of a software is the ease and efficiency with which the system can be updated and kept working in a useful state by its intended stakeholders. Maintainability is a metric, which encompasses the aspects of modifiability, readability, modularity and testability.

In this chapter, we are going to go deep into the readability, reusability, and modularity aspects. We will look at these one by one from the context of the Python programming language. We will start with readability first.

Understanding readability

The readability of a software system is closely tied to its modifiability. Well-written, well-documented code, keeping up with standard or adopted practices for the programming language, tends to produce simple, concise code that is easy to read and modify.

Readability is not only related to the aspect of following good coding guidelines, but it also ties up to how clear the logic is, how much the code uses standard features of the language, how modular the functions are, and so on.

In fact, we can summarize the different aspects of readability as follows:

- **Well-written**: A piece of code is well-written if it uses simple syntax and well-known features and idioms of the language, if the logic is clear and concise, and if it uses variables, functions, and class/module names meaningfully, that is, they express what they do.
- **Well-documented**: Documentation usually refers to the inline comments in the code. A well-documented piece of code tells what it does, what its input arguments are, and what is its return value (if any) along with the logic or algorithm, in some detail. It also documents any external library or API usage and configuration required for running the code either inline or in separate files.
- **Well-formatted**: Most programming languages, especially the open source languages like Python, developed over the internet via distributed but closely-knit programming communities, tend to have well-documented style guidelines. A piece of code that keeps up with these guidelines on aspects such as indentation and formatting will tend to be more readable than something that doesn't.

Lack of readability affects modifiability, and hence, maintainability of the code, thereby incurring ever-increasing costs for the organization in terms of resources—mainly people and time—in maintaining the system in a useful state.

Python and readability

Python is a language that has been designed from the ground-up for readability. To borrow a line from the well-known Zen of Python, we can say:

Readability counts

The Zen of Python is a set of 20 principles that influence the design of the Python programming language, 19 of which have been written down. You can see the Zen of Python by opening the Python interpreter prompt and typing this:

```
>>>import this
```

Python, as a language, emphasizes readability. It achieves this by clear, concise keywords, which mimic their English language counterparts, using minimal operators, and using the following philosophy:

There should be one—and preferably only one—obvious way to do it.

For example, here is one way to iterate through a sequence in Python while also printing its index:

```
for idx in range(len(seq)):
    item = seq[idx]
    print(idx, '=>', item)
```

However, a more common idiom used in Python is the `enumerate()` helper for iterators, which returns a two tuple of (`idx`, `item`) for each item in the sequence:

```
for idx, item in enumerate(seq):
    print(idx, '=>', item)
```

In many other programming languages such as C++ or Java, the first version would be considered with the same merit as the second version. However, in Python, there are certain idioms of writing code that keep up with the language's principles—the Zen—than certain others.

In this case, the second version is closer to the way Python programmers would write code to solve the problem. The first way would be considered less Pythonic than the second one.

The term "Pythonic" is something you would commonly encounter when interacting with the Python community. It means that the code not just solves the problem, but follows the conventions and idioms the Python community generally follows, and uses the language in the way it is intended to be used.

The definition of Pythonic is subjective, but you can think of it as Python code keeping up with the Zen of Python, or in general, following well-known idiomatic programming practices adopted by the community.

Python, by its design principles and clean syntax, makes writing readable code easy. However, it is a common trap for programmers migrating to Python from other more pedantic and less-idiomatic languages to write Python code in a less Pythonic way.

It is important for a Python programmer to understand this aspect early so that you tend to write more idiomatic or Pythonic code as you get used to the language more and more. You can be more productive with Python in the long term if you familiarize yourself with its coding principles and idioms than otherwise.

Readability – antipatterns

Python, in general, encourages and facilitates writing readable code. However, it would be, of course, very unrealistic to say that any code written in Python is highly readable. Even with all of its readability DNA, Python also has its fair share of difficult-to-read, badly written, or unreadable code as can be evident by spending some time scanning through some of the public, open source code written in Python on the web.

There are certain practices that tend to produce difficult-to-read or unreadable code in a programming language. These can be thought of as antipatterns, which are a bane, not just in programming with Python, but in any programming language:

- **Code with little or no comments**: Lack of code comments is often the primary reason for producing code that is unreadable. More often than not, programmers don't do a very good job of documenting their thoughts, which led to a particular implementation, in code. When the same code is read by another programmer or by the same programmer a few months later (this happens quite a lot!), it is not easy to figure out why the specific implementation approach was followed. This makes it difficult to reason about the pros and cons of an alternate approach.

  This also makes taking decisions on modifying the code—perhaps for a customer fix—difficult, and in general, affects code modifiability in the long term. The commenting of code is often an indicator of the discipline and rigor of the programmer who wrote the code and of the organization in enforcing such practices.

- **Code that breaks best practices of the language**: Best practices of a programming language typically evolve from years of experience in using the language by a community of developers, and the efficient feedback that it generates. They capture the best way of putting the programming language to good use to solve problems, and typically, capture the idioms and common patterns for using the language.

For example, in Python, the Zen can be considered as a shining torch to its best practices and the set of common programming idioms adopted by the community.

Often, programmers who are either inexperienced or those who migrate from other programming languages or environments tend to produce code that is not in keeping with these practices, and hence, end up writing code that is low on readability.

- **Programming antipatterns**: There are a large number of coding or programming antipatterns, which tend to produce difficult-to-read, and hence, difficult-to-maintain code. Here are some of the well-known ones:
    - **Spaghetti code**: This is a piece of code with no discernible structure or control-flow. It is typically produced by following complex logic with a lot of unconditional jumps and unstructured exception handling, badly written concurrent code and so on.
    - **Big ball of mud**: This is a system with pieces of code that show no overall structure or goal. Big ball of mud typically consists of many pieces of spaghetti code and is usually a sign of code that has been worked on by multiple people, patched-up multiple times with little or zero documentation.
    - **Copy-Paste programming**: Often produced in organizations where expediency of delivery is favored over thoughtful design, copy/paste coding produces long, repetitive chunks of code, which essentially do the same thing again and again with minor modifications. This leads to code-bloat and, in the long term, the code becomes unmaintainable.

      A similar antipattern is *cargo-cult programming*, where programmers follows the same design or programming pattern over and over again without a thought to whether it fits the specific scenarios or problems that they are trying to solve.
    - **Ego programming**: Ego programming is where a programmer—often an experienced one—favors their personal style over the documented best practices or the organizational style of coding. This sometimes creates code that is cryptic and difficult to read for the other—usually, younger or less-experienced programmers. An example is the tendency to use functional programming constructs in Python to write everything as a one-liner.

Coding antipatterns can be circumvented by adopting practices of structured programming in your organization, and by enforcing the use of coding guidelines and best practices.

The following are some antipatterns that are specific to Python:

- **Mixed indentation**: Python uses indentation to separate blocks of code, as it lacks braces or other syntactical constructs of languages such as C/C++ or Java, which separate code blocks. However, we need to be careful when indenting code in Python. A common antipattern is where people mix both tabs (the `\t` character) and spaces in their Python code. This can be fixed by using editors that always use either tabs or spaces to indent code.

  Python comes with built-in modules such as `tabnanny`, which can be used to check your code for indentation issues.
- **Mixing string literal types**: Python provides three different ways to create string literals: either by using the single quote (`'`), the double quote (`"`), or Python's own special triple quote (`'''` or `"""`). Code that mixes these three types of literals in the same block of code or functional unit becomes more difficult to read.
- **Overuse of functional constructs**: Python, being a mixed paradigm language, provides support for functional programming via its lambda keyword and its `map()`, `reduce()`, and `filter()` functions. However, sometimes, experienced programmers or programmers coming from a background of functional programming to Python overuse these constructs, producing code that is too cryptic and, hence, unreadable to other programmers.

Techniques for readability

Now that we have a good knowledge on what helps readability of code, let's look at the approaches that we can adopt in order to improve the readability of code in Python.

Document your code

A simple and effective way to improve the readability of your code is to document what it does. Documentation is important for readability and long term modifiability of your code.

Code documentation can be categorized as follows:

- **Inline documentation**: The programmer documents their code by using code comments, function documentation, module documentation, and others as part of the code itself. This is the most effective and useful type of code documentation.
- **External documentation**: These are additional documentation captured in separate files, which usually document aspects such as usage of code, code changes, install steps, deployment, and the like. Examples are the `README`, `INSTALL`, or `CHANGELOG` files usually found with open source projects keeping up with the GNU build principles.
- **User manuals**: These are formal documents, usually by a dedicated person or team, using pictures and text that is usually targeted toward users of the system. Such documentation is usually prepared and delivered toward the end of a software project when the product is stable and is ready to ship. We are not concerned with this type of documentation in our discussion here.

Python is a language that is designed for smart inline code documentation from the ground up. In Python, inline documentation can be done at the following levels:

- **Code comments**: This is the text inline with code, prefixed by the hash (#) character. They can be used liberally inside your code explaining what each step of the code does.

  Here is an example:

```
# This loop performs a network fetch of the URL, retrying up to 3
# times in case of errors. In case the URL can't be fetched,
# an error is returned.

# Initialize all state
count, ntries, result, error = 0, 3, None, None
while count < ntries:
    try:
        # NOTE: We are using an explicit   timeout of 30s here
        result = requests.get(url, timeout=30)
    except Exception as error:
        print('Caught exception', error, 'trying again after a
             while')
      # increment count
      count += 1
      # sleep 1 second every time
      time.sleep(1)

  if result == None:
```

```
        print("Error, could not fetch URL",url)
        # Return a tuple of (<return code>, <lasterror>)
        return (2, error)

    # Return data of URL
        return result.content
```

Notice the liberal use of comments even in places it may be deemed superfluous. We will look at some general rules of thumb in commenting your code later.

- **The docstring function**: Python provides a simple way to document what a function does by using a string literal just below the function definition. This can be done by using any of the three styles of string literals.

  Here is an example:

```
def fetch_url(url, ntries=3, timeout=30):
     " Fetch a given url and return its contents "

    # This loop performs a network fetch of the URL, retrying
    # up to
    # 3 times in case of errors. In case the URL can't be
    # fetched,
    # an error is returned.

    # Initialize all state
    count, result, error = 0, None, None
    while count < ntries:
        try:
            result = requests.get(url, timeout=timeout)
        except Exception as error:
            print('Caught exception', error, 'trying again
                   after a while')
            # increment count
            count += 1
            # sleep 1 second every time
            time.sleep(1)

    if result == None:
        print("Error, could not fetch URL",url)
    # Return a tuple of (<return code>, <lasterror>)
    return (2, error)

    # Return data of URL
    return result.content
```

The function docstring is the line that says *fetch a given URL and return its contents*. However, though it is useful, the usage is limited, since it only says what the function does and doesn't explain its parameters. Here is an improved version:

```
def fetch_url(url, ntries=3, timeout=30):
        """ Fetch a given url and return its contents.

        @params
            url - The URL to be fetched.
            ntries - The maximum number of retries.
            timeout - Timout per call in seconds.

        @returns
            On success - Contents of URL.
            On failure - (error_code, last_error)
        """

        # This loop performs a network fetch of the URL,
        # retrying up to
        # 'ntries' times in case of errors. In case the URL
        # can't be fetched, an error is returned.

        # Initialize all state
        count, result, error = 0, None, None
        while count < ntries:
            try:
                result = requests.get(url, timeout=timeout)
            except Exception as error:
                print('Caught exception', error, 'trying again
                       after a while')
                # increment count
                count += 1
                # sleep 1 second every time
                time.sleep(1)

        if result == None:
            print("Error, could not fetch URL",url)
            # Return a tuple of (<return code>, <lasterror>)
            return (2, error)

        # Return data of the URL
        return result.content
```

In the preceding code, the function usage has become much clearer to the programmer. Note that such extended documentation would usually span more than one line, and hence, it is a good idea to always use triple quotes with your function docstrings.

- **Class docstrings**: These work just like a function docstring except that they provide documentation for a class directly. This is provided just below the class keyword defining the class.

  Here is an example:

```
class UrlFetcher(object):
     """ Implements the steps of fetching a URL.

         Main methods:
         fetch - Fetches the URL.
         get - Return the URLs data.
     """

    def __init__(self, url, timeout=30, ntries=3, headers={}):
        """ Initializer.
        @params
            url - URL to fetch.
            timeout - Timeout per connection (seconds).
            ntries - Max number of retries.
            headers - Optional request headers.
        """
        self.url = url
        self.timeout = timeout
        self.ntries = retries
        self.headers = headers
        # Enapsulated result object
        self.result = result

    def fetch(self):
        """ Fetch the URL and save the result """

        # This loop performs a network fetch of the URL,
          # retrying
        # up to 'ntries' times in case of errors.

        count, result, error = 0, None, None
        while count < self.ntries:
```

```
                try:
                    result = requests.get(self.url,
                                          timeout=self.timeout,
                                          headers = self.headers)
                except Exception as error:
                    print('Caught exception', error, 'trying again
                           after a while')
                    # increment count
                    count += 1
                    # sleep 1 second every time
                    time.sleep(1)

            if result != None:
                # Save result
                self.result = result

        def get(self):
            """ Return the data for the URL """

            if self.result != None:
                return self.result.content
```

See how the class docstring defines some of the main methods of the class. This is a very useful practice, as it gives the programmer useful information at the top level without having to go and inspect each function's documentation separately.

- **Module docstrings**: Module docstrings capture information at the module level, usually about the functionality of the module and some detail about what each member of the module (function, class, and others) does. The syntax is the same as the class or function docstring. The information is usually captured at the top of the module, before any code.

  A module documentation can also capture any specific external dependencies of a module:

```
"""
    urlhelper - Utility classes and functions to work with URLs.

    Members:

        # UrlFetcher - A class which encapsulates action of
        # fetching content of a URL.
```

```
        # get_web_url - Converts URLs so they can be used on the
        # web.
        # get_domain - Returns the domain (site) of the URL.
"""

import urllib

def get_domain(url):
    """ Return the domain name (site) for the URL"""

    urlp = urllib.parse.urlparse(url)
    return urlp.netloc

def get_web_url(url, default='http'):
    """ Make a URL useful for fetch requests
    -  Prefix network scheme in front of it if not present already
    """

    urlp = urllib.parse.urlparse(url)
    if urlp.scheme == '' and urlp.netloc == '':
          # No scheme, prefix default
      return default + '://' + url

    return url

class UrlFetcher(object):
     """ Implements the steps of fetching a URL.

        Main methods:
        fetch - Fetches the URL.
        get - Return the URLs data.
    """

    def __init__(self, url, timeout=30, ntries=3, headers={}):
        """ Initializer.
        @params
            url - URL to fetch.
            timeout - Timeout per connection (seconds).
            ntries - Max number of retries.
            headers - Optional request headers.
        """
        self.url = url
        self.timeout = timeout
        self.ntries = retries
```

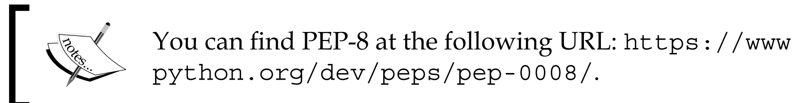
```
            self.headers = headers
            # Enapsulated result object
            self.result = result

        def fetch(self):
            """ Fetch the URL and save the result """

            # This loop performs a network fetch of the URL, retrying
            # up to 'ntries' times in case of errors.

            count, result, error = 0, None, None
            while count < self.ntries:
                try:
                    result = requests.get(self.url,
                                          timeout=self.timeout,
                                          headers = self.headers)
                except Exception as error:
                    print('Caught exception', error, 'trying again
                           after a while')
                    # increment count
                    count += 1
                    # sleep 1 second every time
                    time.sleep(1)

            if result != None:
                # Save result
                self.result = result

        def get(self):
            """ Return the data for the URL """

            if self.result != None:
                return self.result.content
```

Follow coding and style guidelines

Most programming languages have a relatively well-known set of coding and/or style guidelines. These are either developed over many years of use as a convention, or come as a result of discussions in the online community of that programming language. C/C++ is a good example of the former, and Python is a good example of the latter.

It is also a common practice for companies to specify their own guidelines—mostly, by adopting existing standard guidelines and customizing them for the company's own specific development environment and requirements.

For Python, there is a clear set of coding style guidelines published by the Python programming community. This guideline, known as PEP-8, is available online as part of the **Python Enhancement Proposal** (**PEP**) set of documents.



You can find PEP-8 at the following URL: `https://www.python.org/dev/peps/pep-0008/`.

PEP-8 was first created in 2001 and has undergone multiple revisions since then. The primary author is the creator of Python, Guido Van Rossum, with input from Barry Warsaw and Nick Coghlan.

PEP-8 was created by adapting Guido's original *Python Style Guide* essay with additions from Barry's style guide.

We will not go deep into PEP-8 in this book, as the goal of this section is not to teach you PEP-8. However, we will discuss the general principles underlying PEP-8.

The philosophy underlying PEP-8 can be summarized as follows:

- Code is read more than it is written. Hence, providing a guideline would make code more readable and make it consistent across a full spectrum of Python code.
- Consistency within a project is important. However, consistency within a module or package is more important. Consistency within a unit of code—such as class or function is the most important.

- Know when to ignore a guideline. For example, this may happen if adopting the guideline makes your code less readable, breaks the surrounding code, or breaks backward compatibility of the code. Study examples, and choose what is best.
- If a guideline is not directly applicable or useful for your organization, customize it. If you have any doubts about a guideline, get clarification by asking the Python community.

Review and refactor code

Code requires maintenance. Unmaintained code that is used in production can become a problem if not tended to periodically.

Periodically scheduled reviews of code can be very useful in keeping the code readable and in good health aiding modifiability and maintainability. Code that is central to a system or an application in production tends to get a lot of quick-fixes over time, as it is customized or enhanced for different use cases or patched for issues. It is observed that programmers generally don't document such quick fixes, as the situations demand expedite testing and deployment over good engineering practices such as documentation.

Over time, such patches can accumulate, thereby causing code-bloat and creating future engineering debts for the team, which can become a costly affair. The solution is periodical reviews.

Reviews should be done with engineers who are familiar with the application, but ideally, who are not working on the same code. This gives the code a fresh set of eyes, which is often useful in detecting bugs that the original author(s) may have overlooked. It is a good idea to get large changes reviewed by a couple of reviewers who are experienced developers.

This can be combined with the general refactoring of code to improve implementation, reduce coupling, or increase cohesion.

Commenting the code

We are coming toward the end of our discussions on readability of code, and it is a good time to introduce some general rules of thumb to follow when writing code comments. These can be listed as follows:

- Comments should be descriptive, and should explain the code. A comment that simply repeats what is obvious from the function name is not very useful.

  Here is an example. Both of the following codes show the same implementation of a **Root-Mean-Squared** (**RMS**) velocity calculation, but the second version has a much more useful docstring than the first:

  ```
  def rms(varray=[]):
      """ RMS velocity """

      squares = map(lambda x: x*x, varray)
      return pow(sum(squares), 0.5)

  def rms(varray=[]):
      """ Root mean squared velocity. Returns
      square root of sum of squares of velocities """

      squares = map(lambda x: x*x, varray)
      return pow(sum(squares), 0.5)
  ```

- Code comments should be written in the block we are commenting on, rather than as follows:

  ```
  # This code calculates the sum of squares of velocities
  squares = map(lambda x: x*x, varray)
  ```

  The preceding version is much clearer than the following version, which uses comments below the code:

  ```
  squares = map(lambda x: x*x, varray)
  # The above code calculates the sum of squares of velocities
  ```

- Inline comments should be used as little as possible. This is because it is very easy to get these confused as part of the code itself, especially if the separating comment character is accidentally deleted, causing bugs:

  ```
  squares = map(lambda x: x*x, varray)   # Calculate squares of
  velocities
  ```

- Try to avoid comments that are superfluous and add little value:

  ```
  # The following code iterates through odd numbers
  for num in nums:
      # Skip if number is odd
      if num % 2 == 0: continue
  ```

  The second comment in the last piece of code adds little value and can be omitted.

Fundamentals of modifiability – cohesion and coupling

Let's now get back to the main topic of modifiability and discuss the two fundamental aspects that affect modifiability of code—namely, cohesion and coupling.

We've already discussed these concepts briefly in the first chapter. Let's do a quick review here.

Cohesion refers to how tightly the responsibilities of a module are related to each other. A module that performs a specific task or group of related tasks has high cohesion. A module in which a lot of functionality is dumped without a thought as to the core functionality would have low cohesion.

Coupling is the degree to which the functionality of two modules, A and B, are related. Two modules are strongly coupled if their functionality overlaps strongly at the code level—in terms of function or method calls. Any changes in module A would probably require changes in module B.

Strong coupling is always prohibitory for modifiability, as it increases the cost of maintaining the code base. Code which aims to increase modifiability should aim for high cohesion and low coupling.

We will analyze cohesion and coupling in the following subsections with some examples.

Measuring cohesion and coupling

Let's look at a simple example of two modules to figure out how we can measure coupling and cohesion quantitatively. The following is the code for module A, which purportedly implements functions that operate with a series (array) of numbers:

```
"" Module A (a.py) – Provides functions that operate on series of
numbers """

def squares(narray):
    """ Return array of squares of numbers """
    return pow_n(array, 2)

def cubes(narray):
    """ Return array of cubes of numbers """
    return pow_n(narray, 3)

def pow_n(narray, n):
    """ Return array of numbers raised to arbitrary power n each """
    return [pow(x, n) for x in narray]

def frequency(string, word):
    """ Find the frequency of occurrences of word in string
    as percentage """

    word_l = word.lower()
    string_l = string.lower()

    # Words in string
    words = string_l.split()
    count = w.count(word_l)

    # Return frequency as percentage
    return 100.0*count/len(words)
```

Next is the listing of module B:

```
""" Module B (b.py) - Provides functions implementing some statistical
methods """

import a

def rms(narray):
```

```
        """ Return root mean square of array of numbers"""
        return pow(sum(a.squares(narray)), 0.5)

    def mean(array):
        """ Return mean of an array of numbers """
        return 1.0*sum(array)/len(array)

    def variance(array):
        """ Return variance of an array of numbers """

        # Square of variation from mean
        avg = mean(array)
        array_d = [(x - avg) for x in array]
        variance = sum(a.squares(array_d))
        return variance

    def standard_deviation(array):
        """ Return standard deviation of an array of numbers """

        # S.D is square root of variance
        return pow(variance(array), 0.5)
```

Let's do an analysis of the functions in module A and B. Here is the report:

Module	Core functions	Unrelated functions	Function dependencies
B	4	0	3 x 1 = 3
A	3	1	0

This has four functions that can be explained as follows:

- Module B has four functions, all of them dealing with the core functionality. There are no unrelated functions in this module. Module B has `100%` cohesion.
- Module A has four functions, three of which are related to its core functionality, but the last one (frequency) isn't. This gives module A approximately 75% cohesion.
- Three of the module B functions depend on one function in module A, namely, squares. This makes module B strongly coupled to module A. Coupling at function level is 75% from module B → A.
- Module A doesn't depend on any functionality of module B. Module A will work independent of module B. Coupling from module A → B is zero.

Let's now look at how we can improve the cohesion of module A. In this case, it is as simple as dropping the last function, which doesn't really belong there. It could be dropped out entirely or moved to another module.

Here is the rewritten module A code, now with 100% cohesion with respect to its responsibilities:

```
""" Module A (a.py) – Implement functions that operate on series of
numbers """

def squares(narray):
    """ Return array of squares of numbers """
    return pow_n(array, 2)

def cubes(narray):
    """ Return array of cubes of numbers """
    return pow_n(narray, 3)

def pow_n(narray, n):
    """ Return array of numbers raised to arbitrary power n each """
    return [pow(x, n) for x in narray]
```

Let's now analyze the quality of coupling from module B→ A and look at the risk factors of modifiability of code in B with respect to code in A, which are as follows:

- The three functions in B depend on just one function in module A.
- The function is named *squares*, which accepts an array and returns each of its member *squared*.
- The function signature (API) is simple, so chances of changing the function signature in the future is less.
- There is no two-way coupling in the system. The dependency is only from the direction B → A.

In other words, even though there is strong coupling from B to A, it is good coupling and doesn't affect the modifiability of the system in any way at all.

Let's now look at another example.

Measuring cohesion and coupling – string and text processing

Let's consider a different use case now, an example with modules that do a lot of string and text processing:

```
""" Module A (a.py) - Provides string processing functions """
import b

def ntimes(string, char):
    """ Return number of times character 'char'
    occurs in string """

    return string.count(char)

def common_words(text1, text2):
    """ Return common words across text1 and text2"""

    # A text is a collection of strings split using newlines
    strings1 = text1.split("\n")
    strings2 = text2.split("\n")

    common = []
    for string1 in strings1:
        for string2 in strings2:
            common += b.common(string1, string2)

    # Drop duplicates
    return list(set(common))
```

Next is the listing of module B, which is as follows:

```
""" Module B (b.py) - Provides text processing functions to user """

import a

def common(string1, string2):
    """ Return common words across strings1 1 & 2 """

    s1 = set(string1.lower().split())
    s2 = set(string2.lower().split())
    return s1.intersection(s2)
```

```
def common_words(text1, text2):
    """ Return common words across two input files """

    lines1 = open(filename1).read()
    lines2 = open(filename2).read()

    return a.common_words(lines1, lines2)
```

Let's go through the coupling and cohesion analysis of these modules, given in following table:

Module	Core functions	Unrelated functions	Function dependencies
B	2	0	1 x 1 = 1
A	2	0	1 x 1 = 1

Here is an explanation of these numbers in the table:

- Module A and B have two functions each, each of them dealing with the core functionality. Modules A and B both have 100% cohesion.
- One function of module A is dependent on one function of module B. Similarly, one function of module B is dependent on one function of module A. There is strong coupling from A→ B and from B → A. In other words, the coupling is bidirectional.

Bidirectional coupling between two modules ties their modifiability to each other very strongly. Any changes in module A will quickly cascade to behavior of module B and vice versa. In other words, this is bad coupling.

Exploring strategies for modifiability

Now that we have seen some examples of good and bad coupling and cohesion, let's get to the strategies and approaches that a software architect can adopt to improve the modifiability of the software system.

Providing explicit interfaces

A module should mark a set of functions, classes, or methods as the **interface** it provides to external code. This can be thought of as the API of this module. Any external code that uses this API would become a client to the module.

Methods or functions that the module considers internal to its function, and which do not make up its API, should either be explicitly made private to the module or should be documented as such.

In Python, which doesn't provide variable access scope for functions or class methods, this can be done by conventions such as prefixing the function name with a single or double underscore, thereby signaling to potential clients that these functions are internal and shouldn't be referred to from outside.

Reducing two-way dependencies

As seen in the examples earlier, coupling between two software modules is manageable if the coupling direction is one-way. However, bidirectional coupling creates very strong linkages between modules, which can complicate the usage of the modules and increase their maintenance costs.

In Python, which uses reference-based garbage collection, this may also create cryptic referential loops for variables and objects, thereby making garbage collection difficult.

Bidirectional dependencies can be broken by refactoring the code in such a way that a module always uses the other one and not vice versa. In other words, encapsulate all related functions in the same module.

Here are our modules A and B of the earlier example, rewritten to break their bidirectional dependency:

```
""" Module A (a.py) - Provides string processing functions """

def ntimes(string, char):
    """ Return number of times character 'char'
    occurs in string """

    return string.count(char)

def common(string1, string2):
    """ Return common words across strings1 1 & 2 """

    s1 = set(string1.lower().split())
    s2 = set(string2.lower().split())
    return s1.intersection(s2)

def common_words(text1, text2):
    """ Return common words across text1 and text2"""
```

```
        # A text is a collection of strings split using newlines
        strings1 = text1.split("\n")
        strings2 = text2.split("\n")

        common_w = []
        for string1 in strings1:
            for string2 in strings2:
                common_w += common(string1, string2)

        return list(set(common_w))
```

Next is the listing of module B:

```
""" Module B (b.py) - Provides text processing functions to user """

import a

def common_words(filename1, filename2):
  """ Return common words across two input files """

  lines1 = open(filename1).read()
  lines2 = open(filename2).read()

  return a.common_words(lines1, lines2)
```

We achieved this by simply moving the `common` function, which picks common words from two strings from module B to A. This is an example of refactoring to improve modifiability.

Abstract common services

Usage of helper modules that abstract common functions and methods can reduce coupling between two modules and increase their cohesion. For example, in the first example, module A acts as a helper module for module B.

Helper modules can be thought of as intermediaries or mediators, which abstract common services for other modules so that the dependent code is all available in one place without duplication. They can also help modules to increase their cohesion by moving out unwanted or unrelated functions.

Using inheritance techniques

When we find similar code or functionality occurring in classes, it might be a good time to refactor them so as to create class hierarchies so that common code is shared by virtue of inheritance.

Let's take a look at the following example:

```
""" Module textrank - Rank text files in order of degree of a specific
word frequency. """

import operator

class TextRank(object):
    """ Accept text files as inputs and rank them in
    terms of how much a word occurs in them """

    def __init__(self, word, *filenames):
        self.word = word.strip().lower()
        self.filenames = filenames

    def rank(self):
        """ Rank the files. A tuple is returned with
        (filename, #occur) in decreasing order of
        occurences """

        occurs = []

        for fpath in self.filenames:
            data = open(fpath).read()
            words = map(lambda x: x.lower().strip(), data.split())
            # Filter empty words
            count = words.count(self.word)
            occurs.append((fpath, count))

        # Return in sorted order
        return sorted(occurs, key=operator.itemgetter(1),
                      reverse=True)
```

Here is another module, `urlrank`, which performs the same function on URLs:

```
""" Module urlrank - Rank URLs in order of degree of a specific
word frequency """
import operator
```

```
import operator
import requests

class UrlRank(object):
    """ Accept URLs as inputs and rank them in
    terms of how much a word occurs in them """

    def __init__(self, word, *urls):
        self.word = word.strip().lower()
        self.urls = urls

    def rank(self):
        """ Rank the URLs. A tuple is returned with
        (url, #occur) in decreasing order of
        occurences """

        occurs = []

        for url in self.urls:
            data = requests.get(url).content
            words = map(lambda x: x.lower().strip(), data.split())
            # Filter empty words
            count = words.count(self.word)
            occurs.append((url, count))

        # Return in sorted order
        return sorted(occurs, key=operator.itemgetter(1),
                      reverse=True)
```

Both these modules perform similar functions of ranking a set of input data in terms of how much a given keyword appears in them. Over time, these classes could develop a lot of similar functionality, and the organization could end up with a lot of duplicate code, reducing modifiability.

We can use inheritance to help us here to abstract away the common logic in a parent class. Here is the parent class named `RankBase`, which accomplishes this by abstracting all common code as part of its API:

```
""" Module rankbase - Logic for ranking text using degree of word
frequency """

import operator

class RankBase(object):
```

```
    """ Accept text data as inputs and rank them in
    terms of how much a word occurs in them """

    def __init__(self, word):
        self.word = word.strip().lower()

    def rank(self, *texts):
        """ Rank input data. A tuple is returned with
        (idx, #occur) in decreasing order of
        occurences """

        occurs = {}

        for idx,text in enumerate(texts):
            words = map(lambda x: x.lower().strip(), text.split())
            count = words.count(self.word)
            occurs[idx] = count

        # Return dictionary
        return occurs

    def sort(self, occurs):
        """ Return the ranking data in sorted order """

        return sorted(occurs, key=operator.itemgetter(1),
                      reverse=True)
```

We now have the `textrank` and `urlrank` modules rewritten to take advantage of the logic in the parent class:

```
""" Module textrank - Rank text files in order of degree of a specific
word frequency. """

import operator
from rankbase import RankBase

class TextRank(object):
    """ Accept text files as inputs and rank them in
    terms of how much a word occurs in them """

    def __init__(self, word, *filenames):
        self.word = word.strip().lower()
        self.filenames = filenames

    def rank(self):
```

```
        """ Rank the files. A tuple is returned with
        (filename, #occur) in decreasing order of
        occurences """

        texts = map(lambda x: open(x).read(), self.filenames)
        occurs = super(TextRank, self).rank(*texts)
        # Convert to filename list
        occurs = [(self.filenames[x],y) for x,y in occurs.items()]

        return self.sort(occurs)
```

Here is the modified listing for the `urlrank` module:

```
""" Module urlrank - Rank URLs in order of degree of a specific word
frequency """

import requests
from rankbase import RankBase

class UrlRank(RankBase):
    """ Accept URLs as inputs and rank them in
    terms of how much a word occurs in them """

def __init__(self, word, *urls):
    self.word = word.strip().lower()
    self.urls = urls

def rank(self):
    """ Rank the URLs. A tuple is returned with
    (url, #occur) in decreasing order of
    occurences"""

    texts = map(lambda x: requests.get(x).content, self.urls)
    # Rank using a call to parent class's 'rank' method
    occurs = super(UrlRank, self).rank(*texts)
    # Convert to URLs list
    occurs = [(self.urls[x],y) for x,y in occurs.items()]

    return self.sort(occurs)
```

Not only has refactoring reduced the size of the code in each module, but it has also resulted in improved modifiability of the classes by abstracting the common code to a parent class which can be developed independently.

Using late binding techniques

Late binding refers to the practice of postponing the binding of values to parameters as late as possible in the order of execution of a code. Late binding allows the programmer to defer the factors that influence code execution, and hence the results of execution and performance of the code, to a later time by making use of multiple techniques.

Some late-binding techniques that can be used are as follows:

- **Plugin mechanisms**: Rather than statically binding modules together, which increases coupling, this technique uses values resolved at runtime to load plugins that execute a specific dependent code. Plugins can be Python modules whose names are fetched during computations done at runtime or via IDs or variable names loaded from database queries or from configuration files.
- **Brokers/registry lookup services**: Some services can be completely deferred to brokers, which look up the service names from a registry on demand, and call them dynamically and return results. An example may be a currency exchange service, which accepts a specific currency transformation as input (say USDINR), and looks up and configures a service for it dynamically at runtime, thereby requiring only the same code to execute on the system at all times. Since there is no dependent code on the system that varies with the input, the system remains immune from any changes required if the logic for the transformation changes, as it is deferred to an external service.
- **Notification services**: Publish/subscribe mechanisms, which notify subscribers when the value of an object changes or when an event is published, can be useful to decouple systems from a volatile parameter and its value. Rather than tracking changes to such variables/objects internally, which may need a lot of dependent code and structures, such systems keep their clients immune to the changes in the system that affect and trigger the objects' internal behavior, but bind them only to an external API, which simply notifies the clients of the changed value.
- **Deployment time binding**: By keeping the variable values associated to names or IDs in configuration files, we can defer object/variable binding to deployment time. The values are bound at startup by the software system once it loads its configuration files, which can then invoke specific paths in the code that creates appropriate objects.

  This approach can be combined with object-oriented patterns such as factories, which create the required object at runtime given the name or ID, hence keeping the clients that are dependent on these objects immune from any internal changes, increasing their modifiability.

- **Using creational patterns**: Creational design patterns such as factory or builder, which abstract the task of creating of an object from the details of creating it, are ideal for separation of concerns for client modules that don't want their code to be modified when the code for creation of a dependent object changes.

  These approaches, when combined with deployment/configuration time or dynamic binding (using lookup services), can greatly increase the flexibility of a system and aid its modifiability.

We will look at examples of Python patterns in a later chapter in this book.

Metrics – tools for static analysis

Static code analysis tools can provide a rich summary of information on the static properties of your code, which can provide insights into aspects such as complexity and modifiability/readability of the code.

Python has a lot of third-party tool support, which helps in measuring the static aspects of Python code such as these:

- Conformance to coding standards such as PEP-8
- Code complexity metrics such as the McCabe metric
- Errors in code such as syntax errors, indentation issues, missing imports, variable overwrites, and others
- Logic issues in code
- Code smells

The following are some of the most popular tools in the Python ecosystem that can perform such static analysis:

- **Pylint**: Pylint is a static checker for Python code, which can detect a range of coding errors, code smells, and style errors. Pylint uses a style close to PEP-8. The newer versions of Pylint also provide statistics about code complexity and can print reports. Pylint requires the code to be executed before checking it. You can refer to `http://pylint.org`.
- **Pyflakes**: Pyflakes is a more recent project than Pylint. It differs from Pylint in that it need not execute the code before checking it for errors. Pyflakes does not check for coding style errors and only performs logic checks in code. You can refer to `https://launchpad.net/pyflakes`.

- **McCabe**: It is a script that checks and prints a report on the McCabe complexity of your code. You can refer to `https://pypi.python.org/pypi/mccabe`.
- **Pycodestyle**: Pycodestyle is a tool that checks your Python code against some of the PEP-8 guidelines. This tool was earlier called PEP-8. Refer to `https://github.com/PyCQA/pycodestyle`.
- **Flake8**: Flake8 is a wrapper around the Pyflakes, McCabe, and pycodestyle tools and can perform a number of checks including the ones provided by these tools. Refer to `https://gitlab.com/pycqa/flake8/`.

What are code smells?

Code smells are surface symptoms of deeper problems with your code. They usually indicate problems with the design, which can cause bugs in the future or negatively impact development of the particular piece of code.

Code smells are not bugs themselves, but they are patterns that indicate that the approach to solving problems adopted in the code is not right and should be fixed by refactoring.

Some of the common code smells are as follows:

At the class level, there are the following:

- **God object**: A class that tries to do too many things. In short, this class lacks any kind of cohesion.
- **Constant class**: A class that's nothing but a collection of constants, which is used elsewhere, and hence, should not ideally belong here.
- **Refused bequest**: A class that doesn't honor the contract of the base class, and hence, breaks the substitution principle of inheritance.
- **Freeloader**: A class with too few functions that do almost nothing and add little value.
- **Feature envy**: A class that is excessively dependent on methods of another class, indicating high coupling.

At the method/function level, there are the following:

- **Long method**: A method or function that has grown too big and complex.
- **Parameter creep**: This is when there are too many parameters for a function or method. This makes the callability and testability of the function difficult.
- **Cyclomatic complexity**: This is a function or method with too many branches or loops, which creates a convoluted logic that is difficult to follow and can cause subtle bugs. Such a function should be refactored and broken down to multiple functions, or the logic rewritten to avoid too much branching.
- **Overly long or short identifiers**: A function that uses either overly long or overly short variable names so that their purpose is not clear from their names. The same is applicable to the function name as well.

A related antipattern to code smell is design smell, which are the surface symptoms in the design of a system that indicate underlying deeper problems in the architecture.

Cyclomatic complexity – the McCabe metric

Cyclomatic complexity is a measure of complexity of a computer program. It is computed as the number of linearly independent paths through the program's source code from start to finish.

For a piece of code with no branches at all, such as the one given next, the Cyclomatic complexity would be 1, as there is just one path through the code:

```
""" Module power.py """

def power(x, y):
    """ Return power of x to y """
    return x^y
```

A piece of code with two branches, like the following one, will have a complexity of 2:

```
""" Module factorial.py """

def factorial(n):
    """ Return factorial of n """
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

The use of Cyclomatic complexity as a metric using the control graph of a code was developed by Thomas J. McCabe in 1976. Hence, it is also called McCabe complexity or the **McCabe index**.

To measure the metric, the control graph can be pictured as a directed graph, where the nodes represent the blocks of the program and edges represent control flow from one block to another.

With respect to the control graph of a program, the McCabe complexity can be expressed as follows:

$$M = E - N + 2P$$

In the preceding equation, we have the following:

- E => Number of edges in the graph
- N => Number of nodes in the graph
- P => Number of connected components in the graph

In Python, the `mccabe` package, written by Ned Batcheldor, can be used to measure a program's Cyclomatic complexity. It can be used as a standalone module or as a plugin to programs such as Flake8 or Pylint.

For example, here is how we measure the Cyclomatic complexity of the two code pieces given earlier:

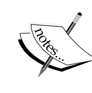
```
Chapter 2: Modifiability
(arch) $ python -m mccabe --min 1 power.py
1:1: 'power' 1
(arch) $
(arch) $ python -m mccabe --min 1 factorial.py
1:1: 'factorial' 2
(arch) $
```

McCabe metrics for some sample Python programs

The `–min` argument tells the `mccabe` module to start measuring and reporting from the given McCabe index.

Testing for metrics

Let's now try a few of the aforementioned tools and use them on an example module to find out what kind of information these tools report.



The purpose of the following sections is not to teach you the usage of these tools or their command-line options—these can be picked up via the tool's documentation. Instead, the purpose is to explore the depth and richness of information that these tools provide with respect to the style, logic, and other issues with the code.

For purposes of this testing, the following contrived module example has been used. It is written purposefully with a lot of coding errors, style errors, and coding smells.

Since the tools we are using lists errors by line numbers, the code has been presented with numbered lines so that it is easy to follow the output of the tools back to the code:

```
 1  """
 2  Module metrictest.py
 3
 4  Metric example - Module which is used as a testbed for static
    checkers.
 5  This is a mix of different functions and classes doing
    different things.
 6
 7  """
 8  import random
 9
10  def fn(x, y):
11      """ A function which performs a sum """
12      return x + y
13
14  def find_optimal_route_to_my_office_from_home(start_time,
15                                      expected_time,
16                                      favorite_route='SBS1K',
17                                      favorite_option='bus'):
18
19
20      d = (expected_time - start_time).total_seconds()/60.0
21
22      if d<=30:
23          return 'car'
24
25      # If d>30 but <45, first drive then take metro
26      if d>30 and d<45:
27          return ('car', 'metro')
28
29      # If d>45 there are a combination of options
```

```
30      if d>45:
31          if d<60:
32              # First volvo,then connecting bus
33              return ('bus:335E','bus:connector')
34          elif d>80:
35              # Might as well go by normal bus
36              return random.choice(('bus:330','bus:331',':'.
                       join((favorite_option,
37                             favorite_route))))
38          elif d>90:
39              # Relax and choose favorite route
40              return ':'.join((favorite_option,
41                                 favorite_route))
42
43
44  class C(object):
45      """ A class which does almost nothing """
46
47      def __init__(self, x,y):
48          self.x = x
49          self.y = y
50
51      def f(self):
52          pass
53
54      def g(self, x, y):
55
56          if self.x>x:
57              return self.x+self.y
58          elif x>self.x:
59              return x+ self.y
60
61  class D(C):
62      """ D class """
63
64      def __init__(self, x):
65          self.x = x
66
67      def f(self, x,y):
68          if x>y:
69              return x-y
70          else:
```

```
71                  return x+y
72
73          def g(self, y):
74
75              if self.x>y:
76                  return self.x+y
77              else:
78                  return y-self.x
```

Running static checkers

Let's see what Pylint has to say about our rather horrible-looking piece of test code.

$ pylint –reports=n metrictest.py

Pylint prints a lot of styling errors, but the purpose of this example being to focus on logic issues and code smells, the log is shown only starting from these reports.

Here is the detailed output captured in two screenshots:

```
Chapter 2: Modifiability
(arch) $ pylint --reports=n metrictest.py
************* Module metrictest
C: 22, 0: Exactly one space required around comparison
        if d<=30:
            ^^ (bad-whitespace)
C: 24, 0: Exactly one space required around comparison
        elif d<45:
              ^ (bad-whitespace)
C: 26, 0: Exactly one space required around comparison
        elif d<60:
              ^ (bad-whitespace)
C: 28, 0: Exactly one space required after comma
            return ('bus:335E','bus:connector')
                              ^ (bad-whitespace)
C: 29, 0: Exactly one space required around comparison
        elif d>80:
              ^ (bad-whitespace)
C: 31, 0: Exactly one space required after comma
            return random.choice(('bus:330','bus:331',':'.join((favorite_option,
                                           ^ (bad-whitespace)
C: 31, 0: Exactly one space required after comma
            return random.choice(('bus:330','bus:331',':'.join((favorite_option,
                                                     ^ (bad-whitespace)
C: 32, 0: Wrong continued indentation (remove 4 spaces).
                                                                    favorite_route))))
```

Pylint output for metric test program (page 1)

Take a look at the screenshot of the next page of the report:

```
Chapter 2: Modifiability
File  Edit  View  Search  Terminal  Help
C: 11, 0: Invalid function name "fn" (invalid-name)
C: 11, 0: Invalid argument name "x" (invalid-name)
C: 11, 0: Invalid argument name "y" (invalid-name)
W: 15, 4: Unreachable code (unreachable)
C: 15, 4: Invalid function name "find_optimal_route_to_my_office_from_home" (invalid-name)
C: 15, 4: Missing function docstring (missing-docstring)
C: 21, 8: Invalid variable name "d" (invalid-name)
E: 32,19: Undefined variable 'random' (undefined-variable)
W: 15, 4: Unused variable 'find_optimal_route_to_my_office_from_home' (unused-variable)
C: 39, 0: Invalid class name "C" (invalid-name)
C: 43, 8: Invalid attribute name "x" (invalid-name)
C: 44, 8: Invalid attribute name "y" (invalid-name)
C: 46, 4: Invalid method name "f" (invalid-name)
C: 46, 4: Missing method docstring (missing-docstring)
C: 49, 4: Invalid method name "g" (invalid-name)
C: 49, 4: Invalid argument name "x" (invalid-name)
C: 49, 4: Invalid argument name "y" (invalid-name)
C: 49, 4: Missing method docstring (missing-docstring)
W: 49,19: Unused argument 'y' (unused-argument)
C: 56, 0: Invalid class name "D" (invalid-name)
W: 59, 4: __init__ method from base class 'C' is not called (super-init-not-called)
W: 62, 4: Arguments number differs from overridden 'f' method (arguments-differ)
W: 68, 4: Arguments number differs from overridden 'g' method (arguments-differ)
C: 75, 0: Invalid argument name "a" (invalid-name)
C: 75, 0: Invalid argument name "b" (invalid-name)
C: 75, 0: Missing function docstring (missing-docstring)
E: 77,15: Undefined variable 'c' (undefined-variable)
W:  9, 0: Unused import sys (unused-import)
```

Pylint output for metric test program (page 2)

Let's focus on those very interesting last 10-20 lines of the Pylint report, skipping the earlier styling and convention warnings.

Here are the errors, classified into a table. We have skipped similar occurrences to keep the table short:

Error	**Occurrences**	**Explanation**	**Type of Code Smell**
Invalid function name	The `fn` function	The name `fn` is too short to explain what the function does	Too short identifier
Invalid variable name	The `x` and `y` variables of the `fn` function, `f`	The names `x` and `y` too short to indicate what the variables represent	Too short identifier

Error	Occurrences	Explanation	Type of Code Smell
Invalid function name	Function name, `find_optimal_route_to_my_office_from_home`	The function name is too long	Too long identifier
Invalid variable name	The `d` variable of function, `find_optimal`...	The name `d` too short to indicate what the variable represents	Too short identifier
Invalid class name	Class `C`	The name `C` doesn't tell anything about the class	Too short identifier
Invalid method name	Class `C`: Method `f`	The name `f` too short to explain what it does	Too short identifier
Invalid `__init__` method	Class `D`: Method `__init__`	Doesn't call base class `__init__`	Breaks contract with base Class
Arguments of f differ in class `D` from class `C`	Class `D`: Method `f`	Method signature breaks contract with base class signature	Refused bequest
Arguments of `g` differ in class `D` from class `C`	Class `D`: Method `g`	Method signature breaks contract with base class signature	Refused bequest

As you can see, Pylint has detected a number of code smells, which we discussed in the previous section. Some of the most interesting ones are how it detected the absurdly long function name and how the subclass `D` breaks the contract with the base class, `C`, in its `__init__` method and other methods.

Let's see what `flake8` has to tell us about our code. We will run it in order to report the statistics and summary of error counts:

```
$  flake8 --statistics --count metrictest.py
```

The preceding command gives the following output:

```
Chapter 2: Modifiability
File Edit View Search Terminal Help
$ flake8 --statistics --count metrictest.py
metrictest.py:8:1: F401 'sys' imported but unused
metrictest.py:10:1: E302 expected 2 blank lines, found 1
metrictest.py:22:13: E225 missing whitespace around operator
metrictest.py:24:15: E225 missing whitespace around operator
metrictest.py:26:15: E225 missing whitespace around operator
metrictest.py:28:31: E231 missing whitespace after ','
metrictest.py:29:15: E225 missing whitespace around operator
metrictest.py:31:20: F821 undefined name 'random'
metrictest.py:31:44: E231 missing whitespace after ','
metrictest.py:31:54: E231 missing whitespace after ','
metrictest.py:32:69: E127 continuation line over-indented for visual indent
metrictest.py:37:1: W293 blank line contains whitespace
metrictest.py:41:25: E231 missing whitespace after ','
metrictest.py:44:1: W293 blank line contains whitespace
metrictest.py:50:18: E225 missing whitespace around operator
metrictest.py:52:15: E225 missing whitespace around operator
metrictest.py:53:21: E225 missing whitespace around operator
metrictest.py:55:1: E302 expected 2 blank lines, found 1
metrictest.py:61:18: E231 missing whitespace after ','
metrictest.py:62:13: E225 missing whitespace around operator
metrictest.py:69:18: E225 missing whitespace around operator
metrictest.py:74:1: E302 expected 2 blank lines, found 1
metrictest.py:75:9: E225 missing whitespace around operator
metrictest.py:76:16: F821 undefined name 'c'
1       E127 continuation line over-indented for visual indent
10      E225 missing whitespace around operator
5       E231 missing whitespace after ','
3       E302 expected 2 blank lines, found 1
1       F401 'sys' imported but unused
2       F821 undefined name 'c'
2       W293 blank line contains whitespace
24
```

Flake8 static check output of the metrictest program

As you would've expected from a tool that is written to mostly follow PEP-8 conventions, the errors reported are all styling and convention errors. These errors are useful to improve the readability of the code and make it follow closer to the style guidelines of PEP-8.

You can get more information about the PEP-8 tests by passing the `-show-pep8` option to Flake8.

It is a good time to now check the complexity of our code. First, we will use `mccabe` directly and then call it via Flake8:

```
Chapter 2: Modifiability
(arch) $ python -m mccabe --min 3 metrictest.py
54:1: 'C.g' 3
14:1: 'find_optimal_route_to_my_office_from_home' 7
(arch) $
```

mccabe complexity of metric test program

As expected, the complexity of the office-route function is too high, as it has too many branches and sub-branches.

As `flake8` prints too many styling errors, we will grep specifically for the report on complexity:

```
Chapter 2: Modifiability
(arch) $ flake8 --max-complexity 3 metrictest.py | grep complex
metrictest.py:14:1: C901 'find_optimal_route_to_my_office_from_home' is too compl
ex (7)
(arch) $
```

mccabe complexity of metric test program as reported by flake8

As expected, Flake8 reports the `find_optimal_route_to_my_office_from_home` function as too complex.

There is a way to run `mccabe` as a plugin from Pylint as well, but since it involves some configuration steps, we will not cover it here.

As a last step, let's run `pyflakes` on the code:

```
(arch) $ pyflakes metrictest.py
(arch) $
```

Static analysis output of pyflakes on the metric test code

There is no output! So, Pyflakes finds no issues with the code. The reason is that Pyflakes is a basic checker that does not report anything beyond the obvious syntax and logic errors, unused imports, missing variable names, and the like.

Let's add some errors into our code and rerun Pyflakes. Here is the adjusted code with line numbers:

```
 1  """
 2  Module metrictest.py
 3
 4  Metric example - Module which is used as a testbed for static
    checkers.
 5  This is a mix of different functions and classes doing
    different things.
 6
 7  """
 8  import sys
 9
10  def fn(x, y):
11      """ A function which performs a sum """
12      return x + y
13
14  def find_optimal_route_to_my_office_from_home(start_time,
15                                  expected_time,
16                                  favorite_route='SBS1K',
17                                  favorite_option='bus'):
18
19
20      d = (expected_time - start_time).total_seconds()/60.0
21
22      if d<=30:
```

```
23          return 'car'
24
25      # If d>30 but <45, first drive then take metro
26      if d>30 and d<45:
27          return ('car', 'metro')
28
29      # If d>45 there are a combination of options
30      if d>45:
31          if d<60:
32              # First volvo,then connecting bus
33              return ('bus:335E','bus:connector')
34          elif d>80:
35              # Might as well go by normal bus
36              return random.choice(('bus:330','bus:331',':'.
                        join((favorite_option,
37                              favorite_route))))
38          elif d>90:
39              # Relax and choose favorite route
40              return ':'.join((favorite_option,
41                                  favorite_route))
42
43
44  class C(object):
45      """ A class which does almost nothing """
46
47      def __init__(self, x,y):
48          self.x = x
49          self.y = y
50
51      def f(self):
52          pass
53
54      def g(self, x, y):
55
56          if self.x>x:
57              return self.x+self.y
58          elif x>self.x:
59              return x+ self.y
60
61  class D(C):
62      """ D class """
63
64      def __init__(self, x):
```

```
65              self.x = x
66
67          def f(self, x,y):
68              if x>y:
69                  return x-y
70              else:
71                  return x+y
72
73          def g(self, y):
74
75              if self.x>y:
76                  return self.x+y
77              else:
78                  return y-self.x
79
80      def myfunc(a, b):
81          if a>b:
82              return c
83          else:
84              return a
```

Take a look at the following output:

```
Chapter 2: Modifiability
(arch) $ pyflakes metrictest.py
metrictest.py:8: 'sys' imported but unused
metrictest.py:36: undefined name 'random'
metrictest.py:82: undefined name 'c'
(arch) $
```

Static analysis output of pyflakes on the metric test code, after modifications

Pyflakes now returns some useful information in terms of a missing name (`random`), unused import (`sys`), and an undefined name (the `c` variable in the newly introduced function, `myfunc`). So it does perform some useful static analysis on the code. For example, the information on the missing and undefined names is useful to fix obvious bugs in the preceding code.

It is a good idea to run Pylint and/or Pyflakes on your code to report and figure out logic and syntax errors after the code is written. To run Pylint to report only errors, use the `-E` option. To run Pyflakes, just follow the preceding example.

Refactoring code

Now that we have seen how static checkers can be used to report a wide range of errors and issues in our Python code, let's do a simple exercise of refactoring our code. We will take our poorly written metric test module as the use case (the first version of it) and perform a few refactoring steps.

Here are the rough guidelines to follow when refactoring software:

1. **Fix complex code first**: This will get a lot of code out of the way as typically, when a complex piece of code is refactored, we end up reducing the number of lines of code. This overall improves the code quality and reduces code smells. You may be creating new functions or classes here, so it always helps to perform this step first.
2. **Do an analysis of the code**: It is a good idea to run the complexity checkers at this step and see how the overall complexity of the code—class/module or functions—has been reduced. If not, iterate again.
3. **Fix code smells next**: Fix any issue with code smells—class, function, or module—next. This gets your code into a much better shape and improves the overall semantics.
4. **Run checkers**: Run checkers such as Pylint on the code now, and get a report on the code smells. Ideally, they should be close to zero or reduced very much from the original.
5. **Fix low-hanging fruits**: Fix low-hanging fruits, such as code style and convention errors, last. This is because, in the process of refactoring, when trying to reduce complexity and code smells, you typically would introduce or delete a lot of code. So, it doesn't make sense to try and improve the code convention issues at earlier stages.
6. **Perform a final check using the tools**: You can run Pylint for code smells, Flake8 for PEP-8 conventions, and Pyflakes for catching the logic, syntax, and missing variable issues.

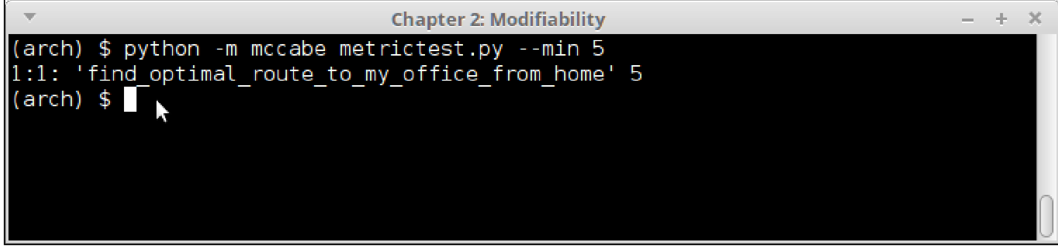
Here is a step-by-step demonstration of fixing our metric test module using this approach in the next section.

Refactoring code – fixing complexity

Most of the complexity is in the office route function, so let's try and fix it. Here is the rewritten version (showing only that function here):

```
def find_optimal_route_to_my_office_from_home(start_time,
                                              expected_time,
                                              favorite_route='SBS1K',
                                              favorite_option='bus'):

        d = (expected_time - start_time).total_seconds()/60.0

        if d<=30:
            return 'car'
        elif d<45:
            return ('car', 'metro')
        elif d<60:
            # First volvo,then connecting bus
            return ('bus:335E','bus:connector')
        elif d>80:
            # Might as well go by normal bus
            return random.choice(('bus:330','bus:331',':'.
                                  join((favorite_option,
                                        favorite_route))))
        # Relax and choose favorite route
        return ':'.join((favorite_option, favorite_route))
```

In the preceding rewrite, we got rid of the redundant `if...else` conditions. Let's check the complexity now:

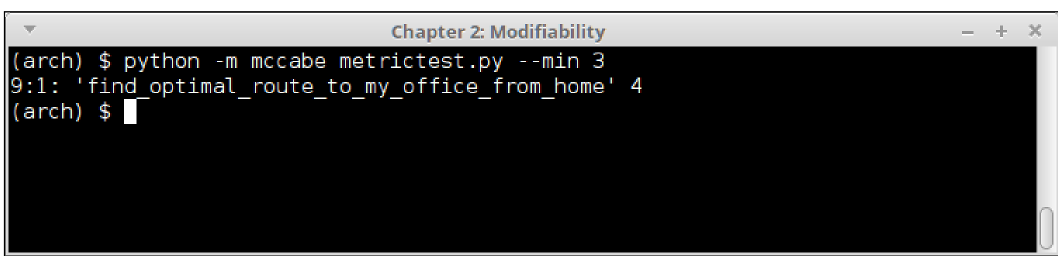


mccabe metric of metric test program after refactoring step 1

We were able to reduce the complexity from `7` to `5`. Can we do better?

In the following piece of code, the code is rewritten to use ranges of values as keys, and the corresponding return value as values. This simplifies our code a lot. Also, the earlier default return at the end would never have got picked, so it is removed now, hence getting rid of a branch and reducing complexity by one. The code has become much simpler:

```
def find_optimal_route_to_my_office_from_home(start_time,
                                              expected_time,
                                              favorite_route='SBS1K',
                                              favorite_option='bus'):

    # If I am very late, always drive.
    d = (expected_time - start_time).total_seconds()/60.0
    options = { range(0,30): 'car',
    range(30, 45): ('car','metro'),
    range(45, 60): ('bus:335E','bus:connector') }


if d<80:
# Pick the range it falls into
for drange in options:
    if d in drange:
    return drange[d]

    # Might as well go by normal bus
    return random.choice(('bus:330','bus:331',':'.join((favorite_
                           option, favorite_route))))
```



mccabe metric of metric test program after refactoring step #2

The complexity of the function is now reduced to `4`, which is manageable.

Refactoring code – fixing code smells

The next step is to fix code smells. Thankfully, we have a very good list from the previous analysis, so this is not too difficult. Mostly, we need to change function names and variable names and fix the contracts from child class to parent class.

Here is the code with all of the fixes:

```
""" Module metrictest.py - testing static quality metrics of Python
code """

import random

def sum_fn(xnum, ynum):
    """ A function which performs a sum """

    return xnum + ynum

def find_optimal_route(start_time,
                       expected_time,
                       favorite_route='SBS1K',
                       favorite_option='bus'):
    """ Find optimal route for me to go from home to office """

    # Time difference in minutes - inputs must be datetime instances
    tdiff = (expected_time - start_time).total_seconds()/60.0

    options = {range(0, 30): 'car',
               range(30, 45): ('car', 'metro'),
               range(45, 60): ('bus:335E', 'bus:connector')}


    if tdiff < 80:
        # Pick the range it falls into
        for drange in options:
            if tdiff in drange:
                return drange[tdiff]

    # Might as well go by normal bus
    return random.choice(('bus:330', 'bus:331',
                          ':'.join((favorite_option,
                                    favorite_route))))


class MiscClassC(object):
```

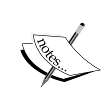
```
    """ A miscellaneous class with some utility methods """

    def __init__(self, xnum, ynum):
        self.xnum = xnum
        self.ynum = ynum

    def compare_and_sum(self, xnum=0, ynum=0):
        """ Compare local and argument variables
        and perform some sums """

        if self.xnum > xnum:
            return self.xnum + self.ynum
        else:
            return xnum + self.ynum

class MiscClassD(MiscClassC):
    """ Sub-class of MiscClassC overriding some methods """

    def __init__(self, xnum, ynum=0):
        super(MiscClassD, self).__init__(xnum, ynum)

    def some_func(self, xnum, ynum):
        """ A function which does summing """

        if xnum > ynum:
            return xnum - ynum
        else:
            return xnum + ynum

    def compare_and_sum(self, xnum=0, ynum=0):
        """ Compare local and argument variables
        and perform some sums """

        if self.xnum > ynum:
            return self.xnum + ynum
        else:
            return ynum - self.xnum
```

Let's run Pylint on this code and see what it outputs this time:

```
Chapter 2: Modifiability
(arch) $ pylint --reports=n metrictest.py
************* Module metrictest
W: 42,38: Unused argument 'ynum' (unused-argument)
R: 35, 0: Too few public methods (1/2) (too-few-public-methods)
R: 57, 4: Method could be a function (no-self-use)
This option 'required-attributes' will be removed in Pylint 2.0This option 'ignor
e-iface-methods' will be removed in Pylint 2.0(arch) $
```

Pylint output of refactored metric test program

You see that the number of code smells has boiled down to near zero except a complaint of lack of `public` methods, and the insight that the `some_func` method of the `MiscClassD` class can be a function, as it does not use any attributes of the class.



We have invoked Pylint with the `–reports=n` option in order to avoid Pylint printing its summary report, as it would make the entire output too long to display here. These reports can be enabled by calling Pylint without any argument.

Refactoring code – fixing styling and coding issues

Now that we have fixed the major code issues, the next step is to fix code style and convention errors. However, in order to shorten the number of steps and the amount of code to be printed in this book for this exercise, this was already merged along with the last step, as you may have guessed from the output of Pylint.

Except for a few whitespace warnings, all of the issues are fixed.

This completes our Refactoring exercise.

Summary

In this chapter, we looked at the architectural quality attribute of modifiability and its various aspects. We discussed readability in some detail, including the readability antipatterns along with a few coding antipatterns.

We looked at various techniques for improving readability of code and understood the different aspects of commenting of code such as function, class and module docstrings. We also looked at PEP-8, the official coding convention guideline for Python.

We then looked at some rules of thumb for code comments and went on to discuss the fundamentals of modifiability, namely, coupling and cohesion of code. We looked at different cases of coupling and cohesion with a few examples. We then went on to discuss the strategies of improving modifiability of code such as providing explicit interfaces or APIs, avoiding two-way dependencies, abstracting common services to helper modules, and using inheritance techniques. We looked at an example where we refactored a class hierarchy via inheritance to abstract away common code and to improve the modifiability of the system.

Toward the end, we listed the different tools, providing static code metrics in Python such as Pylint, Flake8, Pyflakes, and others. We learned about McCabe Cyclomatic complexity with the help of a few examples. We also learned what code smells are and performed a refactoring exercise to improve the quality of the piece of code in stages.

In the next chapter, we'll discuss another important quality attribute of software architecture, namely, testability.

3
Testability – Writing Testable Code

In the previous chapter, we covered a very important architectural attribute of software, namely, modifiability, and its related aspects. In this chapter, the topic is a closely related quality attribute: **testability** of software.

We briefly covered testability in the first chapter of this book, where we understood what testability is, and how it relates to the complexity of the code. In this chapter, we will look into the different aspects of software testability in detail.

Software testing by itself has developed into a large field with its own standards and unique set of tools and processes. The focus of this chapter is not to cover the formal aspects of software testing. Instead, what we will strive to do here is to understand software testing from an architectural perspective and understand its relation to the other quality attributes and, in the second half of this chapter, discuss the Python tools and libraries relevant to our discussion on software testing using Python.

We will cover the following topics in this chapter:

- Understanding testability
- White-box testing principles
- Test-driven development
- TDD with palindromes

Understanding testability

Testability can be defined as follows:

> "*The degree of ease with which a software system exposes its faults through execution-based testing*".

A software system with a high level of testability provides a high degree of exposure of its faults through testing, thereby giving the developers higher accessibility to the system's issues and allowing them to find and fix bugs faster. A less testable system, on the other hand, would make it difficult for developers to figure out issues with it and can often lead to unexpected failures in production.

Testability is, hence, an important aspect in ensuring the quality, stability, and predictability of the software system in production.

Software testability and related attributes

A software system is testable if it gives up (exposes) its faults easily to the tester. Not only that, the system should behave in a predictable way for the tester to develop useful tests. An unpredictable system would give varying output variables to fixed input at varying times, hence, is not testable (or very useful for that matter!).

More than unpredictability, complex or chaotic systems are also less amenable to testing. For example, a system whose behavior varies wildly across a spectrum under load doesn't make a good candidate for load testing. Hence, deterministic behavior is also important to assure the testability of a system.

Another aspect is the amount of control that the tester has on the substructures of the system. In order to design meaningful tests, a system should be easily identifiable to subsystems with their well-defined APIs, for which tests can be written. A software system that is complex and doesn't provide easy access to its subsystems, by definition, becomes much less testable than the one which does.

This means that systems that are more structurally complex are more difficult to test than ones that aren't.

Let's list this in an easy-to-read table:

Determinism	**Complexity**	**Testability**
High	Low	High
Low	High	Low

Testability – architectural aspects

Software testing generally implies that the software artifact being tested is being assessed for its functionality. However, in practical software testing, functionality is just one of the aspects that can fail. Testing implies assessing the software for other quality attributes such as performance, security, and robustness.

Due to these different aspects of testing, software testability is usually grouped at different levels. We will take a look at these from the point of view of software architecture.

Here is a brief listing of the different aspects that usually fall under software testing:

- **Functional testing**: This involves testing the software for verifying its functionality. A unit of software passes its functional test if it behaves exactly the way it is supposed to as per its development specifications. Functional testing is usually of two types:
    - **White-box testing**: These are usually tests implemented by the developers, who have visibility into the software code, themselves. The units being tested here are the individual functions, methods, classes, or modules that make up the software rather than the end user functionality. The most basic form of white-box testing is **unit testing**. Other types are **integration testing** and **system testing**.
    - **Black-box testing**: This type of testing is usually performed by someone who is outside the development team. The tests have no visibility into the software code, and treat the entire system like a black box. Black-box testing tests the end user functionality of the system without bothering about its internal details. Such tests are usually performed by dedicated testing or QA engineers. However, nowadays, a lot of black-box tests on web-based applications can be automated by using testing frameworks such as Selenium.

      Other than functional testing, there are a lot of testing methodologies that are used to assess the various architectural quality attributes of a system. We will discuss these next.

- **Performance testing**: Tests that measure how a software performs with respect to its responsiveness and robustness (stability) under high workloads come within this category. Performance tests are usually categorized into the following:
    - **Load testing**: These are tests that assess how a system performs under a certain specific load, either in terms of the number of concurrent users, input data, or transactions.
    - **Stress testing**: This tests the robustness and response of the system when some inputs present a sudden or high rate of growth and go to extreme limits. Stress tests typically tend to test the system slightly beyond its prescribed design limits. A variation of stress testing is running the system under a certain specified load for extended periods of time and measuring its responsiveness and stability.
    - **Scalability testing**: Measure how much the system can scale out or scale up when the load is increased. For example, if a system is configured to use a cloud service, this can test the horizontal scalability—as in how the system auto scales to a certain number of nodes upon increased load or vertical scalability—in terms of the degree of utilization of CPU cores and/or RAM of the system.
- **Security testing**: Tests that verify the system's security fall into this category. For web-based applications, this usually involves verifying authorization of roles by checking that a given login or role can only perform a specified set of actions and nothing more (or less). Other tests that fall under security would be to verify proper access to data or static files to make sure that all sensitive data of an application is protected by proper authorization via logins.
- **Usability testing**: Usability testing involves testing how much the user interface of a system is easy to use, is intuitive, and understandable by its end users. Usability testing is usually done via target groups comprising selected people who fall into the definition of the intended audience or end users of the system.
- **Installation testing**: For software that is shipped to the customer's location and is installed there, installation testing is important. This tests and verifies that all of the steps involved in building and/or installing the software at the customer's end work as expected. If the development hardware differs from the customer's, then the testing also involves verifying the steps and components in the end user's hardware. Apart from a regular software installation, installation testing is also important when delivering software updates, partial upgrades, and so on.

- **Accessibility testing**: Accessibility, from a software standpoint, refers to the degree of usability and inclusion of a software system towards end users with disabilities. This is usually done by incorporating support for accessibility tools in the system, and designing the user interface by using accessible design principles. A number of standards and guidelines have been developed over the years, which allow organizations to develop software with a view to making the software accessible to such an audience. Examples are the **Web Content Accessibility Guidelines** (**WCAG**) of W3C, Section 508 of the Government of USA, and the like.

  Accessibility testing aims to assess the accessibility of software with respect to these standards, wherever applicable.

There are various other types of software testing, which involves different approaches, and are invoked at various phases of software development, such as regression testing, acceptance testing, alpha or beta testing, and so on.

However, since our focus of discussion is on the architectural aspects of software testing, we will limit our attention to the topics mentioned in the previous list.

Testability – strategies

We saw in a previous section how testability varies according to the complexity and determinism of the software system under testing.

Being able to isolate and control the artifacts that are being tested is critical to software testing. Separation of concerns on the system being tested, as in being able to test components independently and without too much external dependency, is key to this.

Let's look at the strategies that the software architect can employ in order to make sure that the components he/she is subjecting to tests provide predictable and deterministic behavior, which will provide valid and useful test results.

Reduce system complexity

As mentioned earlier, a complex system has lower testability. The system complexity can be reduced by techniques such as splitting systems into subsystems, providing well-defined APIs for systems to be tested, and so on. Here is a list of these techniques in some detail:

- **Reducing coupling**: This is to isolate components so that coupling is reduced in the system. Inter-component dependencies should be well defined, and if possible, documented.

- **Increasing cohesion**: This is to increase cohesion of modules, that is, to make sure that a particular module or class performs only a well-defined set of functions.
- **Providing well-defined interfaces**: Try to provide well-defined interfaces for getting/setting the state of the components and classes involved. For example, getters and setters allow us to provide specific methods for getting and setting the value of a class's attributes. A reset method allows to set the internal state of an object to its state at the time of creation. In Python, this can be done by defining properties.
- **Reducing class complexity**: This means to reduce the number of classes a class derives from. A metric called **Response For Class** (**RFC**) is a set of methods of a class C, plus the methods on other classes called by the methods of class C. It is suggested to keep the RFC of a class in manageable limits, usually not more than 50 for small- to medium-sized systems.

Improving predictability

We saw that having a deterministic behavior is very important to design tests that provide predictable results, and hence, can be used to build a test harness for repeatable testing. Here are some strategies to improve the predictability of the code under test:

- **Correct exception handling**: Missing or improperly-written exception handlers is one of the main reasons for bugs and thence, unpredictable behavior in software systems. It is important to find out places in the code where exceptions can occur and then handle errors. Most of the time, exceptions occur when a code interacts with an external resource such as performing a database query, fetching a URL, waiting on a shared mutex, and the like.
- **Infinite loops and/or blocked wait**: When writing loops that depend on specific conditions such as availability of an external resource, or getting an handle to or data from a shared resource, say a shared mutex or queue, it is important to make sure that there are always safe exit or break conditions provided in the code. Otherwise, the code can get stuck in infinite loops that never break or on never-ending blocked waits on resources causing bugs that are hard to troubleshoot and fix.
- **Logic that is time dependent**: When implementing logic that is dependent on certain times of the day (hours or specific weekdays), make sure that the code works in a predictable fashion. When testing such code, we often need to isolate such dependencies by using mocks or stubs.

- **Concurrency**: When writing code that uses concurrent methods such as multiple threads and/or processes, it is important to make sure that the system logic is not dependent on threads or processes starting in any specific order. The system state should be initialized in a clean and repeatable way via well-defined functions or methods that allow the system behavior to be repeatable, and hence, testable.
- **Memory management**: A very common reason for software errors and unpredictability is incorrect usage and mismanagement of memory. In modern runtimes with dynamic memory management, such as Python, Java, or Ruby, this is less of a problem. However, memory leaks and unreleased memory leading to bloated software are still very much a reality in modern software systems.

It is important to analyze and be able to predict the maximum memory usage of your software system so that you allocate enough memory for it and run it on the right hardware. Also, software should be periodically evaluated and tested for memory leaks and better memory management, and any major issues should be addressed and fixed.

Control and isolate external dependencies

Tests usually have some sort of external dependency. For example, a test may need to load/save data to/from a database. Another may depend on the test running on specific times of the day. A third may require fetching data from a URL on the web.

However, having external dependencies usually complicates a test scenario. This is because external dependencies are usually not within the control of the test designer. In the aforementioned cases, the database may be in another data center, the connection may fail, or the website may not respond within the configured time or may give a 50X error.

Isolating such external dependencies is very important in designing and writing repeatable tests. The following are a few techniques for the same:

- **Data sources**: Most realistic tests require data of some form. More often than not, data is read from a database. However, a database being an external dependency cannot be relied upon. The following are a few techniques to control data source dependencies:
    - Using local files instead of a database: Quite often, test files with prefilled data can be used instead of querying a database. Such files could be text, JSON, CSV, or YAML files. Usually, such files are used with mock or stub objects.

    - Using an in-memory database: Rather than connecting to a real database, a small in-memory database could be used. A good example is the SQLite DB, a file or memory-based database which implements a good, but minimal, subset of SQL.
    - Using a test database: If the test really requires a database, the operation can use a test database that uses **transactions**. The database is set up in the `setUp()` method of the test case, and rolled back in the `tearDown()` method so that no real data remains at the end of the operation.

- **Resource virtualization**: In order to control the behavior of resources that are outside the system, we can virtualize them, that is, build a version of these resources that mimic their APIs, but not the internal implementation. Some common techniques for resource virtualization are as follows:
    - **Stubs**: Stubs provide standard (canned) responses to function calls made during a test. A `Stub()` function replaces the details of the function it replaces, only returning the response as required.

      For example, here is a function that returns `data` for a given URL:

```
import hashlib
import requests

def get_url_data(url):
    """ Return data for a URL """

    # Return data while saving the data in a file
    # which is a hash of the URL
    data = requests.get(url).content
    # Save it in a filename
    filename = hashlib.md5(url).hexdigest()
    open(filename, 'w').write(data)
    return data
```

And the following is the stub that replaces it, which internalizes the external dependency of the URL:

```
import os

def get_url_data_stub(url):
    """ Stub function replacing get_url_data """

    # No actual web request is made, instead
    # the file is opened and data returned
```

```
    filename = hashlib.md5(url).hexdigest()
    if os.path.isfile(filename):
        return open(filename).read()
```

A more common way to write such a function is to combine both the original request and the file cache in the same code. The URL is requested just once—the first time the function is called—and in subsequent requests, the data from the file cache is returned:

```
def get_url_data(url):
    """ Return data for a URL """

    # First check for cached file - if so return its
    # contents. Note that we are not checking for
    # age of the file - so content may be stale.
    filename = hashlib.md5(url).hexdigest()
    if os.path.isfile(filename):
        return open(filename).read()

    # First time - so fetch the URL and write to the
    # file. In subsequent calls, the file contents will
    # be returned.
    data = requests.get(url).content
    open(filename, 'w').write(data)

    return data
```

- **Mocks**: Mocks fake the API of the real-world objects they replace. We program mock objects directly in the test by setting expectations—in terms of the type and order of the arguments the functions will expect and the responses they will return. Later, the expectations can be optionally verified in a verification step.

We will see examples of writing unit test via mocks with Python later.

The main difference between mocks and stubs is that a stub implements just enough behavior for the object under test to execute the test. A mock usually goes beyond by also verifying that the object under test calls the mock as expected—for example, in terms of number and order of arguments.

When using a mock object, part of the test involves verifying that the mock was used correctly. In other words, both mocks and stubs answer the question, *What is the result?*, but mocks also answer the question, *How has the result been achieved?*

- **Fakes**: `Fake` objects have working implementations, but fall short of production usage because they have some limitations. A `Fake` object provides a very lightweight implementation, which goes beyond just stubbing the object.

  For example, here is a `Fake` object that implements a very minimal logging, mimicking the API of the `Logger` object of the Python's logging module:

```
import logging

class FakeLogger(object):
    """ A class that fakes the interface of the
    logging.Logger object in a minimalistic fashion """

    def __init__(self):
        self.lvl = logging.INFO

    def setLevel(self, level):
        """ Set the logging level """
        self.lvl = level

    def _log(self, msg, *args):
        """ Perform the actual logging """

        # Since this is a fake object - no actual logging is
        # done.
        # Instead the message is simply printed to standard
        # output.

        print (msg, end=' ')
        for arg in args:
            print(arg, end=' ')
        print()

    def info(self, msg, *args):
        """ Log at info level """
        if self.lvl<=logging.INFO:
            return self._log(msg, *args)

    def debug(self, msg, *args):
        """ Log at debug level """
```

```
        if self.lvl<=logging.DEBUG:
            return self._log(msg, *args)

    def warning(self, msg, *args):
        """ Log at warning level """
        if self.lvl<=logging.WARNING:
            return self._log(msg, *args)

    def error(self, msg, *args):
        """ Log at error level """
        if self.lvl<=logging.ERROR:
            return self._log(msg, *args)

    def critical(self, msg, *args):
        """ Log at critical level """
        if self.lvl<=logging.CRITICAL:
            return self._log(msg, *args)
```

The `FakeLogger` class in the preceding code implements some main methods of the `logging.Logger` class, which it is trying to fake.

It is ideal as a fake object for replacing the `Logger` object for implementing tests.

White-box testing principles

From a software architecture perspective, one of the most important steps of testing is at the time the software is developed. The behavior or functionality of a software, which is apparent only to its end users, is an artifact of the implementation details of the software.

Hence, it follows that a system that is tested early and tested often has a higher likelihood to produce a testable and robust system, which provides the required functionality to the end user in a satisfactory manner.

The best way, therefore, to start implementing testing principles is right from the source, that is, where the software is written, and by the developers. Since the source code is visible to the developer, this testing is often called white-box testing.

So, how do we make sure that we can follow the correct testing principles, and perform due diligence while the software is getting developed? Let's take a look at the different types of testing that are involved during the development stage before the software ends up in front of the customer.

Unit testing

Unit testing is the most fundamental type of testing performed by developers. A unit test applies the most basic unit of software code—typically, functions or class methods—by using executable assertions, which check the output of the unit being tested against an expected outcome.

In Python, support for unit testing is provided by the `unittest` module in the standard library.

The unit test module provides the following high-level objects:

- **Test cases**: The `unittest` module provides the `TestCase` class, which provides support for test cases. A new test case class can be set up by inheriting from this class and setting up the test methods. Each test method will implement unit tests by checking the response against an expected outcome.
- **Test fixtures**: Test fixtures represent any setup or preparation required for one or more tests followed by any cleanup actions. For example, this may involve creating temporary or in-memory databases, starting a server, creating a directory tree, and the like. In the `unittest` module, support for fixtures is provided by the `setUp()` and `tearDown()` methods of the `TestCase` class and the associated class and module methods of the `TestSuite` class.
- **Test suites**: A test suite is an aggregation of related test cases. A test suite can also contain other test suites. A test suite allows to group test cases that perform functionally similar tests on a software system, and whose results should be read or analyzed together. The `unittest` module provides support for test suites through the `TestSuite` class.
- **Test runners**: A test runner is an object that manages and runs the test cases, and provides the results to the tester. A test runner can use a text interface or a GUI.
- **Test results**: Test result classes manage the test result output shown to the tester. Test results summarize the number of successful, failed, and erred-out test cases. In the `unittest` module, this is implemented by the `TestResult` class with a concrete, default implementation of the `TextTestResult` class.

Other modules that provide support for Unit testing in Python are `nose` (`nose2`) and `py.test`. We will discuss each of these briefly in the following sections.

Unit testing in action

Let's take a specific unit-testing task and then try to build a few test cases and test suites. Since the `unittest` module is the most popular, and available by default in the Python standard library, we will start with it first.

For our test purposes, we will create a class that has a few methods, which are used for date/time conversions.

The following code shows our class:

```
""" Module datetime helper - Contains the class DateTimeHelper
providing some helpful methods for working with date and datetime
objects """

import datetime
class DateTimeHelper(object):
    """ A class which provides some convenient date/time
    conversion and utility methods """

    def today(self):
        """ Return today's datetime """
        return datetime.datetime.now()

    def date(self):
        """ Return today's date in the form of DD/MM/YYYY """
        return self.today().strftime("%d/%m/%Y")

    def weekday(self):
        """ Return the full week day for today """
        return self.today().strftime("%A")

    def us_to_indian(self, date):
        """ Convert a U.S style date i.e mm/dd/yy to Indian style
              dd/mm/yyyy """

        # Split it
        mm,dd,yy = date.split('/')
        yy = int(yy)
        # Check if year is >16, else add 2000 to it
        if yy<=16: yy += 2000
        # Create a date object from it
```

```
        date_obj = datetime.date(year=yy, month=int(mm), day=int(dd))
        # Retur it in correct format
        return date_obj.strftime("%d/%m/%Y")
```

Our `DateTimeHelper` class has a few methods, which are as follows:

- `date`: Returns the day's timestamp in the dd/mm/yyyy format
- `weekday`: Returns the day's weekday, for example, Sunday, Monday, and so on
- `us_to_indian`: Converts a US date format (mm/dd/yy(yy)) into the Indian format (dd/mm/yyyy)

Here is a `unittest TestCase` class, which implements a test for the last method:

```
""" Module test_datetimehelper -  Unit test module for testing
datetimehelper module """

import unittest
import datetimehelper

class DateTimeHelperTestCase(unittest.TestCase):
     """ Unit-test testcase class for DateTimeHelper class """

    def setUp(self):
        print("Setting up...")
        self.obj = datetimehelper.DateTimeHelper()

    def test_us_india_conversion(self):
        """ Test us=>india date format conversion """

        # Test a few dates
        d1 = '08/12/16'
        d2 = '07/11/2014'
        d3 = '04/29/00'
        self.assertEqual(self.obj.us_to_indian(d1), '12/08/2016')
        self.assertEqual(self.obj.us_to_indian(d2), '11/07/2014')
        self.assertEqual(self.obj.us_to_indian(d3), '29/04/2000')

if __name__ == "__main__":
    unittest.main()
```

Note that, in the main part of the test case code, we just invoke `unittest.main()`. This automatically figures out the test cases in the module, and executes them. The following screenshot shows the output of the test run:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ python3 test_datetimehelper.py
Setting up...
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$
```

Output of the unit-test case for the datetimehelper module—version #1

As we can see from the output, this simple test case passes.

Extending our unit test case

You may have noted that the first version of the unit test case for the `datetimehelper` module contained a test only for one method, namely, the method that converts the US date format in to the Indian one.

However, what about the other two methods? Shouldn't we write unit tests for them too?

The problem with the other two methods is that they get data from today's date. In other words, the output is dependent on the exact day that the code is run. Hence, it is not possible to write a specific test case for them by feeding in a date value and expecting the result to match an outcome as the code is time dependent. We need a way to control this external dependency.

This is where mocking comes to our rescue. Remember that we discussed mock objects as a way to control external dependencies. We can use the patching support of the `unittest.mock` library, and patch the method that returns today's date to return a date that we control. This way, we are able to test the methods that depend on it.

Here is the modified test case with support added for the two methods using this technique:

```
""" Module test_datetimehelper -  Unit test module for testing
datetimehelper module """

import unittest
import datetime
import datetimehelper
```

```
from unittest.mock import patch

class DateTimeHelperTestCase(unittest.TestCase):
    """ Unit-test testcase class for DateTimeHelper class """

    def setUp(self):
        self.obj = datetimehelper.DateTimeHelper()

    def test_date(self):
        """ Test date() method """

        # Put a specific date to test
        my_date = datetime.datetime(year=2016, month=8, day=16)

        # Patch the 'today' method with a specific return value
        with patch.object(self.obj, 'today', return_value=my_date):
            response = self.obj.date()
            self.assertEqual(response, '16/08/2016')

    def test_weekday(self):
        """ Test weekday() method """

        # Put a specific date to test
        my_date = datetime.datetime(year=2016, month=8, day=21)

        # Patch the 'today' method with a specific return value
        with patch.object(self.obj, 'today', return_value=my_date):
            response = self.obj.weekday()
            self.assertEqual(response, 'Sunday')

    def test_us_india_conversion(self):
        """ Test us=>india date format conversion """

        # Test a few dates
        d1 = '08/12/16'
        d2 = '07/11/2014'
        d3 = '04/29/00'
        self.assertEqual(self.obj.us_to_indian(d1), '12/08/2016')
        self.assertEqual(self.obj.us_to_indian(d2), '11/07/2014')
        self.assertEqual(self.obj.us_to_indian(d3), '29/04/2000')

if __name__ == "__main__":
    unittest.main()
```

As you can see, we have patched the `today` method to return a specific date in the two test methods. This allows us to control the method's output and, in turn, compare the result with a specific outcome.

Here is the new output of the test case:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ python3 test_datetimehelper.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$
```

Output of the unit-test case for datetimehelper module with two more tests—version #2

`unittest.main` is a convenience function on the `unittest` module, which makes it easy to load a set of test cases automatically from a module and run them.

To find out more details of what is happening when the tests are run, we can make the test runner show more information by increasing the verbosity. This can be done either by passing the `verbosity` argument to `unittest.main` or by passing the `-v` option on the command line as follows:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ python3 test_datetimehelper.py -v
test_date (__main__.DateTimeHelperTestCase)
Test date() method ... ok
test_us_india_conversion (__main__.DateTimeHelperTestCase)
Test us=>india date format conversion ... ok
test_weekday (__main__.DateTimeHelperTestCase)
Test weekday() method ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Producing verbose output from the unit-test case by passing the -v argument

Nosing around with nose2

There are other unit-testing modules in Python that are not part of the standard library, but are available as third-party packages. We will look at the first one named `nose`. The most recent version (at the time of writing) is version 2, and the library has been renamed as `nose2`.

The `nose2` package can be installed by using the Python package installer, `pip`:

```
$ pip install nose2
```

Running `nose2` is very simple. It automatically detects Python test cases to run in the folder that it is run from by looking for classes derived from `unittest.TestCase` and functions starting with `test`.

In the case of our `datetimehelper` test case, `nose2` picks it up automatically. Simply run it from the folder containing the module. Here is the test output:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Running unit tests using nose2

The preceding output doesn't, however, report anything, since, by default, `nose2` runs quietly. We can turn on some reporting of tests by using the verbose option (`-v`):

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2 -v
test_date (test_datetimehelper.DateTimeHelperTestCase)
Test date() method ... ok
test_us_india_conversion (test_datetimehelper.DateTimeHelperTestCase)
Test us=>india date format conversion ... ok
test_weekday (test_datetimehelper.DateTimeHelperTestCase)
Test weekday() method ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Running unit-tests using nose2 with verbose output

`nose2` also supports reporting code coverage by using plugins. We will look at code coverage in a later section.

Testing with py.test

The `py.test` package, commonly known as `pytest`, is a full-featured, mature testing framework for Python. Like `nose2`, `py.test` also supports test discovery by looking for files starting with certain patterns.

The `py.test` can also be installed with `pip`:

```
$ pip install pytest
```

Like `nose2`, test execution with pytest is also easy. Simply run the `pytest` executable in the folder containing the test cases:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ pytest
======================================== test session starts ========================================
platform linux -- Python 3.5.2, pytest-3.0.0, py-1.4.31, pluggy-0.3.1
rootdir: /home/anand/Documents/ArchitectureBook/code/chap3, inifile:
collected 3 items

test_datetimehelper.py ...

===================================== 3 passed in 0.02 seconds =====================================
```

Test discovery and execution with py.test

Like nose2, `pytest` also comes with its own plugin support, the most useful among them being the code coverage plugin. We will see examples in a later section.

It is to be noted that pytest doesn't require test cases to be derived formally from the `unittest.TestCase` module. `pytest` automatically discovers tests from any modules containing classes prefixed with `Test` or from functions prefixed with `test_`.

For example, here is a new test case without any dependency on the `unittest` module but with the test case class derived from `object`, the most base type in Python. The new module is called `test_datetimehelper_object`:

```
""" Module test_datetimehelper_object - Simple test case with test
class derived from object """

import datetimehelper

class TestDateTimeHelper(object):

    def test_us_india_conversion(self):
        """ Test us=>india date format conversion """

        obj = datetimehelper.DateTimeHelper()
        assert obj.us_to_indian('1/1/1') == '01/01/2001'
```

Note how this class has zero dependency on the `unittest` module and defines no fixtures. Here is the output of running `pytest` on the folder now:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ py.test -v
============================================ test session starts ============================================
platform linux -- Python 3.5.2, pytest-3.0.0, py-1.4.31, pluggy-0.3.1 -- /home/anand/arch3/env/bin/python3
cachedir: .cache
rootdir: /home/anand/Documents/ArchitectureBook/code/chap3, inifile:
plugins: cov-2.3.1
collected 4 items

test_datetimehelper.py::DateTimeHelperTestCase::test_date PASSED
test_datetimehelper.py::DateTimeHelperTestCase::test_us_india_conversion PASSED
test_datetimehelper.py::DateTimeHelperTestCase::test_weekday PASSED
test_datetimehelper2.py::TestDateTimeHelper::test_us_india_conversion PASSED

========================================= 4 passed in 0.02 seconds ==========================================
```

Test case discovery and execution without the unittest module support using py.test

The `py.test` has picked up the test case in this module and executed it automatically as the output shows.

`nose2` also has similar capabilities to pick up such test cases. The following screenshot shows the output of `nose2` with the new test case defined:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2 -v
test_date (test_datetimehelper.DateTimeHelperTestCase)
Test date() method ... ok
test_us_india_conversion (test_datetimehelper.DateTimeHelperTestCase)
Test us=>india date format conversion ... ok
test_weekday (test_datetimehelper.DateTimeHelperTestCase)
Test weekday() method ... ok
test_datetimehelper2.TestDateTimeHelper.test_us_india_conversion ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

Test case discovery and execution without the unittest module support using nose2

The preceding output shows that the new test has been picked up and executed.

The `unittest` module, `nose2`, and `py.test` packages provide a lot of support for developing and implementing test cases, fixtures, and test suites in a very flexible and customizable manner. Discussing all of the multitude of options of these tools is beyond the scope of this chapter, as our focus is on getting to know these tools to understand how we can use them to satisfy the architectural quality attribute of testability.

So, at this point, we will go on to the next major topic in unit testing, that of **code coverage**. We will look at these three tools, namely, `unittest`, `nose2`, and `pytest`, and see how they allow the architect to help his/her developers and testers find information about the code coverage in their unit tests.

Code coverage

Code coverage is measured as the degree to which the source code under test is covered by a specific test suite. Ideally, test suites should aim for higher code coverage, as this would expose a larger percentage of the source code to tests and help to uncover bugs.

Code coverage metrics are reported typically as a percentage of **Lines of Code** (**LOC**) or a percentage of the subroutines (functions) covered by a test suite.

Let's now look at different tools support for measuring code coverage. We will continue to use our test example (`datetimehelper`) for these illustrations too.

Measuring coverage using coverage.py

`coverage.py` is a third-party Python module, which works with test suites and cases written with the `unittest` module, and reports their code coverage.

`coverage.py` can be installed, like other tools shown here so far, using pip:

```
$ pip install coverage
```

This last command installs the coverage application, which is used to run and report code coverages.

Coverage.py has two stages: first, where it runs a piece of source code and collects coverage information, and next, where it reports the coverage data.

To run `coverage.py`, use the following syntax:

```
$ coverage run <source file1> <source file 2> …
```

Once the run is complete, report the coverage using this command:

```
$ coverage report -m
```

For example, here is the output with our test modules:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ coverage run test_datetimehelper.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ coverage report -m
Name                     Stmts   Miss  Cover   Missing
------------------------------------------------------
datetimehelper.py           14      1    93%   9
test_datetimehelper.py      26      0   100%
------------------------------------------------------
TOTAL                       40      1    98%
```

Test coverage report for the datetimehelper module using coverage.py

`coverage.py` reports that our tests cover `93%` of the code in the `datetimehelper` module, which is pretty good code coverage. (You can ignore the report on the test module itself.)

Measuring coverage using nose2

The `nose2` package comes with plugin support for code coverage. This is not installed by default. To install the code coverage plugin for `nose2`, use this command:

```
$ pip install cov-core
```

Now, `nose2` can be run with the code coverage option to run the test cases and to report coverage in one shot. This can be done as follows:

```
$ nose2 -v -C
```

Behind the scenes, `cov-core` makes use of coverage.py to get its work done, so the metric report of coverage by both `coverage.py` and `nose2` is the same.

Here is the output of running test coverage using `nose2`:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2 -v -C
test_date (test_datetimehelper.DateTimeHelperTestCase)
Test date() method ... ok
test_us_india_conversion (test_datetimehelper.DateTimeHelperTestCase)
Test us=>india date format conversion ... ok
test_weekday (test_datetimehelper.DateTimeHelperTestCase)
Test weekday() method ... ok
test_datetimehelper2.TestDateTimeHelper.test_us_india_conversion ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
----------- coverage: platform linux, python 3.5.2-final-0 -----------
Name                        Stmts   Miss  Cover
-----------------------------------------------
datetimehelper.py              14      1    93%
test_datetimehelper.py         26      1    96%
test_datetimehelper2.py         5      0   100%
-----------------------------------------------
TOTAL                          45      2    96%
```

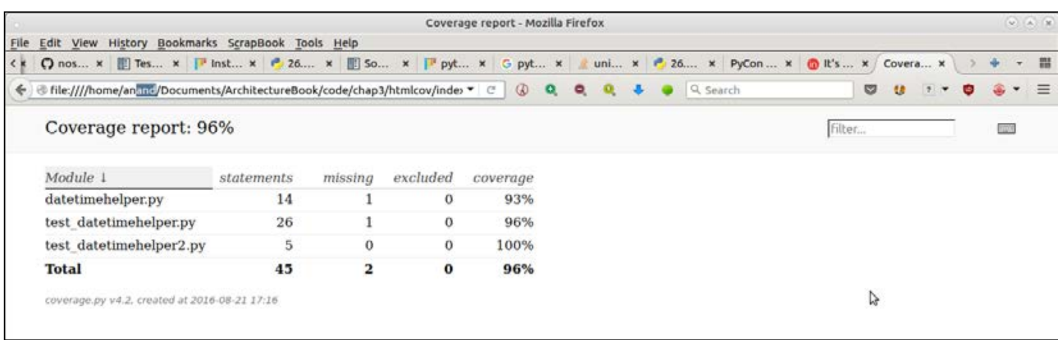
Test coverage report for the datetimehelper module using nose2

By default, the coverage report is written to the console. To produce other forms of output, the `-coverage-report` option can be used. For example, `--coverage-report html` will write the coverage report in the HTML format to a subfolder named `htmlcov`:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2 -C --coverage-report html
....
----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
---------- coverage: platform linux, python 3.5.2-final-0 ----------
coverage HTML written to dir htmlcov
```

Producing HTML coverage output using nose2

Here is how the HTML output looks in the browser:



HTML coverage report as viewed in the browser

Measuring coverage using pytest

`pytest` also comes with its own coverage plugin for reporting code coverage. Like `nose2`, it utilizes coverage.py behind the scenes to get the work done.

To provide support for code coverage for `py.test`, the `pytest-cov` package needs to be installed as follows:

```
$ pip install pytest-cov
```

To report code coverage of test cases in the current folder, use the following command:

```
$ pytest -cov
```

Here is a sample output of pytest code coverage:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ pytest --cov .
============================================= test session starts =============================================
platform linux -- Python 3.5.2, pytest-3.0.0, py-1.4.31, pluggy-0.3.1
rootdir: /home/anand/Documents/ArchitectureBook/code/chap3, inifile:
plugins: cov-2.3.1
collected 4 items

test_datetimehelper.py ...
test_datetimehelper2.py .

---------- coverage: platform linux, python 3.5.2-final-0 ----------
Name                       Stmts   Miss  Cover
----------------------------------------------
datetimehelper.py             14      1    93%
test_datetimehelper.py        26      1    96%
test_datetimehelper2.py        5      0   100%
----------------------------------------------
TOTAL                         45      2    96%


============================================= 4 passed in 0.04 seconds =============================================
```

Running code coverage for current folder using py.test

Mocking things up

We saw an example of using the patch support of `unittest.mock` in our test example earlier. However, the mock support provided by `unittest` is even more powerful than this, so let's look at one more example to understand its power and applicability in writing unit tests.

For the purpose of this illustration, we will consider a class that performs a keyword search on a large dataset and returns the results ordered by weightage. Assume that the dataset is stored in a database, and the results are returned as a list of (sentence, relevance) tuples, where sentence is the original string with a match for the keyword, and relevance is its hit weightage in the result set.

Here is the code:

```
"""
Module textsearcher - Contains class TextSearcher for performing
search on a database and returning results
"""

import operator

class TextSearcher(object):
    """ A class which performs a text search and returns results """

    def __init__(self, db):
```

```
        """ Initializer - keyword and database object """

        self.cache = False
        self.cache_dict = {}
        self.db = db
        self.db.connect()

    def setup(self, cache=False, max_items=500):
        """ Setup parameters such as caching """

        self.cache = cache
        # Call configure on the db
        self.db.configure(max_items=max_items)

    def get_results(self, keyword, num=10):
        """ Query keyword on db and get results for given keyword """

        # If results in cache return from there
        if keyword in self.cache_dict:
            print ('From cache')
            return self.cache_dict[keyword]

        results = self.db.query(keyword)
        # Results are list of (string, weightage) tuples
        results = sorted(results, key=operator.itemgetter(1),
                    reverse=True)[:num]
        # Cache it
        if self.cache:
            self.cache_dict[keyword] = results

        return results
```

The class has the following three methods:

- `__init__`: This is the initializer; it accepts an object that acts as a handle to the data source (database). It also initializes a few attributes and connects to the database
- `setup`: It sets up the searcher and configures the database object
- `get_results`: It performs a search using the data source (database) and returns the results for a given keyword

We now want to implement a unit test case for this searcher. Since the database is an external dependency, we will virtualize the database object by mocking it. We will test only the searcher's logic, callable signatures, and return data.

We will develop this program step by step so that each step of mocking is clear to you. We will use a Python interactive interpreter session for the same.

First, let's get the mandatory imports:

```
>>> from unittest.mock import Mock, MagicMock
>>> import textsearcher
>>> import operator
```

Since we want to mock the DB, the first step is to do that exactly:

```
>>> db = Mock()
```

Now let's create the `searcher` object. We are not going to mock this, as we need to test the calling signature and the return value of its methods:

```
>>> searcher = textsearcher.TextSearcher(db)
```

At this point, the database object has been passed to the `__init__` method of `searcher`, and `connect` has been called on it. Let's verify this expectation:

```
>>> db.connect.assert_called_with()
```

No issues, so the assertion has succeeded! Let's now set up `searcher`:

```
>>> searcher.setup(cache=True, max_items=100)
```

Looking at the code of the `TextSearcher` class, we realize that the preceding call should have called `configure` on the database object with the `max_items` parameter set to the value `100`. Let's verify this:

```
>>> searcher.db.configure.assert_called_with(max_items=100)
<Mock name='mock.configure_assert_called_with()' id='139637252379648'>
```

Bravo! Finally, let's try and test the logic of the `get_results` method. Since our database is a mock object, it won't be able to do any actual query, so we pass some canned results to its `query` method, effectively mocking it:

```
>>> canned_results = [('Python is wonderful', 0.4),
...                        ('I like Python',0.8),
...                        ('Python is easy', 0.5),
...                        ('Python can be learnt in an afternoon!',
0.3)]
>>> db.query = MagicMock(return_value=canned_results)
```

Now we set up the keyword and the number of results and call `get_results` using these parameters:

```
>>> keyword, num = 'python', 3
>>> data = searcher.get_results(python, num=num)
```

Let's inspect the data:

```
>>> data
[('I like Python', 0.8), ('Python is easy', 0.5), ('Python is
wonderful', 0.4)]
```

It looks good! In the next step, we verify that `get_results` has indeed called `query` with the given keyword:

```
>>> searcher.db.query.assert_called_with(keyword)
```

Finally, we verify that the data returned has been sorted right and truncated to the number of results (`num`) value we passed:

```
>>> results = sorted(canned_results, key=operator.itemgetter(1),
reverse=True)[:num]
>>> assert data == results
True
```

All good! The example shows how to use mock support in the `unittest` module in order to mock an external dependency and effectively virtualize it, while at the same time testing the program's logic, control flow, callable arguments, and return values.

Here is a test module combining all of these tests into a single test module and the output of `nose2` on it:

```
"""
Module test_textsearch - Unittest case with mocks for textsearch
module
"""

from unittest.mock import Mock, MagicMock
import textsearcher
import operator

def test_search():
```

```
""" Test search via a mock """

# Mock the database object
db = Mock()
searcher = textsearcher.TextSearcher(db)
# Verify connect has been called with no arguments
db.connect.assert_called_with()
# Setup searcher
searcher.setup(cache=True, max_items=100)
# Verify configure called on db with correct parameter
searcher.db.configure.assert_called_with(max_items=100)

canned_results = [('Python is wonderful', 0.4),
                  ('I like Python',0.8),
                  ('Python is easy', 0.5),
                  ('Python can be learnt in an afternoon!', 0.3)]
db.query = MagicMock(return_value=canned_results)

# Mock the results data
keyword, num = 'python', 3
data = searcher.get_results(keyword,num=num)
searcher.db.query.assert_called_with(keyword)

# Verify data
results = sorted(canned_results, key=operator.itemgetter(1),
          reverse=True)[:num]
assert data == results
```

Here is the output of `nose2` on this test case:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ nose2 -v test_textsearch
test_textsearch.transplant_class.<locals>.C (test_search)
Test search via a mock ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$
```

Running testsearcher test-case using nose2

For good measure, let's also look at the coverage of our mock test example, the `test_textsearch` module, using the py.test coverage plugin:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ pytest --cov textsearcher
======================================== test session starts ========================================
platform linux -- Python 3.5.2, pytest-3.0.0, py-1.4.31, pluggy-0.3.1
rootdir: /home/anand/Documents/ArchitectureBook/code/chap3, inifile:
plugins: cov-2.3.1
collected 5 items

test_datetimehelper.py ...
test_datetimehelper2.py .
test_textsearch.py .

----------- coverage: platform linux, python 3.5.2-final-0 -----------
Name              Stmts   Miss  Cover
-------------------------------------
textsearcher.py      19      2    89%


======================================== 5 passed in 0.04 seconds ========================================
```

Measuring coverage of the textsearcher module via test_textsearch test case using py.test

So our mock test has a coverage of `89%`, missing just two statements out of 20. Not bad!

Tests inline in documentation – doctests

Python has unique support for another form of inline code tests, which are commonly called **doctests**. These are inline unit tests in a function, class, or module documentation, which add a lot of value by combining code and tests in one place without having to develop or maintain separate test suites.

The `doctest` module works by looking for pieces of text in code documentation that look like Python strings, and executing those sessions to verify that they work exactly as found. Any test failures are reported on the console.

Let's look at a code example to see this in action. The following piece of code implements the simple factorial function by using an iterative approach:

```
"""
Module factorial - Demonstrating an example of writing doctests
"""

import functools
import operator

def factorial(n):
```

```
        """ Factorial of a number.

        >>> factorial(0)
        1
        >>> factorial(1)
        1
        >>> factorial(5)
        120
        >>> factorial(10)
        3628800

        """

        return functools.reduce(operator.mul, range(1,n+1))

    if __name__ == "__main__":
        import doctest
        doctest.testmod(verbose=True)
```

Let's look at the output of executing this module:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ python3 factorial.py
**********************************************************************
File "factorial.py", line 13, in __main__.factorial
Failed example:
    factorial(0)
Exception raised:
    Traceback (most recent call last):
      File "/usr/lib/python3.5/doctest.py", line 1321, in __run
        compileflags, 1), test.globs)
      File "<doctest __main__.factorial[3]>", line 1, in <module>
        factorial(0)
      File "factorial.py", line 17, in factorial
        return functools.reduce(operator.mul, range(1,n+1))
    TypeError: reduce() of empty sequence with no initial value
**********************************************************************
1 items had failures:
   1 of   4 in __main__.factorial
***Test Failed*** 1 failures.
```

Output of doctest for the factorial module

The doctest reports that one out of four tests failed.

A quick scan of the output tells us that we forgot to code in the special case to compute the factorial for zero. The error occurs because the code tries to compute `range(1, 1)`, which raises an exception with `reduce`.

The code can be easily rewritten to fix this. Here is the modified code:

```
"""
Module factorial - Demonstrating an example of writing doctests
"""

import functools
import operator

def factorial(n):
    """ Factorial of a number.

    >>> factorial(0)
    1
    >>> factorial(1)
    1
    >>> factorial(5)
    120
    >>> factorial(10)
    3628800
    """

    # Handle 0 as a special case
    if n == 0:
        return 1

    return functools.reduce(operator.mul, range(1,n+1))

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

The next screenshot shows the fresh output of executing the module now:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ python3 factorial.py
Trying:
    factorial(1)
Expecting:
    1
ok
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    factorial(10)
Expecting:
    3628800
ok
Trying:
    factorial(0)
Expecting:
    1
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.factorial
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$
```

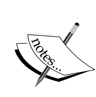
Output of doctest for the factorial module after the fix

Now all of the tests pass.

> We turned on the verbose option of the doctest module's `testmod` function in this example in order to show the details of the tests. Without this option, doctest would be silent if all of the tests passed, producing no output.

The `doctest` module is very versatile. Rather than just Python code, it can also load Python interactive sessions from sources such as text files and execute them as tests.

The `doctest` module examines all docstrings including function, class, and module docstrings to search for Python interactive sessions.



> The `pytest` package comes with built-in support for doctests. To allow `pytest` to discover and run doctests in the current folder, use the following command:

```
$ pytest -doctest-modules
```

Integration tests

Unit tests, though very useful to discover and fix bugs during white-box testing early on in the software development life cycle, aren't enough by themselves. A software system is fully functional only if the different components work together in expected ways in order to deliver the required functionality to the end user, satisfying the pre-defined architectural quality attributes. This is where integration tests assume importance.

The purpose of integration tests is to verify the functional, performance, and other quality requirements on the different functional subsystems of a software system, which act as a logical unit, providing certain functionality. Such subsystems deliver some piece of functionality through the cumulative action of their individual units. Though each component may have defined its own unit test, it is also important to verify the combined functionality of the system by writing integration tests.

Integration tests are usually written after unit testing is completed and before validation testing is done.

It would be instructional to list down the advantages provided by integration tests at this point, as this could be useful for any software architect who is at a phase where he/she has designed and implemented his/her unit tests for the different components:

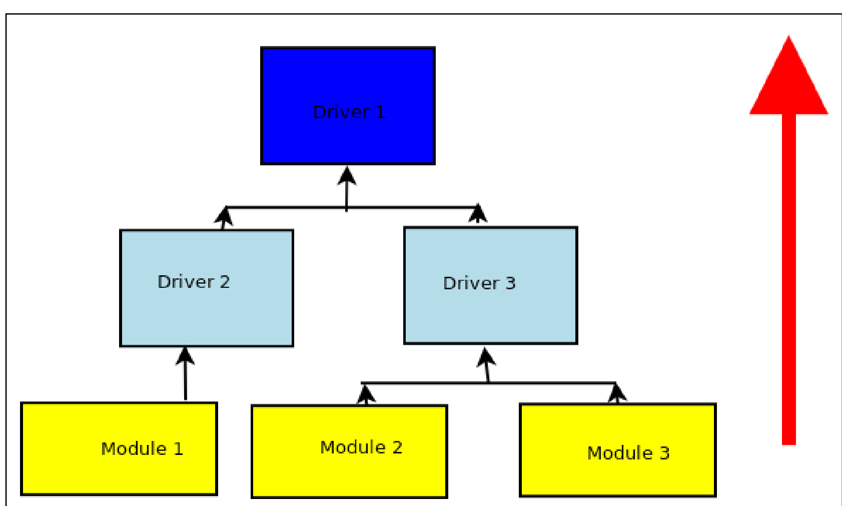
- **Testing component interoperability**: Each unit in a functional subsystem could be written by different programmers. Though each programmer is aware of how this component should perform, and may have written unit tests for the same, the entire system may have issues working in unison, as there could be errors or misunderstanding in the integration points where components talk to each other. Integration testing would reveal such mistakes.
- **Testing for system requirement modifications**: The requirements may have changed during the time of implementation. These updated requirements may not have been unit tested, hence, an integration test becomes very useful to reveal issues. Also, some parts of the system may not have implemented the requirements correctly, which can also be revealed by an appropriate integration test.
- **Testing external dependencies and APIs**: Software components these days use a lot of third-party APIs, which are usually mocked or stubbed during unit tests. Only an integration test would reveal how these APIs would perform and expose any issues either in the calling convention, response data, or performance with them.

- **Debugging hardware issues**: Integration tests are helpful in getting information about any hardware problems, and debugging such tests gives the developer(s) data about whether an update or change in the hardware configuration is required.
- **Uncovering exceptions in code paths**: Integration tests can also help developers figure out exceptions that they may not have handled in their code, as unit tests wouldn't have executed paths or conditions which raised such errors. Higher code coverage can identify and fix a lot of such issues. However, a good integration test combining known code paths for each functionality with high coverage is a good formula for making sure most potential errors that may occur during usage are uncovered and executed during testing.

There are three approaches to writing integration tests. These are as follows:

- **Bottom-up**: In this approach, components at the lower level are tested first, and these test results are used to integrate tests of the higher-level components in the chain. The process repeats until we reach the top of the hierarchy of the components with respect to the control flow. In this approach, critical modules at the top of the hierarchy may be tested inadequately.

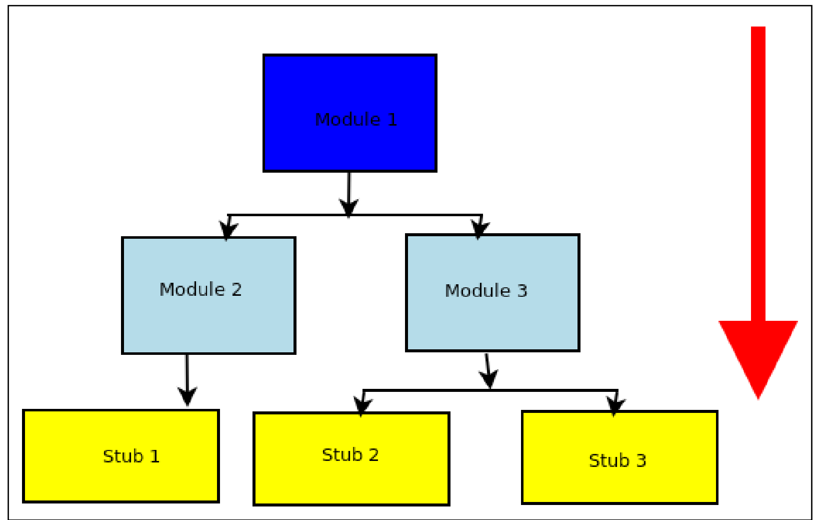
  If the top-level components are under development, drivers may be required to simulate (mock) them:



Bottom-up strategy of integration testing

- **Top-down**: Test development and testing happens top-down, following the workflow in the software system. Hence, components at the top level of the hierarchy are tested first and the lower-level modules are tested last. In this approach, critical modules are tested on priority, so we can identify major design or development flaws first and fix them. However, lower-level modules may be tested inadequately.

  Lower-level modules can be replaced by stubs which mock their functionality. Early prototypes are possible in this approach, as lower-level module logic can be stubbed out:



Top-down strategy of integration testing

- **Big-bang**: This is the approach is one where all of the components are integrated and tested at the very end of development. Since the integration tests come at the end, this approach saves time for development. However, this may not give enough time to test critical modules, as there may not be enough time to spend equally on all of the components.

There is no specific software for general integration testing. A certain class of applications, such as web frameworks, define their own specific integration test frameworks. For example, some web frameworks such as Django, Pyramid, and Flask have some specific testing frameworks developed by their own communities.

Another example is the popular `WebTest` framework, which is useful for automated testing of the Python WSGI applications. A detailed discussion of such frameworks is outside the scope of this chapter and this book.

Test automation

There are a number of tools on the internet that are useful for automating integration testing of software applications. We will take a quick look at some of the popular ones here.

Test automation using Selenium WebDriver

Selenium has been a popular choice for automating integration, regression, and validation tests for a number of software applications. Selenium is free and open source and comes with support for most popular web browser engines.

In Selenium, the primary object is a **web driver**, which is a stateful object on the client side, representing a browser. The web driver can be programmed to visit URLs, perform actions (such as clicking, filling forms, and submitting forms), effectively replacing the human test subject, who usually performs these steps manually.

Selenium provides client driver support for most popular programming languages and runtimes.

To install the Selenium WebDriver in Python, use the following command:

```
$ pip install selenium
```

We will look at a small example that uses Selenium along with pytest in order to implement a small automation test, which will test the Python website (`http://www.python.org`) for some simple test cases.

Here is our test code. The module is named `selenium_testcase.py`:

```
"""
Module selenium_testcase - Example of implementing an automated UI
test using selenium framework
"""

from selenium import webdriver
import pytest
import contextlib

@contextlib.contextmanager
@pytest.fixture(scope='session')
def setup():
    driver = webdriver.Firefox()
    yield driver
```

```
        driver.quit()

    def test_python_dotorg():
        """ Test details of python.org website URLs """

        with setup() as driver:
            driver.get('http://www.python.org')
            # Some tests
            assert driver.title == 'Welcome to Python.org'
            # Find out the 'Community' link
            comm_elem = driver.find_elements_by_link_text('Community')[0]
            # Get the URL
            comm_url = comm_elem.get_attribute('href')
            # Visit it
            print ('Community URL=>',comm_url)
            driver.get(comm_url)
            # Assert its title
            assert driver.title == 'Our Community | Python.org'
            assert comm_url == 'https://www.python.org/community/'
```

Before running the preceding example and showing the output, let's inspect the functions a bit:

- The `setUp` function is a test fixture, which sets up the main object required for our test, that is, the Selenium WebDriver for Firefox. We convert the `setUp` function in to a context manager by decorating it with the `contextmanager` decorator from the `contextlib` module. At the end of the `setUp` function, the driver exits, since its `quit` method is called.
- In the `test_python_dot_org` test function, we set up a rather simple, contrived test for visiting the main Python website URL and checking its title via an assertion. We load the URL for the Python community by locating it on the main page and then visit this URL. We finally assert its title and URL before ending our tests.

Let's see the program in action. We will specifically ask pytest to load only this module and run it. The command line for this is as follows:

```
$ pytest -s selenium_testcase.py
```

The Selenium driver will launch the browser (Firefox) and open a window automatically, visiting the Python website URL while running the tests.
The console output for the test is shown in the following screenshot:

```
(env) anand@ubuntu-pro-book:~/Documents/ArchitectureBook/code/chap3$ pytest -s selenium_testcase.py
========================================= test session starts ==========================================
platform linux -- Python 3.5.2, pytest-3.0.0, py-1.4.31, pluggy-0.3.1
rootdir: /home/anand/Documents/ArchitectureBook/code/chap3, inifile:
plugins: cov-2.3.1
collected 1 items

selenium_testcase.py Community URL=> https://www.python.org/community/
.

====================================== 1 passed in 16.35 seconds ======================================
```

Console output of a simple Selenium test case on the Python programming language website

Selenium can be used for more complex test cases, as it provides a number of methods for inspecting the HTML of pages, locating elements, and interacting with them. There are also plugins for Selenium, which can execute the JavaScript content of the pages to make the testing support complex interactions via JavaScript (such as AJAX requests).

Selenium can also be run on the server. It provides support for remote clients via its remote driver support. Browsers are instantiated on the server (typically, using virtual X sessions), whereas, the tests can be run and controlled from client machines via the network.

Test-driven development

Test-Driven Development (**TDD**) is an agile practice of software development, which uses a very short development cycle, where code is written to satisfy an incremental test case.

In TDD, a functional requirement is mapped to a specific test case. Code is written to pass the first test case. Any new requirement is added as a new test case. Code is refactored to support the new test case. The process continues till the code is able to support the entire spectrum of user functionality.

The steps in TDD are as follows:

1. Define a few starting test cases as a specification for the program.
2. Write code to make the early test cases pass.
3. Add a new test case defining new functionality.
4. Run all of the tests and see whether the new test fails or passes.
5. If the new test fails, write some code for the test to pass.

6. Run the tests again.
7. Repeat steps 4 to 6 till the new test passes.
8. Repeat steps 3 to 7 to add a new functionality via test cases.

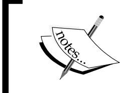
In TDD, the focus is on keeping everything simple, including the unit test cases and the new code that is added to support the test cases. TDD practitioners believe that writing tests upfront allows the developer to understand the product requirements better, allowing a focus on software quality from the very beginning of the development lifecycle.

In TDD, often, a final refactoring step is also done after many tests have been added to the system in order to make sure no coding smells or antipatterns are introduced and to maintain code readability and maintainability.

There is no specific software for TDD, rather, it is a methodology and process for software development. Most of the time, TDD uses unit tests, so the toolchain support is mostly the `unittest` module and the related packages that we've discussed in this chapter.

TDD with palindromes

Let's understand TDD as discussed earlier with a simple example of developing a program in Python that checks whether an input string is a palindrome.



> A palindrome is a string that reads the same in both directions. For example, *bob*, *rotator*, and *Malayalam* are palindromes. So is the sentence, *Madam, I'm Adam* when you get rid of the punctuation marks.

Let us follow the steps of TDD. Initially, we need a test case that defines the basic specification of the program. Our first version of the test code looks like this:

```
"""
Module test_palindrome - TDD for palindrome module
"""

import palindrome

def test_basic():
    """ Basic test for palindrome """

    # True positives
    for test in ('Rotator','bob','madam','mAlAyAlam', '1'):
```

```
        assert palindrome.is_palindrome(test)==True

    # True negatives
    for test in ('xyz','elephant', 'Country'):
        assert palindrome.is_palindrome(test)==False
```

Note that the preceding code not only gives us a specification for the program in terms of its early functionality, but also gives a function name and signature—in terms of the argument and return value. We can list down the requirements for the first version by looking at the test:

- The function is named is _palindrome. It should accept a string and return True if it is a palindrome and False otherwise. The function sits in the palindrome module.
- The function should treat strings as case-insensitive.

With these specifications, here is our first version of the palindrome module:

```
def is_palindrome(in_string):
    """ Returns True whether in_string is palindrome, False otherwise
    """

    # Case insensitive
    in_string = in_string.lower()
    # Check if string is same as in reverse
    return in_string == in_string[-1::-1]
```

Let's check whether this passes our test. We will run py.test on the test module to verify this:

```
Chapter 3: Testability
(env) $ py.test -s test_palindrome.py
==================================== test session starts ====================================
platform linux -- Python 3.5.2, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /home/user/programs/chap3, inifile:
plugins: cov-2.4.0
collected 1 items

test_palindrome.py .

================================= 1 passed in 0.02 seconds =================================
(env) $
```

Test output of test_palindrome.py version #1

As you can see in the last image, the basic test passes; so, we've got a first version of the `palindrome` module, which works and passes its tests.

Now as per the TDD step, let's go to step 3 and add a new test case. This adds a check for testing palindrome strings with spaces. Here is the new test module with this extra test:

```
"""
Module test_palindrome - TDD for palindrome module
"""

import palindrome

def test_basic():
    """ Basic test for palindrome """

    # True positives
    for test in ('Rotator','bob','madam','mAlAyAlam', '1'):
        assert palindrome.is_palindrome(test)==True

    # True negatives
    for test in ('xyz','elephant', 'Country'):
        assert palindrome.is_palindrome(test)==False

def test_with_spaces():
    """ Testing palindrome strings with extra spaces """

    # True positives
    for test in ('Able was I ere I saw Elba',
                 'Madam Im Adam',
                 'Step on no pets',
                 'Top spot'):
        assert palindrome.is_palindrome(test)==True

    # True negatives
    for test in ('Top post','Wonderful fool','Wild Imagination'):
        assert palindrome.is_palindrome(test)==False
```

Let's run the updated test and see the results:

```
Chapter 3: Testability
(env) $ py.test -s test_palindrome.py
========================================= test session starts ==========================================
platform linux -- Python 3.5.2, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /home/user/programs/chap3, inifile:
plugins: cov-2.4.0
collected 2 items

test_palindrome.py .F

=============================================== FAILURES ===============================================
__________________________________________ test_with_spaces ___________________________________________

    def test_with_spaces():
        """ Testing palindrome strings with extra spaces """

        # True positives
        for test in ('Able was I ere I saw Elba',
                     'Madam Im Adam',
                     'Step on no pets',
                     'Top spot'):
>           assert palindrome.is_palindrome(test)==True
E           AssertionError: assert False == True
E            +  where False = <function is_palindrome at 0x7fd856207488>('Madam Im Adam')
E            +    where <function is_palindrome at 0x7fd856207488> = palindrome.is_palindrome

test_palindrome.py:28: AssertionError
=================================== 1 failed, 1 passed in 0.07 seconds ===================================
(env) $ 
```

Test output of test_palindrome.py version #2

The test fails, because the code is not enabled to process palindrome strings with spaces in them. So let's do as TDD step 5 says and write some code to make this test pass.

Since it is clear we need to ignore spaces, a quick fix is to purge all spaces from the input string. Here is the modified `palindrome` module with this simple fix:

```
"""
Module palindrome - Returns whether an input string is palindrome or
not
"""

import re

def is_palindrome(in_string):
    """ Returns True whether in_string is palindrome, False otherwise
    """

    # Case insensitive
    in_string = in_string.lower()
    # Purge spaces
    in_string = re.sub('\s+','', in_string)
    # Check if string is same as in reverse
    return in_string == in_string[-1::-1]
```

Let's now repeat step 4 of TDD to see whether the updated code makes the test pass:

```
Chapter 3: Testability
(env) $ py.test -s test_palindrome.py -v
======================================== test session starts ========================================
platform linux -- Python 3.5.2, pytest-3.0.7, py-1.4.33, pluggy-0.4.0 -- /home/anand/py3/env/bin/python
3
cachedir: .cache
rootdir: /home/user/programs/chap3, inifile:
plugins: cov-2.4.0
collected 2 items

test_palindrome.py::test_basic PASSED
test_palindrome.py::test_with_spaces PASSED

===================================== 2 passed in 0.01 seconds ======================================
(env) $
```

Console output of test_palindrome.py version #2, after code updates

Surely, the code passes the test now!

What we just saw was an instance of TDD with one update cycle for implementing a module in Python, which checks strings for palindromes. In a similar way, we can keep adding tests and keep updating the code as per step 8 of TDD, thereby adding new functionality while maintaining the updated tests naturally via the process.

We conclude this section with the final version of our palindrome test case, which adds a test case for checking for strings with extra punctuation marks:

```
"""
Module test_palindrome - TDD for palindrome module
"""

import palindrome

def test_basic():
    """ Basic test for palindrome """

    # True positives
    for test in ('Rotator','bob','madam','mAlAyAlam', '1'):
        assert palindrome.is_palindrome(test)==True

    # True negatives
    for test in ('xyz','elephant', 'Country'):
        assert palindrome.is_palindrome(test)==False

def test_with_spaces():
```

```
        """ Testing palindrome strings with extra spaces """

        # True positives
        for test in ('Able was I ere I saw Elba',
                     'Madam Im Adam',
                     'Step on no pets',
                     'Top spot'):
            assert palindrome.is_palindrome(test)==True

        # True negatives
        for test in ('Top post','Wonderful fool','Wild Imagination'):
            assert palindrome.is_palindrome(test)==False


    def test_with_punctuations():
        """ Testing palindrome strings with extra punctuations """

        # True positives
        for test in ('Able was I, ere I saw Elba',
                     "Madam I'm Adam",
                     'Step on no pets.',
                     'Top spot!'):
            assert palindrome.is_palindrome(test)==True

        # True negatives
        for test in ('Top . post','Wonderful-fool','Wild Imagination!!'):
            assert palindrome.is_palindrome(test)==False
```

And here is the updated `palindrome` module that makes this test pass:

```
"""
Module palindrome - Returns whether an input string is palindrome or
not
"""

import re
from string import punctuation

def is_palindrome(in_string):
    """ Returns True whether in_string is palindrome, False otherwise
    """

    # Case insensitive
    in_string = in_string.lower()
```

```
    # Purge spaces
    in_string = re.sub('\s+','', in_string)
    # Purge all punctuations
    in_string = re.sub('[' + re.escape(punctuation) + ']+', '',
                in_string)
    # Check if string is same as in reverse
    return in_string == in_string[-1::-1]
```

Let's inspect the final output of the `test_palindrome` module on the console:

```
Chapter 3: Testability
(env) $ py.test -s test_palindrome.py -v
======================================== test session starts ========================================
platform linux -- Python 3.5.2, pytest-3.0.7, py-1.4.33, pluggy-0.4.0 -- /home/anand/py3/env/bin/python
3
cachedir: .cache
rootdir: /home/user/programs/chap3, inifile:
plugins: cov-2.4.0
collected 3 items

test_palindrome.py::test_basic PASSED
test_palindrome.py::test_with_spaces PASSED
test_palindrome.py::test_with_punctuations PASSED

===================================== 3 passed in 0.03 seconds ======================================
(env) $
```

Console output of test_palindrome.py version #3, with matching code updates

Summary

In this chapter, we revisited the definition of testability and its related architectural quality aspects, such as complexity and determinism. We looked at the different architectural aspects that are tested and got an understanding of the type of tests that are usually performed by the software testing process.

We then discussed the various strategies for improving the testability of software, and looked at techniques to reduce system complexity and improve predictability and to control and manage external dependencies. Along the way, we learned the different ways to virtualize and manage external dependencies, such as fakes, mocks and stubs, by way of examples.

We then looked at unit testing and its various aspects mainly from the perspective of the Python `unittest` module. We saw an example by using a `datetime` helper class, and explained how to write effective unit tests—a simple example followed by an interesting example of patching functions using the mock library of `unittest`.

We then introduced, and learned quite a bit about, the two other well-known testing frameworks in Python, namely, `nose2` and `py.test`. Next, we discussed the very important aspect of code coverage and saw examples of measuring code coverage using the `coverage.py` package directly, and by using it via plugins of `nose2` and `pytest`.

In the next section, we sketched an example of a `textsearch` class for using advanced mock objects, where we mocked its external dependency and wrote a unit test case. We went on to discuss the Python doctest support of embedding tests in the documentation of classes, modules, methods, and functions via the `doctest` module while looking at examples.

The next topic was integration tests, where we discussed the different aspects and advantages of integration tests, and looked at the three different ways in which tests can be integrated in a software organization. Test automation via Selenium was discussed next with an example of automating a couple of tests on the Python language website using Selenium and `py.test`.

We ended this chapter with a quick overview of TDD, and discussed an example of writing a program for detecting palindromes in Python using TDD principles, where we developed the program using tests in a step-by-step fashion.

In the next chapter, we will look at one of the most critical quality attribute of architecture when developing software—namely, performance.

4
Good Performance is Rewarding!

Performance is one of the cornerstones of modern-day software applications. Every day we interact with high-performing computing systems in many different ways, as part of our work and our leisure.

When you book an airline ticket from one of the travel sites on the web, you are interacting with a high-performance system that carries out hundreds of such transactions at any given time. When you transfer money to someone or pay your credit card bill online via an internet banking transaction, you are interacting with a high performance and high throughput transactional system. Similarly, when you play online games on your mobile phone and interact with other players, again there is a network of servers built for high concurrency and low latency that is receiving input from you and thousands of other players, performing computations at the backend and sending data to you—all with reasonable and quiet efficiency.

Modern day web applications that serve millions of users concurrently became possible with the advent of high-speed internet and huge drops in the price and performance ratio of hardware. Performance is still a key quality attribute of modern day software architecture and writing high-performing and scalable software still continues to be something of a difficult art. You may write an application which ticks all the boxes of functionality and other quality attributes, but if it fails its performance tests, then it cannot be moved to production.

In this chapter and the next, we focus on two aspects of writing software with high throughput—namely performance and scalability. In this chapter, the focus is on performance, the various aspects of it, how to measure it, the performance of various data structures, and when to choose what—with the focus on Python.

The topics we will be discussing in this chapter roughly fall under the following sections:

- Defining performance
- Software performance engineering
- Types of performance-testing tool
- Performance complexity and the Big O notation:
    - Measuring performance
    - Finding performance complexity using graphs
    - Improving performance
- Profiling:
    - Deterministic profiling
    - `cProfile` and `profile`
    - Third-party profilers
- Other tools:
    - Objgraph
    - Pympler
- Programming for performance—data structures:
    - Lists
    - Dictionaries
    - Sets
    - Tuples
- High performance containers—the collections module:
    - `deque`
    - `defaultdict`
    - `OrderedDict`
    - `Counter`
    - `ChainMap`
    - `namedtuple`
- Probabilistic data structures—bloom filters

What is performance?

The performance of a software system can be broadly defined as:

> *The degree to which the system is able to meet its throughput and/or latency requirements in terms of the number of transactions per second or time taken for a single transaction.*

We've already taken an overview of measuring performance in the introductory chapter. Performance can be measured either in terms of response time/latency or in terms of throughput. The former is the time it takes for the application to complete a request/response loop on average. The latter is the rate at which the system processes its input in terms of the number of requests or transactions successfully completed per minute.

The performance of a system is a function of its software and of its hardware capabilities. A badly written piece of software could still be made to perform better by scaling the hardware—for example, the amount of RAM.

Similarly, a piece of software can be made to work better on existing hardware by increasing its performance—for example, by rewriting routines or functions to be more efficient in terms of time or memory, or by modifying the architecture.

However, the right type of performance engineering is the one where the software is tuned for the hardware in an optimal fashion so that software scales linearly or better with respect to the available hardware.

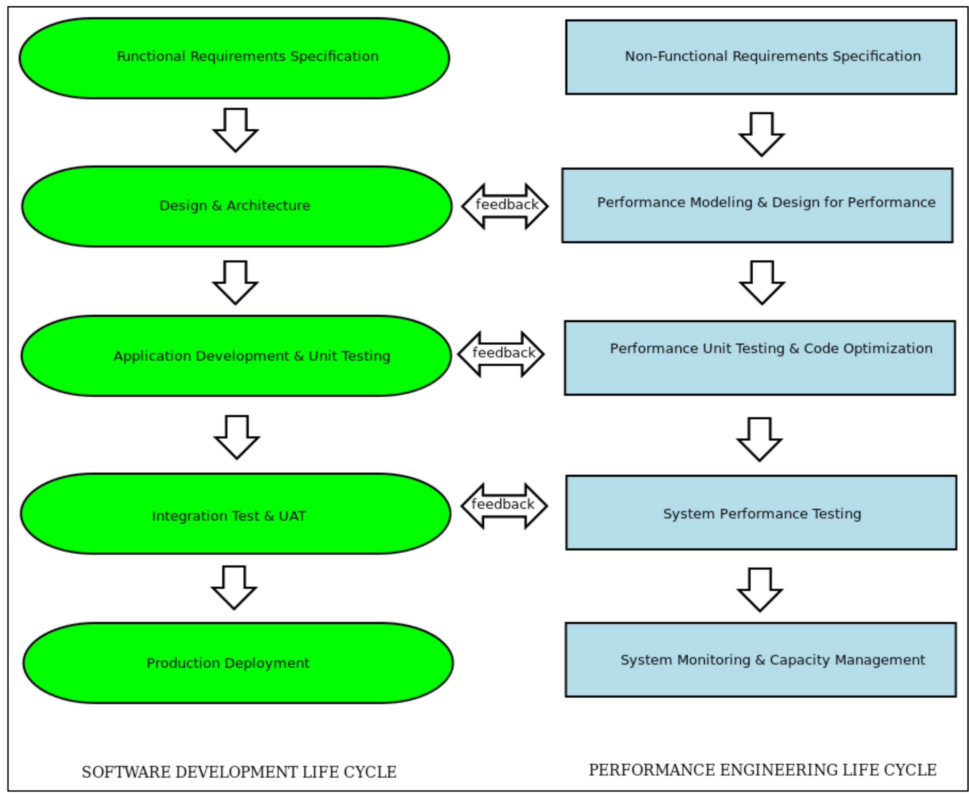
Software performance engineering

Software performance engineering includes all the activities of software engineering and analysis applied during the **Software Development Life Cycle** (**SDLC**) and is directed towards meeting performance requirements.

In conventional software engineering, performance testing and feedback are done usually towards the end of the SDLC. This approach is purely measurement-based and waits for the system to be developed before applying tests and diagnostics and tuning the system based on the results.

Another more formal model named **Software Performance Engineering** (**SPE**), itself develops performance models early in the SDLC and uses results from the models to modify the software design and architecture to meet performance requirements in multiple iterations.

In this approach, both performance as a non-functional requirement and software development meeting its functional requirement go hand in hand. There is a specific **Performance Engineering Life Cycle** (**PELC**) that parallels the steps in the SDLC. At every step, starting from the design and architecture all the way to deployment, feedback between both the life cycles is used to iteratively improve the software quality:



SPE—Performance Engineering Life Cycle mirroring Software Development Life Cycle

In both approaches, performance testing and diagnostics are important, followed by tuning the design/architecture or the code based on the results obtained. Hence performance testing and measurement tools play an important role in this step.

Performance testing and measurement tools

These tools fall under two broad categories—namely, the ones used for performance testing and diagnostics, and the ones used for performance metrics gathering and instrumentation.

Performance testing and diagnostic tools can be classified further as follows:

- **Stress-testing tools**: These tools are used to supply workload to the system under test, simulating peak workloads in production. These tools can be configured to send a continuous stream of input to the application to simulate high stress or to periodically send a burst of very high traffic—much exceeding even peak stress—to test the robustness of the system. These tools are also called **load generators**. Examples of common stress testing tools used for web application testing include **httpperf**, **ApacheBench**, **LoadRunner**, **Apache JMeter**, and **Locust**. Another class of tools involves those that actually record real user traffic and then replay it via the network to simulate real user load. For example, the popular network packet capturing and monitoring tool, **Wireshark** and its console cousin program, `tcpdump`, can be used to do this. We won't be discussing these tools in this chapter as they are general-purpose and examples of usage for them can be found in abundance on the web.
- **Monitoring tools**: These tools work with the application code to generate performance metrics such as the time and memory taken for functions to execute, the number of function calls made per request-response loop, the average and peak times spent on each function, and so on.
- **Instrumentation tools**: Instrumentation tools trace metrics, such as the time and memory required for each computing step, and also track events, such as exceptions in code, covering such details as the module/function/line number where the exception occurred, the timestamp of the event, and the environment of the application (environment variables, application configuration parameters, user information, system information, and so on). Often external instrumentation tools are used in modern web-application programming systems to capture and analyze such data in detail.
- **Code or application profiling tools**: These tools generate statistics about functions, their frequency of duration of calls, and the time spent on each function call. This is a kind of dynamic program analysis. It allows the programmer to find critical sections of code where the most time is spent, allowing them to optimize those sections. Optimization without profiling is not advised as the programmer may end up optimizing the wrong code, thereby not surfacing the intended benefits up to the application.

Most programming languages come with their own set of instrumentation and profiling tools. In Python, a set of tools in the standard library (such as the `profile` and `cProfile` modules) do this—this is supplemented by a rich ecosystem of third-party tools. We will discuss these tools in the coming sections.

Performance complexity

It would be helpful to spend some time discussing what we mean by the performance complexity of code before we jump into code examples in Python and discuss tools to measure and optimize performance.

The performance complexity of a routine or function is defined in terms of how they respond to changes in the input size typically in terms of the time spent in executing the code.

This is usually represented by the so-called Big-O notation which belongs to a family of notations called the **Bachmann–Landau notation or asymptotic** notation.

The letter O is used as the rate of growth of a function with respect to input size—also called the **order** of the function.

Commonly used Big-O notations or function orders are shown in the following table in order of increasing complexity:

#	Order	Complexity	Example
1	*O(1)*	Constant	Looking for a key in a constant lookup table such as a HashMap or dictionary in Python
2	*O(log (n))*	Logarithmic	Searching for an item in a sorted array with a binary search. All operations on a heapq in Python
3	*O(n)*	Linear	Searching an item in an array (list in Python) by traversing it
4	*O(n*k)*	Linear	Worst-case complexity of Radix sort
5	*O(n * log (n))*	n log-star n	Worst-case complexity in a mergesort or heapsort algorithm
6	$O(n^2)$	Quadratic	Simple sorting algorithms such as bubblesort, insertion sort, and selection sort. Worst-case complexity on some sorting algorithms such as quicksort, shellsort, and so on

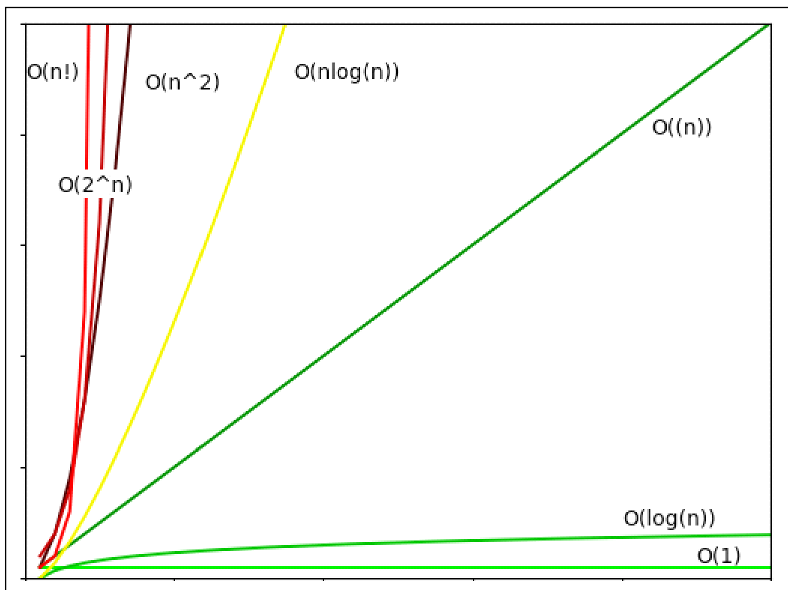
#	Order	Complexity	Example
7	$O(2^n)$	Exponential	Trying to break a password of size *n* using brute force, solving the travelling salesman problem using dynamic programming
8	$O(n!)$	Factorial	Generating all partitions of a set

Table 1: Common Big-O notations for function orders with respect to input size "n"

When implementing a routine or algorithm accepting an input of a certain size *n*, the programmer ideally should aim for implementing it in an order that falls in the first five. Anything which is of the order of $O(n)$ *or* $O(n* log(n))$ or lesser indicates reasonable to good performance.

Algorithms with an order of $O(n^2)$ can usually be optimized to work at a lower order. We will see some examples of this in the sections in the following diagram.

The following diagram shows how each of these orders grow with respect to *n*:



Graph of growth rate of each order of complexity (y axis) w.r.t input size (x axis)

Measuring performance

Now that we've had an overview of what performance complexity is and also of performance testing and measurement tools, let us take an actual look at the various ways of measuring performance complexity with Python.

One of the simplest time measurements can be done by using the `time` command of a POSIX/Linux system.

This is done by using the following command line:

```
$ time <command>
```

For example, here is a screenshot of the time it takes to fetch a very popular page from the web:

```
anand@ubuntu-pro-book: /home/user/programs
File  Edit  View  Search  Terminal  Help
$ time wget http://www.google.com -O /dev/null
--2016-12-28 16:59:27--  http://www.google.com/
Resolving www.google.com (www.google.com)... 216.58.197.36, 2404:6800:4007:807::2004
Connecting to www.google.com (www.google.com)|216.58.197.36|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.google.co.in/?gfe_rd=cr&ei=F6JjWITaLeeK8Qfvj7OQBg [following]
--2016-12-28 16:59:27--  http://www.google.co.in/?gfe_rd=cr&ei=F6JjWITaLeeK8Qfvj7OQBg
Resolving www.google.co.in (www.google.co.in)... 216.58.197.35, 2404:6800:4007:807::2003
Connecting to www.google.co.in (www.google.co.in)|216.58.197.35|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: '/dev/null'

/dev/null                 [ <=>                       ]  12.14K  --.-KB/s    in 0.004s

2016-12-28 16:59:27 (2.80 MB/s) - '/dev/null' saved [12429]


real    0m0.121s
user    0m0.004s
sys     0m0.008s
$
```

Output of the time command on fetching a web page from the internet via wget

See that it shows three classes of time output, namely `real`, `user`, and `sys`. It is important to know the distinction between these three so let us look at them briefly:

- `real`: Real time is the actual wall-clock time that elapsed for the operation. This is the time of the operation from start to finish. It will include any time the process sleeps or spends blocked—such as time taken for I/O to complete.
- `User`: User time is the amount of actual CPU time spent within the process in user mode (outside the kernel). Any sleep time or time spent in waiting such as I/O doesn't add to the user time.
- `Sys`: System time is the amount of CPU time spent on executing system calls within the kernel for the program. This counts only those functions that execute in kernel space such as privileged system calls. It doesn't count any system calls that execute in user space (which is counted in `User`).

The total CPU time spent by a process is `user` + `sys` time. The real or wall-clock time is the time mostly measured by simple time counters.

Measuring time using a context manager

In Python, it is not very difficult to write a simple function that serves as a context manager for blocks of code whose execution time you want to measure.

But first we need a program whose performance we can measure.

Take a look at the following steps to learn how to use a context manager for measuring time:

1. Let us write a program that calculates the common elements between two sequences as a test program. Here is the code:

```
def common_items(seq1, seq2):
    """ Find common items between two sequences """

    common = []
    for item in seq1:
        if item in seq2:
            common.append(item)

    return common
```

2. Let us write a simple context-manager timer to time this code. For timing we will use `perf_counter` of the `time` module, which gives the time to the most precise resolution for short durations:

```
from time import perf_counter as timer_func
from contextlib import contextmanager

@contextmanager
def timer():
    """ A simple timing function for routines """

    try:
        start = timer_func()
        yield
    except Exception as e:
        print(e)
        raise
    finally:
        end = timer_func()
        print ('Time spent=>',1000.0*(end - start),'ms.')
```

3. Let us time the function for some simple input data. For this a `test` function is useful that generates random data, given an input size:

```
def test(n):
    """ Generate test data for numerical lists given input size
    """

    a1=random.sample(range(0, 2*n), n)
    a2=random.sample(range(0, 2*n), n)

    return a1, a2
```

Here is the output of the `timer` method on the `test` function on the Python interactive interpreter:

```
>>> with timer() as t:
... common = common_items(*test(100))
... Time spent=> 2.0268699999999864 ms.
```

4. In fact both test data generation and testing can be combined in the same function to make it easy to test and generate data for a range of input sizes:

```
def test(n, func):
    """ Generate test data and perform test on a given function
    """

    a1=random.sample(range(0, 2*n), n)
    a2=random.sample(range(0, 2*n), n)

    with timer() as t:
        result = func(a1, a2)
```

5. Now let us measure the time taken for different ranges of input sizes in the Python interactive console:

```
>>> test(100, common_items)
    Time spent=> 0.6799279999999963 ms.
>>> test(200, common_items)
    Time spent=> 2.7455590000000085 ms.
>>> test(400, common_items)
    Time spent=> 11.440810000000024 ms.
>>> test(500, common_items)
    Time spent=> 16.83928100000001 ms.
>>> test(800, common_items)
    Time spent=> 21.15130400000004 ms.
>>> test(1000, common_items)
    Time spent=> 13.200749999999983 ms.
```

Oops, the time spent for `1000` items is less than that for `800`! How's that possible? Let's try again:

```
>>> test(800, common_items)
    Time spent=> 8.328282999999992 ms.
>>> test(1000, common_items)
    Time spent=> 34.85899500000001 ms.
```

Now the time spent for `800` items seems to be lesser than that for `400` and `500`. And time spent for `1000` items has increased to more than twice what it was before.

The reason is that our input data is random, which means it will sometimes have a lot of common items—which takes more time—and sometimes have much fewer. Hence on subsequent calls the time taken can show a range of values.

In other words, our timing function is useful to get a rough picture, but not very useful when it comes to getting the true statistical measure of time taken for program execution, which is more important.

6. For this we need to run the timer many times and take an average. This is somewhat similar to the **amortized** analysis of algorithms, which takes into account both the lower end and upper end of the time taken for executing algorithms and gives the programmer a realistic estimate of the average time spent.

Python comes with such a module, which helps to perform such timing analysis, in its standard library, namely the `timeit` module. Let us look at this module in the next section.

Timing code using the timeit module

The `timeit` module in the Python standard library allows the programmer to measure the time taken to execute small code snippets. The code snippets can be a Python statement, an expression, or a function.

The simplest way to use the `timeit` module is to execute it as a module in the Python command line.

For example, here is timing data for some simple Python inline code measuring the performance of a list comprehension calculating squares of numbers in a range:

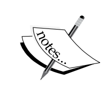
```
$ python3 -m timeit '[x*x for x in range(100)]'
100000 loops, best of 3: 5.5 usec per loop

$ python3 -m timeit '[x*x for x in range(1000)]'
10000 loops, best of 3: 56.5 usec per loop

$ python3 -m timeit '[x*x for x in range(10000)]'
1000 loops, best of 3: 623 usec per loop
```

The result shows the time taken for execution of the code snippet. When run on the command line, the `timeit` module automatically determines the number of cycles to run the code and also calculates the average time spent in a single execution.

The results show that the statement we are executing is linear or O(n) as a range of size 100 takes 5.5 usec and that of 1,000 takes 56.5 usec or about 10 times its time. A usec—or microsecond—is 1 millionth of a second or 1*10-6 seconds.

Here is how to use the `timeit` module on the Python interpreter in a similar manner:

```
>>> 1000000.0*timeit.timeit('[x*x for x in range(100)]',
number=100000)/100000.0
6.007622049946804

>>> 1000000.0*timeit.timeit('[x*x for x in range(1000)]',
number=10000)/10000.0
58.761584300373215
```



Observe that when used in this way, the programmer has to pass the correct number of iterations as the `number` argument and, to average, has to divide by the same number. The multiplication by `1000000` is to convert the time to microseconds (usec).

The `timeit` module uses a `Timer` class behind the scenes. The class can be made use of directly as well as for finer control.

When using this class, `timeit` becomes a method of the instance of the class to which the number of cycles is passed as an argument.

The `Timer` class constructor also accepts an optional `setup` argument, which sets up the code for the `Timer` class. This can contain statements for importing the module that contains the function, setting up globals, and so on. It accepts multiple statements separated by semi-colons.

Measuring the performance of our code using timeit

Let us rewrite our `test` function to test the common items between two sequences. Now that we are going to use the `timeit` module, we can remove the context manager timer from the code. We will also hardcode the call to `common_items` in the function.

> We also need to create the random input outside the test function otherwise the time taken for it will add to the test function's time and corrupt our results.
>
> Hence we need to move the variables out as globals in the module and write a `setup` function, which will generate the data for us as a first step.

Our rewritten `test` function looks like this:

```
def test():
    """ Testing the common_items function """

    common = common_items(a1, a2)
```

The `setup` function with the global variables looks like this:

```
# Global lists for storing test data
a1, a2 = [], []

def setup(n):
    """ Setup data for test function """

    global a1, a2
    a1=random.sample(range(0, 2*n), n)
    a2=random.sample(range(0, 2*n), n)
```

Let's assume the module containing both the `test` and `common_items` functions is named `common_items.py`.

The timer test can now be run as follows:

```
>>> t=timeit.Timer('test()', 'from common_items import test,setup;
setup(100)')
>>> 1000000.0*t.timeit(number=10000)/10000
116.58759460115107
```

So the time taken for a range of `100` numbers is around 117 usec (0.12 microseconds) on average.

Executing it now for a few other ranges of input sizes gives the following output:

```
>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(200)')
>>> 1000000.0*t.timeit(number=10000)/10000
```

```
482.8089299000567

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(400)')
>>> 1000000.0*t.timeit(number=10000)/10000
1919.577144399227

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(800)')
>>> 1000000.0*t.timeit(number=1000)/1000
7822.607815993251

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(1000)')
>>> 1000000.0*t.timeit(number=1000)/1000
12394.932234004957
```

So the maximum time taken for this test run is 12.4 microseconds for an input size of `1000` items.

Finding out time complexity – graphs

Is it possible to find out from these results what the time-performance complexity of our function is? Let us try plotting it in a graph and see the results.

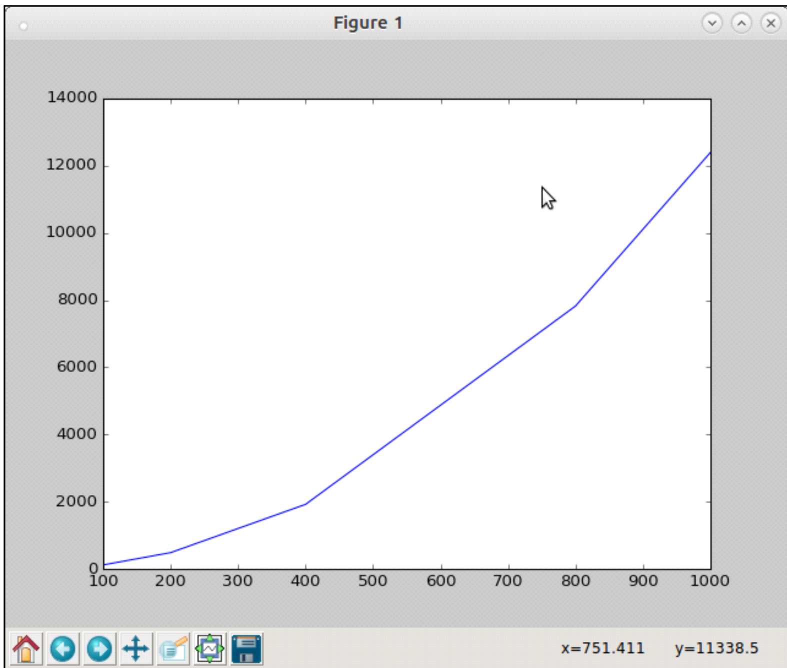
The `matplotlib` library is very useful in plotting graphs in Python for any type of input data. We just need the following simple piece of code for this to work:

```
import matplotlib.pyplot as plt

def plot(xdata, ydata):
    """ Plot a range of ydata (on y-axis) against xdata (on x-axis)
    """

    plt.plot(xdata, ydata)
    plt.show()
```

The preceding code gives you the following output:

```
This is our x data.
>>> xdata = [100, 200, 400, 800, 1000]
This is the corresponding y data.
>>> ydata = [117,483,1920,7823,12395]
>>> plot(xdata, ydata)
```

Take a look at the following graph:



Plot of the input range versus time taken for the common_items function

This is clearly not linear, yet of course not quadratic (in comparison with the figure on Big-O notations). Let us try and plot a graph of O(n*log(n)) superimposed on the current plot to see if there's a match.

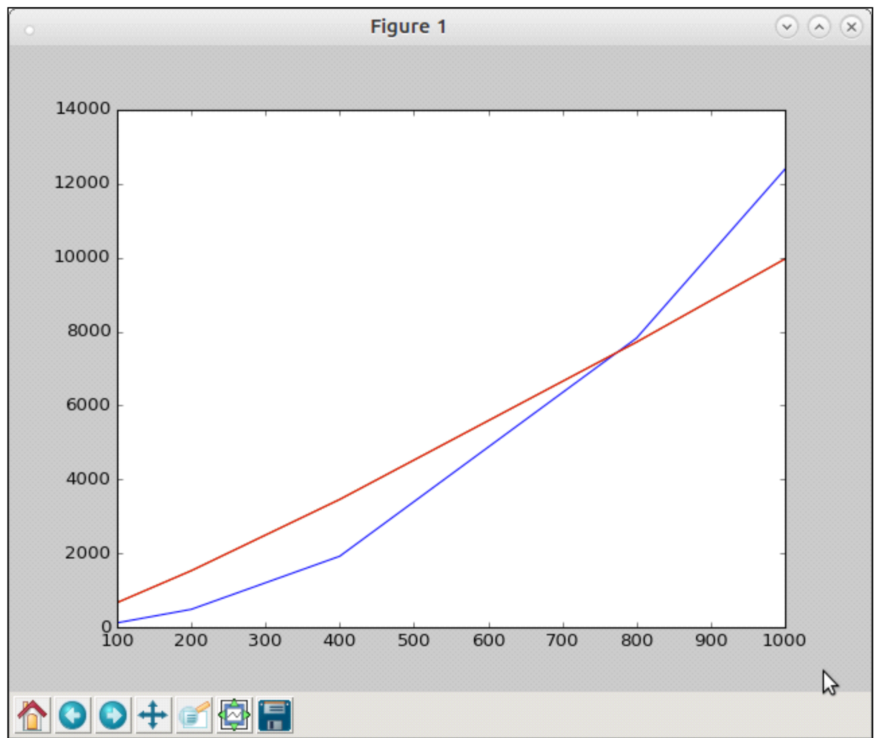
Since we now need two series of `ydata`, we need another slightly modified function:

```
def plot_many(xdata, ydatas):
    """ Plot a sequence of ydatas (on y-axis) against xdata
    (on x-axis) """

    for ydata in ydatas:
        plt.plot(xdata, ydata)
    plt.show()
```

The preceding code gives you the following output:

```
>>> ydata2=map(lambda x: x*math.log(x, 2), input)

>>> plot_many(xdata, [ydata2, ydata])
```

You get the following graph:



Plot of time complexity of common_items superimposed on the plot of y = x*log(x)

The superimposed plot shows that the function is a close match for the n*log(n) order, if not exactly the same. So our current implementation's complexity seems to be roughly O(n*log(n)).

Now that we've done the performance analysis, let us see if we can rewrite our routine to perform better.

Here is the current code:

```
def common_items(seq1, seq2):
    """ Find common items between two sequences """

    common = []
    for item in seq1:
        if item in seq2:
            common.append(item)

    return common
```

The routine first does a pass over an outer `for` loop (of size `n`) and does a check in a sequence (also of size `n`) for the item. Now the second search is also of time complexity `n` on average.

However, some items would be found immediately and some items would take linear time (k) where $1<k<n$. On average, the distribution would be somewhere in between, which is why the code has an average complexity approximating O(n*log(n)).

A quick analysis will tell you that the inner search can be avoided by converting the outer sequence to a dictionary, setting values to 1. The inner search will be replaced with a loop on the second sequence that increments values by 1.

In the end, all common items will have a value greater than 1 in the new dictionary.

The new code is as follows:

```
def common_items(seq1, seq2):
    """ Find common items between two sequences, version 2.0 """

    seq_dict1 = {item:1 for item in seq1}

    for item in seq2:
        try:
            seq_dict1[item] += 1
        except KeyError:
            pass

    # Common items will have value > 1
    return [item[0] for item in seq_dict1.items() if item[1]>1]
```

With this change, the timer gives the following updated results:

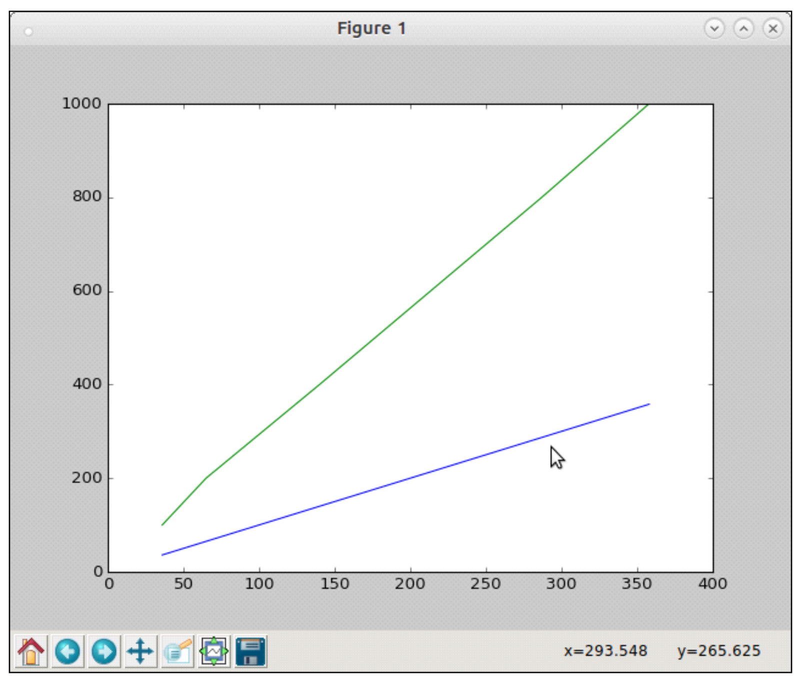
```
>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(100)')
>>> 1000000.0*t.timeit(number=10000)/10000
35.777671200048644

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(200)')
>>> 1000000.0*t.timeit(number=10000)/10000
65.20369809877593

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(400)')
>>> 1000000.0*t.timeit(number=10000)/10000
```

```
139.67061050061602

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(800)')
>>> 1000000.0*t.timeit(number=10000)/10000
287.0645995993982

>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(1000)')
>>> 1000000.0*t.timeit(number=10000)/10000
357.764518300246
```

Let us plot this and superimpose it on an O(n) graph:

```
>>> input=[100,200,400,800,1000]
>>> ydata=[36,65,140,287,358]

# Note that ydata2 is same as input as we are superimposing with y = x
# graph
>>> ydata2=input
>>> plot.plot_many(xdata, [ydata, ydata2])
```

Let's take a look at the following graph:



Plot of time taken by common_items function (v2) against y = x graph

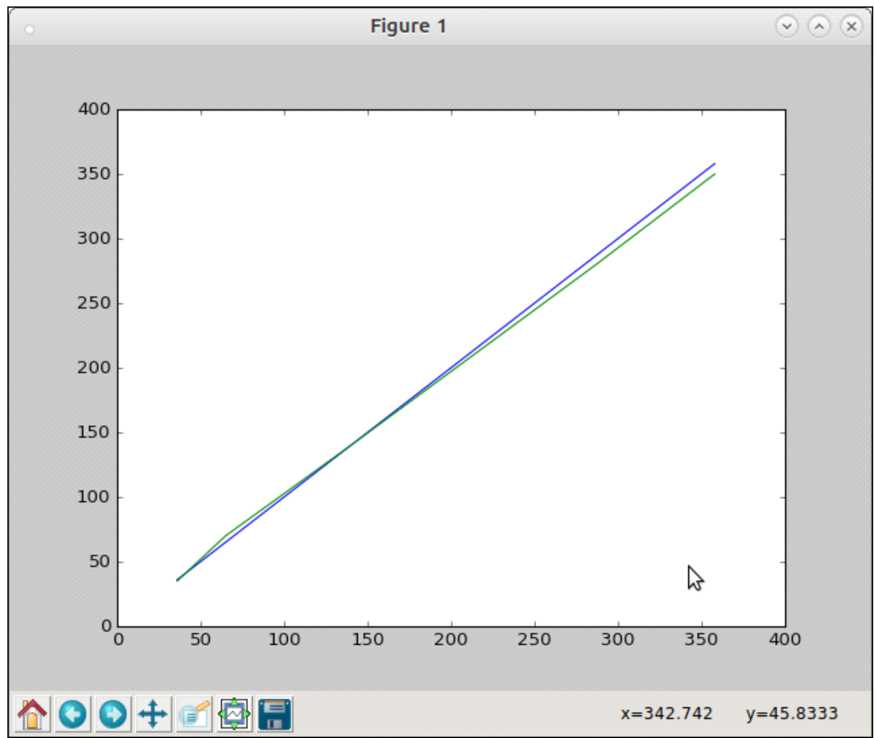
The upper green line is the reference **y = x** graph and the lower blue line is the plot of the time taken by our new function. It is pretty obvious that the time complexity is now linear or O(n).

However, there seems to be a constant factor here as the slopes of the two lines are different. From a quick calculation one can compute this factor as roughly `0.35`.

After applying this change, you will get the following output:

```
>>> input=[100,200,400,800,1000]
>>> ydata=[36,65,140,287,358]

# Adjust ydata2 with the constant factor
>>> ydata2=map(lambda x: 0.35*x, input)
>>> plot.plot_many(xdata, [ydata, ydata2])
```

The output can be seen in the following graph as follows:



Plot of time taken by common_items function (v2) against y = 0.35*x graph

You can see that the plots pretty much superimpose on each other. So our function is now performing at O(c*n) where c ~= 0.35.

Another implementation of the `common_items` function is to convert both sequences to sets and return their intersection. It would be an interesting exercise for the reader to make this change, time it, and plot the graphs to determine the time complexity.

Measuring CPU time with timeit

The `Timer` module by default uses the `perf_counter` function of the `time` module as the default `timer` function. As mentioned earlier, this function returns the wall-clock time spent to the maximum precision for small time durations, hence it will include any sleep time, time spent for I/O, and so on.

This can be made clear by adding a little sleep time to our test function as follows:

```
def test():
    """ Testing the common_items function using a given input size """

    sleep(0.01)
    common = common_items(a1, a2)
```

The preceding code will give you the following output:

```
>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(100)')
>>> 1000000.0*t.timeit(number=100)/100
10545.260819926625
```

The time jumped by as much as 300 times since we are sleeping `0.01` seconds (10 milliseconds) upon every invocation, so the actual time spent on the code is now determined almost completely by the sleep time as the result shows `10545.260819926625` microseconds (or about 10 milliseconds).

Sometimes you may have such sleep times and other blocking or wait times but you want to measure only the actual CPU time taken by the function. To use this, the `Timer` object can be created using the `process_time` function of the `time` module as the `timer` function.

This can be done by passing in a `timer` argument when you create the `Timer` object:

```
>>> from time import process_time
>>> t=timeit.Timer('test()','from common_items import
test,setup;setup(100)', timer=process_time)
>>> 1000000.0*t.timeit(number=100)/100
345.22438
```

If you now increase the sleep time by a factor of, say, 10, the testing time increases by that factor, but the return value of the timer remains the same.

For example, here is the result when sleeping for 1 second. The output comes after about 100 seconds (since we are iterating `100` times), but notice that the return value (time spent per invocation) doesn't change:

```
>>> t=timeit.Timer('test()','from common_items import
test,setup;setup(100)', timer=process_time)
>>> 1000000.0*t.timeit(number=100)/100
369.8039100000002
```

Let us move on to profiling next.

Profiling

In this section, we will discuss profilers and take a deep look at the modules in the Python standard library, which provides support for deterministic profiling. We will also look at third-party libraries that provide support for profiling such as `line_profiler` and `memory_profiler`.

Deterministic profiling

Deterministic profiling means that all function calls, function returns, and exception events are monitored, and precise timings are made for the intervals between these events. Another type of profiling, namely **statistical profiling**, randomly samples the instruction pointer and deduces where time is being spent—but this may not be very accurate.

Python, being an interpreted language, already has a certain overhead in terms of metadata kept by the interpreter. Most deterministic profiling tools make use of this information and hence only add very little extra processing overhead for most applications. Hence deterministic profiling in Python is not a very expensive operation.

Profiling with cProfile and profile

The `profile` and `cProfile` modules provide support for deterministic profiling in the Python standard library. The `profile` module is purely written in Python. The `cProfile` module is a C extension that mimics the interface of the `profile` module but adds lesser overhead to it when compared to `profile`.

Both modules report statistics that are converted into reportable results using the `pstats` module.

We will use the following code, which is a prime number iterator, in order to show our examples using the `profile` modules:

```
class Prime(object):
    """ A prime number iterator for first 'n' primes """

    def __init__(self, n):
        self.n = n
        self.count = 0
        self.value = 0

    def __iter__(self):
        return self

    def __next__(self):
        """ Return next item in iterator """

        if self.count == self.n:
            raise StopIteration("end of iteration")
        return self.compute()

    def is_prime(self):
        """ Whether current value is prime ? """

        vroot = int(self.value ** 0.5) + 1
        for i in range(3, vroot):
            if self.value % i == 0:
                return False
        return True

    def compute(self):
        """ Compute next prime """

        # Second time, reset value
        if self.count == 1:
            self.value = 1

        while True:
            self.value += 2

            if self.is_prime():
```

```
                    self.count += 1
                    break

            return self.value
```

The prime number iterator generates the first `n` prime numbers given the value of `n`:

```
>>> for p in Prime(5):
... print(p)
...
2
3
5
7
11
```

To profile this code, we just need to pass the code to be executed as a string to the `run` method of `profile` or `cProfile` module. In the following examples, we will be using the `cProfile` module:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
>>> cProfile.run("list(primes.Prime(100))")
         477 function calls in 0.004 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.004    0.004 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 primes.py:25(__init__)
        1    0.000    0.000    0.000    0.000 primes.py:30(__iter__)
      101    0.000    0.000    0.003    0.000 primes.py:33(__next__)
      271    0.003    0.000    0.003    0.000 primes.py:40(is_prime)
      100    0.001    0.000    0.003    0.000 primes.py:49(compute)
        1    0.000    0.000    0.004    0.004 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


>>> 
```

Profiling output of the prime iterator function for the first 100 primes

See how the profiler reports its output. The output is ordered into six columns as follows:

- `ncalls`: The number of calls per function
- `tottime`: The total time spent in the call
- `percall`: The `percall` time (quotient of `tottime`/`ncalls`)
- `cumtime`: The cumulative time in this function plus any child function
- `percall`: Another `percall` column (the quotient of `cumtime`/number of primitive calls)
- `filename: lineno(function)`: The filename and line number of the function call

In this case, our function took `4` microseconds to complete with most of that time (`3` microseconds) being spent inside the `is_prime` method, which also dominates the number of calls at 271.

Here are the outputs of the profiler at `n = 1000` and `10000` respectively:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
>>> cProfile.run("list(primes.Prime(1000))")
         5966 function calls in 0.043 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.043    0.043 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 primes.py:25(__init__)
        1    0.000    0.000    0.000    0.000 primes.py:30(__iter__)
     1001    0.001    0.000    0.042    0.000 primes.py:33(__next__)
     3960    0.035    0.000    0.035    0.000 primes.py:40(is_prime)
     1000    0.005    0.000    0.040    0.000 primes.py:49(compute)
        1    0.000    0.000    0.043    0.043 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


>>> 
```

Profiling output of the prime iterator function for the first 1,000 primes

Take a look at the following additional output:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
>>> cProfile.run("list(primes.Prime(10000))")
         72371 function calls in 0.458 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.006    0.006    0.458    0.458 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 primes.py:25(__init__)
        1    0.000    0.000    0.000    0.000 primes.py:30(__iter__)
    10001    0.006    0.000    0.452    0.000 primes.py:33(__next__)
    52365    0.417    0.000    0.417    0.000 primes.py:40(is_prime)
    10000    0.028    0.000    0.445    0.000 primes.py:49(compute)
        1    0.000    0.000    0.458    0.458 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


>>> 
```

Profiling output of the Prime iterator function for the first 10,000 primes

As you can see, at `n=1000` it took about `0.043` seconds (43 microseconds) and at `n=10000` it took `0.458` seconds (458 microseconds). Our `Prime` iterator seems to be performing at an order close to O(n).

As usual, most of that time is spent in `is_primes`. Is there a way to reduce that time?

At this point, let us analyze the code.

Prime number iterator class – performance tweaks

A quick analysis of the code tells us that inside `is_prime` we are dividing the value by every number in the range from `3` to the successor of the square root of the value.

This contains many even numbers as well—we are doing unnecessary computation, which we can avoid by dividing only by the odd numbers.

The modified `is_prime` method is as follows:

```
    def is_prime(self):
        """ Whether current value is prime ? """

        vroot = int(self.value ** 0.5) + 1
        for i in range(3, vroot, 2):
            if self.value % i == 0:
                return False
        return True
```

With this, the profile for `n=1000` and `n=10000` looks as follows.

The following is the output of the profiler for `n = 1000`:

```
>>> cProfile.run("list(primes.Prime(1000))")
         5966 function calls in 0.038 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.038    0.038 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 primes.py:25(__init__)
        1    0.000    0.000    0.000    0.000 primes.py:30(__iter__)
     1001    0.002    0.000    0.037    0.000 primes.py:33(__next__)
     3960    0.029    0.000    0.029    0.000 primes.py:40(is_prime)
     1000    0.006    0.000    0.035    0.000 primes.py:49(compute)
        1    0.000    0.000    0.038    0.038 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

>>>
```

Profiling output of the Prime iterator function for the first 1,000 primes with tweaked code

The following is the output of the profiler for `n = 10000`:

```
>>> cProfile.run("list(primes.Prime(10000))")
         72371 function calls in 0.232 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.003    0.003    0.232    0.232 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 primes.py:25(__init__)
        1    0.000    0.000    0.000    0.000 primes.py:30(__iter__)
    10001    0.005    0.000    0.228    0.000 primes.py:33(__next__)
    52365    0.202    0.000    0.202    0.000 primes.py:40(is_prime)
    10000    0.022    0.000    0.224    0.000 primes.py:49(compute)
        1    0.000    0.000    0.232    0.232 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

>>>
```

Profiling output of the Prime iterator function for the first 10,000 primes with tweaked code

You can see that, at `1000`, the time has dropped a bit (43 microseconds to 38 microseconds) but at `10000`, there is nearly a 50% drop from 458 microseconds to 232 microseconds. At this point, the function is performing better than O(n).

Profiling – collecting and reporting statistics

The way we used `cProfile` in the example earlier, it ran and reported the statistics directly. Another way to use the module is to pass a `filename` argument to which it writes the statistics, which can later be loaded and interpreted by the `pstats` module.

We modify the code as follows:

```
>>> cProfile.run("list(primes.Prime(100))", filename='prime.stats')
```

By doing this, the stats, instead of getting printed out, are saved to the file named `prime.stats`.

Here is how to parse the statistics using the `pstats` module and print the results ordered by the number of calls:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
>>> pstats.Stats('prime.stats').sort_stats('ncalls').print_stats()
Thu Dec 29 06:34:39 2016    prime.stats

         577 function calls in 0.002 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      271    0.001    0.000    0.001    0.000 /home/user/programs/chap4/primes.py:41(is_prime)
      101    0.000    0.000    0.002    0.000 /home/user/programs/chap4/primes.py:34(__next__)
      100    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
      100    0.000    0.000    0.001    0.000 /home/user/programs/chap4/primes.py:51(compute)
        1    0.000    0.000    0.000    0.000 /home/user/programs/chap4/primes.py:31(__iter__)
        1    0.000    0.000    0.002    0.002 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 /home/user/programs/chap4/primes.py:25(__init__)
        1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats object at 0x7f5b97262da0>
>>>
```

Parsing and printing saved profile results using the `pstats` module

The `pstats` module allows sorting the profile results by a number of headers such as total time (`tottime`), number of primitive calls (`pcalls`), cumulative time (`cumtime`), and so on. You can see from the output of `pstats` again that most of the processing in terms of number of calls are being spent in the `is_prime` method, as we are sorting the output by '`ncalls`' or the number of function calls.

The `Stats` class of the `pstats` module returns a reference to itself after every operation. This is a very useful aspect of some Python classes and allows us to write compact one-line code by chaining method calls.

Another useful method of the `Stats` object is to find out the callee/caller relationship. This can be done by using the `print_callers` method instead of `print_stats`. Here is the output from our current statistics:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
>>> pstats.Stats('prime.stats').sort_stats('pcalls').print_callers()
   Ordered by: primitive call count

Function                                          was called by...
                                                      ncalls  tottime  cumtime
/home/user/programs/chap4/primes.py:41(is_prime)  <-     271    0.001    0.001  /home/user/programs/chap
4/primes.py:51(compute)
/home/user/programs/chap4/primes.py:34(__next__)  <-     101    0.000    0.002  <string>:1(<module>)
{method 'append' of 'list' objects}               <-     100    0.000    0.000  /home/user/programs/chap
4/primes.py:51(compute)
/home/user/programs/chap4/primes.py:51(compute)   <-     100    0.000    0.001  /home/user/programs/chap
4/primes.py:34(__next__)
/home/user/programs/chap4/primes.py:31(__iter__)  <-       1    0.000    0.000  <string>:1(<module>)
<string>:1(<module>)                              <-       1    0.000    0.002  {built-in method builtin
s.exec}
/home/user/programs/chap4/primes.py:25(__init__)  <-       1    0.000    0.000  <string>:1(<module>)
{built-in method builtins.exec}                   <-
{method 'disable' of '_lsprof.Profiler' objects}  <-


<pstats.Stats object at 0x7f5b9907e550>
>>>
```

Printing callee/caller relationships ordered by primitive calls using the pstats module

Third-party profilers

The Python ecosystem comes with a plethora of third-party modules for solving most problems. This is true in the case of profilers as well. In this section, we will take a quick look at a few popular third-party profiler applications contributed by developers in the Python community.

Line profiler

Line profiler is a profiler application developed by Robert Kern for performing line-by-line profiling of Python applications. It is written in Cython, an optimizing static compiler for Python that reduces the overhead of profiling.

Line profiler can be installed via `pip` as follows:

```
$ pip3 install line_profiler
```

As opposed to the profiling modules in Python, which profile functions, line profiler is able to profile code line by line, thus providing more granular statistics.

Line profiler comes with a script called `kernprof.py` that makes it easy to profile code using line profiler. One needs only to decorate the functions that need to be profiled with the `@profile` decorator when using `kernprof`.

For example, we realized that most of the time in our prime number iterator was being spent in the `is_prime` method. However, line profiler allows us to go into more detail and find which lines of those functions take the most time.

To do this, just decorate the method with the `@profile` decorator:

```
@profile
def is_prime(self):
    """ Whether current value is prime ? """

    vroot = int(self.value ** 0.5) + 1
    for i in range(3, vroot, 2):
        if self.value % i == 0:
            return False
    return True
```

Since `kernprof` accepts a script as an argument, we need to add some code to invoke the prime number iterator. To do that, we can append the following at the end of the `primes.py` module:

```
# Invoke the code.
if __name__ == "__main__":
    l=list(Prime(1000))
```

Now, run it with line profiler as follows:

```
$ kernprof -l -v primes.py
```

By passing `-v` to the `kernprof` script, we tell it to display the profile results in addition to saving them.

Here is the output:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
$ kernprof -l -v primes.py
Wrote profile results to primes.py.lprof
Timer unit: 1e-06 s

Total time: 0.04177 s
File: primes.py
Function: is_prime at line 40

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    40                                           @profile
    41                                           def is_prime(self):
    42                                               """ Whether current value is prime ? """
    43
    44      3960         3339      0.8      8.0      vroot = int(self.value ** 0.5) + 1
    45     41579        17141      0.4     41.0      for i in range(3, vroot, 2):
    46     40579        19821      0.5     47.5          if self.value % i == 0:
    47      2960         1072      0.4      2.6              return False
    48      1000          397      0.4      1.0      return True

$
```

Line profiler results from profiling the `is_prime` method using n = 1000

Line profiler tells us that the majority of the time—close to 90% of the total time spent in the method—is spent in the first two lines: the `for` loop and the reminder check.

This tells us that, if ever we want to optimize this method, we need to concentrate on these two aspects.

Memory profiler

Memory profiler is a profiler similar to line profiler in that it profiles Python code line by line. However, instead of profiling the time taken in each line of code, it profiles lines by memory consumption.

Memory profiler can be installed the same way as line profiler:

```
$ pip3 install memory_profiler
```

Once installed, memory for lines can be printed by decorating the function with the `@profile` decorator in a similar way to line profiler.

Here is a simple example:

```
# mem_profile_example.py
@profile
def squares(n):
    return [x*x for x in range(1, n+1)]

squares(1000)
```

Here's how to run this:

```
$ python3 -m memory_profiler mem_profile_example.py
Filename: mem_profile_example.py

Line #    Mem usage    Increment   Line Contents
================================================
     1   31.559 MiB    0.000 MiB   @profile
     2                             def squares(n):
     3   31.559 MiB    0.000 MiB       return [x*x for x in range(1, n+1)]


$
```

Memory profiler profiling a list comprehension of squares of the first 1,000 numbers

Memory profiler shows memory increments line by line. In this case, there is almost no increment for the line containing the number of squares (the list comprehension) as the numbers are rather small. The total memory usage remains what it was at the beginning: about 32 MB.

What happens if we change the value of `n` to 1,000,000? This can be done by rewriting the last line of the code as follows:

```
squares(100000)
```

```
$ python3 -m memory_profiler mem_profile_example.py
Filename: mem_profile_example.py

Line #    Mem usage    Increment   Line Contents
================================================
     1   31.418 MiB    0.000 MiB   @profile
     2                             def squares(n):
     3   70.027 MiB   38.609 MiB       return [x*x for x in range(1, n+1)]


$
```

Memory profiler profiling a list comprehension of squares of the first 1,000,000 numbers

Now you can see that there is a clear memory increment of about 39 MB for the list comprehension calculating the squares, with a total final memory usage of about 70 MB.

To demonstrate the real usefulness of memory profiler, let us look at another example.

This involves finding the strings from a sequence that are subsequences of any of the strings present in another sequence, generally containing larger strings.

Substring (subsequence) problem

Let us say you have a sequence containing the following strings:

```
>>> seq1 = ["capital","wisdom","material","category","wonder"]
```

And say there is another sequence as follows:

```
>>> seq2 = ["cap","mat","go","won","to","man"]
```

The problem is to find the strings in `seq2` that are substrings—as is found anywhere contiguously in any of the strings in `seq1`:

In this case, the answer is as follows:

```
>>> sub=["cap","mat","go","won"]
```

This can be solved using a brute-force search—checking for each string one by one in each of the parent strings as follows:

```
def sub_string_brute(seq1, seq2):
    """ Sub-string by brute force """

    subs = []
    for item in seq2:
        for parent in seq1:
            if item in parent:
                subs.append(item)

    return subs
```

However, a quick analysis will tell you that the time complexity of this function scales rather badly as the size of the sequences increase. Since every step needs iteration through two sequences and then a search in each string in the first sequence, the average performance would be O(n1*n2), where n1, n2 are the sizes of the sequences respectively.

Here are the results of some tests of this function with input sizes (both sequences of the same size) of random strings varying from length 2 to 10:

Input size	Time taken
100	450 usec
1000	52 microseconds
10000	5.4 seconds

Table 2: Input size versus time taken for subsequence solution via brute force

The results indicate the performance is almost exactly $O(n^2)$.

Is there a way to rewrite the function to be more performance-efficient? This approach is captured in the following `sub_string` function:

```
def slices(s, n):
    return map(''.join, zip(*(s[i:] for i in range(n))))

def sub_string(seq1, seq2):
    """ Return sub-strings from seq2 which are part of strings in seq1
    """

    # Create all slices of lengths in a given range
    min_l, max_l = min(map(len, seq2)), max(map(len, seq2))
    sequences = {}

    for i in range(min_l, max_l+1):
        for string in seq1:
         # Create all sub sequences of given length i
         sequences.update({}.fromkeys(slices(string, i)))

    subs = []
    for item in seq2:
        if item in sequences:
            subs.append(item)

    return subs
```

In this approach, we pre-compute all the substrings of a size range from the strings in `seq1` and store it in a dictionary. Then it is a matter of going through the strings in `seq2` and checking if they are in this dictionary and if so adding them to a list.

To optimize the calculation, we only compute strings whose size is in the range of the minimum and maximum length of the strings in `seq2`.

As with almost all solutions to performance issues, this one trades space for time. By pre-computing all the substrings, we are expending more space in memory but this eases the computation time.

The test code looks like this:

```
import random
import string

seq1, seq2 = [], []

def random_strings(n, N):
     """ Create N random strings in range of 4..n and append
     to global sequences seq1, seq2 """

    global seq1, seq2
    for i in range(N):
        seq1.append(''.join(random.sample(string.ascii_lowercase,
                             random.randrange(4, n))))

    for i in range(N):
        seq2.append(''.join(random.sample(string.ascii_lowercase,
                             random.randrange(2, n/2))))

def test(N):
    random_strings(10, N)
    subs=sub_string(seq1, seq2)

def test2():
    # random_strings has to be called before this
    subs=sub_string(seq1, seq2)
```

Here are the timing results of this function using the `timeit` module:

```
>>> t=timeit.Timer('test2()',setup='from sub_string import test2,
random_
strings;random_strings(10, 100)')
>>> 1000000*t.timeit(number=10000)/10000.0
1081.6103347984608
>>> t=timeit.Timer('test2()',setup='from sub_string import test2,
random_
strings;random_strings(10, 1000)')
>>> 1000000*t.timeit(number=1000)/1000.0
11974.320339999394
```

```
>>> t=timeit.Timer('test2()',setup='from sub_string import test2,
random_
strings;random_strings(10, 10000)')
>>> 1000000*t.timeit(number=100)/100.0124718.30968977883
124718.30968977883
>>> t=timeit.Timer('test2()',setup='from sub_string import test2,
random_
strings;random_strings(10, 100000)')
>>> 1000000*t.timeit(number=100)/100.0
1261111.164370086
```

Here are the summarized results for this test:

Input size	Time taken
100	1.08 microseconds
1000	11.97 microseconds
10000	0.12 microseconds
100000	1.26 seconds

Table 3: Input size versus time taken for optimized sub-sequence solution using pre-computed strings

A quick calculation tells us that the algorithm is now performing at O(n). Pretty good!

But this is at the expense of memory in terms of the pre-computed strings. We can get an estimate of this by invoking memory profiler.

Here is the decorated function for doing this:

```
@profile
def sub_string(seq1, seq2):
    """ Return sub-strings from seq2 which are part of strings in seq1
    """

    # Create all slices of lengths in a given range
    min_l, max_l = min(map(len, seq2)), max(map(len, seq2))
    sequences = {}

    for i in range(min_l, max_l+1):
        for string in seq1:
            sequences.update({}.fromkeys(slices(string, i)))

    subs = []
    for item in seq2:
        if item in sequences:
            subs.append(item)
```

The test function would now be as follows:

```
def test(N):
    random_strings(10, N)
    subs = sub_string(seq1, seq2)
```

Let's test this for the sequence of sizes 1,000 and 10,000 respectively.

Here is the result for an input size of 1,000:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
$ python3 -m memory_profiler sub_string.py
Filename: sub_string.py

Line #    Mem usage    Increment   Line Contents
================================================
    24   31.352 MiB    0.000 MiB   @profile
    25                             def sub_string(seq1, seq2):
    26                                 """ Return sub-strings from seq2 which are in seq1 """
    27
    28                                 # E.g: seq1 = ['introduction','discipline','animation']
    29                                 # seq2 = ['in','on','is','mat','ton']
    30                                 # Result = ['in','on','mat','is']
    31
    32                                 # Create all slices of lengths in a given range
    33   31.352 MiB    0.000 MiB       min_l, max_l = min(map(len, seq2)), max(map(len, seq2))
    34   31.352 MiB    0.000 MiB       sequences = {}
    35
    36   32.797 MiB    1.445 MiB       for i in range(min_l, max_l+1):
    37   32.797 MiB    0.000 MiB           for string in seq1:
    38   32.797 MiB    0.000 MiB               sequences.update({}.fromkeys(slices(string, i)))
    39
    40   32.797 MiB    0.000 MiB       subs = []
    41   32.797 MiB    0.000 MiB       for item in seq2:
    42   32.797 MiB    0.000 MiB           if item in sequences:
    43   32.797 MiB    0.000 MiB               subs.append(item)
    44
    45   32.797 MiB    0.000 MiB       return subs
```

Memory profiler results for testing sub-strings of sequences of size 1,000

And here is the result for an input size of 10,000:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
$ python3 -m memory_profiler sub_string.py
Filename: sub_string.py

Line #    Mem usage    Increment   Line Contents
================================================
    24   32.523 MiB    0.000 MiB   @profile
    25                             def sub_string(seq1, seq2):
    26                                 """ Return sub-strings from seq2 which are in seq1 """
    27
    28                                 # E.g: seq1 = ['introduction','discipline','animation']
    29                                 # seq2 = ['in','on','is','mat','ton']
    30                                 # Result = ['in','on','mat','is']
    31
    32                                 # Create all slices of lengths in a given range
    33   32.523 MiB    0.000 MiB       min_l, max_l = min(map(len, seq2)), max(map(len, seq2))
    34   32.523 MiB    0.000 MiB       sequences = {}
    35
    36   38.770 MiB    6.246 MiB       for i in range(min_l, max_l+1):
    37   38.770 MiB    0.000 MiB           for string in seq1:
    38   38.770 MiB    0.000 MiB               sequences.update({}.fromkeys(slices(string, i)))
    39
    40   38.770 MiB    0.000 MiB       subs = []
    41   38.770 MiB    0.000 MiB       for item in seq2:
    42   38.770 MiB    0.000 MiB           if item in sequences:
    43   38.770 MiB    0.000 MiB               subs.append(item)
    44
    45   38.770 MiB    0.000 MiB       return subs
```

Memory profiler results for testing sub-strings of sequences of size 10,000

For the sequence of size of 1,000, the memory usage increased by a paltry 1.4 MB. For the sequence of size 10,000 it increased by 6.2 MB. Clearly, these are not very significant numbers.

So the test with memory profiler makes it clear that our algorithm, while being efficient on time performance, is also memory-efficient.

Other tools

In this section, we will discuss a few more tools that will aid the programmer in debugging memory leaks and also enable them to visualize their objects and their relations.

objgraph

`objgraph` (**object graph**) is a Python object visualization tool that makes use of the `graphviz` package to draw object reference graphs.

It is not a profiling or instrumentation tool but can be used along with such tools to visualize object trees and references in complex programs while hunting for elusive memory leaks. It allows you to find out references to objects to figure out what references are keeping an object alive.

As with almost everything in the Python world, it is installable via `pip`:

```
$ pip3 install objgraph
```

However objgraph is really useful only if it can generate graphs. Hence we need to install the `graphviz` package and the `xdot` tool.

In a Debian/Ubuntu system, you will install this as follows:

```
$ sudo apt install graphviz xdot -y
```

Let's look at a simple example of using `objgraph` to find out hidden references:

```
import objgraph

class MyRefClass(object):
    pass

ref=MyRefClass()
class C(object):pass

c_objects=[]
for i in range(100):
    c=C()
    c.ref=ref
    c_objects.append(c)

import pdb; pdb.set_trace()
```

We have a class named `MyRefClass` with a single instances `ref` that is referred to by 100 instances of the class `C` created in a `for` loop. These are references that may cause memory leaks. Let us see how `objgraph` allows us to identify them.

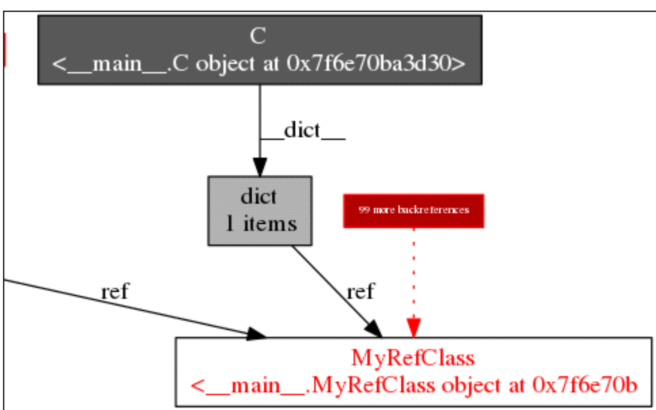
When this piece of code is executed, it stops at the debugger (`pdb`):

```
$ python3 objgraph_example.py
--Return--
[0] > /home/user/programs/chap4/objgraph_example.py(15)<module>()->None
-> import pdb; pdb.set_trace()
```

```
(Pdb++) objgraph.show_backrefs(ref, max_depth=2, too_many=2,
filename='refs.png')
Graph written to /tmp/objgraph-xxhaqwxl.dot (6 nodes)
Image generated as refs.png
```

> The left side of the image has been cropped to show only the relevant part.

Next is the diagram generated by `objgraph`:



objgraph back references visualization for the object ref

The red box in the preceding diagram says **99 more references**, which means that it is showing one instance of class **C** and informing us there are 99 more like it—totaling to 100 instances of C, that refer to the single object **ref**.

In a complex program where we are unable to track object references that cause memory leaks, such reference graphs can be put to good use by the programmer.

Pympler

Pympler is a tool that can be used to monitor and measure the memory usage of objects in a Python application. It works on both Python 2.x and 3.x. It can be installed using `pip` as follows:

```
$ pip3 install pympler
```

The documentation of `pympler` is rather lacking. However, it's well-known use is to track objects and print their actual memory usage via its `asizeof` module.

The following is our `sub_string` function modified to print the memory usage of the sequences dictionary (where it stores all the generated substrings):

```
from pympler import asizeof

def sub_string(seq1, seq2):
    """ Return sub-strings from seq2 which are part of strings in seq1
    """


    # Create all slices of lengths in a given range
    min_l, max_l = min(map(len, seq2)), max(map(len, seq2))
    sequences = {}

    for i in range(min_l, max_l+1):
        for string in seq1:
            sequences.update({}.fromkeys(slices(string, i)))

    subs = []
    for item in seq2:
        if item in sequences:
            subs.append(item)
    print('Memory usage',asizeof.asized(sequences).format())

    return subs
```

When running this for a sequence size of 10,000:

```
$ python3 sub_string.py
Memory usage {'awg': None, 'qlbo': None, 'gvap': No....te':
                           None, 'luwr':
                           None, 'ipat': None}
size=5874384
flat=3145824
```

The memory size of `5870408` bytes (or around 5.6 MB) is in line with what memory profiler reported (around 6 MB)

Pympler also comes with a package called `muppy` which allows us to keep track of all objects in a program. This can be summarized with the `summary` package to print out the summary of memory usage of all objects (classified according to their types) in an application.

Here is a report of our `sub_string` module run with n =10,000. To do this, the execution part has to be modified as follows:

```
if __name__ == "__main__":
    from pympler import summary
    from pympler import muppy
    test(10000)
    all_objects = muppy.get_objects()
    sum1 = summary.summarize(all_objects)
    summary.print_(sum1)
```

The following shows the output that `pympler` summarizes at the end of the program:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File Edit View Search Terminal Help
$ python3 sub_string.py
Memory usage {'gra': None, 'usrq': None, 'slx': Non....gj': None, 'yzfp': None, 'egfa': None} size=588
7488 flat=3145824
                              types |   # objects |   total size
=================================== | =========== | ============
                       <class 'str |       30048 |      2.14 MB
                      <class 'dict |        1540 |      1.19 MB
                      <class 'type |         462 |    458.64 KB
                      <class 'code |        3067 |    431.45 KB
                      <class 'list |         413 |    229.49 KB
                       <class 'set |         363 |    139.91 KB
         <class 'wrapper_descriptor |        1106 |     86.41 KB
                     <class 'tuple |        1266 |     83.86 KB
                   <class 'weakref |         975 |     76.17 KB
  <class 'builtin_function_or_method |         920 |     64.69 KB
          <class 'method_descriptor |         828 |     58.22 KB
                <class 'abc.ABCMeta |          51 |     48.13 KB
                       <class 'int |        1489 |     41.66 KB
          <class 'getset_descriptor |         504 |     35.44 KB
                 <class 'frozenset |          42 |     27.94 KB
$
```

Summary of memory usage classified by object type by pympler

Programming for performance – data structures

We've looked at the definition of performance, measuring performance complexity, and the different tools for measuring program performance. We've also gained insights by profiling code for statistics, memory usage, and so on.

We also saw a couple of examples of program optimization to improve the time performance of the code.

In this section, we will take a look at common Python data structures and discuss what their best and worst performance scenarios are and also discuss some situations of where they are an ideal fit and where they may not be the best choice.

Mutable containers – lists, dictionaries, and sets

Lists, dictionaries, and sets are the most popular and useful mutable containers in Python.

Lists are appropriate for object access via a known index. Dictionaries provide a near-constant time lookup for objects with known keys. Sets are useful to keep groups of items while dropping duplicates and finding their difference, intersection, union, and so on in near-linear time.

Let us look at each of these in turn.

Lists

Lists provide a near constant time O(1) order for the following operations:

- `get(index)` via the `[]` operator
- The `append(item)` via the `.append` method

However, lists perform badly `(O(n))` in the following cases:

- Seeking an item via the `in` operator
- Inserting at an index via the `.insert` method

A list is ideal in the following cases:

- If you need a mutable store to keep different types or classes of items (heterogeneous).
- If your search of objects involves getting the item by a known index.
- If you don't have a lot of lookups via searching the list (**item in list**).
- If any of your elements are non-hashable. Dictionaries and sets require their entries to be hashable. So in this case, you almost default to using a list.

If you have a huge list—of, say, more than 100,000 items—and you keep finding that you search it for elements via the `in` operator, you should replace it with a dictionary.

Similarly, if you find that you keep inserting to a list instead of appending to it most of the time, you can think of replacing the list with `deque` from the `collections` module.

Dictionaries

Dictionaries provide a constant time order for:

- Setting an item via a key
- Getting an item via a key
- Deleting an item via a key

However, dictionaries take slightly more memory than lists for the same data. A dictionary is useful in the following situations:

- You don't care about the insertion order of the elements
- You don't have duplicate elements in terms of keys

A dictionary is also ideal where you load a lot of data uniquely indexed by keys from a source (database or disk) in the beginning of the application and need quick access to them—in other words, a lot of random reads as against fewer writes or updates.

Sets

The usage scenario of sets lies somewhere between lists and dictionaries. Sets are in implementation closer to dictionaries in Python—since they are unordered, don't support duplicate elements, and provide near O(1) time access to items via keys. They are kind of similar to lists in that they support the pop operation (even if they don't allow index access!).

Sets are usually used in Python as intermediate data structures for processing other containers—for operations such as dropping duplicates, finding common items across two containers, and so on.

Since the order of set operations is exactly the same as that of a dictionary, you can use them for most cases where a dictionary needs to be used, except that no value is associated to the key.

Examples include:

- Keeping heterogeneous, unordered data from another collection while dropping duplicates
- Processing intermediate data in an application for a specific purpose—such as finding common elements, combining unique elements across multiple containers, dropping duplicates, and so on

Immutable containers – tuples

Tuples are an immutable version of lists in Python. Since they are unchangeable after creation, they don't support any of the methods of list modification such as insert, append, and so on.

Tuples have the same time complexity as when using the index and search (via **item in tuple**) as lists. However, they take much less memory overhead when compared to lists; the interpreter optimizes them more as they are immutable.

Hence tuples can be used whenever there are use cases for reading, returning, or creating a container of data that is not going to be changed but requires iteration. Some examples are as follows:

- Row-wise data loaded from a data store that is going to have read-only access. For example, results from a DB query, processed rows from reading a CSV file, and so on.
- A constant set of values that needs iteration over and over again. For example, a list of configuration parameters loaded from a configuration file.
- When returning more than one value from a function. In this case, unless one explicitly returns a list, Python always returns a tuple by default.
- When a mutable container needs to be a dictionary key. For example, when a list or set needs to be associated to a value as a dictionary key, the quick way is to convert it to a tuple.

High performance containers – the collections module

The collection module supplies high performance alternatives to the built-in default container types in Python, namely `list`, `set`, `dict`, and `tuple`.

We will briefly look at the following container types in the collections module:

- `deque`: An alternative to a list container supporting fast insertions and pops at either ends
- `defaultdict`: A sub-class of `dict` that provides factory functions for types to provide missing values
- `OrderedDict`: A sub-class of `dict` that remembers the order of insertion of keys
- `Counter`: A `dict` sub-class for keeping count and statistics of hashable types

- `ChainMap`: A class with a dictionary-like interface for keeping track of multiple mappings
- `namedtuple`: A type for creating tuple-like classes with named fields

deque

A deque or *double ended queue* is like a list but supports nearly constant (O(1)) time appends and pops from either side as opposed to a list, which has an O(n) cost for pops and inserts at the left.

Deques also support operations such as rotation for moving `k` elements from back to front and reverse with an average performance of O(k). This is often slightly faster than the similar operation in lists, which involves slicing and appending:

```
def rotate_seq1(seq1, n):
    """ Rotate a list left by n """
    # E.g: rotate([1,2,3,4,5], 2) => [4,5,1,2,3]

    k = len(seq1) - n
    return seq1[k:] + seq1[:k]

def rotate_seq2(seq1, n):
    """ Rotate a list left by n using deque """

    d = deque(seq1)
    d.rotate(n)
    return d
```

By a simple `timeit` measurement, you should find that deques have a slight performance edge over lists (about 10-15%), in the above example.

defaultdict

Default dicts are `dict` sub-classes that use type factories to provide default values to dictionary keys.

A common problem one encounters in Python when looping over a list of items and trying to increment a dictionary count is that there may not be any existing entry for the item.

For example, if one is trying to count the number of occurrences of a word in a piece of text:

```
counts = {}
for word in text.split():
    word = word.lower().strip()
    try:
        counts[word] += 1
    except KeyError:
        counts[word] = 1
```

We are forced to write code like the preceding or a variation of it.

Another example is when grouping objects according to a key using a specific condition, for example, trying to group all strings with the same length to a dictionary:

```
cities = ['Jakarta','Delhi','Newyork','Bonn','Kolkata','Bangalore','S
eoul']
cities_len = {}
for city in cities:
  clen = len(city)
  # First create entry
  if clen not in cities_len:
    cities_len[clen] = []
  cities_len[clen].append(city)
```

A `defaultdict` container solves these problems elegantly by defining a type factory to supply the default argument for any key that is not yet present in the dictionary. The default factory type supports any of the default types and defaults to `None`.

For each type, its empty value is the default value. This means:

```
0 → default value for integers
[] → default value for lists
'' → default value for strings
{} → default value for dictionaries
```

The word-count code can then be rewritten as follows:

```
counts = defautldict(int)
for word in text.split():
    word = word.lower().strip()
    # Value is set to 0 and incremented by 1 in one go
    counts[word] += 1
```

Similarly, for the code which groups strings by their length we can write this:

```
cities = ['Jakarta','Delhi','Newyork','Bonn','Kolkata','Bangalore','S
eoul']
cities_len = defaultdict(list)
for city in cities:
    # Empty list is created as value and appended to in one go
    cities_len[len(city)].append(city)
```

OrderedDict

`OrderedDict` is a sub-class of dict that remembers the order of the insertion of entries. It kind of behaves as a dictionary and list hybrid. It behaves like a mapping type but also has list-like behavior in remembering the insertion order plus supporting methods such as `popitem` to remove the last or first entry.

Here is an example:

```
>>> cities = ['Jakarta','Delhi','Newyork','Bonn','Kolkata',
'Bangalore','Seoul']
>>> cities_dict = dict.fromkeys(cities)
>>> cities_dict
{'Kolkata': None, 'Newyork': None, 'Seoul': None, 'Jakarta': None,
'Delhi': None, 'Bonn': None, 'Bangalore': None}

# Ordered dictionary
>>> cities_odict = OrderedDict.fromkeys(cities)
>>> cities_odict
OrderedDict([('Jakarta', None), ('Delhi', None), ('Newyork', None),
('Bonn', None), ('Kolkata', None), ('Bangalore', None), ('Seoul',
None)])
>>> cities_odict.popitem()
('Seoul', None)
>>> cities_odict.popitem(last=False)
('Jakarta', None)
```

You can compare and contrast how the dictionary changes the order around and how the `OrderedDict` container keeps the original order.

This allows for a few recipes using the `OrderedDict` container.

Dropping duplicates from a container without losing the order

Let us modify the cities list to include duplicates:

```
>>> cities = ['Jakarta','Delhi','Newyork','Bonn','Kolkata',
              'Bangalore','Bonn','Seoul','Delhi','Jakarta','Mumbai']
>>> cities_odict = OrderedDict.fromkeys(cities)
>>> print(cities_odict.keys())
odict_keys(['Jakarta', 'Delhi', 'Newyork', 'Bonn', 'Kolkata',
            'Bangalore', 'Seoul', 'Mumbai'])
```

See how the duplicates are dropped but the order is preserved.

Implementing a Least Recently Used (LRU) cache dictionary

An LRU cache gives preference to entries that are recently used (accessed) and drops those entries that are least used. This is a common caching algorithm used in HTTP caching servers such as Squid and in places where one needs to keep a limited size container that keeps recently accessed items preferentially over others.

Here we make use of the behavior of `OrderedDict`: when an existing key is removed and re-added, it is added at the end (the right side):

```
class LRU(OrderedDict):
    """ Least recently used cache dictionary """

    def __init__(self, size=10):
        self.size = size

    def set(self, key):
        # If key is there delete and reinsert so
        # it moves to end.
        if key in self:
            del self[key]

        self[key] = 1
        if len(self)>self.size:
            # Pop from left
            self.popitem(last=False)
```

Here is a demonstration:

```
>>> d=LRU(size=5)
>>> d.set('bangalore')
>>> d.set('chennai')
>>> d.set('mumbai')
>>> d.set('bangalore')
>>> d.set('kolkata')
>>> d.set('delhi')
>>> d.set('chennai')

>>> len(d)
5
>>> d.set('kochi')
>>> d
LRU([('bangalore', 1), ('chennai', 1), ('kolkata', 1), ('delhi', 1),
('kochi', 1)])
```

Since a key `mumbai` was set first and never set again, it became the leftmost one and got dropped off.

Notice how the next candidate to drop off is `bangalore`, followed by `chennai`. This is because `chennai` was set once more after `bangalore` was set.

Counter

A counter is a subclass of a dictionary to keep a count of hashable objects. Elements are stored as dictionary keys and their counts get stored as the values. The `Counter` class is a parallel for multisets in languages such as C++ or Bag in languages such as Smalltalk.

A counter is a natural choice for keeping the frequency of items encountered when processing any container. For example, a counter can be used to keep the frequency of words when parsing text or the frequency of characters when parsing words.

For example, both of the following code snippets perform the same operation but the counter one is less verbose and compact.

They both return the most common 10 words from the text of the famous Sherlock Holmes Novel, *The Hound of Baskervilles* from its Gutenberg version online:

- Using the `defaultdict` container in the following code:

```
import requests, operator
    text=requests.get('https://www.gutenberg.org/
files/2852/2852-0.txt').text
    freq=defaultdict(int)
    for word in text.split():
        if len(word.strip())==0: continue
        freq[word.lower()] += 1
        print(sorted(freq.items(), key=operator.itemgetter(1),
reverse=True) [:10])
```

- Using the `Counter` class in the following code:

```
import requests
text = requests.get('https://www.gutenberg.org/files/2852/2852-0.
txt').text
freq = Counter(filter(None, map(lambda x:x.lower().strip(), text.
split())))
print(freq.most_common(10))
```

ChainMap

A `ChainMap` is a dictionary-like class that groups multiple dictionaries or similar mapping data structures together to create a single view that is updateable.

All of the usual dictionary methods are supported. Lookups search successive maps until a key is found.

The `ChainMap` class is a more recent addition to Python, having been added in Python 3.3.

When you have a scenario where you keep updating keys from a source dictionary to a target dictionary over and over again, a `ChainMap` class can work in your favor in terms of performance, especially if the number of updates is large.

Here are some practical uses of a `ChainMap`:

- A programmer can keep the `GET` and `POST` arguments of a web framework in separate dictionaries and keep the configuration updated via a single `ChainMap`.
- Keeping multilayered configuration overrides in applications.
- Iterating over multiple dictionaries as a view when there are no overlapping keys.
- A `ChainMap` class keeps the previous mappings in its maps attribute. However, when you update a dictionary with mappings from another dictionary, the original dictionary state is lost. Here is a simple demonstration:

```
>>> d1={i:i for i in range(100)}
>>> d2={i:i*i for i in range(100) if i%2}
>>> c=ChainMap(d1,d2)
# Older value accessible via chainmap
>>> c[5]
5
>>> c.maps[0][5]
5
# Update d1
>>> d1.update(d2)
# Older values also got updated
>>> c[5]
25
>>> c.maps[0][5]
25
```

namedtuple

A `namedtuple` is like a class with fixed fields. Fields are accessible via attribute lookups like a normal class but are also indexable. The entire namedtuple is also iterable like a container. In other words, a namedtuple behaves like a class and a tuple combined in one:

```
>>> Employee = namedtuple('Employee', 'name, age, gender, title,
department')
>>> Employee
<class '__main__.Employee'>
```

Let's create an instance of Employee:

```
>>> jack = Employee('Jack',25,'M','Programmer','Engineering')
>>> print(jack)
Employee(name='Jack', age=25, gender='M', title='Programmer',
department='Engineering')
```

We can iterate over the fields of the instance, as if it is an iterator:

```
>>> for field in jack:
... print(field)
...
Jack
25
M
Programmer
Engineering
```

Once created, the `namedtuple` instance, like a tuple, is read-only:

```
>>> jack.age=32
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

To update values, the `_replace` method can be used. It returns a new instance with the specified keyword arguments replaced with new values:

```
>>> jack._replace(age=32)
Employee(name='Jack', age=32, gender='M', title='Programmer',
department='Engineering')
```

A namedtuple is much more memory-efficient when compared to a class which has the same fields. Hence a namedtuple is very useful in the following scenarios:

- A large amount of data needs to be loaded as read-only with keys and values from a store. Examples are loading columns and values via a DB query or loading data from a large CSV file.
- When a lot of instances of a class need to be created but not many write or set operations need to be done on the attributes. Instead of creating class instances, `namedtuple` instances can be created to save on memory.

- The `_make` method can be used to load an existing iterable that supplies fields in the same order to return a `namedtuple` instance. For example, if there is an `employees.csv` file with the columns name, age, gender, title, and department in that order, we can load them all into a container of `namedtuples` using the following command line:

  ```
  employees = map(Employee._make, csv.reader(open('employees.csv'))
  ```

Probabilistic data structures – bloom filters

Before we conclude our discussion on the container data types in Python, let us take a look at an important probabilistic data structure named **Bloom Filter**. Bloom filter implementations in Python behave like containers, but they are probabilistic in nature.

A bloom filter is a sparse data structure that allows us to test for the presence of an element in the set. However, we can only positively be sure of whether an element is not there in the set—that is, we can assert only for true negatives. When a bloom filter tells us an element is there in the set, it might be there—in other words, there is a non-zero probability that the element may actually be missing.

Bloom filters are usually implemented as bit vectors. They work in a similar way to a Python dictionary in that they use hash functions. However, unlike dictionaries, bloom filters don't store the actual elements themselves. Also elements, once added, cannot be removed from a bloom filter.

Bloom filters are used when the amount of source data implies an unconventionally large amount of memory if we store all of it without hash collisions.

In Python, the `pybloom` package provides a simple bloom filter implementation (however, at the time of writing, it doesn't support Python 3.x, so the examples here are shown in Python 2.7.x):

```
$ pip install pybloom
```

Let us write a program to read and index words from the text of `The Hound of Baskervilles`, which was the example we used in the discussion of the Counter data structure, but this time using a bloom filter:

```
# bloom_example.py
from pybloom import BloomFilter
import requests

f=BloomFilter(capacity=100000, error_rate=0.01)
```

```
text=requests.get('https://www.gutenberg.org/files/2852/2852-0.txt').
text

for word in text.split():
    word = word.lower().strip()
    f.add(word)

print len(f)
print len(text.split())
for w in ('holmes','watson','hound','moor','queen'):
    print 'Found',w,w in f
```

Executing this, we get the following output:

```
$ python bloomtest.py
9403
62154
Found holmes True
Found watson True
Found moor True
Found queen False
```

The words `holmes`, `watson`, `hound`, and `moor` are some of the most common in the story of *The Hound of Baskervilles*, so it is reassuring that the bloom filter finds these words. On the other hand, the word `queen` never appears in the text so the bloom filter is correct on that fact (true negative). The number of the words in the text is 62,154, out of which only 9,403 got indexed in the filter.

Let us try and measure the memory usage of the bloom filter as opposed to the Counter. For that we will rely on memory profiler.

For this test, we will rewrite the code using the `Counter` class as follows:

```
# counter_hound.py
import requests
from collections import Counter

@profile
def hound():
    text=requests.get('https://www.gutenberg.org/files/2852/2852-0.
txt').text
    c = Counter()
```

```
        words = [word.lower().strip() for word in text.split()]
        c.update(words)

    if __name__ == "__main__":
        hound()
```

And the one using the bloom filter as follows:

```
# bloom_hound.py
from pybloom import BloomFilter
import requests

@profile
def hound():
    f=BloomFilter(capacity=100000, error_rate=0.01)
    text=requests.get('https://www.gutenberg.org/files/2852/2852-0.
txt').text

    for word in text.split():
        word = word.lower().strip()
        f.add(word)

if __name__ == "__main__":
    hound()
```

Here is the output from running the memory profiler for the first one:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
$ python -m memory_profiler counter_hound.py
Filename: counter_hound.py

Line #    Mem usage    Increment   Line Contents
================================================
     4   40.906 MiB    0.000 MiB   @profile
     5                             def hound():
     6   45.082 MiB    4.176 MiB       text=requests.get('https://www.gutenberg.org/files/2852/285
2-0.txt').text
     7   45.082 MiB    0.000 MiB       c = Counter()
     8   49.570 MiB    4.488 MiB       words = [word.lower().strip() for word in text.split()]
     9   50.266 MiB    0.695 MiB       c.update(words)

$
```

Memory usage by the Counter object when parsing the text of The Hound of the Baskervilles

The following result is for the second one:

```
anand@ubuntu-pro-book: /home/user/programs/chap4
File  Edit  View  Search  Terminal  Help
$ python -m memory_profiler bloom_hound.py
Filename: bloom_hound.py

Line #    Mem usage    Increment   Line Contents
================================================
     4   40.996 MiB    0.000 MiB   @profile
     5                             def hound():
     6   41.160 MiB    0.164 MiB       f=BloomFilter(capacity=100000, error_rate=0.01)
     7   45.621 MiB    4.461 MiB       text=requests.get('https://www.gutenberg.org/files/2852/285
2-0.txt').text
     8
     9   49.742 MiB    4.121 MiB       for word in text.split():
    10   49.742 MiB    0.000 MiB           word = word.lower().strip()
    11   49.742 MiB    0.000 MiB           f.add(word)
```

Memory usage by the Bloom filter for parsing text of The Hound of the Baskervilles

The final memory usage is roughly the same at about 50 MB each. In the case of the Counter, nearly no memory is used when the Counter class is created but close to 0.7 MB is used when words are added to the counter.

However, there is a distinct difference in the memory growth pattern between both these data structures.

In the case of the bloom filter, an initial memory of 0.16 MB is allotted to it upon creation. The addition of the words seems to add nearly no memory to the filter and hence to the program.

So when should we use a bloom filter as opposed to, say, a dictionary or set in Python? Here are some general principles and real-world usage scenarios:

- When you are fine with not storing the actual element itself but only interested in the presence (or absence) of the element. In other words, where your application use case relies more on checking the absence of data than its presence.
- When the size of your input data is so large that storing each and every item in a deterministic data structure (as a dictionary or hashtable) in memory is not feasible. A bloom filter takes much less data in memory as opposed to a deterministic data structure.
- When you are fine with a certain well-defined error rate of *false positives* with your dataset—let us say this is 5% out of 1,000,000 pieces of data—you can configure a bloom filter for this specific error rate and get a data hit rate that will satisfy your requirements.

Some real-world examples of using bloom filters are as follows:

- **Security testing**: Storing data for malicious URLs in browsers, for example
- **Bio-informatics**: Testing the presence of a certain pattern (a k-mer) in a genome
- To avoid storing URLs with just one hit in a distributed web-caching infrastructure

Summary

This chapter was all about performance. At the start of the chapter, we discussed performance and SPE. We looked at the two categories of performance testing and diagnostic tools—namely, stress-testing tools and profiling/instrumentation tools.

We then discussed what performance complexity really means in terms of the Big-O notation and discussed briefly the common time orders of functions. We looked at the time taken by functions to execute and learned the three classes of time usage—namely `real`, `user`, and `sys` in POSIX systems.

We moved on to measuring performance and time in the next section—starting with a simple context manager timer and moving on to more accurate measurements using the `timeit` module. We measured the time taken for certain algorithms for a range of input sizes. By plotting the time taken against the input size and superimposing it on the standard time complexity graphs, we were able to get a visual understanding of the performance complexity of functions. We optimized the common item problem from its O(n*log(n)) performance to O(n) and the plotted graphs of time usage confirmed this.

We then started our discussion on profiling code and saw some examples of profiling using the `cProfile` module. The example we chose was a prime number iterator returning the first `n` primes performing at O(n). Using the profiled data, we optimized the code a bit, making it perform better than O(n). We briefly discussed the `pstats` module and used its `Stats` class to read profile data and produce custom reports ordered by a number of available data fields. We discussed two other third-party profilers—the `liner_profiler` and the `memory_profiler`, which profile code line by line—and discussed the problem of finding sub-sequences among two sequences of strings, writing an optimized version of them, and measuring its time and memory usage using these profilers.

Among other tools, we discussed objgraph and pympler—the former as a visualization tool to find relations and references between objects, helping to explore memory leaks, and the latter as a tool to monitor and report the memory usage of objects in the code and provide summaries.

In the last section on Python containers, we looked at the best and worst use case scenarios of standard Python containers—such as list, dict, set, and tuple. We then studied high performance container classes in the collections module—`deque`, `defaultdict`, `OrderedDict`, `Counter`, `Chainmap`, and `namedtuple`, with examples and recipes for each. Specifically, we saw how to create an LRU cache very naturally using `OrderedDict`.

Towards the end of the chapter, we discussed a special data structure called the bloom filter, which is very useful as a probabilistic data structure to report true negatives with certainty and true positives within a pre-defined error rate.

In the next chapter, we will discuss a close cousin of performance, scalability, where we will look at the techniques of writing scalable applications and the details of writing scalable and concurrent programs in Python.

5
Writing Applications that Scale

Imagine the checkout counter of a supermarket on a Saturday evening, the usual rush-hour time. It is common to see long queues of people waiting to check out with their purchases. What could a store manager do to reduce the rush and waiting time?

A typical manager would try a few approaches, including telling those manning the checkout counters to pick up their speed, and to try and redistribute people to different queues so that each queue roughly has the same waiting time. In other words, they would manage the current load with available resources by *optimizing the performance* of the existing resources.

However, if the store has existing counters that are not in operation—and enough people at hand to manage them—the manager could enable those counters, and move people to these new counters. In other words, they would add resources to the store to *scale* the operation.

Software systems, too, scale in a similar way. An existing software application can be scaled by adding compute resources to it.

When the system scales by either adding or making better use of resources inside a compute node, such as CPU or RAM, it is said to *scale vertically* or *scale up*. On the other hand, when a system scales by adding more compute nodes to it, such as a creating a load-balanced cluster of servers, it is said to *scale horizontally* or *scale out*.

The degree to which a software system is able to scale when compute resources are added is called its *scalability*. Scalability is measured in terms of how much the system's performance characteristics, such as throughput or latency, improve with respect to the addition of resources. For example, if a system doubles its capacity by doubling the number of servers, it is scaling linearly.

Increasing the concurrency of a system often increases its scalability. In the supermarket example given earlier, the manager is able to scale out his operations by opening additional counters. In other words, they increase the amount of concurrent processing done in their store. Concurrency is the amount of work that gets done simultaneously in a system.

In this chapter, we look at the different techniques of scaling a software application with Python.

We will be following the approximate sketch of the following topics in our discussion in this chapter:

- Scalability and performance
- Concurrency
    - Concurrency and parallelism
    - Concurrency in Python – multithreading
        - Thumbnail generator
        - Thumbnail generator – producer/consumer architecture
        - Thumbnail generator – program end condition
        - Thumbnail generator – resource constraint using locks
        - Thumbnail generator – resource constraint using semaphores
        - Resource constraint – semaphore versus lock
        - Thumbnail generator – URL rate controller using conditions
        - Multi-threading – Python and GIL
    - Concurrency in Python – multiprocessing:
        - A primality checker
        - Sorting disk files
        - Sorting disk files – using a counter
        - Sorting disk files – using multiprocessing
    - Multi-threading versus multiprocessing
    - Concurrency in Python – Asynchronous execution
        - Pre-emptive versus co-operative multitasking
        - `asyncio` in Python

        - Waiting for future – `async` and `await`
        - Concurrent futures – high-level concurrent processing
    - Concurrency options - how to choose

- Parallel processing libraries:
    - joblib
    - PyMP
        - Fractals – the Mandelbrot set
        - Fractals – scaling the Mandelbrot set implementation

- Scaling for the web:
    - Scaling workflows – message queues and task queues
    - Celery – a distributed task queue
      The Mandelbrot set - Using Celery
    - Serving Python on the web – WSGI
      uWSGI – WSGI middleware on steroids
      Gunicorn – unicorn for WSGI
      Gunicorn versus uWSGI

- Scalability architectures:
    - Vertical scalability architectures
    - Horizontal scalability architectures

Scalability and performance

How do we measure the scalability of a system? Let's take an example, and see how this is done.

Let's say our application is a simple report generation system for employees. It is able to load employee data from a database, and generate a variety of reports in bulk, such as pay slips, tax deduction reports, employee leave reports, and more.

The system is able to generate 120 reports per minute—this is the *throughput* or *capacity* of the system expressed as the number of successfully completed operations in a given unit of time. Let's say the time it takes to generate a report at the server side (latency) is roughly 2 seconds.

Let's say the architect decides to scale up the system by doubling the RAM on its server.

Once this is done, a test shows that the system is able to increase its throughput to 180 reports per minute. The latency remains the same at 2 seconds.

So, at this point, the system has scaled *close to linear* in terms of the memory added. The scalability of the system expressed in terms of throughput increase is as follows:

Scalability (throughput) = *180/120 = 1.5X*

As a second step, the architect decides to double the number of servers on the backend—all with the same memory. After this step, it's found that the system's performance throughput has now increased to 350 reports per minute. The scalability achieved by this step is given as follows:

Scalability (throughput) = *350/180 = 1.9X*

The system has now responded much better with a close to linear increase in scalability.

After further analysis, the architect finds that by rewriting the code that was processing reports on the server to run in multiple processes instead of a single process, he is able to reduce the processing time at the server, and hence, the latency of each request by roughly 1 second per request at peak time. The latency has now gone down from 2 seconds to 1 second.

The system's performance with respect to latency has become better as follows:

Performance (latency): *X = 2/1 = 2X*

How does this improve scalability? Since the time taken to process each request is less now, the system overall will be able to respond to similar loads at a faster rate than what it was able to earlier. With the exact same resources, the system's throughput performance, and hence, scalability has increased assuming other factors remain the same.

Let's summarize what we've discussed so far, as follows:

1. In the first step, the architect increased the throughput of a single system by scaling it up by adding extra memory as a resource, which increased the overall scalability of the system. In other words, he scaled the performance of a single system by *scaling up*, which boosted the overall performance of the whole system.

2. In the second step, he added more nodes to the system, and hence, its ability to perform work concurrently, and found that the system responded well by rewarding him with a near-linear scalability factor. In other words, he increased the throughput of the system by scaling its resource capacity. Thus, he increased scalability of the system by *scaling out*, that is, by adding more compute nodes.
3. In the third step, he made a critical fix by running a computation in more than one process. In other words, he increased the *concurrency* of a single system by dividing the computation into more than one part. He found that this increased the performance characteristic of the application by reducing its *latency*, potentially setting up the application to handle workloads better at high stress.

We find that there is a relationship between scalability, performance, concurrency, and latency. This can be explained as follows:

1. When the performance of one of the components in a system goes up, generally the performance of the overall system goes up.
2. When an application scales in a single machine by increasing its concurrency, it has the potential to improve performance, and hence, the net scalability of the system in deployment.
3. When a system reduces its performance time, or its latency, at the server, it positively contributes to scalability.

We have captured these relationships in the following table:

Concurrency	Latency	Performance	Scalability
High	Low	High	High
High	High	Variable	Variable
Low	High	Poor	Poor

An ideal system is one that has good concurrency and low latency; such a system has high performance, and would respond better to scaling up and/or scaling out.

A system with high concurrency, but also high latency, would have variable characteristics—its performance, and hence, scalability would be potentially very sensitive to other factors such as current system load, network congestion, geographical distribution of compute resources and requests, and so on.

A system with low concurrency and high latency is the worst case—it would be difficult to scale such a system, as it has poor performance characteristics. The latency and concurrency issues should be addressed before the architect decides to scale the system horizontally or vertically.

Scalability is always described in terms of variation in performance throughput.

Concurrency

A system's concurrency is the degree to which the system is able to perform work simultaneously instead of sequentially. An application written to be concurrent in general, can execute more units of work in a given time than one which is written to be sequential or serial.

When we make a serial application concurrent, we make the application better utilize the existing compute resources in the system—CPU and/or RAM—at a given time. Concurrency, in other words, is the cheapest way of making an application scale inside a machine in terms of the cost of compute resources.

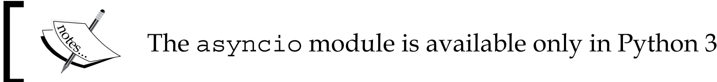
Concurrency can be achieved using different techniques. The common ones include the following:

- **Multithreading**: The simplest form of concurrency is to rewrite the application to perform parallel tasks in different threads. A thread is the simplest sequence of programming instructions that can be performed by a CPU. A program can consist of any number of threads. By distributing tasks to multiple threads, a program can execute more work simultaneously. All threads run inside the same process.
- **Multiprocessing**: Another way to concurrently scale up a program is to run it in multiple processes instead of a single process. Multiprocessing involves more overhead than multithreading in terms of message passing and shared memory. However, programs that perform a lot of CPU-intensive computations can benefit more from multiple processes than multiple threads.
- **Asynchronous Processing**: In this technique, operations are performed asynchronously with no specific ordering of tasks with respect to time. Asynchronous processing usually picks tasks from a queue of tasks, and schedules them to execute at a future time, often receiving the results in callback functions or special future objects. Asynchronous processing usually happens in a single thread.

There are other forms of concurrent computing, but in this chapter, we will focus our attention on only these three.

Python, especially Python 3, has built-in support for all these types of concurrent computing techniques in its standard library. For example, it supports multi-threading via its `threading` module, and multiple processes via its `multiprocessing` module. Asynchronous execution support is available via the *asyncio* module. A form of concurrent processing that combines asynchronous execution with threads and processes is available via the `concurrent.futures` module.

In the coming sections we will take a look at each of these in turn with sufficient examples.



The `asyncio` module is available only in Python 3.

Concurrency versus parallelism

We will take a brief look at the concept of concurrency and its close cousin, namely parallelism.

Both concurrency and parallelism are about executing work simultaneously rather than sequentially. However, in concurrency, the two tasks need not be executed at the exact same time; instead, they just need to be scheduled to be executed simultaneously. Parallelism, on the other hand, requires that both the tasks execute together at a given moment in time.

To take a real-life example, let's say you are painting two exterior walls of your house. You have employed just one painter, and you find that he is taking a lot more time than you thought. You can solve the problem in these two ways:

1. Instruct the painter to paint a few coats on one wall before switching to the next wall, and doing the same there. Assuming he is efficient, he will work on both the walls simultaneously (though not at the same time), and achieve the same degree of finish on both walls for a given time. This is a *concurrent* solution.
2. Employ one more painter. Instruct the first painter to paint the first wall, and the second painter to paint the second wall. This is a *parallel* solution.

Two threads performing bytecode computations in a single core CPU do not exactly perform parallel computation, as the CPU can accommodate only one thread at a time. However, they are concurrent from a programmer's perspective, since the CPU scheduler performs fast switching in and out of the threads so that they appear to run in parallel.

However, on a multi-core CPU, two threads can perform parallel computations at any given time in its different cores. This is true parallelism.

Parallel computation requires that the computation resources increase at least linearly with respect to its scale. Concurrent computation can be achieved by using the techniques of multitasking, where work is scheduled and executed in batches, making better use of existing resources.

In this chapter, we will use the term *concurrent* uniformly to indicate both types of execution. In some places, it may indicate concurrent processing in the traditional way, and in some others, it may indicate true parallel processing. Use the context to disambiguate.

Concurrency in Python – multithreading

We will start our discussion of concurrent techniques in Python with multithreading.

Python supports multiple threads in programming via its *threading* module. The threading module exposes a `Thread` class, which encapsulates a thread of execution. Along with this, it also exposes the following synchronization primitives:

- A `Lock` object, which is useful for synchronized protected access to share resources, and its cousin `RLock`
- A `Condition` object, which is useful for threads to synchronize while waiting for arbitrary conditions
- An `Event` object, which provides a basic signaling mechanism between threads
- A `Semaphore` object, which allows synchronized access to limited resources
- A `Barrier` object, which allows a fixed set of threads to wait for each other, synchronize to a particular state, and proceed

Thread objects in Python can be combined with the synchronized `Queue` class in the queue module for implementing thread-safe producer/consumer workflows.

Thumbnail generator

Let's start our discussion of multi-threading in Python with the example of a program used to generate thumbnails of image URLs.

In the example, we are using **Pillow**, a fork of the **Python Imaging Library** (**PIL**) to perform this operation:

```
# thumbnail_converter.py
from PIL import Image
import urllib.request

def thumbnail_image(url, size=(64, 64), format='.png'):
    """ Save thumbnail of an image URL """

    im = Image.open(urllib.request.urlopen(url))
    # filename is last part of the URL minus extension + '.format'
    pieces = url.split('/')
    filename = ''.join((pieces[-2],'_',pieces[-1].split('.')[0],'_
thumb',format))
    im.thumbnail(size, Image.ANTIALIAS)
    im.save(filename)
    print('Saved',filename)
```

The preceding code works very well for single URLs.

Let's say we want to convert five image URLs to their thumbnails:

```
img_urls = ['https://dummyimage.com/256x256/000/fff.jpg',
            'https://dummyimage.com/320x240/fff/00.jpg',
            'https://dummyimage.com/640x480/ccc/aaa.jpg',
            'https://dummyimage.com/128x128/ddd/eee.jpg',
            'https://dummyimage.com/720x720/111/222.jpg']
for url in img_urls:
    thumbnail_image(urls)
```

Let's see how such a function performs with respect to time taken in the following screenshot:

```
Chapter 5 - Multithreading
File Edit View Search Terminal Help
$ time python3 thumbnail_converter.py
Saved 000_fff_thumb.png
Saved fff_00_thumb.png
Saved ccc_aaa_thumb.png
Saved ddd_eee_thumb.png
Saved 111_222_thumb.png

real    0m8.446s
user    0m0.184s
sys     0m0.028s
$
```

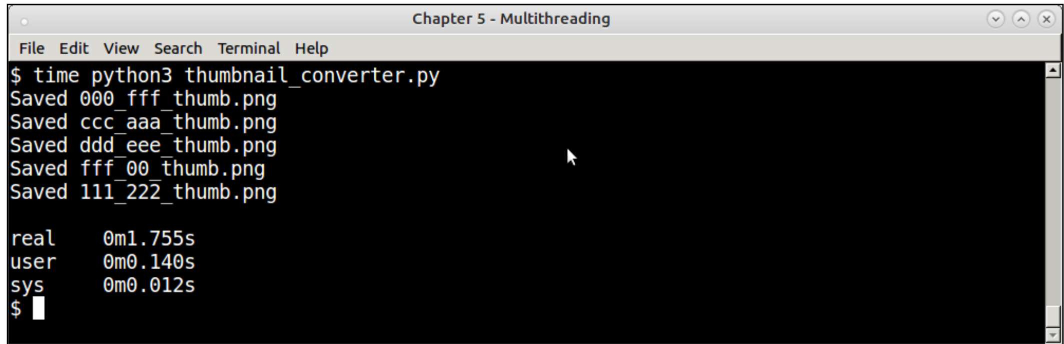
Response time of serial thumbnail converter for 5 URLs

The function took approximately 1.7 seconds per URL.

Let's now scale the program to multiple threads so we can perform the conversions concurrently. Here is the rewritten code to run each conversion in its own thread:

```
import threading

for url in img_urls:
    t=threading.Thread(target=thumbnail_image,args=(url,))
    t.start()
```

The timing that this last program now gives is shown in this screenshot:



Response time of threaded thumbnail converter for 5 URLs

With this change, the program returns in 1.76 seconds, almost equal to the time taken by a single URL in serial execution before. In other words, the program has now linearly scaled with respect to the number of threads. Note that, we had to make no change to the function itself to get this scalability boost.

Thumbnail generator – producer/consumer architecture

In the previous example, we saw a set of image URLs being processed by a thumbnail generator function concurrently by using multiple threads. With the use of multiple threads, we were able to achieve near linear scalability as compared to serial execution.

However, in real life, rather than processing a fixed list of URLs, it is more common for the URL data to be produced by some kind of URL producer. It could be fetching this data from a database, a **comma separated value** (**CSV**) file or from a TCP socket for example.

In such a scenario, creating one thread per URL would be a tremendous waste of resources. It takes a certain overhead to create a thread in the system. We need some way to reuse the threads we create.

For such systems that involve a certain set of threads producing data and another set of threads consuming or processing data, the producer/consumer model is an ideal fit. Such a system has the following features:

1. Producers are a specialized class of workers (threads) producing the data. They may receive the data from a specific source(s), or generate the data themselves.
2. Producers add the data to a shared synchronized queue. In Python, this queue is provided by the `Queue` class in the aptly named `queue` module.
3. Another set of specialized class of workers, namely consumers, wait on the queue to get (consume) the data. Once they get the data, they process it and produce the results.
4. The program comes to an end when the producers stop generating data and the consumers are starved of data. Techniques like timeouts, polling, or poison pills can be used to achieve this. When this happens, all threads exit, and the program completes.

We have rewritten our thumbnail generator to a producer consumer architecture. The resulting code is given next. Since this is a bit detailed, we will discuss each class one by one.

First, let's look at the imports—these are pretty self-explanatory:

```
# thumbnail_pc.py
import threading
import time
import string
import random
import urllib.request
from PIL import Image
from queue import Queue
```

Next is the code for the producer class:

```
class ThumbnailURL_Generator(threading.Thread):
    """ Worker class that generates image URLs """

    def __init__(self, queue, sleep_time=1,):
        self.sleep_time = sleep_time
        self.queue = queue
        # A flag for stopping
        self.flag = True
        # choice of sizes
        self._sizes = (240,320,360,480,600,720)
        # URL scheme
        self.url_template = 'https://dummyimage.com/%s/%s/%s.jpg'
        threading.Thread.__init__(self, name='producer')

    def __str__(self):
        return 'Producer'

    def get_size(self):
        return '%dx%d' % (random.choice(self._sizes),
                          random.choice(self._sizes))

    def get_color(self):
        return ''.join(random.sample(string.hexdigits[:-6], 3))

    def run(self):
        """ Main thread function """

        while self.flag:
            # generate image URLs of random sizes and fg/bg colors
            url = self.url_template % (self.get_size(),
                                       self.get_color(),
                                       self.get_color())
```

```
            # Add to queue
            print(self,'Put',url)
            self.queue.put(url)
            time.sleep(self.sleep_time)


    def stop(self):
        """ Stop the thread """

        self.flag = False
```

Let's analyze the producer class code:

1. The class is named `ThumbnailURL_Generator`. It generates the URLs (by using the service of a website named `http://dummyimage.com`) of different sizes, foreground, and background colors. It inherits from the `threading.Thread` class.
2. It has a `run` method, which goes in a loop, generates a random image URL, and pushes it to the shared queue. Every time, the thread sleeps for a fixed time, as configured by the `sleep_time` parameter.
3. The class exposes a `stop` method, which sets the internal flag to `False` causing the loop to break and the thread to finish its processing. This can be called externally by another thread, typically, the main thread.

Now we have the URL consumer class that consumes the thumbnail URLs and creates the thumbnails:

```
class ThumbnailURL_Consumer(threading.Thread):
    """ Worker class that consumes URLs and generates thumbnails """

    def __init__(self, queue):
        self.queue = queue
        self.flag = True
        threading.Thread.__init__(self, name='consumer')

    def __str__(self):
        return 'Consumer'

    def thumbnail_image(self, url, size=(64,64), format='.png'):
        """ Save image thumbnails, given a URL """

        im=Image.open(urllib.request.urlopen(url))
```

```
        # filename is last part of URL minus extension + '.format'
        filename = url.split('/')[-1].split('.')[0] + '_thumb' +
format
        im.thumbnail(size, Image.ANTIALIAS)
        im.save(filename)
        print(self,'Saved',filename)

    def run(self):
        """ Main thread function """

        while self.flag:
            url = self.queue.get()
            print(self,'Got',url)
            self.thumbnail_image(url)

    def stop(self):
        """ Stop the thread """

        self.flag = False
```

Here's the analysis of the consumer class:

1. The class is named `ThumbnailURL_Consumer`, as it consumes URLs from the queue, and creates thumbnail images of them.
2. The `run` method of this class goes in a loop, gets a URL from the queue, and converts it to a thumbnail by passing it to the `thumbnail_image` method. (Note that this code is exactly the same as that of the `thumbnail_image` function we created earlier.)
3. The `stop` method is very similar, checking for a stop flag every time in the loop, and ending once the flag has been unset.

Here is the main part of the code—setting up a couple of producers and consumers each, and running them:

```
q = Queue(maxsize=200)
producers, consumers = [], []

for i in range(2):
    t = ThumbnailURL_Generator(q)
    producers.append(t)
    t.start()

for i in range(2):
    t = ThumbnailURL_Consumer(q)
    consumers.append(t)
    t.start()
```

Here is a screenshot of the program in action:

```
Chapter 5 - Multithreading
File Edit View Search Terminal Help
Producer Put https://dummyimage.com/240x600/4c8/1d0.jpg
Producer Put https://dummyimage.com/240x240/18d/c05.jpg
Consumer Got https://dummyimage.com/240x600/4c8/1d0.jpg
Consumer Got https://dummyimage.com/240x240/18d/c05.jpg
Producer Put https://dummyimage.com/320x600/e41/58d.jpg
Producer Put https://dummyimage.com/360x720/9c0/b39.jpg
Consumer Saved 1d0_thumb.png
Consumer Got https://dummyimage.com/320x600/e41/58d.jpg
Consumer Saved c05_thumb.png
Consumer Got https://dummyimage.com/360x720/9c0/b39.jpg
Producer Put https://dummyimage.com/600x480/b9d/8e5.jpg
Producer Put https://dummyimage.com/360x600/62a/fc9.jpg
Producer Put https://dummyimage.com/320x240/eb7/c9b.jpg
Producer Put https://dummyimage.com/360x320/a90/340.jpg
Consumer Saved b39_thumb.png
Consumer Got https://dummyimage.com/600x480/b9d/8e5.jpg
Consumer Saved 58d_thumb.png
Consumer Got https://dummyimage.com/360x600/62a/fc9.jpg
Producer Put https://dummyimage.com/240x320/4f7/036.jpg
Producer Put https://dummyimage.com/480x720/8d9/d03.jpg
```

Running the thumbnail producer/consumer program with 4 threads, 2 of each type

In the above program, since the producers keep generating random data without any end, the consumers will keep consuming it without any end. Our program has no proper end condition.

Hence, this program will keep running until the network requests are denied or timed out or the disk space of the machine runs out because of thumbnails.

However, a program solving a real world problem should end in some way that is predictable.

This could be due to a number of external constraints

- It could be a timeout introduced where the consumers wait for data for a certain maximum time, and then exit if no data is available during that time. This, for example, can be configured as a timeout in the `get` method of the queue.
- Another technique would be to signal program end after a certain number of resources are consumed or created. In this program, for example, it could be a fixed limit to the number of thumbnails created.

In the following section, we will see how to enforce such resource limits by using threading synchronization primitives such as Locks and Semaphores.

You may have observed that we start a thread using its `start` method, though the overridden method in the Thread subclass is `run`. This is because, in the parent `Thread` class, the `start` method sets up some state, and then calls the `run` method internally. This is the right way to call the thread's run method. It should never be called directly.

Thumbnail generator – resource constraint using locks

In this section, we will see how to modify the program using a `Lock`, a synchronization primitive to implement a counter that will limit the number of images created as a way to end the program.

Lock objects in Python allows exclusive access by threads to a shared resource.

The pseudo-code would be as follows:

```
try:
  lock.acquire()
  # Do some modification on a shared, mutable resource
  mutable_object.modify()
finally:
  lock.release()
```

However, Lock objects support context-managers via the with statement, so this is more commonly written as follows:

```
with lock:
  mutable_object.modify()
```

To implement a fixed number of images per run, our code needs to be supported to add a counter. However, since multiple threads would check and increment this counter, it needs to be synchronized via a `Lock` object.

This is our first implementation of the resource counter class using Locks.

```
class ThumbnailImageSaver(object):
    """ Class which saves URLs to thumbnail images and keeps a counter
"""

    def __init__(self, limit=10):
        self.limit = limit
        self.lock = threading.Lock()
        self.counter = {}

    def thumbnail_image(self, url, size=(64,64), format='.png'):
        """ Save image thumbnails, given a URL """

        im=Image.open(urllib.request.urlopen(url))
        # filename is last two parts of URL minus extension +
'.format'
        pieces = url.split('/')
        filename = ''.join((pieces[-2],'_',pieces[-1].split('.')[0],'_
thumb',format))
        im.thumbnail(size, Image.ANTIALIAS)
        im.save(filename)
        print('Saved',filename)
        self.counter[filename] = 1
        return True

    def save(self, url):
        """ Save a URL as thumbnail """

        with self.lock:
            if len(self.counter)>=self.limit:
                return False
            self.thumbnail_image(url)
            print('Count=>',len(self.counter))
            return True
```

Since this modifies the consumer class as well, it makes sense to discuss both changes together. Here is the modified consumer class to accommodate the extra counter needed to keep track of the images:

```
class ThumbnailURL_Consumer(threading.Thread):
    """ Worker class that consumes URLs and generates thumbnails """

    def __init__(self, queue, saver):
        self.queue = queue
```

```
            self.flag = True
            self.saver = saver
            # Internal id
            self._id = uuid.uuid4().hex
            threading.Thread.__init__(self, name='Consumer-'+ self._id)

        def __str__(self):
            return 'Consumer-' + self._id

        def run(self):
            """ Main thread function """

            while self.flag:
                url = self.queue.get()
                print(self,'Got',url)
                if not self.saver.save(url):
                   # Limit reached, break out
                   print(self, 'Set limit reached, quitting')
                   break

        def stop(self):
            """ Stop the thread """

            self.flag = False
```

Let's analyze both of these classes. First, we'll look at the new class, `ThumbnailImageSaver`:

1. This class derives from the `object`. In other words, it is not a `Thread`. It is not meant to be one.
2. It initializes a lock object and a counter dictionary in its initializer method. The lock is for synchronizing access to the counter by threads. It also accepts a `limit` parameter equal to the number of images it should save.
3. The `thumbnail_image` method moves to here from the consumer class. It is called from a `save` method, which encloses the call in a synchronized context using the lock.
4. The `save` method first checks if the count has crossed the configured limit; when this happens, the method returns `False`. Otherwise, the image is saved with a call to `thumbnail_image`, and the image filename is added to the counter, effectively incrementing the count.

Next, we'll consider the modified `ThumbnailURL_Consumer` class:

1. The class's initializer is modified to accept an instance of the `ThumbnailImageSaver` as a `saver` argument. The rest of the arguments remain the same.
2. The `thumbnail_image` method no longer exists in this class, as it is moved to the new class.
3. The `run` method is much simplified. It makes a call to the `save` method of the saver instance. If it returns `False`, it means the limit has been reached, the loop breaks, and the consumer thread exits.
4. We have also modified the `__str__` method to return a unique ID per thread, which is set in the initializer using the `uuid` module. This helps to debug threads in a real-life example.

The calling code also changes a bit, as it needs to set up the new object, and configure the consumer threads with it:

```
q = Queue(maxsize=2000)
# Create an instance of the saver object
saver = ThumbnailImageSaver(limit=100)

    producers, consumers = [], []
    for i in range(3):
        t = ThumbnailURL_Generator(q)
        producers.append(t)
        t.start()

    for i in range(5):
        t = ThumbnailURL_Consumer(q, saver)
        consumers.append(t)
        t.start()

    for t in consumers:
        t.join()
        print('Joined', t, flush=True)

    # To make sure producers don't block on a full queue
    while not q.empty():
        item=q.get()

    for t in producers:
        t.stop()
        print('Stopped',t, flush=True)

    print('Total number of PNG images',len(glob.glob('*.png')))
```

The following are the main points to be noted here:

1. We create an instance of the new `ThumbnailImageSaver` class, and pass it on to the consumer threads when creating them.
2. We wait on consumers first. Note that, the main thread doesn't call `stop`, but `join` on them. This is because the consumers exit automatically when the limit is reached, so the main thread should just wait for them to stop.
3. We stop the producers after the consumers exit—explicitly so—since they would otherwise keep working forever, since there is no condition for the producers to exit.

We use a dictionary instead of an integer as because of the nature of the data.

Since the images are randomly generated, there is a minor chance of one image URL being the same as another one created previously, causing the filenames to clash. Using a dictionary takes care of such possible duplicates.

The following screenshot shows a run of the program with a limit of 100 images. Note that we can only show the last few lines of the console log, since it produces a lot of output:

```
Chapter 5 - Multithreading
File  Edit  View  Search  Terminal  Help
Producer-27f794680560473f939b0bd1fae5166f Put https://dummyimage.com/480x720/372/35f.jpg
Producer-8e2aa45e1ec14693939ec5b87a45429b Put https://dummyimage.com/320x360/e2f/d71.jpg
Producer-0ac969637a5c4b95af4f9d30638b42da Put https://dummyimage.com/720x480/754/e14.jpg
Producer-27f794680560473f939b0bd1fae5166f Put https://dummyimage.com/720x720/815/d3b.jpg
Producer-8e2aa45e1ec14693939ec5b87a45429b Put https://dummyimage.com/320x360/4b5/f02.jpg
Saved da4_fde_thumb.png
Count=> 100
Consumer-2df279565d52400985500148e402da08 Got https://dummyimage.com/600x480/38a/316.jpg
Consumer-71399ff8d7504ef49ac104c6cdab6ec3 Set limit reached, quitting
Consumer-d597bdfebe58452bb84f526ead921b64 Set limit reached, quitting
Joined Consumer-71399ff8d7504ef49ac104c6cdab6ec3
Consumer-c845a8d36fe543fd9945345a7a1e67fb Set limit reached, quitting
Consumer-2df279565d52400985500148e402da08 Set limit reached, quitting
Joined Consumer-c845a8d36fe543fd9945345a7a1e67fb
Joined Consumer-d597bdfebe58452bb84f526ead921b64
Joined Consumer-2df279565d52400985500148e402da08
Stopped Producer-0ac969637a5c4b95af4f9d30638b42da
Stopped Producer-27f794680560473f939b0bd1fae5166f
Stopped Producer-8e2aa45e1ec14693939ec5b87a45429b
Total number of PNG images 100
$
```

Run of the thumbnail generator program with a limit of 100 images using a Lock

You can configure this program with any limit of the images, and it will always fetch exactly the same count—nothing more or less.

In the next section, we will familiarize ourselves with another synchronization primitive, namely *semaphore*, and learn how to implement a resource limiting class in a similar way using the semaphore.

Thumbnail generator – resource constraint using semaphores

Locks aren't the only way to implement synchronization constraints and write logic on top of them in order to limit resources used/generated by a system.

A semaphore, one of the oldest synchronization primitives in computer science, is ideally suited for such use cases.

A semaphore is initialized with a value greater than zero:

1. When a thread calls `acquire` on a semaphore that has a positive internal value, the value gets decremented by one, and the thread continues on its way.
2. When another thread calls `release` on the semaphore, the value is incremented by 1.
3. Any thread calling `acquire` once the value has reached zero is blocked on the semaphore until it is woken up by another thread calling `release`.

Due to this behavior, a semaphore is perfectly suited for implementing a fixed limit on shared resources.

In the following code example, we will implement another class for resource limiting our thumbnail generator program, this time using a semaphore:

```
class ThumbnailImageSemaSaver(object):
    """ Class which keeps an exact counter of saved images
    and restricts the total count using a semaphore """

    def __init__(self, limit = 10):
        self.limit = limit
        self.counter = threading.BoundedSemaphore(value=limit)
        self.count = 0

    def acquire(self):
        # Acquire counter, if limit is exhausted, it
        # returns False
        return self.counter.acquire(blocking=False)

    def release(self):
```

```
        # Release counter, incrementing count
        return self.counter.release()

    def thumbnail_image(self, url, size=(64,64), format='.png'):
        """ Save image thumbnails, given a URL """

        im=Image.open(urllib.request.urlopen(url))
        # filename is last two parts of URL minus extension +
'.format'
        pieces = url.split('/')
        filename = ''.join((pieces[-2],'_',pieces[-1].split('.')
[0],format))
        try:
            im.thumbnail(size, Image.ANTIALIAS)
            im.save(filename)
            print('Saved',filename)
            self.count += 1
        except Exception as e:
            print('Error saving URL',url,e)
            # Image can't be counted, increment semaphore
            self.release()

        return True

    def save(self, url):
        """ Save a URL as thumbnail """

        if self.acquire():
            self.thumbnail_image(url)
            return True
        else:
            print('Semaphore limit reached, returning False')
            return False
```

Since the new semaphore-based class keeps the exact same interface as the previous lock-based class—with a save method—there is no need to change any code on the consumer!

Only the calling code needs to be changed.

This line in the previous code initialized the `ThumbnailImageSaver` instance:

```
saver = ThumbnailImageSaver(limit=100)
```

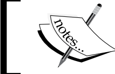
The preceding line needs to be replaced with the following one:

```
saver = ThumbnailImageSemaSaver(limit=100)
```

The rest of the code remains exactly the same.

Let's quickly discuss the new class using the semaphore before seeing this code in action:

1. The `acquire` and `release` methods are simple wrappers over the same methods on the semaphore.
2. We initialize the semaphore with a value equal to the image limit in the initializer.
3. In the save method, we call the `acquire` method. If the semaphore's limit is reached, it will return `False`. Otherwise, the thread saves the image and returns `True`. In the former case, the calling thread quits.



The internal count attribute of this class is only there for debugging. It doesn't add anything to the logic of limiting images.

This class behaves in a way similar way to the previous one, and limits resources exactly. The following is an example with a limit of 200 images:

```
Chapter 5 - Multithreading
File  Edit  View  Search  Terminal  Help
Consumer-275279a36ff049c38c77b506c5fa4010 Got https://dummyimage.com/720x360/47e/76b.jpg
Semaphore limit reached, returning False
Consumer-275279a36ff049c38c77b506c5fa4010 Set limit reached, quitting
Producer-cac6092edbbf40fa89ebdb530f1afc40 Put https://dummyimage.com/480x720/172/1fe.jpg
Producer-cc88bd03b49b43348aa3d6128869aea4 Put https://dummyimage.com/320x320/2f1/75c.jpg
Producer-b6fa5b443b8045e792278b4c49c6037f Put https://dummyimage.com/720x360/eab/570.jpg
Saved f96_3d0.png
Consumer-8b3103b9076640b1a4387b6cdc81f7c7 Got https://dummyimage.com/360x320/5be/567.jpg
Semaphore limit reached, returning False
Consumer-8b3103b9076640b1a4387b6cdc81f7c7 Set limit reached, quitting
Joined Consumer-8b3103b9076640b1a4387b6cdc81f7c7
Joined Consumer-275279a36ff049c38c77b506c5fa4010
Joined Consumer-ef6c97e1618744c9af41cc31c70552da
Joined Consumer-92460417f9984080bea7417159a9112b
Producer-cac6092edbbf40fa89ebdb530f1afc40 Put https://dummyimage.com/240x320/4fb/bc5.jpg
Producer-b6fa5b443b8045e792278b4c49c6037f Put https://dummyimage.com/360x320/792/491.jpg
Stopped Producer-cc88bd03b49b43348aa3d6128869aea4
Stopped Producer-cac6092edbbf40fa89ebdb530f1afc40
Stopped Producer-b6fa5b443b8045e792278b4c49c6037f
Total number of PNG images 200
$
```

Run of the thumbnail generator program with a limit of 200 images using a Semaphore

Resource constraint – semaphore versus lock

We saw two competing versions of implementing a fixed resource constraint in the previous two examples—one using `Lock` and another using `Semaphore`.

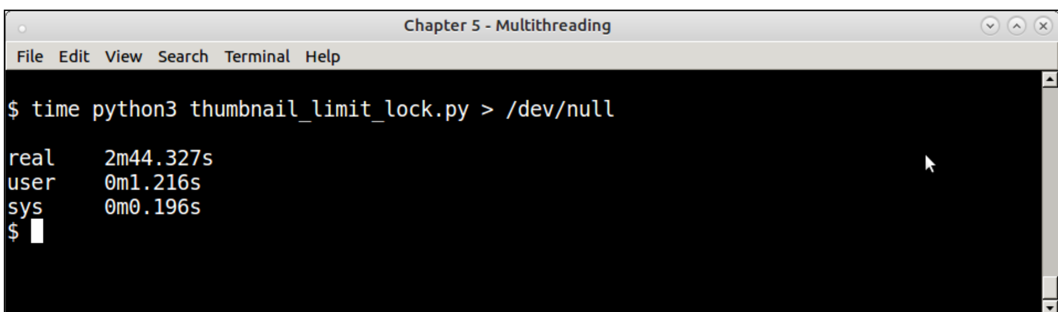
The differences between the two versions are as follows:

1. The version using Lock protects all the code that modifies the resource—in this case, checking the counter, saving the thumbnail, and incrementing the counter—to make sure that there are no data inconsistencies.
2. The Semaphore version is implemented more like a gate—a door that is open while the count is below the limit, and through which any number of threads can pass, and that only closes when the limit is reached. In other words, it doesn't mutually exclude threads from calling the thumbnail saving function.

Hence, the effect is that the semaphore version would be faster than the version using Lock.

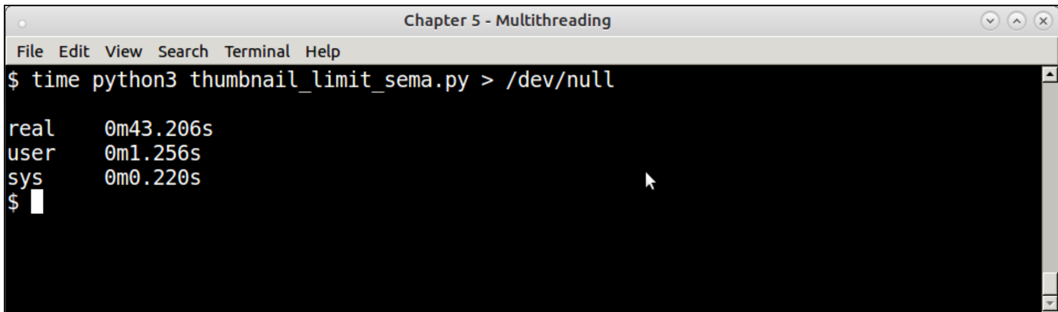
How much faster? The following timing example for a run of 100 images gives an idea.

This screenshot shows the time it takes for the Lock version to save 100 images:



Timing the run of the thumbnail generator program—the Lock version—for 100 images

The following screenshot shows the time for the semaphore version to save a similar number:



Timing the run of the thumbnail generator program—the semaphore version—for 100 images

By a quick calculation you can see that the semaphore version is about 4 times faster than the lock version for the same logic. In other words, it *scales 4 times better*.

Thumbnail generator – URL rate controller using conditions

In this section, we will briefly see the application of another important synchronization primitive in threading, namely the `Condition` object.

First, we will get a real life example of using a `Condition` object. We will implement a throttler for our thumbnail generator to manage the rate of URL generation.

In the producer/consumer systems in real life, the following three kinds of scenario can occur with respect to the rate of data production and consumption:

1. Producers produce data at a faster pace than consumers can consume. This causes the consumers to always play catch up with the producers. Excess data by the producers can accumulate in the queue, which causes the queue to consume a higher memory and CPU usage in every loop causing the program to slow down.
2. Consumers consume data at a faster rate than producers. This causes the consumers to always wait on the queue—for data. This, in itself, is not a problem as long as the producers don't lag too much. In the worst case, this leads to half of the system, that is, the consumers, remaining idle, while the other half—the producers—try to keep up with the demand.
3. Both producers and consumers work at nearly the same pace keeping the queue size within limits. This is the ideal scenario.

There are many ways to solve this problem. Some of them are as follows:

1. **Queue with a fixed size**: Producers would be forced to wait until data is consumed by a consumer once the queue size limit is reached. However this would almost always keep the queue full.
2. **Provide the workers with timeouts plus other responsibilities**: Rather than remain blocked on the queue, producers and/or consumers can use a timeout to wait on the queue. When they time out they can either sleep or perform some other responsibilities before coming back and waiting on the queue.
3. **Dynamically configure the number of workers**: This is an approach where the worker pool size automatically increases or decreases upon demand. If one class of workers is ahead, the system will launch just the required number of workers of the opposite class to keep the balance.
4. **Adjust the data generation rate**: In this approach, we statically or dynamically adjust the data generation rate by the producers. For example, the system can be configured to produce data at a fixed rate, say, 50 URLs in a minute or it can calculate the rate of consumption by the consumers, and adjust the data production rate of the producers dynamically to keep things in balance.

In the following example, we will implement the last approach—to limit the production rate of URLs to a fixed limit using `Condition` objects.

A `Condition` object is a sophisticated synchronization primitive that comes with an implicit built-in lock. It can wait on an arbitrary condition till it becomes true. The moment the thread calls `wait` on the condition, the internal lock is released, but the thread itself becomes blocked:

```
cond = threading.Condition()
# In thread #1
with cond:
    while not some_condition_is_satisfied():
        # this thread is now blocked
        cond.wait()
```

Now, another thread can wake up this preceding thread by setting the condition to True, and then calling `notify` or `notify_all` on the condition object. At this point, the preceding blocked thread is woken up, and continues on its way:

```
# In thread #2
with cond:
    # Condition is satisfied
```

```
    if some_condition_is_satisfied():
        # Notify all threads waiting on the condition
        cond.notify_all()
```

Here is our new class namely `ThumbnailURLController` which implements the rate control of URL production using a condition object.

```
class ThumbnailURLController(threading.Thread):
    """ A rate limiting controller thread for URLs using conditions
    """

    def __init__(self, rate_limit=0, nthreads=0):
        # Configured rate limit
        self.rate_limit = rate_limit
        # Number of producer threads
        self.nthreads = nthreads
        self.count = 0
        self.start_t = time.time()
        self.flag = True
        self.cond = threading.Condition()
        threading.Thread.__init__(self)

    def increment(self):
        # Increment count of URLs
        self.count += 1

    def calc_rate(self):
        rate = 60.0*self.count/(time.time() - self.start_t)
        return rate

    def run(self):
        while self.flag:
            rate = self.calc_rate()
            if rate<=self.rate_limit:
                with self.cond:
                    # print('Notifying all...')
                    self.cond.notify_all()

    def stop(self):
        self.flag = False

    def throttle(self, thread):
        """ Throttle threads to manage rate """
        # Current total rate
        rate = self.calc_rate()
```

```
        print('Current Rate',rate)
        # If rate > limit, add more sleep time to thread
        diff = abs(rate - self.rate_limit)
        sleep_diff = diff/(self.nthreads*60.0)

        if rate>self.rate_limit:
            # Adjust threads sleep_time
            thread.sleep_time += sleep_diff
            # Hold this thread till rate settles down with a 5% error
            with self.cond:
                print('Controller, rate is high, sleep more
by',rate,sleep_diff)
                while self.calc_rate() > self.rate_limit:
                    self.cond.wait()
        elif rate<self.rate_limit:
            print('Controller, rate is low, sleep less by',rate,sleep_
diff)
            # Decrease sleep time
            sleep_time = thread.sleep_time
            sleep_time -= sleep_diff
            # If this goes off < zero, make it zero
            thread.sleep_time = max(0, sleep_time)
```

Let's discuss the preceding code before we discuss the changes in the producer class that will make use of this class:

1. The class is an instance of `Thread`, so it runs in its own thread of execution. It also holds a `Condition` object.
2. It has a `calc_rate` method, which calculates the rate of generation of URLs by keeping a counter and using timestamps.
3. In the `run` method, the rate is checked. If it's below the configured limit, the condition object notifies all threads waiting on it.
4. Most importantly, it implements a `throttle` method. This method uses the current rate, calculated via `calc_rate`, and uses it to throttle and adjust the sleep times of the producers. It mainly does these two things:
    1. If the rate is more than the configured limit, it causes the calling thread to wait on the condition object until the rate levels off. It also calculates an extra sleep time that the thread should sleep in its loop to adjust the rate to the required level.
    2. If the rate is less than the configured limit, then the thread needs to work faster and produce more data, so it calculates the sleep difference and lowers the sleep limit accordingly.

Here is the code of the producer class to incorporate the changes:

```
class ThumbnailURL_Generator(threading.Thread):
    """ Worker class that generates image URLs and supports throttling
         via an external controller """

    def __init__(self, queue, controller=None, sleep_time=1):
        self.sleep_time = sleep_time
        self.queue = queue
        # A flag for stopping
        self.flag = True
        # sizes
        self._sizes = (240,320,360,480,600,720)
        # URL scheme
        self.url_template = 'https://dummyimage.com/%s/%s/%s.jpg'
        # Rate controller
        self.controller = controller
        # Internal id
        self._id = uuid.uuid4().hex
        threading.Thread.__init__(self, name='Producer-'+ self._id)

    def __str__(self):
        return 'Producer-'+self._id

    def get_size(self):
        return '%dx%d' % (random.choice(self._sizes),
                          random.choice(self._sizes))

    def get_color(self):
        return ''.join(random.sample(string.hexdigits[:-6], 3))

    def run(self):
        """ Main thread function """

        while self.flag:
            # generate image URLs of random sizes and fg/bg colors
            url = self.url_template % (self.get_size(),
                                       self.get_color(),
                                       self.get_color())
            # Add to queue
            print(self,'Put',url)
            self.queue.put(url)
            self.controller.increment()
```

```
            # Throttle after putting a few images
            if self.controller.count>5:
                self.controller.throttle(self)

            time.sleep(self.sleep_time)

    def stop(self):
        """ Stop the thread """

        self.flag = False
```

Let's see how the preceding code works:

1. The class now accepts an additional controller object in its initializer. This is the instance of the controller class given earlier.
2. After putting a URL, it increments the count on the controller. Once the count reaches a minimum limit (set as 5 to avoid early throttling of the producers), it calls `throttle` on the controller, passing itself as the argument.

The calling code also needs quite a few changes. The modified code is shown as follows:

```
    q = Queue(maxsize=2000)
    # The controller needs to be configured with exact number of
    # producers
    controller = ThumbnailURLController(rate_limit=50, nthreads=3)
    saver = ThumbnailImageSemaSaver(limit=200)

    controller.start()

    producers, consumers = [], []
    for i in range(3):
        t = ThumbnailURL_Generator(q, controller)
        producers.append(t)
        t.start()

    for i in range(5):
        t = ThumbnailURL_Consumer(q, saver)
        consumers.append(t)
        t.start()

    for t in consumers:
        t.join()
```

```
            print('Joined', t, flush=True)

        # To make sure producers dont block on a full queue
        while not q.empty():
            item=q.get()
        controller.stop()

        for t in producers:
            t.stop()
            print('Stopped',t, flush=True)


        print('Total number of PNG images',len(glob.glob('*.png')))
```

The main changes here are the ones listed next:

1. The controller object is created with the exact number of producers that will be created. This helps the correct calculation of sleep time per thread.
2. The producer threads, themselves, are passed the instance of the controller in their initializer.
3. The controller is started as a thread before all other threads.

Here is a run of the program configured with 200 images at the rate of 50 images per minute. We show two images of the running program's output, one at the beginning of the program and one towards the end.

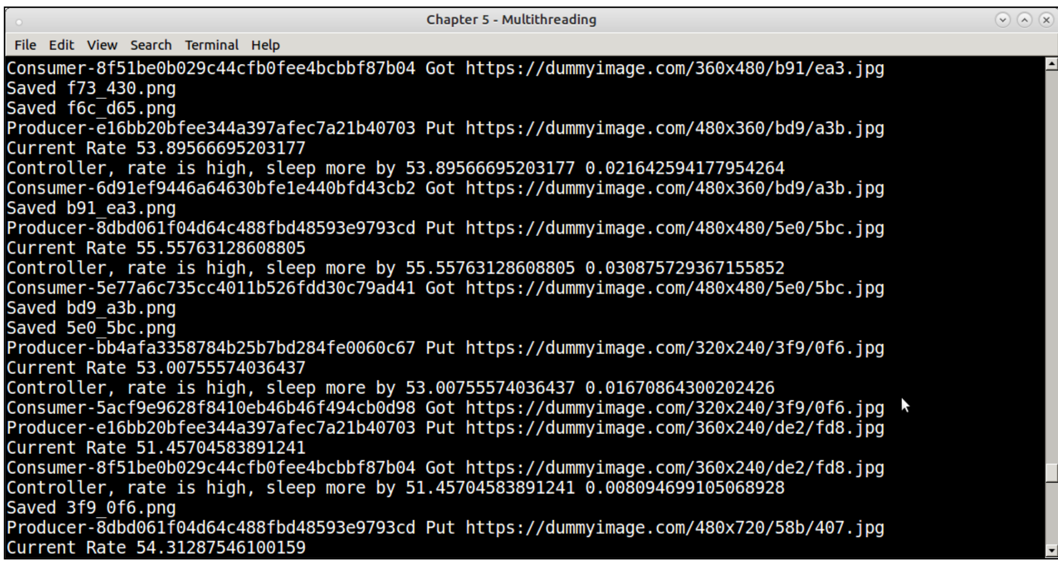
```
Chapter 5 - Multithreading
File  Edit  View  Search  Terminal  Help
Producer-e16bb20bfee344a397afec7a21b40703 Put https://dummyimage.com/240x600/ac4/732.jpg
Consumer-5e77a6c735cc4011b526fdd30c79ad41 Got https://dummyimage.com/240x600/ac4/732.jpg
Producer-8dbd061f04d64c488fbd48593e9793cd Put https://dummyimage.com/480x360/6a5/dc0.jpg
Producer-bb4afa3358784b25b7bd284fe0060c67 Put https://dummyimage.com/240x240/417/bd1.jpg
Current Rate 340.09723953422684
Controller, rate is high, sleep more by 340.09723953422684 1.6116513307457048
Producer-e16bb20bfee344a397afec7a21b40703 Put https://dummyimage.com/480x480/7e4/fde.jpg
Current Rate 206.21639516208333
Controller, rate is high, sleep more by 206.21639516208333 0.8678688620115741
Producer-8dbd061f04d64c488fbd48593e9793cd Put https://dummyimage.com/360x720/e98/3f9.jpg
Current Rate 233.2954122890928
Controller, rate is high, sleep more by 233.2954122890928 1.0183078460505155
Saved 1af_516.png
Consumer-5acf9e9628f8410eb46b46f494cb0d98 Got https://dummyimage.com/480x360/6a5/dc0.jpg
Saved 6ef_189.png
Consumer-6d91ef9446a64630bfe1e440bfd43cb2 Got https://dummyimage.com/240x240/417/bd1.jpg
Saved ac4_732.png
Saved 61f_df6.png
Consumer-8f51be0b029c44cfb0fee4bcbbf87b04 Got https://dummyimage.com/480x480/7e4/fde.jpg
Consumer-5e77a6c735cc4011b526fdd30c79ad41 Got https://dummyimage.com/360x720/e98/3f9.jpg
Saved e98_3f9.png
Saved 6a5_dc0.png
Saved 7e4_fde.png
Saved 417_bd1.png
Producer-e16bb20bfee344a397afec7a21b40703 Put https://dummyimage.com/720x720/f73/430.jpg
```

Starting the thumbnail program with URL rate controller—at 50 URLs per minute

You will find that, when the program starts, it almost immediately slows down, and nearly comes to a halt, since the original rate is high. What happens here is that the producers call on the `throttle` method, and since the rate is high, they all get blocked on the condition object.

After a few seconds, the rate comes down to the prescribed limit, since no URLs are generated. This is detected by the controller in its loop, and it calls `notify_all` on the threads, waking them up.

After a while you will see that the rate is getting settled around the set limit of 50 URLs per minute.



The thumbnail program with URL rate controller 5-6 seconds after start

Towards the end of the program, you will see that the rate has almost settled to the exact limit:

```
Chapter 5 - Multithreading
File  Edit  View  Search  Terminal  Help
Consumer-c2a30ea8d7494631a8203f5c90abea50 Got https://dummyimage.com/320x240/92f/f1c.jpg
Semaphore limit reached, returning False
Consumer-c2a30ea8d7494631a8203f5c90abea50 Set limit reached, quitting
Producer-ace34b3fc9644cd0a5760373db9aa0fa Put https://dummyimage.com/720x480/0a1/c69.jpg
Current Rate 50.15528364068437
Consumer-e0c93e3825df407a9d452b993eedd7ca Got https://dummyimage.com/720x480/0a1/c69.jpg
Controller, rate is high, sleep more by 50.15528364068437 0.0008626868926909263
Semaphore limit reached, returning False
Consumer-e0c93e3825df407a9d452b993eedd7ca Set limit reached, quitting
Producer-198415b202dc4b979d68d423c6ba0f16 Put https://dummyimage.com/360x720/7ad/6f4.jpg
Current Rate 50.08487818479042
Controller, rate is high, sleep more by 50.08487818479042 0.0004715454710578844 6
Consumer-d82287478e0d4a07afa4085b0a666160 Got https://dummyimage.com/360x720/7ad/6f4.jpg
Semaphore limit reached, returning False
Consumer-d82287478e0d4a07afa4085b0a666160 Set limit reached, quitting
Joined Consumer-d82287478e0d4a07afa4085b0a666160
Joined Consumer-1cbb1331c886469abd9892b5219abb9e
Joined Consumer-e0c93e3825df407a9d452b993eedd7ca
Joined Consumer-c2a30ea8d7494631a8203f5c90abea50
Stopping controller
Stopping producers...
Stopped Producer-198415b202dc4b979d68d423c6ba0f16
Stopped Producer-bd2baedf319e45298bc320a1346a46c6
Stopped Producer-ace34b3fc9644cd0a5760373db9aa0fa
Total number of PNG images 200
```

The thumbnail program with URL rate controller towards the end

We are coming towards the end of our discussion on threading primitives and how to use them in improving the concurrency of your programs and in implementing shared resource constraints and controls.

Before we conclude, we will look at an aspect of Python threads which prevents multi-threaded programs from making full use of the CPU in Python—namely the GIL or Global Interpreter Lock.

Multithreading – Python and GIL

In Python there is, a global lock that prevents multiple threads from executing native bytecode at once. This lock is required, since the memory management of CPython (the native implementation of Python) is not thread-safe.

This lock is called **Global Interpreter Lock** or just **GIL**.

Python cannot execute bytecode operations concurrently on CPUs due to the GIL. Hence, Python becomes almost unsuitable for the following cases:

- When the program depends on a number of heavy bytecode operations, which it wants to run concurrently
- When the program uses multithreading to utilize the full power of multiple CPU cores on a single machine

I/O calls and long-running operations typically occur outside the GIL. Therefore, multithreading is efficient in Python only when it involves some amount of I/O or such operations – such as image processing.

In such cases, scaling your program to concurrently scale beyond a single process becomes a handy approach. Python makes this possible via its `multiprocessing` module, which is our next topic of discussion.

Concurrency in Python – multiprocessing

The Python standard library provides a multiprocessing module, which allows a programmer to write programs that scale concurrently using multiple processes instead of threads.

Since multiprocessing scales computation across multiple processes, it effectively removes any issues with the GIL in Python. Programs can make use of multiple CPU cores efficiently using this module.

The main class exposed by this module is the `Process` class, the analog to the `Thread` class in the threading module. It also provides a number of synchronization primitives, which are almost exact counterparts of their cousins in the threading module.

We will get started by using an example using the `Pool` object provided by this module. It allows a function to execute in parallel over multiple inputs using processes.

A primality checker

The following function is a simple checker function for primality, that is, whether the input number is prime or not:

```
def is_prime(n):
    """ Check for input number primality """

    for i in range(3, int(n**0.5+1), 2):
```

```
            if n % i == 0:
                print(n,'is not prime')
                return False

        print(n,'is prime')
        return True
```

The following is a threaded class that uses this last function to check numbers from a queue for primality:

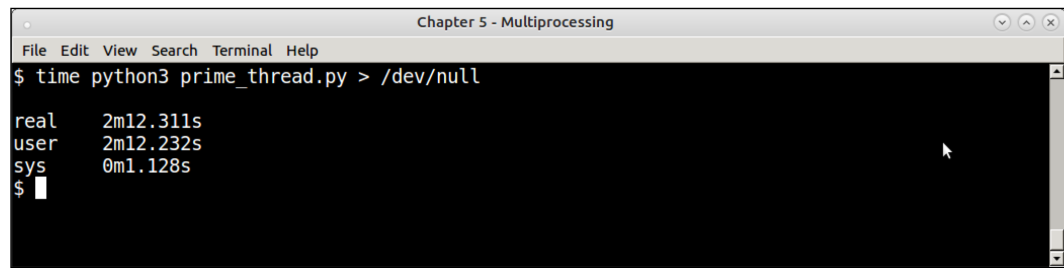
```
# prime_thread.py
import threading

class PrimeChecker(threading.Thread):
    """ Thread class for primality checking """

    def __init__(self, queue):
        self.queue = queue
        self.flag = True
        threading.Thread.__init__(self)

    def run(self):

        while self.flag:
            try:
                n = self.queue.get(timeout=1)
                is_prime(n)
            except Empty:
                break
```

We will test it with 1,000 large prime numbers. In order to save space for the list represented here, what we've done is to take 10 of these numbers and multiply the list with 100:

```
    numbers = [1297337, 1116281, 104395303, 472882027, 533000389,
               817504243, 982451653, 112272535095293, 115280095190773,
               1099726899285419]*100

    q = Queue(1000)

    for n in numbers:
        q.put(n)

    threads = []
    for i in range(4):
```

```
            t = PrimeChecker(q)
            threads.append(t)
            t.start()

        for t in threads:
            t.join()
```

We've used four threads for this test. Let's see how the program performs, in the following screenshot:

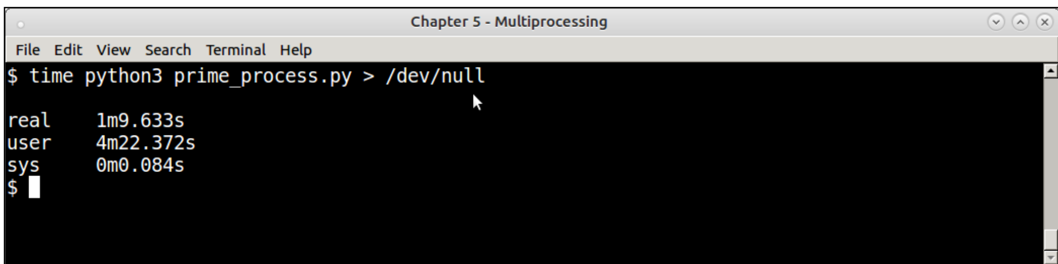


Primality checker of 1,000 numbers using a pool of 4 threads

Now, here is the equivalent code using the multiprocessing `Pool` object:

```
        numbers = [1297337, 1116281, 104395303, 472882027, 533000389,
                   817504243, 982451653, 112272535095293, 115280095190773,
                   1099726899285419]*100
        pool = multiprocessing.Pool(4)
        pool.map(is_prime, numbers)
```

The following screenshot shows its performance over the same set of numbers:



Primality checker of 1,000 numbers using a multiprocessing Pool of 4 processes

We learn the following by comparing these numbers:

1. The real time, that is, the wall clock time spent by the process pool version at 1 minute 9.6 seconds (69.6 seconds) is nearly 50% lesser than that of the thread pool version at 2 minute 12 seconds (132 seconds).
2. However, notice that the user time—that is, the time spent inside the CPU for user code—for the process pool version at 4 minute 22 seconds (262 seconds) is nearly two times more than that of the thread pool version at 2 minutes 12 seconds (132 seconds).
3. The real and user CPU time of the thread pool version is exactly the same at 2 minutes 12 seconds. This is a clear indication that the threaded version was able to execute effectively, only in one of the CPU cores.

This means that the process pool version was able to better make use of all the CPU cores, since, for the 50% of the real time of the thread pool version, it was able to make use of the CPU time twice over.

Hence, the real performance boost in terms of CPU time/real time for the two programs is as follows:

1. Threaded version → 132 seconds/132 seconds = 1
2. Process version → 262 seconds/69.6 seconds = 3.76 ~= 4

The real performance ratio of the process version to the threaded version is, hence, given as follows:

4/1 = 4

The machine on which the program was executed has a four-core CPU. This clearly shows that the multiprocess version of the code was able to utilize all the four cores of the CPU nearly equally.

This is because the threaded version is being restricted by the GIL, whereas the process version has no such restriction and can freely make use of all the cores.

In the next section, let's move on to a more involved problem—that of sorting disk-based files.

Sorting disk files

Imagine you have hundreds of thousands of files on the disk, each containing a certain fixed number of integers in a given range. Let's say we need the files to be sorted and merged into a single file.

If we decide to load all this data into memory, it will need large amounts of RAM. Let's do a quick calculation for a million files, each containing around 100 integers in the range of 1 to 10,000 for a total of 100,000,000 or 100 million integers.

Let's assume each of the files is loaded as a list of integers from the disk—we will ignore string processing, and the like for the time being.

Using `sys.getsizeof`, we can get a rough calculation going:

```
>>> sys.getsizeof([100000]*1000)*100000/(1024.0*1024.0)
769.04296875
```

So, the entire data will take close to 800 MB if loaded into memory at once. Now this may not look like a large memory footprint at first, but the larger the list, the more system resources it takes to sort it in memory as one large list.

Here is the simplest code for sorting of all the integers present in the disk files after loading them into memory:

```
# sort_in_memory.py
import sys

all_lists = []

for i in range(int(sys.argv[1])):
    num_list = map(int, open('numbers/numbers_%d.txt' %
i).readlines())
    all_lists += num_list

print('Length of list',len(all_lists))
print('Sorting...')
all_lists.sort()
open('sorted_nums.txt','w').writelines('\n'.join(map(str, all_lists))
+ '\n')
print('Sorted')
```

This preceding code loads a certain number of files from the disk, each containing 100 integers in the range 1 to 10,000. It reads each file, maps it to a list of integers, and adds each list to a cumulative list. Finally, the list is sorted and written to a file.

The following table shows the time taken to sort a certain number of disk files:

Number of files (n)	Time taken for sorting
1000	17.4 seconds
10000	101 seconds
100000	138 seconds
1000000	NA

As you can see, the time taken scales pretty reasonably—less than *O(n)*. However, this is one problem where more than the time, it is the space—in terms of memory and operations on it—that matters.

For example, in the machine that was used to conduct the test, an 8 -GB RAM, 4-core CPU laptop with 64-bit Linux, the test with a million numbers didn't finish. Instead, it caused the system to hang, so it was not completed.

Sorting disk files – using a counter

If you look at the data, you find that there is an aspect that allows us to treat the problem as more about space than time. This is the observation that the integers are in a fixed range with a maximum limit of 10,000.

Hence, instead of loading all the data as separate lists and merging them, one can use a data structure like a counter.

Here is the basic idea of how this works:

1. Initialize a data structure—a counter, where each integer starts from 1… 10,000 the maximum entry is initialized to zero.
2. Load each file and convert the data to a list. For any number found in the list, increment its count in the counter data structure initialized in Step 1.
3. Finally, loop through the counter, and output each number with a count greater than zero *so many times*, and save the output to a file. The output is your merged and sorted single file:

```
# sort_counter.py
import sys
import collections

MAXINT = 100000

def sort():
    """ Sort files on disk by using a counter """
```

```
counter = collections.defaultdict(int)
for i in range(int(sys.argv[1])):
filename = 'numbers/numbers_%d.txt' % i
for n in open(filename):
counter[n] += 1
print('Sorting...')

with open('sorted_nums.txt','w') as fp:
for i in range(1, MAXINT+1):
    count = counter.get(str(i) + '\n', 0)
if count>0:
fp.write((str(i)+'\n')*count)

print('Sorted')
```

In the preceding code, we use a `defaultdict` from the collections module as the counter. Whenever we encounter an integer, we increment its count. In the end, the counter is looped through, and each item is output as many times as it was found.

The sort and merge happen due to the way we have converted the problem from one of sorting integers to one of keeping a count and outputting in a naturally sorted order.

The following table summarizes the time taken for the sorting of numbers against the size of the input – in terms of number of disk files:

Number of files (n)	**Time taken for sorting**
1000	16.5 seconds
10000	83 seconds
100000	86 seconds
1000000	359 seconds

Though the performance for the smallest case – that of 1,000 files is similar to that for the in-memory sort, the performance becomes better as the size of the input increases. This code also manages to finish the sorting of a million files or 100 million integers - in about 5m 59s.

In timing measurements for processes that read files, there is always the effect of buffer caches in the kernel. You will find that running the same performance test successively shows a tremendous improvement, as Linux caches the contents of the files in its buffer cache. Hence, subsequent tests for the same input size should be done after clearing the buffer cache. In Linux, this can be done by the following command:

```
$ echo 3 > /proc/sys/vm/drop_caches
```

In our tests for successive numbers, we *don't* reset the buffer caches as shown before. This means that runs for higher numbers enjoy a performance boost from the caches created during the previous runs. However, since this is done uniformly for each test, the results are comparable. The cache is reset before starting the test suite for a specific algorithm.

This algorithm also requires much less memory, since, for each run, the memory requirements are *the same* since we are using an array of integers up to MAXINT and just incrementing the count.

Here is the memory usage of the sort in-memory program for 100,000 files using the `memory_profiler`, which we have encountered in the previous chapter.

```
Chapter 5 - Multiprocessing
File Edit View Search Terminal Help
$ python3 -m memory_profiler sort_in_memory.py 100000
Length of list 10000000
Sorting...
Sorted
Filename: sort_in_memory.py

Line #    Mem usage    Increment   Line Contents
================================================
     5   31.328 MiB    0.000 MiB   @profile
     6                             def sort():
     7   31.328 MiB    0.000 MiB       all_lists = []
     8
     9  447.043 MiB  415.715 MiB       for i in range(int(sys.argv[1])):
    10  447.043 MiB    0.000 MiB           num_list = map(int, open('numbers/numbers_%d.txt
' % i).readlines())
    11  447.043 MiB    0.000 MiB           all_lists += num_list
    12
    13  447.043 MiB    0.000 MiB       print('Length of list',len(all_lists))
    14  447.043 MiB    0.000 MiB       print('Sorting...')
    15  456.887 MiB    9.844 MiB       all_lists.sort()
    16  465.199 MiB    8.312 MiB       open('sorted_nums.txt','w').writelines('\n'.join(map
(str, all_lists)) + '\n')
    17  465.199 MiB    0.000 MiB       print('Sorted')
```

Memory usage of in-memory sort program for an input of 100,000 files

The following screenshot shows the memory usage for the sort counter for the same number of files:

```
Chapter 5 - Multiprocessing
File  Edit  View  Search  Terminal  Help
$ python3 -m memory_profiler sort_counter.py 100000
Sorting...
Sorted
Filename: sort_counter.py

Line #    Mem usage    Increment   Line Contents
================================================
     7   31.305 MiB    0.000 MiB   @profile
     8                             def sort():
     9
    10   31.305 MiB    0.000 MiB       counter = collections.defaultdict(int)
    11
    12   69.836 MiB   38.531 MiB       for i in range(int(sys.argv[1])):
    13   69.836 MiB    0.000 MiB           num_list = map(int, open('numbers/numbers_%d.txt' % i
).readlines())
    14   69.836 MiB    0.000 MiB           for n in num_list:
    15   69.836 MiB    0.000 MiB               counter[n] += 1
    16
    17
    18   69.836 MiB    0.000 MiB       print('Sorting...')
    19   69.836 MiB    0.000 MiB       with open('sorted_nums.txt','w') as fp:
    20   69.836 MiB    0.000 MiB           for i in range(1, MAXINT+1):
    21   69.836 MiB    0.000 MiB               count = counter.get(i, 0)
    22   69.836 MiB    0.000 MiB               if count>0:
    23   69.836 MiB    0.000 MiB                   fp.write((str(i)+'\n')*count)
    24   69.836 MiB    0.000 MiB       print('Sorted')
```

Memory usage of counter sort program for an input of 100,000 files

The memory usage of the in-memory sort program at 465 MB is more than six times that of the counter sort program at 70 MB. Also note, that the sorting operation itself takes extra memory of nearly 10 MB in the in-memory version.

Sorting disk files – using multiprocessing

In this section, we rewrite the counter sorting program using multiple processes. The approach is to scale the processing input files for more than one process by splitting the list of file paths to a pool of processes – and planning to take advantage of the resulting data parallelism.

Here is the rewrite of the code:

```
# sort_counter_mp.py
import sys
import time
import collections
from multiprocessing import Pool

MAXINT = 100000

def sorter(filenames):
```

```
    """ Sorter process sorting files using a counter """

    counter = collections.defaultdict(int)

    for filename in filenames:
for i in open(filename):
counter[i] += 1

return counter

def batch_files(pool_size, limit):
""" Create batches of files to process by a multiprocessing Pool """
batch_size = limit // pool_size

filenames = []

for i in range(pool_size):
batch = []
for j in range(i*batch_size, (i+1)*batch_size):
filename = 'numbers/numbers_%d.txt' % j
batch.append(filename)

filenames.append(batch)

return filenames

def sort_files(pool_size, filenames):
""" Sort files by batches using a multiprocessing Pool """

with Pool(pool_size) as pool:
counters = pool.map(sorter, filenames)
with open('sorted_nums.txt','w') as fp:
for i in range(1, MAXINT+1):
count = sum([x.get(str(i)+'\n',0) for x in counters])
if count>0:
fp.write((str(i)+'\n')*count)
print('Sorted')
if __name__ == "__main__":
limit = int(sys.argv[1])
pool_size = 4
filenames = batch_files(pool_size, limit)
sort_files(pool_size,
```

It is exactly the same code as earlier with the following changes:

1. Instead of processing all the files as a single list, the filenames are put in batches, with batches equaling the size of the pool.
2. We use a sorter function, which accepts the list of filenames, processes them, and returns a dictionary with the counts.
3. The counts are summed for each integer in the range from `1` to `MAXINT`, and so many numbers are written to the sorted file.

The following table shows the data for processing a different number of files for pool sizes of 2 and 4 respectively:

Number of files (n)	Pool size	Time taken for sorting
1,000	2	18 seconds
	4	20 seconds
10,000	2	92 seconds
	4	77 seconds
100,000	2	96 seconds
	4	86 seconds
1,000,000	2	350 seconds
	4	329 seconds

The numbers tell an interesting story:

1. The multiple process version one with 4 processes (equal to number of cores in the machine) has better numbers overall when compared to the one with 2 processes and the single process one.
2. However, the multiple-process version doesn't seem to offer much of a performance benefit when compared to the single-process version. The performance numbers are very similar and any improvement is within bounds of error and variation. For example, for 1 million number input the multiple process with 4 processes has just an 8% improvement over the single-process one.
3. This is because the bottleneck here is the processing time it takes to load the files into memory—in file I/O—not the computation (sorting), as the sorting is just an increment in the counter. Hence the single process version is pretty efficient as it is able to load all the file data in the same address space. The multiple-process ones are able to improve this a bit by loading the files in multiple address spaces, but not by a lot.

This example shows that, in situations where there is not much computation done but the bottleneck is disk or file I/O, the impact of scaling by multiprocessing is much less.

Multithreading versus multiprocessing

Now that we have come to the end of our discussion on multiprocessing, it is a good time to compare and contrast the scenarios where one needs to choose between scaling using threads in a single process or using multiple processes in Python.

Here are some guidelines.

Use multithreading in the following cases:

1. The program needs to maintain a lot of shared states, especially mutable ones. A lot of the standard data structures in Python, such as lists, dictionaries, and others, are thread-safe, so it costs much less to maintain a mutable shared state using threads than via processes.
2. The program needs to keep a low memory foot-print.
3. The program spends a lot of time doing I/O. Since the GIL is released by threads doing I/O, it doesn't affect the time taken by the threads to perform I/O.
4. The program doesn't have a lot of data-parallel operations which it can scale across multiple processes

Use multiprocessing in these scenarios:

- The program performs a lot of CPU-bound heavy computing such as byte-code operations, number crunching, and the like on reasonably large inputs.
- The program has inputs which can be parallelized into chunks and whose results can be combined afterwards—in other words, the input of the program yields well to data-parallel computations.
- The program doesn't have any limitations on memory usage, and you are on a modern machine with a multicore CPU and large enough RAM.
- There is not much shared mutable state between processes that need to be synchronized—this can slow down the system, and offset any benefits gained from multiple processes.
- Your program is not heavily dependent on I/O—file or disk I/O or socket I/O.

Concurrecy in Python – Asynchronous Execution

We have seen two different ways to perform concurrent execution using multiple threads and multiple processes. We saw different examples of using threads and their synchronization primitives. We also saw a couple of examples using multi-processing with slightly varied outcomes.

Apart from these two ways to do concurrent programming, another common technique is that of asynchronous programming or asynchronous I/O.

In an asynchronous model of execution, tasks are picked to be executed from a queue of tasks by a scheduler, which executes these tasks in an interleaved manner. There is no guarantee that the tasks will be executed in any specific order. The order of execution of tasks depends upon how much processing time a task is willing to *yield* to another task in the queue. Put in other words, asynchronous execution happens through co-operative multitasking.

Asynchronous execution usually happens in a single thread. This means no true data parallelism or true parallel execution can happen. Instead, the model only provides a semblance of parallelism.

As execution happens out of order, asynchronous systems need a way to return the results of function execution to the callers. This usually happens with *callbacks*, which are functions to be called when the results are ready or using special objects that receive the results, often called *futures*.

Python 3 provides support for this kind of execution via its *asyncio* module using coroutines. Before we go on to discuss this, we will spend some time understanding pre-emptive multitasking versus cooperative multitasking, and how we can implement a simple cooperative multitasking scheduler in Python using generators.

Pre-emptive versus cooperative multitasking

The programs we wrote earlier using multiple threads were examples of concurrency. However, we didn't have to worry about how and when the operating system chose to run the thread—we just had to prepare the threads (or processes), provide the target function, and execute them. The scheduling is taken care of by the operating system.

Every few ticks of the CPU clock, the operating system pre-empts a running thread, and replaces it with another one in a particular core. This can happen due to different reasons, but the programmer doesn't have to worry about the details. He just creates the threads, sets them up with the data they need to process, uses the correct synchronization primitives, and starts them. The operating system does the rest including switching and scheduling.

This is how almost all modern operating systems work. It guarantees each thread a fair share of the execution time, all other things being equal. This is known as **pre-emptive multitasking**.

There is another type of scheduling which is the opposite of pre-emptive multitasking. This is called as co-operative multitasking, where the operating system plays no role in deciding the priority and execution of competing threads or processes. Instead, a process or thread willingly yields control for another process or thread to run. Alternatively, a thread can replace another thread which is idling (sleeping) or waiting for I/O.

This is the technique used in the asynchronous model of concurrent execution using co-routines. A function, while waiting for data, say a call on the network that is yet to return, can yield control for another function or task to run.

Before we go to discuss actual co-routines using `asyncio` let's write our own co-operative multitasking scheduler using simple Python generators. It is not very difficult to do this as you can see below.

```
# generator_tasks.py
import random
import time
import collections
import threading

def number_generator(n):
    """ A co-routine that generates numbers in range 1..n """

    for i in range(1, n+1):
        yield i

def square_mapper(numbers):
    """ A co-routine task for converting numbers to squares """

    for n in numbers:
        yield n*n

def prime_filter(numbers):
```

```
    """ A co-routine which yields prime numbers """

    primes = []
    for n in numbers:
        if n % 2 == 0: continue
        flag = True
        for i in range(3, int(n**0.5+1), 2):
            if n % i == 0:
                flag = False
                break

        if flag:
            yield n

def scheduler(tasks, runs=10000):
    """ Basic task scheduler for co-routines """

    results = collections.defaultdict(list)

    for i in range(runs):
        for t in tasks:
            print('Switching to task',t.__name__)
            try:
                result = t.__next__()
                print('Result=>',result)
                results[t.__name__].append(result)
            except StopIteration:
                break

    return results
```

Let's analyze the preceding code:

- We have four functions—three generators, since they use the `yield` keyword to return the data, and a scheduler, which runs a certain set of tasks
- The `square_mapper` function accepts an iterator, which returns integers iterating through it, and yields the squares of the members
- The `prime_filter` function accepts a similar iterator, and filters out numbers that are not prime, yielding only prime numbers
- The `number_generator` function acts as the input iterator to both these functions, providing them with an input stream of integers

Let's now look at the calling code which ties all the four functions together.

```
import sys

tasks = []
start = time.clock()

limit = int(sys.argv[1])

# Append sqare_mapper tasks to list of tasks
tasks.append(square_mapper(number_generator(limit)))
# Append prime_filter tasks to list of tasks
tasks.append(prime_filter(number_generator(limit)))

results = scheduler(tasks, runs=limit)
print('Last prime=>',results['prime_filter'][-1])
end = time.clock()
print('Time taken=>',end-start)
```

Here is an analysis of the calling code:

- The number generator is initialized with a count, which is received via the command-line argument. It is passed to the `square_mapper` function. The combined function is added as a task to the `tasks` list.
- A similar operation is performed for the `prime_filter` function.
- The `scheduler` method is run by passing the task list to it, which it runs by iterating through a `for` loop, running each task one after another. The results are appended to a dictionary using the function's name as the key, and returned at the end of execution.
- We print the last prime number's value to verify correct execution, and also the time taken for the scheduler to process.

Let's see the output of our simple cooperative multitasking scheduler for a limit of `10`. This allows us to capture all the input in a single command window, as seen in the following screenshot:

```
$ python3 generator_tasks.py  10
Switching to task square_mapper
Result=> 1
Switching to task prime_filter
Result=> 1
Switching to task square_mapper
Result=> 4
Switching to task prime_filter
Result=> 3
Switching to task square_mapper
Result=> 9
Switching to task prime_filter
Result=> 5
Switching to task square_mapper
Result=> 16
Switching to task prime_filter
Result=> 7
Switching to task square_mapper
Result=> 25
Switching to task prime_filter
Switching to task square_mapper
Result=> 36
Switching to task prime_filter
Switching to task square_mapper
Result=> 49
Switching to task prime_filter
Switching to task square_mapper
Result=> 64
Switching to task prime_filter
Switching to task square_mapper
Result=> 81
Switching to task prime_filter
Switching to task square_mapper
Result=> 100
Switching to task prime_filter
Last prime=> 7
Time taken=> 0.00021399999999999197
$
```

Output of the simple co-operative multitasking program example for an input of 10

Let's analyze the output:

1. The output of the `square_mapper` and `prime_filter` functions alternates on the console. This is because the scheduler switches between them in the `for` loop. Each of the functions are co-routines (generators) so they *yield* execution—that is the control is passed from one function to the next—and vice-versa. This allows both functions to run concurrently, while maintaining state and producing output.
2. Since we used generators here, they provide a natural way of generating the result plus yielding control in one go, using the *yield* keyword.

The asyncio module in Python

The `asyncio` module in Python provides support for writing concurrent, single-threaded programs using co-routines. It is available only in Python 3.

A co-routine using the `asyncio` module is one that uses either of the following approaches:

- Using the `async def` statement for defining functions
- Being decorated using the `@asyncio.coroutine` expression

Generator-based co-routines use the second technique, and they yield from expressions.

Co-routines created using the first technique typically use the `await <future>` expression to wait for the future to be completed.

Co-routines are scheduled for execution using an `event` loop, which connects the objects and schedules them as tasks. Different types of event loop are provided for different operating systems.

The following code rewrites our earlier example of a simple cooperative multitasking scheduler to use the `asyncio` module:

```
# asyncio_tasks.py
import asyncio

def number_generator(m, n):
    """ A number generator co-routine in range(m...n+1) """
    yield from range(m, n+1)

async prime_filter(m, n):
    """ Prime number co-routine """

    primes = []
    for i in number_generator(m, n):
        if i % 2 == 0: continue
        flag = True

        for j in range(3, int(i**0.5+1), 2):
            if i % j == 0:
                flag = False
                break

        if flag:
```

```
print('Prime=>',i)
primes.append(i)

# At this point the co-routine suspends execution
# so that another co-routine can be scheduled
await asyncio.sleep(1.0)
return tuple(primes)

async def square_mapper(m, n):
""" Square mapper co-routine """
squares = []

for i in number_generator(m, n):
print('Square=>',i*i)
squares.append(i*i)
# At this point the co-routine suspends execution
# so that another co-routine can be scheduled
await asyncio.sleep(1.0)
return squares

def print_result(future):
print('Result=>',future.result())
```

Here is how the preceding code works:

1. The `number_generator` function is a co-routine that yields from the sub-generator `range(m, n+1)`, which is an iterator. This allows this co-routine to be called in other co-routines.
2. The `square_mapper` function is a co-routine of the first type using the `async def` keyword. It returns a list of squares using numbers from the number generator.
3. The `prime_filter` function is of the same type. It also uses the number generator, and appends prime numbers to a list and returns it.
4. Both co-routines yield to the other by sleeping, using the *asyncio.sleep* function and waiting on it. This allows both co-routines to work concurrently in an interleaved fashion.

Here is the calling code with the `event` loop and the rest of the plumbing:

```
loop = asyncio.get_event_loop()
future = asyncio.gather(prime_filter(10, 50), square_mapper(10, 50))
future.add_done_callback(print_result)
loop.run_until_complete(future)

loop.close()
```

Here is the output of the program. Observe how the results of each of the tasks is being printed in an interleaved fashion.

```
Chapter 5 - Asynchronous Execution
(env) anand@mangu-probook:/home/user/programs/chap5$ python3 asyncio_tasks.py
Prime=> 11
Square=> 100
Prime=> 13
Square=> 121
Square=> 144
Prime=> 17
Square=> 169
Prime=> 19
Square=> 196
Square=> 225
Prime=> 23
Square=> 256
Square=> 289
Square=> 324
Prime=> 29
Square=> 361
Prime=> 31
Square=> 400
Square=> 441
Square=> 484
Prime=> 37
Square=> 529
Square=> 576
Prime=> 41
Square=> 625
Prime=> 43
Square=> 676
```

Result of executing the asyncio task calculating prime numbers and squares

Let's analyze how the preceding code worked line by line, while following a top-to-bottom approach:

1. We first get an asyncio event `loop` using the `factory` function `asyncio.get_event_loop`. This returns the default event loop implementation for the operating system.
2. We set up an asyncio `future` object by using the `gather` method of the module. This method is used to aggregate results from a set of co-routines or futures passed as its argument. We pass both the `prime_filter` and the `square_mapper` to it.
3. A callback is added to the `future` object—the `print_result` function. It will be automatically called once the future's execution is completed.
4. The loop is run until the future's execution is completed. At this point, the callback is called and it prints the result. Note how the output appears interleaved—as each task yields to the other one using the *sleep* function of the asyncio module.
5. The loop is closed and terminates is operation.

Waiting for a future – async and await

We discussed how one could wait for data from a future inside a co-routine using await. We saw an example that uses await to yield control to other co-routines. Let's now look at an example that waits for I/O completion on a future, which returns data from the web.

For this example, you need the `aiohttp` module which provides an HTTP client and server to work with the asyncio module and supports futures. We also need the `async_timeout` module which allows timeouts on asynchronous co-routines. Both these modules can be installed using pip.

Here is the code—this is a co-routine that fetches a URL using a timeout and awaits the future, that is, the result of the operation:

```
# async_http.py
import asyncio
import aiohttp
import async_timeout

@asyncio.coroutine
def fetch_page(session, url, timeout=60):
""" Asynchronous URL fetcher """

with async_timeout.timeout(timeout):
response = session.get(url)
return response
```

The following is the calling code with the event loop:

```
loop = asyncio.get_event_loop()
urls = ('http://www.google.com',
        'http://www.yahoo.com',
        'http://www.facebook.com',
        'http://www.reddit.com',
        'http://www.twitter.com')

session = aiohttp.ClientSession(loop=loop)
tasks = map(lambda x: fetch_page(session, x), urls)
# Wait for tasks
done, pending = loop.run_until_complete(asyncio.wait(tasks,
                                        timeout=120))
loop.close()

for future in done:
```

```
        response = future.result()
        print(response)
        response.close()
        session.close()

    loop.close()
```

What are we doing in the preceding code?

1. We create an event loop and a list of URLs to be fetched. We also create an instance of `aiohttp ClientSession` object which is a helper for fetching URLs.
2. We create a map of tasks by mapping the `fetch_page` function to each of the URLs. The session object is passed as first argument to the `fetch_page` function.
3. The tasks are passed to the wait method of `asyncio` with a timeout of `120` seconds.
4. The loop is run until complete. It returns two sets of futures—`done` and `pending`.
5. We iterate through the future that is done, and print the response by fetching it using the `result` method of the `future`.

You can see the result of the operation (the first few lines, as many lines are output) in the following screenshot:

```
Chapter 5 - Asynchronous Execution
File Edit View Search Terminal Help
$ python3 async_fetch_url.py
<ClientResponse(https://twitter.com/) [200 OK]>
<CIMultiDictProxy('Cache-Control': 'no-cache, no-store, must-revalidate, pre-check=0, post-check=0', 'Content-Encoding':
'gzip', 'Content-Type': 'text/html;charset=utf-8', 'Date': 'Sun, 15 Jan 2017 18:40:42 GMT', 'Expires': 'Tue, 31 Mar 1981
05:00:00 GMT', 'Last-Modified': 'Sun, 15 Jan 2017 18:40:42 GMT', 'Pragma': 'no-cache', 'Server': 'tsa_f', 'Set-Cookie': '
fm=0; Expires=Sun, 15 Jan 2017 18:40:32 GMT; Path=/; Domain=.twitter.com; Secure; HTTPOnly', 'Set-Cookie': '_twitter_sess
=BAh7CSIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFzaHsABjoKQHVzZWR7ADoPY3JlYXRlZF9hdGwrCLZ0bqNZAToMY3N
yZl9p%250AZCIlOGQ3ODVmMzk3ZThiYTljMmY2ZjQwOWYyYTEyMmQxOWY6B2lkIiU3ZDYw%250ANjgyMjkxOTRhZjZjOTc0YmFiZWYyZDBhN2M3OQ%253D%25
3D--a1761cdd01da6bcca1128a85138cd3b6526910f9; Path=/; Domain=.twitter.com; Secure; HTTPOnly', 'Set-Cookie': 'guest_id=v1%
3A148450564216729728; Domain=.twitter.com; Path=/; Expires=Tue, 15-Jan-2019 18:40:42 UTC', 'Status': '200 OK', 'Strict-Tr
ansport-Security': 'max-age=631138519', 'Transfer-Encoding': 'chunked', 'X-Connection-Hash': '82f038828b9b47d9f48721edf50
08f77', 'X-Content-Type-Options': 'nosniff', 'X-Frame-Options': 'SAMEORIGIN', 'X-Response-Time': '258', 'X-Transaction':
'00006c7a007c82af', 'X-Twitter-Response-Tags': 'BouncerCompliant', 'X-Ua-Compatible': 'IE=edge,chrome=1', 'X-Xss-Protecti
on': '1; mode=block')>

<ClientResponse(https://www.reddit.com/) [200 OK]>
<CIMultiDictProxy('Content-Type': 'text/html; charset=UTF-8', 'X-Ua-Compatible': 'IE=edge', 'X-Frame-Options': 'SAMEORIGI
N', 'X-Content-Type-Options': 'nosniff', 'X-Xss-Protection': '1; mode=block', 'Content-Encoding': 'gzip', 'Cache-Control'
: 'max-age=0, must-revalidate', 'X-Moose': 'majestic', 'Strict-Transport-Security': 'max-age=15552000; includeSubDomains;
 preload', 'Content-Length': '25398', 'Accept-Ranges': 'bytes', 'Date': 'Sun, 15 Jan 2017 18:40:41 GMT', 'Via': '1.1 varn
ish', 'Connection': 'keep-alive', 'X-Served-By': 'cache-cdg8726-CDG', 'X-Cache': 'MISS', 'X-Cache-Hits': '0', 'X-Timer':
'S1484505641.366114,VS0,VE470', 'Vary': 'accept-encoding', 'Server': 'snooserv')>

<ClientResponse(https://www.facebook.com/) [200 OK]>
<CIMultiDictProxy('X-Xss-Protection': '0', 'Public-Key-Pins-Report-Only': 'max-age=500; pin-sha256="WoiWRyIOVNa9ihaBciRSC
7XHjliYS9VwUGOIud4PB18="; pin-sha256="r/mIkG3eEpVdm+u/ko/cwxzOMo1bk4TyHIlByibiA5E="; pin-sha256="q4PO2G2cbkZhZ82+JgmRUyGM
oAeozA+BSXVXQWB8XWQ="; report-uri="http://reports.fb.com/hpkp/"', 'Pragma': 'no-cache', 'Cache-Control': 'private, no-cac
```

Output of program doing an async fetch of URLs for 5 URLs

As you can see, we are able to print the responses in terms of a simple summary. How about processing the response to get more details about it such as the actual response text, the content length, status code, and so on?

The function below parses a list of *done* futures—waiting for the response data via *await* on the *read* method of the response. This returns the data for each response asynchronously:

```
async def parse_response(futures):
""" Parse responses of fetch """
for future in futures:
response = future.result()
data = await response.text()
        print('Response for URL',response.url,'=>', response.status,
len(data))
        response.close()
```

The details of the `response` object—the final URL, status code, and length of data—are output by this method for each response before closing the response.

We only need to add one more processing step on the list of completed responses for this to work:

```
session = aiohttp.ClientSession(loop=loop)
# Wait for futures
tasks = map(lambda x: fetch_page(session, x), urls)
done, pending = loop.run_until_complete(asyncio.wait(tasks,
                                        timeout=300))

# One more processing step to parse responses of futures
loop.run_until_complete(parse_response(done))

session.close()
loop.close()
```

Note how we chain the co-routines together. The final link in the chain is the `parse_response` co-routine, which processes the list of done futures before the loop ends.

The following screenshot shows the output of the program:

```
$ python3 async_fetch_url2.py
Response for URL https://twitter.com/ => 200 296219
Response for URL http://www.google.co.in/?gfe_rd=cr&ei=zcZ7WMrDLNTFuATnqbrADQ => 200 12390
Response for URL https://in.yahoo.com/?p=us => 200 380341
Response for URL https://www.reddit.com/ => 200 138933
Response for URL https://www.facebook.com/ => 200 147941
$
```

Output of program doing fetching and response processing of 5 URLs asynchronously

A lot of complex programming can be done using the `asyncio` module. One can wait for futures, cancel their execution, and run `asyncio` operations from multiple threads. A full discussion is beyond the scope of this chapter.

We will move on to another model for executing concurrent tasks in Python, namely the `concurrent.futures` module.

Concurrent futures – high-level concurrent processing

The `concurrent.futures` module provides high-level concurrent processing using either threads or processes, while asynchronously returning data using future objects.

It provides an executor interface which exposes mainly two methods, which are as follows:

- `submit`: Submits a callable to be executed asynchronously, returning a `future` object representing the execution of the callable.
- `map`: Maps a callable to a set of iterables, scheduling the execution asynchronously in the `future` object. However, this method returns the results of processing directly instead of returning a list of futures.

There are two concrete implementations of the executor interface: `ThreadPoolExecutor` executes the callable in a pool of threads, and `ProcessPoolExecutor` does so in a pool of processes.

Here is a simple example of a `future` object that calculates the factorial of a set of integers asynchronously:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import functools
import operator

def factorial(n):
    return functools.reduce(operator.mul, [i for i in range(1, n+1)])

with ThreadPoolExecutor(max_workers=2) as executor:
    future_map = {executor.submit(factorial, n): n for n in range(10, 
21)}
    for future in as_completed(future_map):
        num = future_map[future]
        print('Factorial of',num,'is',future.result())
```

The following is a detailed explanation of the preceding code:

- The `factorial` function computes the factorial of a given number iteratively by using `functools.reduce` and the multiplication operator
- We create an executor with two workers, and submit the numbers (from 10 to 20) to it via its `submit` method
- The submission is done via a dictionary comprehension, returning a dictionary with the future as the key and the number as the value
- We iterate through the completed futures, which have been computed, using the `as_completed` method of the `concurrent.futures` module
- The result is printed by fetching the future's result via the `result` method

When executed, the program prints its output, rather in order, as shown in the next screenshot:

```
$ python3 concurrent_factorial.py
Factorial of 10 is 3628800
Factorial of 11 is 39916800
Factorial of 12 is 479001600
Factorial of 13 is 6227020800
Factorial of 14 is 87178291200
Factorial of 15 is 1307674368000
Factorial of 16 is 20922789888000
Factorial of 17 is 355687428096000
Factorial of 18 is 6402373705728000
Factorial of 19 is 121645100408832000
Factorial of 20 is 2432902008176640000
$
```

Output of concurrent futures factorial program

Disk thumbnail generator

In our earlier discussion of threads, we used the example of the generation of thumbnails for random images from the Web to demonstrate how to work with threads, and process information.

In this example, we will do something similar. Here, rather than processing random image URLs from the Web, we will load images from disk, and convert them to thumbnails using the `concurrent.futures` function.

We will reuse our thumbnail creation function from before. On top of that, we will add concurrent processing.

First, here are the imports:

```
import os
import sys
import mimetypes
from concurrent.futures import ThreadPoolExecutor,
ProcessPoolExecutor, as_completed
```

Here is our familiar thumbnail creation function:

```
def thumbnail_image(filename, size=(64,64), format='.png'):
    """ Convert image thumbnails, given a filename """

    try:
        im=Image.open(filename)
        im.thumbnail(size, Image.ANTIALIAS)

        basename = os.path.basename(filename)
        thumb_filename = os.path.join('thumbs',
            basename.rsplit('.')[0] + '_thumb.png')
        im.save(thumb_filename)
        print('Saved',thumb_filename)
        return True

    except Exception as e:
        print('Error converting file',filename)
        return False
```

We will process images from a specific folder—in this case, the `Pictures` subdirectory of the `home` folder. To process this, we will need an iterator that yields image filenames. We have written one next with the help of the `os.walk` function:

```
def directory_walker(start_dir):
    """ Walk a directory and generate list of valid images """

    for root,dirs,files in os.walk(os.path.expanduser(start_dir)):
        for f in files:
            filename = os.path.join(root,f)
            # Only process if it's a type of image
            file_type = mimetypes.guess_type(filename.lower())[0]
            if file_type != None and file_type.startswith('image/'):
                yield filename
```

As you can see, the preceding function is a generator.

Here is the main calling code, which sets up an executor and runs it over the folder:

```
root_dir = os.path.expanduser('~/Pictures/')
if '--process' in sys.argv:
    executor = ProcessPoolExecutor(max_workers=10)
else:
    executor = ThreadPoolExecutor(max_workers=10)

with executor:
    future_map = {executor.submit(thumbnail_image, filename):
    filename for filename in directory_walker(root_dir)}
    for future in as_completed(future_map):
        num = future_map[future]
        status = future.result()
        if status:
            print('Thumbnail of',future_map[future],'saved')
```

The preceding code uses the same technique of submitting arguments to a function asynchronously, saving the resultant futures in a dictionary and then processing the result as and when the futures are finished, in a loop.

To change the executor to use processes, one simply needs to replace `ThreadPoolExecutor` with `ProcessPoolExecutor`; the rest of the code remains the same. We have provided a simple command-line flag, `--process`, to make this easy.

Here is an output of a sample run of the program using both thread and process pools on the `~/Pictures` folder—generating around 2000+ images in roughly the same time.

```
Chapter 5 - Asynchronous Execution
File  Edit  View  Search  Terminal  Help
$ time python3 concurrent_thumbnail.py > /dev/null

real    0m14.372s
user    0m54.368s
sys     0m1.132s
$ time python3 concurrent_thumbnail.py --process> /dev/null

real    0m14.580s
user    0m55.996s
sys     0m1.076s
$ ls thumbs/* | wc -l
2138
$
```

Output of concurrent futures disk thumbnail program—using thread and process executor

Concurrency options – how to choose?

We are at the end of our discussion of concurrency techniques in Python. We discussed threads, processes, asynchronous I/O, and concurrent futures. Naturally, a question arises—when to pick what?

This question has been already answered for the choice between threads and processes, where the decision is mostly influenced by the GIL.

Here are some rough guidelines for picking your concurrency options.

- **Concurrent futures versus multiprocessing:** Concurrent futures provide an elegant way to parallelize your tasks using either a thread or process pool executor. Hence, it is ideal if the underlying application has similar scalability metrics with either threads or processes, since it's very easy to switch from one to the other as we've seen in a previous example. Concurrent futures can be chosen also when the result of the operation needn't be immediately available. Concurrent futures is a good option when the data can be finely parallelized and the operation can be executed asynchronously, and when the operations involve simple callables without requiring complex synchronization techniques.

  Multiprocessing should be chosen if the concurrent execution is more complex, and not just based on data parallelism, but has aspects like synchronization, shared memory, and so on. For example, if the program requires processes, synchronization primitives, and IPC, the only way to truly scale up then, is to write a concurrent program using the primitives provided by the multiprocessing module.

  Similarly when your multithreaded logic involves simple parallelization of data across multiple tasks, one can choose concurrent futures with a thread pool. However if there is a lot of shared state to be managed with complex thread synchronization objects—one has to use thread objects and switch to multiple threads using `threading` module to get finer control of the state.

- **Asynchronous I/O vs threaded concurrency:** When your program doesn't need true concurrency (parallelism), but is dependent more on asynchronous processing and callbacks, then `asyncio` is the way to go. Asyncio is a good choice when there are lot of waits or sleep cycles involved in the application, such as waiting for user input, waiting for I/O, and so on, and one needs to take advantage of such wait or sleep times by yielding to other tasks via co-routines. Asyncio is not suitable for CPU-heavy concurrent processing, or for tasks involving true data parallelism.

AsyncIO seems to be suitable for request-response loops, where a lot of I/O happens – so its good for writing web application servers which do not have real-time data requirements.

You can use the points just listed as rough guidelines when deciding on the correct concurrency package for your applications.

Parallel processing libraries

Apart from the standard library modules that we've discussed so far, Python is also rich in its ecosystem of third-party libraries, which support parallel processing in **symmetric multi-processing** (**SMP**) or multi-core systems.

We will take a look at a couple of such packages, that are somewhat distinct and present some interesting features.

Joblib

`joblib` is a package that provides a wrapper over multiprocessing to execute code in loops in parallel. The code is written as a generator expression, and interpreted to execute in parallel over CPU cores using multiprocessing modules behind the scenes.

For example, take the following code which calculates square roots for first 10 numbers:

```
>>> [i ** 0.5 for i in range(1, 11)]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979,
2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0,
3.1622776601683795]
```

This preceding code can be converted to run on two CPU cores by the following:

```
>>> import math
>>> from joblib import Parallel, delayed
    [1.0, 1.4142135623730951, 1.7320508075688772, 2.0,
     2.23606797749979, 2.449489742783178, 2.6457513110645907,
     2.8284271247461903, 3.0, 3.1622776601683795]
```

Here is another example: this is our primality checker that we had written earlier to run using multiprocessing, rewritten to use the `joblib` package:

```
# prime_joblib.py
from joblib import Parallel, delayed

def is_prime(n):
    """ Check for input number primality """

    for i in range(3, int(n**0.5+1), 2):
        if n % i == 0:
            print(n,'is not prime')
            return False

    print(n,'is prime')
    return True

if __name__ == "__main__":
    numbers = [1297337, 1116281, 104395303, 472882027, 533000389,
               817504243, 982451653, 112272535095293, 115280095190773,
               1099726899285419]*100
    Parallel(n_jobs=10)(delayed(is_prime)(i) for i in numbers)
```

If you execute and time the preceding code, you will find the performance metrics very similar to that of the version using multiprocessing.

PyMP

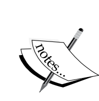
`OpenMP` is an open API, which supports shared memory multiprocessing in C/C++ and Fortran. It uses special work-sharing constructs such as pragmas (special instructions to compilers) indicating how to split work among threads or processes.

For example, the following C code using the `OpenMP` API indicates that the array should be initialized in parallel using multiple threads:

```
int parallel(int argc, char **argv)
{
    int array[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
array[i] = i * i;
    }

return 0;
}
```

`PyMP` is inspired by the idea behind `OpenMP`, but uses the `fork` system call to parallelize code executing in expressions such as for loops across processes. For this, `PyMP` also provides support for shared data structures such as lists and dictionaries, and also provides a wrapper for `numpy` arrays.

We will look at an interesting and exotic example—that of fractals—to illustrate how `PyMP` can be used to parallelize code and obtain performance improvement.



NOTE: The PyPI package for PyMP is named `pymp-pypi` so make sure you use this name when trying to install it via `pip`. Also note that it doesn't do a good job of pulling its dependencies such as numpy, so these have to be installed separately.

Fractals – the Mandelbrot set

The following is the code listing of a very popular class of complex numbers, which when plotted, produces very interesting fractal geometries, namely, the **Mandelbrot set**:

```
# mandelbrot.py
import sys
import argparse
from PIL import Image

def mandelbrot_calc_row(y, w, h, image, max_iteration = 1000):
    """ Calculate one row of the Mandelbrot set with size wxh """

    y0 = y * (2/float(h)) - 1 # rescale to -1 to 1

    for x in range(w):
        x0 = x * (3.5/float(w)) - 2.5 # rescale to -2.5 to 1

        i, z = 0, 0 + 0j
        c = complex(x0, y0)
        while abs(z) < 2 and i < max_iteration:
            z = z**2 + c
            i += 1

        # Color scheme is that of Julia sets
        color = (i % 8 * 32, i % 16 * 16, i % 32 * 8)
```

```
        image.putpixel((x, y), color)


def mandelbrot_calc_set(w, h, max_iteration=10000, output='mandelbrot.
png'):
    """ Calculate a mandelbrot set given the width, height and
    maximum number of iterations """

    image = Image.new("RGB", (w, h))

    for y in range(h):
        mandelbrot_calc_row(y, w, h, image, max_iteration)

    image.save(output, "PNG")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(prog='mandelbrot',
description='Mandelbrot fractal generator')
    parser.add_argument('-W','--width',help='Width of the
image',type=int, default=640)
    parser.add_argument('-H','--height',help='Height of the
image',type=int, default=480)
    parser.add_argument('-n','--niter',help='Number of
iterations',type=int, default=1000)
    parser.add_argument('-o','--output',help='Name of output image
file',default='mandelbrot.png')

    args = parser.parse_args()
    print('Creating Mandelbrot set with size %(width)sx%(height)s,
#iterations=%(niter)s' % args.__dict__)
    mandelbrot_calc_set(args.width, args.height, max_iteration=args.
niter, output=args.output)
```

The preceding code calculates a Mandelbrot set using a certain number of `c` and a variable geometry (*width x height*). It is complete with argument parsing to produce fractal images of varying geometries, and supports different iterations.

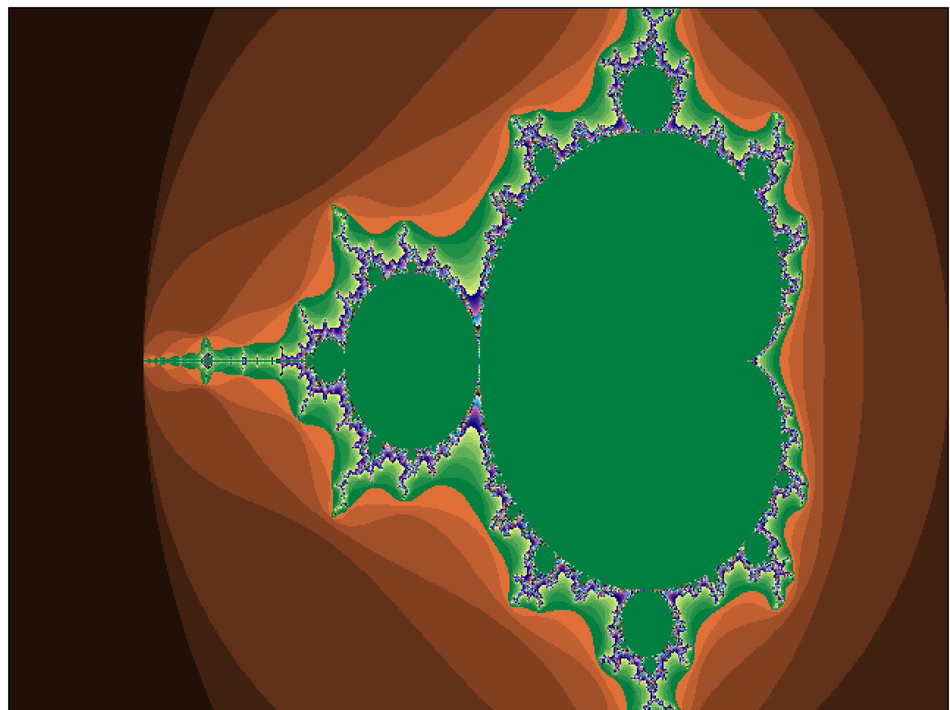
For simplicity's sake, and for producing rather more beautiful pics than what Mandelbrot usually does, the code takes some liberties, and uses the color scheme of a related fractal class, namely, Julia sets.

How does it work ? Here is an explanation of the code .

1. The `mandelbrot_calc_row` function calculates a row of the Mandelbrot set for a certain value of the *y* coordinate for a certain number of maximum iterations. The pixel color values for the entire row, from `0` to width `w` for the *x* coordinate, is calculated. The pixel values are put into the `Image` object that is passed to this function.
2. The `mandelbrot_calc_set` function calls the `mandelbrot_calc_row` function for all values of the *y* coordinate ranging from `0` to the height `h` of the image. An `Image` object (via the **Pillow library**) is created for the given geometry (*width x height*), and filled with pixel values. Finally, we save this image to a file, and we've got our fractal!

Without further ado, let's see the code in action.

Here is the image that our Mandelbrot program produces for the default number of iterations namely 1000.



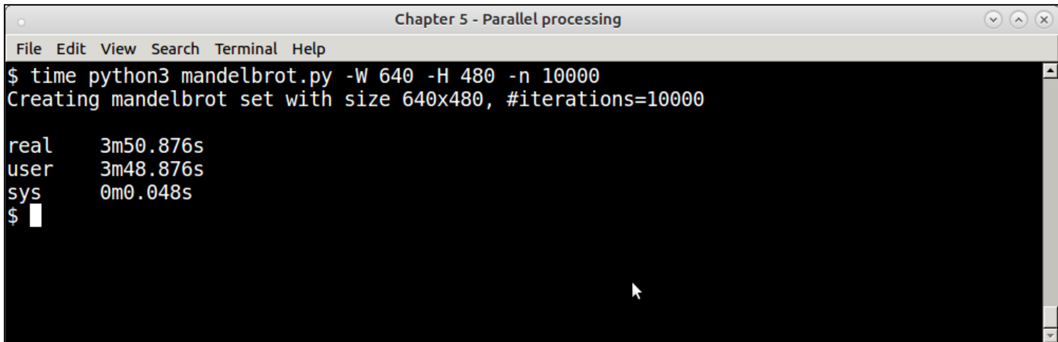
Mandelbrot set fractal image for 1,000 iterations

Here is the time it takes to create this image.

```
Chapter 5 - Parallel processing
File Edit View Search Terminal Help
$ time python3 mandelbrot.py -W 640 -H 480 -n 1000
Creating mandelbrot set with size 640x480, #iterations=1000

real    0m22.887s
user    0m22.868s
sys     0m0.016s
$
```

Timing of single process Mandelbrot program—for 1,000 iterations

However, if you increase the number of iterations—the single process version slows down quite a bit. Here is the output when we increase the number of iterations by 10X—for 10,000 iterations:



Timing of single process Mandelbrot program—for 10,000 iterations

If we look at the code, we can see that there is an outer for loop in the `mandelbrot_calc_set` function, which sets things in motion. It calls `mandelbrot_calc_row` for each row of the image ranging from `0` to the height of the function, varied by the *y* coordinate.

Since each invocation of the `mandelbrot_calc_row` function calculates one row of the image, it naturally fits into a data parallel problem, and can be parallelized sufficiently easily.

In the next section, we will see how to do this using PyMP.

Fractals – scaling the Mandelbrot set implementation

We will use `PyMP` to parallelize the outer for loop across many processes in a rewrite of the previous simple implementation of the Mandelbrot set, to take advantage of the inherent data parallelism in the solution.

Here is the `PyMP` version of the two functions of the Mandelbrot program. The rest of the code remains the same.

```
# mandelbrot_mp.py
import sys
from PIL import Image
import pymp
import argparse


def mandelbrot_calc_row(y, w, h, image_rows, max_iteration = 1000):
    """ Calculate one row of the mandelbrot set with size wxh """

    y0 = y * (2/float(h)) - 1 # rescale to -1 to 1

    for x in range(w):
        x0 = x * (3.5/float(w)) - 2.5 # rescale to -2.5 to 1

        i, z = 0, 0 + 0j
        c = complex(x0, y0)
        while abs(z) < 2 and i < max_iteration:
            z = z**2 + c
            i += 1

        color = (i % 8 * 32, i % 16 * 16, i % 32 * 8)
        image_rows[y*w + x] = color


def mandelbrot_calc_set(w, h, max_iteration=10000, output='mandelbrot_
mp.png'):
    """ Calculate a mandelbrot set given the width, height and
    maximum number of iterations """

    image = Image.new("RGB", (w, h))
    image_rows = pymp.shared.dict()

    with pymp.Parallel(4) as p:
```

```
            for y in p.range(0, h):
                mandelbrot_calc_row(y, w, h, image_rows, max_iteration)

        for i in range(w*h):
            x,y = i % w, i // w
            image.putpixel((x,y), image_rows[i])

        image.save(output, "PNG")
        print('Saved to',output)
```

The rewrite mainly involved converting the code to one that builds the Mandelbrot image line by line, each line of data being computed separately and in a way that it can be computed in parallel—in a separate process.

- In the single process version, we put the pixel values directly in the image in the `mandelbrot_calc_row` function. However, since the new code executes this function in parallel processes, we cannot modify the image data in it directly. Instead, the new code passes a shared dictionary to the function, and it sets the pixel color values in it using the location as `key` and the pixel RGB value as `value`.
- A new shared data structure—a shared dictionary—is hence added to the `mandelbrot_calc_set` function, which is finally iterated over, and the pixel data, filled, in the `Image` object, which is then saved to the final output.
- We use four `PyMP` parallel processes, as the machine has four CPU cores, using a with context and enclosing the outer for loop inside it. This causes the code to execute in parallel in four cores, each core calculating approximately 25% of the rows. The final data is written to the image in the main process.

Here is the result timing of the `PyMP` version of the code:

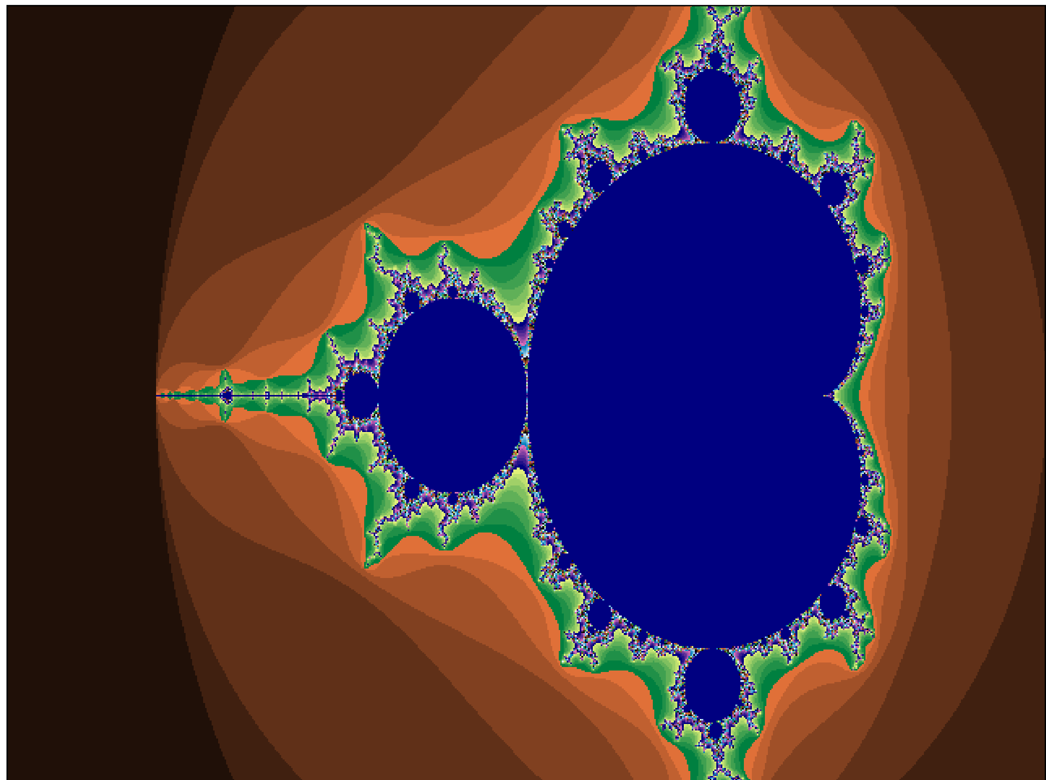
```
$ time python3 mandelbrot_mp.py -W 640 -H 480 -n 10000
Creating mandelbrot set with size 640x480, #iterations=10000
Saved to mandelbrot_mp.png

real    2m20.586s
user    7m17.132s
sys     0m8.036s
$
```

Timing of parallel process Mandelbrot program using PyMP—for 10000 iterations

The program is about 33% faster in real time. In terms of CPU usage, you can see that the `PyMP` version has a higher ratio of user CPU time to real CPU time, indicating a higher usage of the CPU by the processes than the single process version.

> We can write an even more efficient version of the program by avoiding the shared data structure image_rows which is used to keep the pixel values of the image. This version however uses that to show the features of PyMP. The code archives of this book contain two more versions of the program—one that uses multiprocessing and another that uses PyMP without the shared dictionary.

This is the output fractal image produced by this run of the program:



Mandelbrot set fractal image for 10000 iterations using PyMP

You can observe that the colors are different, and this image provides more detail and a finer structure than the previous one due to the increased number of iterations.

Scaling for the web

So far, all the scalability and concurrency techniques we discussed were involved with scalability within the confines of a single server or machine—in other words, scaling up. In real world, applications also scale by scaling out, that is, by spreading their computation over multiple machines. This is how most real-world web applications run and scale at present.

We will look at a few techniques, scaling out an application in terms of scaling communications/workflows, scaling computation, and horizontal scaling using different protocols.

Scaling workflows – message queues and task queues

One important aspect of scalability is to reducing coupling between systems. When two systems are tightly coupled, they prevent each other from scaling beyond a certain limit.

For example, a code written serially, where data and computation is tied into the same function, prevents the program from taking advantage of the existing resources like multiple CPU cores. When the same program is rewritten to use multiple threads (or processes) and a message passing system like a queue in between, we find it scales well to multiple CPUs. We've seen such examples aplenty in our concurrency discussion.

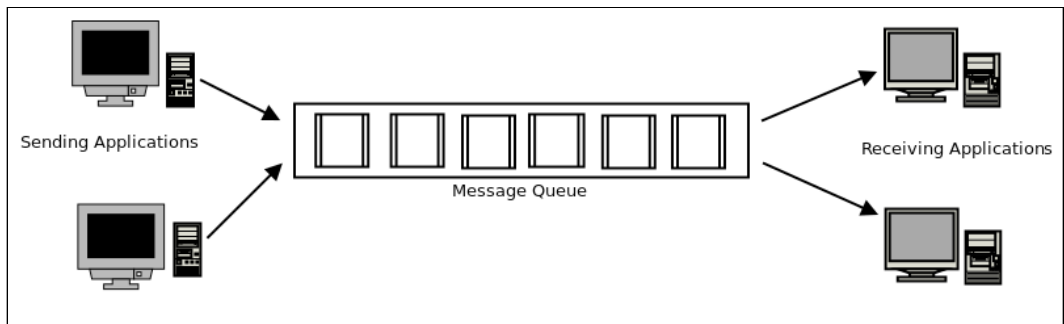
In a much similar way, systems over the Web scale better when they are decoupled. The classic example is the client/server architecture of the Web itself, where clients interact via well-known RestFUL protocols like HTTP, with servers located in different places across the world.

Message queues are systems that allow applications to communicate in a decoupled manner by sending messages to each other. The applications typically run in different machines or servers connected to the Internet, and communicate via queuing protocols.

One can think of a message queue as a scaled-up version of the multi-threaded synchronized queue, with applications on different machines replacing the threads, and a shared, distributed queue replacing the simple in-process queue.

Message queues carry packets of data called messages, which are delivered from the **Sending Applications** to the **Receiving Applications**. Most **Message Queues** provide **store and forward** semantics, where the message is stored on the queue till the receiver is available to process the message.

Here is a simple schematic model of a **Message Queue**:



Schematic model of a distributed message queue

The most popular and standardized implementation of a message queue or **message-oriented middleware** (**MoM**) is the **Advanced Message Queuing Protocol** (**AMQP**). AMQP provides features such as queuing, routing, reliable delivery, and security. The origins of AMQP are in the financial industry, where reliable and secure message delivery semantics are of critical importance.

The most popular implementations of AMQP (version 1.0) are Apache Active MQ, RabbitMQ, and Apache Qpid.

RabbitMQ is a MoM written in Erlang. It provides libraries in many languages including Python. In RabbitMQ, a message is always delivered via exchanges via routing keys which indicate the queues to which the message should be delivered.

We won't be discussing RabbitMQ in this section anymore, but will move on to a related, but slightly different, middleware with a varying focus, namely, Celery.

Celery – a distributed task queue

Celery is a distributed task queue written in Python, which works using distributed messages. Each execution unit in celery is called a **task**. A task can be executed concurrently on one or more servers using processes called **workers**. By default, Celery achieves this using `multiprocessing`, but it can also use other backends such as gevent, for example.

Tasks can be executed synchronously or asynchronously with results available in the future, like objects. Also, task results can be stored in storage backend such as Redis, databases, or in files.

Celery differs from message queues in that the basic unit in celery is an executable task—a callable in Python—rather than just a message.

Celery, however, can be made to work with message queues. In fact, the default broker for passing messages in celery is RabbitMQ, the popular implementation of AMQP. Celery can also work with Redis as the broker backend.

Since Celery takes a task, and scales it over multiple workers, over multiple servers, it is suited to problems involving data parallelism as well as computational scaling. Celery can accept messages from a queue and distribute it over multiple machines as tasks for implementing a distributed e-mail delivery system, for example, and achieve horizontal scalability. Or, it can take a single function and perform parallel data computation by splitting the data over multiple processes, achieving parallel data processing.

In the following example, we will take our Mandelbrot fractal program and, rewrite it to work with Celery. We will try to scale the program by performing data parallelism, in terms of computing the rows of the Mandelbrot set over multiple celery workers—in a similar way to what we did with `PyMP`.

The Mandelbrot set using Celery

For implementing a program to take advantage of Celery, it needs to be implemented as a task. This is not as difficult as it sounds. Mostly, it just involves preparing an instance of the celery app with a chosen broker backend, and decorating the callable we want to parallelize—using the special decorator `@app.task` where *app* is an instance of Celery.

We will look at this program listing step by step, since it involves a few new things. The software requirements for this session are as follows:

- Celery
- An AMQP backend; RabbitMQ is preferred
- Redis as a result storage backend

First we will provide the listing for the Mandelbrot tasks module:

```
# mandelbrot_tasks.py
from celery import Celery

app = Celery('tasks', broker='pyamqp://guest@localhost//',
             backend='redis://localhost')

@app.task
def mandelbrot_calc_row(y, w, h, max_iteration = 1000):
    """ Calculate one row of the mandelbrot set with size w x h """

    y0 = y * (2/float(h)) - 1 # rescale to -1 to 1

    image_rows = {}
    for x in range(w):
        x0 = x * (3.5/float(w)) - 2.5 # rescale to -2.5 to 1

        i, z = 0, 0 + 0j
        c = complex(x0, y0)
        while abs(z) < 2 and i < max_iteration:
            z = z**2 + c
            i += 1

        color = (i % 8 * 32, i % 16 * 16, i % 32 * 8)
        image_rows[y*w + x] = color

    return image_rows
```

Let's analyze the preceding code:

- We first do the imports required for Celery. This requires importing the `Celery` class from the `celery` module.
- We prepare an instance of the `Celery` class as the Celery app using AMQP as the message broker and Redis as the result backend. The AMQP configuration will use whatever AMQP MoM is available on the system. (In this case, it is RabbitMQ.)
- We have a modified version of `mandelbrot_calc_row`. In the `PyMP` version, the `image_rows` dictionary was passed as an argument to the function. Here, the function calculates it locally and returns a value. We will use this return value at the receiving side to create our image.
- We decorated the function using `@app.task`, where app is the `Celery` instance. This makes it ready to be executed as a Celery task by the Celery workers.

Next is the main program, which calls the task for a range of `y` input values and creates the image:

```
# celery_mandelbrot.py
import argparse
from celery import group
from PIL import Image
from mandelbrot_tasks import mandelbrot_calc_row

def mandelbrot_main(w, h, max_iterations=1000,
output='mandelbrot_celery.png'):
    """ Main function for mandelbrot program with celery """

    # Create a job - a group of tasks
    job = group([mandelbrot_calc_row.s(y, w, h, max_iterations) for y
in range(h)])
    # Call it asynchronously
    result = job.apply_async()

    image = Image.new('RGB', (w, h))

    for image_rows in result.join():
        for k,v in image_rows.items():
            k = int(k)
            v = tuple(map(int, v))
```

```
            x,y = k % args.width, k // args.width
            image.putpixel((x,y), v)

    image.save(output, 'PNG')
    print('Saved to',output)
```

The argument parser is the same, so is not reproduced here.

This last bit of code introduces some new concepts in Celery, so needs some explanation. Let's analyze the code in some detail:

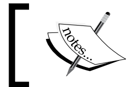
1. The `mandelbrot_main` function is similar to the previous `mandelbrot_calc_set` function in its arguments.
2. This function sets up a group of tasks, each performing `mandelbrot_calc_row` execution on a given `y` input over the entire range of `y` inputs from `0` to the height of the image. It uses the `group` object of Celery to do this. A group is a set of tasks which can be executed together.
3. The tasks are executed by calling the `apply_async` function on the group. This executes the tasks asynchronously in the background in multiple workers. We get an async `result` object in return—the tasks are not completed yet.
4. We then wait on this result object by calling `join` on it, which returns the results—the rows of the image as a dictionary from each single execution of the `mandelbrot_calc_row` task. We loop through this, and do integer conversions for the values, since Celery returns data as strings, and put the pixel values in the image.
5. Finally, the image is saved in the output file.

So, how does Celery execute the tasks? This needs the Celery program to run, processing the tasks module with a certain number of workers. Here is how we start it in this case:

```
Chapter 5 - Parallel processing
File  Edit  View  Search  Terminal  Tabs  Help
Chapter 5 - Parallel processing                 anand@ubuntu-pro-book: /home/user/programs/chap5
$ celery -A mandelbrot_tasks worker -c 4 --loglevel info

 -------------- celery@ubuntu-pro-book v4.0.2 (latentcall)
---- **** -----
--- * ***  * -- Linux-4.4.0-57-generic-x86_64-with-Ubuntu-16.04-xenial 2017-01-17 01:55:05
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         tasks:0x7fee7c7e84e0
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     redis://localhost/
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery


[tasks]
  . mandelbrot_tasks.mandelbrot_calc_row

[2017-01-17 01:55:05,511: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2017-01-17 01:55:05,519: INFO/MainProcess] mingle: searching for neighbors
[2017-01-17 01:55:06,540: INFO/MainProcess] mingle: all alone
[2017-01-17 01:55:06,583: INFO/MainProcess] celery@ubuntu-pro-book ready.
```

Celery console—workers starting up with the Mandelbrot task as target

The command starts Celery with tasks loaded from the module `mandelbrot_tasks.py` with a set of 4 worker processes. Since the machine has 4 CPU cores, we have chosen this as the concurrency.



Note that Celery will automatically default the workers to the number of cores if not specifically configured.

The program ran under 15 seconds, twice as fast in as the single-process version, and also the `PyMP` version.

If you observe the Celery console, you will find a lot of messages getting echoed, since we configured Celery with the `INFO` log level. All these are info messages with data on the tasks and their results:

The following screenshot shows the result of the run for `10000` iterations. This performance is slightly better than that of the similar run by the `PyMP` version earlier, by around 20 seconds:

```
$ time python3 celery_mandelbrot.py -W 640 -H 480 -n 10000
Creating mandelbrot set with size 640x480, #iterations=10000
Saved to mandelbrot_celery.png

real    1m59.934s
user    0m2.328s
sys     0m0.156s
$
```

Celery Mandelbrot program for a set of 10000 iterations.

Celery is used in production systems in many organizations. It has plugins for some of the more popular Python web application frameworks. For example, Celery supports Django out-of-the-box with some basic plumbing and configuration. There are also extension modules such as `django-celery-results`, which allow the programmer to use the Django ORM as a Celery results backend.

It is beyond the scope of this chapter and book to discuss this in detail, so the reader is advised to refer to the documentation available on this on the Celery project website.

Serving with Python on the Web – WSGI

Web Server Gateway Interface (**WSGI**) is a specification for a standard interface between Python web application frameworks and web servers.

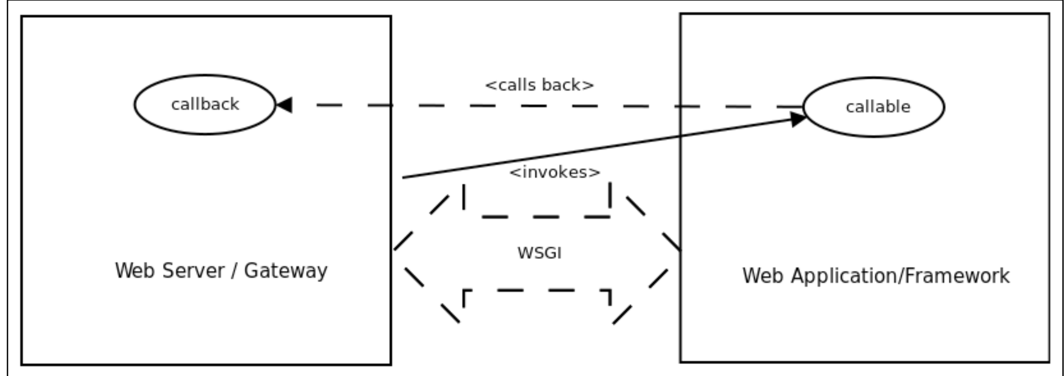
In the early days of Python web applications, there was a problem connecting web application frameworks to web servers, since there was no common standard. Python web applications were designed to work with one of the existing standards of CGI, FastCGI, or `mod_python` (Apache). This meant that an application written to work with one web server might not be able to work with another. In other words, interoperability between the uniform application and web server was missing.

WSGI solved this problem by specifying a simple, but uniform, interface between servers and web application frameworks to allow for portable web application development.

WSGI specifies two sides: the server (or gateway) side, and the application or framework side. A WSGI request gets processed as follows:

- The server side executes the application, providing it with an environment and a callback function
- The application processes the request, and returns the response to the server using the provided callback function

Here is a schematic diagram showing the interaction between a web server and web application using WSGI:



Schematic diagram showing WSGI protocol interaction

The following is the simplest function that is compatible with the application or framework side of WSGI:

```
def simple_app(environ, start_response):
    """Simplest possible application object"""

    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n']
```

The preceding function can be explained as follows:

1. The `environ` variable is a dictionary of environment variables passed from the server to the application as defined by the **Common Gateway Interface** (**CGI**) specification. WSGI makes a few of these environment variables mandatory in its specification.
2. The `start_response` is a callable provided as a callback from the server side to the application side, to start response processing on the server side. It must take two positional arguments. The first should be a status string with an integer status code, and the second, a list of (`header_name`, `header_value`), tuples describing the HTTP response header.

For more details, the reader can refer to the WSGI specification v1.0.1, which is published on the Python language website as PEP 3333.

Python Enhancement Proposal (**PEP**) is a design document on the Web, that describes a new feature or feature suggestion for Python, or provides information to the Python community about an existing feature. The Python community uses PEPs as a standard process for describing, discussing, and adopting new features and enhancements to the Python programming language and its standard library.

WSGI middleware components are software that implement both sides of the specification, and hence, provide capabilities such as the following:

- Load balancing of multiple requests from a server to an application
- Remote processing of requests by forwarding requests and responses over a network
- Multi-tenancy or co-hosting of multiple servers and/or applications in the same process
- URL-based routing of requests to different application objects

The middleware sits in between the server and application. It forwards requests from server to the application and responses from application to the server.

There are a number of WSGI middleware an architect can choose from. We will briefly look at two of the most popular ones, namely, uWSGI and Gunicorn.

uWSGI – WSGI middleware on steroids

uWSGI is an open source project and application, which aims to build a full stack for hosting services. The WSGI of the uWSGI project stems from the fact that the WSGI interface plugin for Python was the first one developed in the project.

Apart from WSGI, the uWSGI project also supports **Perl Webserver Gateway Interface** (**PSGI**) for Perl web applications, and the rack web server interface for Ruby web applications. It also provides gateways, load balancers, and routers for requests and responses. The Emperor plugin of uWSGI provides management and monitoring of multiple uWSGI deployments of your production system across servers.

The components of uWSGI can run in preforked, threaded, asynchronous. or green-thread/co-routine modes.

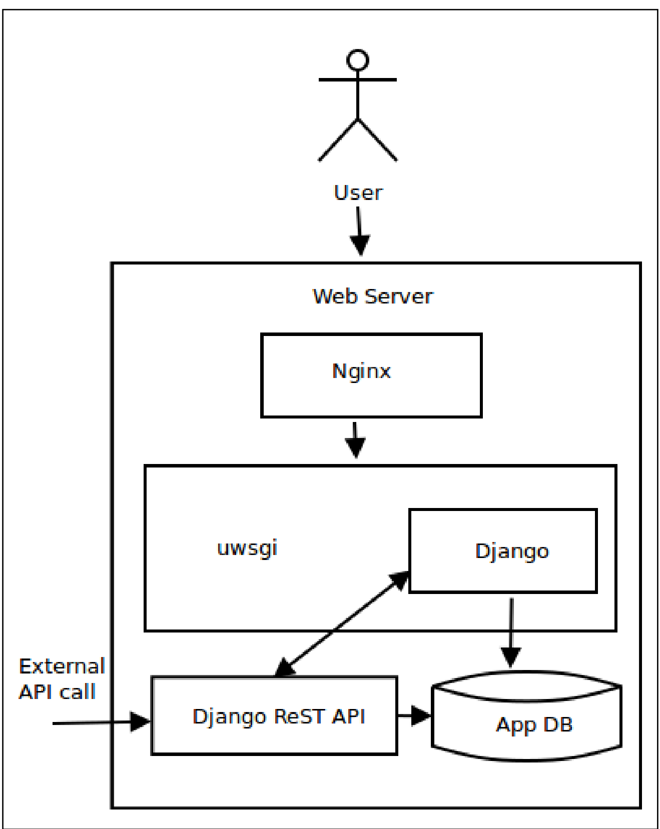
uWSGI also comes with a fast and in-memory caching framework, which allows the responses of the web applications to be stored in multiple caches on the uWSGI server. The cache can also be backed with a persistence store such as a file. Apart from a multitude of other things, uWSGI also supports virtualenv based deployments in Python.

uWSGI also provides a native protocol that is used by the uWSGI server. uWSGI version 1.9 also adds native support for the web sockets.

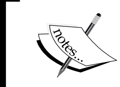
Here is a typical example of a uWSGI configuration file:

```
[uwsgi]

# the base directory (full path)
chdir           = /home/user/my-django-app/
# Django's wsgi file
module          = app.wsgi
# the virtualenv (full path)
home            = /home/user/django-virtualenv/
# process-related settings
master          = true
# maximum number of worker processes
processes       = 10
# the socket
socket          = /home/user/my-django-app/myapp.sock
# clear environment on exit
vacuum          = true
```

A typical deployment architecture with uWSGI looks like that depicted in the following diagram. In this case, the web server is Nginx, and the web application framework is Django. uWSGI is deployed in a reverse-proxy configuration with Nginx, forwarding requests and responses between Nginx and Django:



uWSGI deployment with Nginx and Django



The Nginx web server supports a native implementation of the uWSGI protocol since version 0.8.40. There is also a proxy module support for uWSGI in Apache named `mod_proxy_uwsgi`.

uWSGI is an ideal choice for Python web application production deployments where one needs a good balance of customization with high performance and features. It is the swiss-army-knife of components for WSGI web application deployments.

Gunicorn – unicorn for WSGI

The Gunicorn project is another popular WSGI middleware implementation, which is open source. It uses a preforked model, and is a ported version from the unicorn project of Ruby. There are different worker types in Gunicorn, like uWSGI supporting synchronous and asynchronous handling of requests. The asynchronous workers make use of the `Greenlet` library which is built on top of gevent.

There is a master process in Gunicorn that runs an event loop, processing and reacting to various signals. The master manages the workers, and the workers process the requests, and send responses.

Gunicorn versus uWSGI

Here are a few guidelines when choosing whether to go with Gunicorn or uWSGI for your Python web application deployments:

- For simple application deployments which don't need a lot of customization, Gunicorn is a good choice. uWSGI has a bigger learning curve when compared to Gunicorn, and it takes a while to get used to. The defaults in Gunicorn work pretty well for most deployments.
- If your deployment is homogenously Python, then Gunicorn is a good choice. On the other hand, uWSGI allows you to perform heterogeneous deployments due to its support for other stacks such as PSGI and Rack.
- If you want a more full-featured WSGI middleware, which is heavily customizable, then uWSGI is a safe bet. For example, uWSGI makes Python virtualenv-based deployments simple, whereas, Gunicorn doesn't natively support virtualenv; instead, Gunicorn itself has to be deployed in the virtual environment.
- Since Nginx supports uWSGI natively, it is very commonly deployed along with Nginx on production systems. Hence, if you use Nginx, and want a full-featured and highly customizable WSGI middleware with caching, uWSGI is the default choice.
- With respect to performance, both Gunicorn and uWSGI score similarly on different benchmarks published on the web.

Scalability architectures

As discussed, a system can scale vertically, or horizontally, or both. In this section, we will briefly look at a few of the architectures that an architect can choose from when deploying his systems to production to take advantage of the scalability options.

Vertical scalability architectures

Vertical scalability techniques come in the following two flavors:

- **Adding more resources to an existing system**: This could mean adding more RAM to a physical or virtual machine, adding more vCPUs to a virtual machine or VPS, and so on. However, none of these options are dynamic, as they require stopping, reconfiguring, and restarting the instance.
- **Making better use of existing resources in the system**: We have spent a lot of this chapter discussing this approach. This is when an application is rewritten to make use of the existing resources, such as multiple CPU cores, and more effectively by concurrency techniques such as threading, multiple processes, and/or asynchronous processing. This approach scales dynamically, since no new resource is added to the system, and hence, there is no need for a stop/start.

Horizontal scalability architectures

Horizontal scalability involves a number of techniques that an architect can add to his tool box, and pick and choose from. They include the ones listed next:

- **Active redundancy**: This is the simplest technique of scaling out, which involves adding multiple, homogenous processing nodes to a system typically fronted with a load balancer. This is a common practice for scaling out web application server deployments. Multiple nodes make sure that, even if one or a few of the systems fail, the remaining systems continue to carry out request processing, ensuring no downtime for your application.

  In a redundant system, all the nodes are actively in operation, though only one or a few of them may be responding to requests at a specific time.

- **Hot standby**: A hot standby (hot spare) is a technique used to switch to a system that is ready to server requests, but is not active until the moment the main system goes down. A hot spare is in many ways exactly similar to the main node(s) that is serving the application. In the event of a critical failure, the load balancer is configured to switch to the hot spare.

The hot spare itself may be a set of redundant nodes instead of just a single node. Combining redundant systems with a hot spare ensures maximum reliability and failover.

A variation of a hot standby is a software standby, which provides a mode in the application that switches the system to a minimum **Quality of Service** (**QoS**) instead of offering the full feature at extreme load. An example is a web application that switches to the read-only mode under high loads, serving most users but not allowing writes.

- **Read replicas**: The response of a system that is dependent on read-heavy operations on a database can be improved by adding read-replicas of the database. Read replicas are essentially database nodes that provide hot backups (online backups), which constantly sync from the main database node. Read replicas, at a given point in time, may not be exactly consistent with the main database node, but they provide eventual consistency with SLA guarantees.

  Cloud service providers such as Amazon make their RDS database service available with a choice of read replicas. Such replicas can be distributed geographically closer to your active user locations to ensure less response time and failover in case the master node goes down, or doesn't respond.

  Read replicas basically offer your system a kind of data redundancy.

- **Blue-green deployments**: This is a technique where two separate systems (labeled `blue` and `green` in the literature) are run side by side. At any given moment, only one of the systems is active and is serving requests. For example, blue is *active*, green is *idle*.

  When preparing a new deployment, it is done on the idle system. Once the system is ready, the load balancer is switched to the idle system (green), and away from the active system (blue). At this point, green is active, and blue is idle. The positions are reversed again in the next switch.

  Blue-green deployments, if done correctly, ensure zero to minimum downtime of your production applications.

- **Failure monitoring and/or restart**: A failure monitor is a system that detects failure of critical components—software or hardware—of your deployments, and either notifies you, and/or takes steps to mitigate the downtime.

  For example, you can install a monitoring application on your server that detects when a critical component, say, a Celery or RabbitMQ server, goes down, sends an e-mail to the DevOps contact, and also tries to restart the daemon.

Heartbeat monitoring is another technique where a software actively sends pings or heartbeats to a monitoring software or hardware, which could be in the same machine or another server. The monitor will detect the downtime of the system if it fails to send the heartbeat after a certain interval, and could then inform and/or try to restart the component.

Nagios is an example of a common production monitoring server, usually deployed in a separate environment, and monitors your deployment servers. Other examples of system-switch monitors and restart components are **Monit** and **Supervisord**.

Apart from these techniques, the following best practices should be followed when performing system deployments to ensure scalability, availability, and redundancy/failover:

- **Cache it**: Use caches, and, if possible, distributed caches, in your system as much as possible. Caches can be of various types. The simplest possible cache is caching static resources on the **content delivery network** (**CDN**) of your application service provider. Such a cache ensures geographic distribution of resources closer to your users, which reduces response, and hence, page-load times.

  A second kind of cache is your application's cache, where it caches responses and database query results. Memcached and Redis are commonly used for these scenarios, and they provide distributed deployments, typically, in master/slave modes. Such caches should be used to load and cache most commonly requested content from your application with proper expiry times to ensure that the data is not too stale.

  Effective and well-designed caches minimize system load, and avoid multiple, redundant operations that can artificially increase load on a system and decrease performance:

- **Decouple**: As much as possible, decouple your components to take advantage of the shared geography of your network. For example, a message queue may be used to decouple components in an application that need to publish and subscribe data instead of using a local database or sockets in the same machine. When you decouple, you automatically introduce redundancy and data backup to your system, since the new components you add for decoupling—message queues, task queues, and distributed caches—typically come with their own stateful storage and clustering.

  The added complexity of decoupling is the configuration of the extra systems. However, in this day and age, with most systems being able to perform auto configuration or provide simple web-based configurations, this is not an issue.

You can refer to literature for application architectures that provide effective decoupling, such as observer patterns, mediators, and other such middleware:

- **Gracefully degrade**: Rather than being unable to answer a request and providing timeouts, arm your systems with graceful degradation behaviors. For example, a write-heavy web application can switch to the read-only mode under heavy load when it finds that the database node is not responding. Another example is when a system which provides heavy, JS-dependent dynamic web pages could switch to a similar static page under heavy loads on the server when the JS middleware is not responding well.

  Graceful degradation can be configured on the application itself, or on the load balancers, or both. It is a good idea to prepare your application itself to provide a gracefully downgraded behavior, and configure the load balancer to switch to that route under heavy loads.

- **Keep data close to the code:** A golden rule of performance-strong software is to provide data closer to where the computation is. For example, if your application is making 50 SQL queries to load data from a remote database for every request, then you are not doing this correctly.

  Providing data close to the computation reduces data access and transport times, and hence, processing times, decreasing latency in your application, and making it more scalable.

  There are different techniques for this: caching, as discussed earlier, is a favored technique. Another one is to split your database to a local and remote one, where most of the reads happen from the local read replica, and writes (which can take time) happen to a remote write master. Note that local, in this sense, may not mean the same machine, but typically, the same data center, sharing the same subnet if possible.

  Also, common configurations can be loaded from an on-disk database like SQLite or local JSON files, reducing the time it takes for preparing the application instances.

  Another technique is to not store any transactional state in the application tier or the frontend, but to move the state closer to the backend where the computation is. Since this makes all application server nodes equal in terms of not having any intermediate state, it also allows you to front them with a load-balancer, and provide a redundant cluster of equals, any of which can serve a given request.

- **Design according to SLAs**: It is very important for an architect to understand the guarantees that the application provides to its users, and design the deployment architecture accordingly.

The CAP theorem ensures that, if a network partition in a distributed system fails, the system can guarantee only one of consistency or availability at a given time. This groups distributed systems into two common types, namely, CP and AP systems.

Most web applications in today's world are AP. They ensure availability, but data is only eventually consistent, which means they will serve stale data to users in case one of the systems in the network partition, say the master DB node, fails.

On the other hand, a number of businesses such as banking, finance, and healthcare need to ensure consistent data, even if there is a network partition failure. These are CP systems. The data in such systems should never be stale, so, in case of a choice between availability and consistent data, they will choose the latter.

The choice of software components, application architecture, and the final deployment architecture are influenced by these constraints. For example, an AP system can work with NoSQL databases which guarantee eventual consistent behavior. It can make better use of caches. A CP system, on the other hand, may need ACID guarantees provided by **Relational Database Systems** (**RDBMs**).

Summary

In this chapter, we reused a lot of ideas and concepts that you learned in the previous chapter on performance.

We started with a definition of scalability, and looked at its relationship with other aspects like concurrency, latency, and performance. We briefly compared and contrasted concurrency and its close cousin, parallelism.

We then went on to discuss various concurrency techniques in Python with detailed examples and performance comparisons. We used a thumbnail generator with random URLs from the Web as an example to illustrate the various techniques of implementing concurrency using multi-threading in Python. You also learned and saw an example of the producer/consumer pattern, and, using a couple of examples, learned how to implement resource constraints and limits using synchronization primitives.

Next we discussed how to scale applications using multiprocessing and saw a couple of examples using the `multiprocessing` module—such as a primality checker which showed us the effects of `GIL` on multiple threads in Python and a disk file sorting program which showed the limits of multiprocessing when it comes to scaling programs using a lot disk I/O .

We looked at asynchronous processing as the next technique of concurrency. We saw a generator based co-operative multitasking scheduler and also its counterpart using `asyncio`. We saw a couple of examples using asyncio and learned how to perform URL fetches using the aiohttp module asynchronously. The section on concurrent processing compared and contrasted concurrent futures with other options on concurrency in Python while sketching out a couple of examples.

We used Mandelbrot fractals as an example to show how to implement data parallel programs and showed an example of using `PyMP` to scale a Mandelbrot fractal program across multiple processes and hence multiple cores.

Next we went on to discuss how to scale your programs out on the Web. We briefly discussed the theoretical aspect of message queues and task queues. We looked at Celery, the Python task queue library, and rewrote the Mandelbrot program to scale using Celery workers, and did performance comparisons.

WSGI, Python's way of serving web applications over web servers, was the next topic of discussion. We discussed the WSGI specification, and compared and contrasted two popular WSGI middleware, namely, uWSGI and Gunicorn.

Towards the end of the chapter, we discussed scalability architectures, and looked at the different options of scaling vertically and horizontally on the Web. We also discussed some best practices an architect should follow while designing, implementing, and deploying distributed applications on the web for achieving high scalability.

In the next chapter, we discuss the issue of security in software architecture and discuss aspects of security the architect should be aware of and strategies for making your applications secure.

6
Security – Writing Secure Code

Security of software applications (or lack of it) has been attracting a lot of importance in the past few years in the industry and the media. It seems that every other day, we hear about an instance or two of malicious hackers causing massive data breaches in software systems in different parts of the world, and causing millions of dollars worth of losses. The victims are either government departments, financial institutions, firms handling sensitive customer data such as passwords, credit cards, and so on.

Software security and secure coding has assumed more importance than ever due to the unprecedented amounts of data being shared across software and hardware systems—the explosion of smart personal technologies such as smart phones, smart watches, smart music players, and other smart systems has aided this immense traffic of data across the Internet in a big way. With the advent of IPv6 and expected large scale adoption of **IoT** devices (**Internet of Things**) in the next few years, the amount of data is only going to increase exponentially.

As we discussed in the first chapter, security is an important aspect of software architecture. Apart from architecting systems with secure principles, architects should also try to imbibe their team with secure coding principles to minimize security pitfalls in the code written by them.

In this chapter, we will look at the principles of architecting secure systems, and also look at tips and techniques for writing secure code in Python.

The topics we will be discussing can be summed up in the following list:

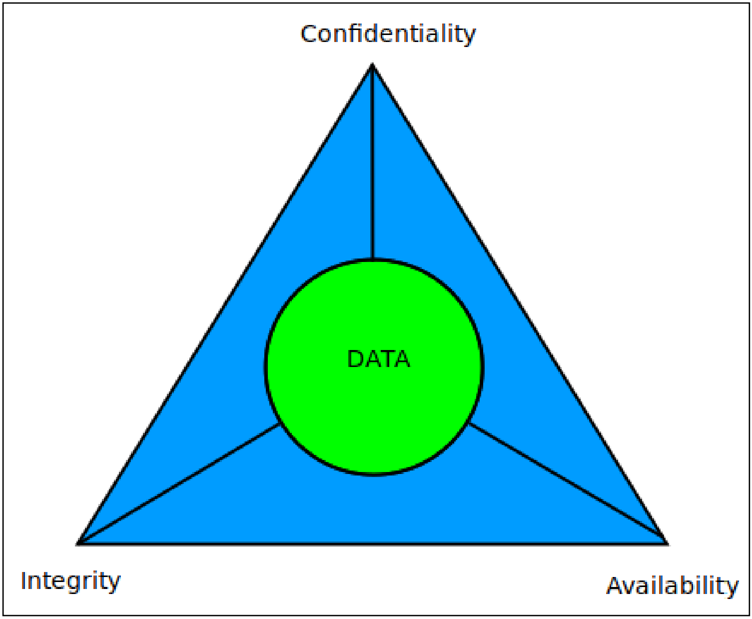
- Information security architecture
- Secure coding
- Common security vulnerabilities
- Is Python secure?
    - Reading input
    - Evaluating arbitrary input
    - Overflow errors
    - Serializing objects
    - Security issues with web applications
- Strategies for Security – Python
- Secure coding strategies

Information security architecture

A secure architecture involves creating a system that is able to provide access to data and information to authorized people and systems while preventing any unauthorized access. Creating an architecture for information security for your systems involves the following aspects:

- **Confidentiality**: A set of rules or procedures that restricts the envelope of access to information in the system. Confidentiality ensures that data is not exposed to unauthorized access or modification.
- **Integrity**: Integrity is the property of the system which ensures that the information channels are trustworthy and reliable and that the system is free from external manipulations. In other words, integrity ensures the data can be trusted as it flows through the system across its components.
- **Availability**: Property that the system will ensure a level of service to its authorized users according to its **Service Level Agreements** (**SLAs**). Availability ensures that the system will not deny service to its authorized users.

The three aspects of confidentiality, integrity, and availability, often called the CIA triad form the corner stones of building an information security architecture for your system.



CIA triad of information security architecture

These aspects are aided by other characteristics, such as the following:

- **Authentication**: This verifies the identity of the participants of a transaction, and ensures that they are actually those who they purport to be. Examples are digital certificates used in e-mail, public keys used to log in to systems, and the like.
- **Authorization**: This gives rights to a specific user/role to perform a specific task or groups of related tasks. Authorization ensures that certain groups of users are tied to certain roles, which limit their access and modification rights in the system.
- **Non-reputability**: This refers to security techniques that guarantee that users involved in a transaction cannot later deny that the transaction happened. For example, a sender of an e-mail cannot later deny that they had sent the e-mail; a recipient of a bank funds transfer cannot later deny that they received the money, and so on.

Secure coding

Secure coding is the practice of software development that guards programs against security vulnerabilities, and makes it resistant to malicious attacks right from program design to implementation. It is about writing code that is inherently secure, as opposed to thinking of security as a layer which is added on later.

The philosophies behind secure coding include the following:

- Security is an aspect to be considered right from the design and development of a program or application; it is not an afterthought.
- Security requirements should be identified early in the development cycle, and these should be propagated to subsequent stages of development of the system to make sure that compliance is maintained.
- Use threat modeling to anticipate security threats to the system from the beginning. Threat modeling involves the following:
    1. Identifying important assets (code/data)
    2. Decomposing the application into components
    3. Identifying and categorizing threats to each asset or component
    4. Ranking the threats based on an established risk model
    5. Developing threat mitigation strategies

The practice or strategies of secure coding include the following main tasks:

1. **Definition of areas of interest of the application**: Identify important assets in code/data of the application which are critical and need to be secured.
2. **Analysis of software architecture**: Analyze the software architecture for obvious security flaws. Secure interaction between components in order to help ensure data confidentiality and integrity. Ensure confidential data is protected via proper authentication and authorization techniques. Ensure availability is built into the architecture from the ground up.
3. **Review of implementation details**: Review the code using secure coding techniques. Ensure peer review is done with a view to finding security holes. Provide feedback to the developer and make sure the required changes are made.
4. **Verification of logic and syntax**: Review code logic and syntax to ensure there are no obvious loopholes in the implementation. Make sure programming is done keeping with in commonly available secure coding guidelines of the programming language/platform.

5. **Whitebox/Unit Testing**: The developer unit tests his code with security tests apart from tests ensuring functionality. Mock data and/or APIs can be used to virtualize third party data/API required for testing.
6. **Blackbox Testing**: The application is tested by an experienced QA engineer who looks for security loopholes such as unauthorized access to data, pathways accidentally exposing code and or data, weak passwords or hashes etc. The testing reports are fed back the stakeholders including the architect to make sure the loopholes identified are fixed.

Common security vulnerabilities

So what are the common security vulnerabilities, a professional programmer today should be prepared to face and mitigate during the course of their career? Looking at the available literature, these can be organized into a few specific categories:

- **Overflow errors**: These include the popular and often abused **buffer overflow** errors, and the lesser known but still vulnerable **arithmetic or integer overflow** errors:
    - **Buffer overflow**: Buffer overflows are produced by programming errors that allow an application to write past the end or beginning of a buffer. Buffer overflows allow attackers to take control of systems by gaining access to the applications stack or heap memory by carefully crafted attack data.
    - **Integer or arithmetic overflow**: These errors occur when an arithmetic or mathematical operation on integers produces a result that is too large for the maximum size of the type used to store it.

  Integer overflows can create security vulnerabilities if they are not properly handled. In programming languages supporting signed and unsigned integers, overflows can cause the data to wrap and produce negative numbers, allowing the attacker with a result similar to buffer overflows to gain access to heap or stack memory outside the program execution limits.

- **Unvalidated/Improperly validated input**: A very common security issue with modern web applications, unvalidated input can cause major vulnerabilities, where attackers can trick a program into accepting malicious input such as code data or system commands, which, when executed, can compromise a system. A system that aims to mitigate this type of attack should have filters to check and remove content that is malicious, and only accept data that is reasonable and safe to the system.

  Common subtypes of this type of attack include SQL injections, Server-Side Template Injections, **Cross-Site-Scripting** (**XSS**), and Shell Execution Exploits.

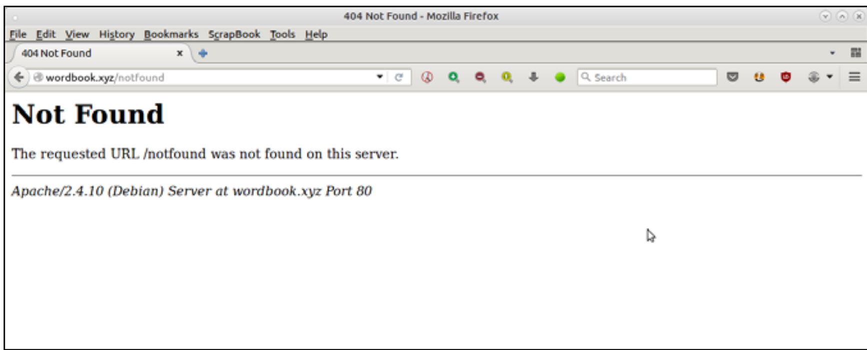
  Modern web application frameworks are vulnerable to this kind of attack due to use of HTML templates which mix code and data, but many of them have standard mitigation procedures such as escaping or filtering of input.

- **Improper access control**: Modern day applications should define separate roles for their classes of users, such as regular users, and those with special privileges, such as superusers or administrators. When an application fails to do this or does it incorrectly, it can expose routes (URLs) or workflows (series of actions specified by specific URLs containing attack vectors), which can either expose sensitive data to attackers, or, in the worst case, allow an attacker to compromise and take control of the system.
- **Cryptography issues**: Simply ensuring that access control is in place is not enough for hardening and securing a system. Instead, the level and strength of security should be verified and ascertained; otherwise, your system can still be hacked or compromised. Some examples are as follows:
    - **HTTP instead of HTTPS**: When implementing RESTFul web services, make sure you favor HTTPS (SSL/TLS) over HTTP. In HTTP, all communication is in plain text between the client and server, and can be easily captured by passive network sniffers or carefully crafted packet capture software or devices installed in routers.

      Projects like letsencrypt have made life easy for system administrators for procuring and updating free SSL certificates, so securing your servers using SSL/TLS is easier these days than ever before.

- **Insecure authentication**: Prefer secure authentication techniques on a web server over insecure ones. For example, prefer HTTP Digest authentication to Basic authentication on web servers, as, in the latter, passwords are sent in the clear. Similarly, use **Kerberos** authentication in a large shared network over less secure alternatives such as **Lightweight Directory Access Protocol** (**LDAP**) or **NT LAN Manager** (**NTLM**).
- **Use of weak passwords**: Easy-to-guess or default/trivial passwords are the bane of many modern-day web applications.
- **Reuse of secure hashes/secret keys**: Secure hashes or secret keys are usually specific to an application or project and should never be reused across applications. Whenever required, generate fresh hashes and or keys.
- **Weak encryption techniques**: Ciphers used in encrypting communication, either on the server (SSL certificates) or personal computers (GPG/PGP keys), should use high-grade security – of at least 2048 bits and use peer-reviewed and crypto-safe algorithms.
- **Weak hashing techniques**: Just as in ciphers, hashing techniques used to keep secrets and salts of sensitive data such as passwords, should be careful in choosing strong algorithms. For example, if one is writing an application that requires hashes to be computed and stored today, they would be better off using the SHA-1 or SHA-2 algorithms rather than the weaker MD5.
- **Invalid or expired certificates/keys**: Web masters often forget to keep their SSL certificates updated, and this can become a big problem, compromising the security of their web servers, as invalid certificates offer no protection. Similarly, personal keys such as GPG or PGP public/private key pairs used for e-mail communication should be kept updated.
- **Password enabled SSH**: SSH access to remote systems using clear text passwords is a security hole. Disable password-based access and only enable access via authorized SSH keys for specific users only. Disable remote root SSH access.

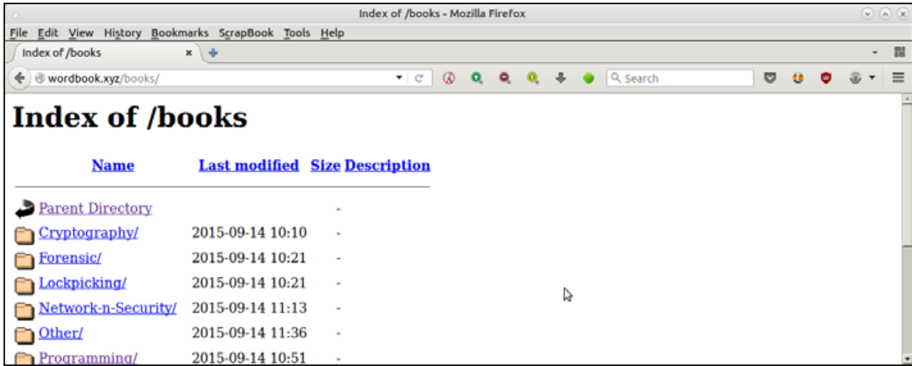
- **Information leak**: A lot of web servers systems—mostly due to open configuration, or misconfiguration, or due to lack of validation of inputs—can reveal a lot of information about themselves to an attacker. Some examples are as follows:
    - **Server meta information**: Many web servers leak information about themselves via their 404 pages, and sometimes, via their landing pages. Here is an example:



404 page of a web server exposing server meta information

By simply requesting for a non-existing page, we came to know that the site seen in the preceding screenshot runs Apache version 2.4.10 on a Debian Server. For a crafty attacker, this is often information enough to try out specific attacks for that particular web-server/OS combination.

- **Open index pages**: Many websites don't protect their directory pages, and leave them open for world access. This screenshot shows an example:



Open index page of a web server

Open ports: It is a common error to provide world-access to an application's ports running on remote web servers instead of limiting access to them by specific IP addresses or security groups by using firewalls – such as *iptables*. A similar error is to allow a service to run on 0.0.0.0 (all IP addresses on the server) for a service which is only consumed on the localhost. This makes it easy for attackers to scan for such ports using network reconnaissance tools such as `nmap/hping3`, and the like, and plan their attack.

- **Open access to files/folders/databases**: A very poor practice is to provide open or world access to application configuration files, log files, process ID files, and other artifacts so that any logged-in user can access and obtain information from these files. Instead, such files should be part of security policies to ensure that only specific roles with the required privileges have access to the files.
- **Race conditions**: A race condition exists when a program has two or more actors trying to access a certain resource, but the output depends on the correct order of access, which cannot be ensured. An example is two threads trying to increment a numerical value in shared memory without proper synchronization.

  Crafty attackers can take advantage of the situation to insert malicious code, change a filename, or sometimes, take advantage of small time gaps in the processing of code to interfere with the sequence of operations.
- **System clock drifts**: This is the phenomena on where the system or local clock time on a server slowly drifts away from the reference time due to improper or missing synchronization. Over time, the clock drift can cause serious security flaws such as error in SSL certificate validation, which can be exploited by highly sophisticated techniques like *timing attacks* where an attacker tries to take control over the system by analyzing the time taken to execute cryptographic algorithms. Time synchronization protocols like NTP can be used to mitigate this.
- **Insecure file/folder operations**: Programmers often make assumptions about the ownership, location, or attributes of a file or folder that might not be true in practice. This can result in conditions where a security flaw can occur or where we may not detect tampering with the system. Some examples are as follows:
  - Failing to check results after a write operation, assuming it succeeded
  - Assuming local file paths are always local files (whereas, they might be symbolic links to system files for which the application may not have access)

- Improperly using sudo in executing system commands, which, if not done correctly, can cause loopholes, which can be used to gain root access of the system
- Generous use of permissions on shared files or folders, for example, turning on all the execute bits of a program which should be limited to a group, or open home folders which can be read by any logged in user
- Using unsafe serialization and deserialization of code or data objects

It is beyond the scope of this chapter to visit each and every type of vulnerability in this list. However, we will make an earnest attempt to review and explain the common classes of software vulnerabilities that affect Python, and some of its web frameworks in the coming section.

Is Python secure?

Python is a very readable language with simple syntax, and typically, one clearly stated way to do things. It comes with a set of well-tested and compact standard library modules. All of this seems to indicate that Python should be a very secure language.

But is it so?

Let's look at a few examples in Python, and try to analyze the security aspect of Python and its standard libraries.

For the purposes of usefulness, we will demonstrate the code examples shown in this section using both Python 2.x and Python 3.x versions. This is because a number of security vulnerabilities that are present in Python 2.x versions are fixed in the recent 3.x versions. However, since many Python developers are still using some form or the other of Python 2.x, the code examples would be useful to them, and also illustrate the importance of migrating to Python 3.x.

All examples are executed on a machine running the Linux (Ubuntu 16.0), x86_64 architecture:

```
$ python3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print (sys.version)
3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609]
```

```
$ python2
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print sys.version
2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609]
```

Python 3.x version used for these examples are Python 3.5.2, and the Python 2.x version used is Python 2.7.12. All examples are executed on a machine running the Linux (Ubuntu 16.0), 64 bit x86 architecture.

Most of the examples will use one version of code, which will run both in Python 2.x and Python 3.x. In cases where this is not possible, two versions of the code will be listed.

Reading input

Let's look at this program that is a simple guessing game. It reads a number from the standard input, and compares it with a random number. If it matches, the user wins, otherwise, the user has to try again:

```
# guessing.py
import random

# Some global password information which is hard-coded
passwords={"joe": "world123",
          "jane": "hello123"}

def game():
     """A guessing game """

    # Use 'input' to read the standard input
    value=input("Please enter your guess (between 1 and 10): ")
    print("Entered value is",value)
```

```
        if value == random.randrange(1, 10):
            print("You won!")
        else:
            print("Try again")

    if __name__ == "__main__":
        game()
```

The preceding code is simple, except that it has some sensitive global data, which is the passwords of some users in the system. In a realistic example, these could be populated by some other functions, which read the passwords and cache them in memory.

Let's try the program with some standard inputs. We will initially run it with Python 2.7, as follows:

```
$ python2 guessing.py
Please enter your guess (between 1 and 10): 6
('Entered value is', 6)
Try again

$ python2 guessing.py
Please enter your guess (between 1 and 10): 8
('Entered value is', 8)
You won!
```

Now, let's try a "non-standard" input:

```
$ python2 guessing.py
Please enter your guess (between 1 and 10): passwords
('Entered value is', {'jane': 'hello123', 'joe': 'world123'})
Try again
```

Note how the preceding run exposed the global password data!

The problem is that in Python 2, the input value is evaluated as an expression without doing any check, and when it is printed, the expression prints its value. In this case, it happens to match a global variable, so its value is printed out.

Now let's look at this one:

```
$ python2 guessing.py
Please enter your guess (between 1 and 10): globals()
('Entered value is', {'passwords': {'jane': 'hello123',
'joe' : 'world123'}, '__builtins__': <module '__builtin__' (built-
in)>,
 '__file__': 'guessing.py', 'random':
```

```
<module 'random' from '/usr/lib/python2.7/random.pyc'>,
 '__package__': None, 'game':
<function game at 0x7f6ef9c65d70>,
 '__name__': '__main__', '__doc__': None})
Try again
```

Now, not only has it exposed the passwords, it has exposed the complete global variables in the code including the passwords. Even if there were no sensitive data in the program, a hacker using this approach can reveal valuable information about the program such as variable names, function names, packages used, and so on.

What is the fix for this? For Python 2, one solution is to replace `input`, which evaluates its contents by passing directly to `eval`, with `raw_input`, which doesn't evaluate the contents. Since `raw_input` doesn't return a number, it needs to be converted to the target type. (This can be done by casting the return data to an `int`.) The following code does not only that, but also adds an exception handler for the type conversion for extra safety:

```
# guessing_fix.py
import random

passwords={"joe": "world123",
           "jane": "hello123"}

def game():
    value=raw_input("Please enter your guess (between 1 and 10): ")
    try:
        value=int(value)
    except TypeError:
        print ('Wrong type entered, try again',value)
        return

    print("Entered value is",value)
    if value == random.randrange(1, 10):
        print("You won!")
    else:
        print("Try again")

if __name__ == "__main__":
    game()
```

Let's see how this version fixes the security hole in evaluating inputs

```
$ python2 guessing_fix.py
Please enter your guess (between 1 and 10): 9
('Entered value is', 9)
Try again

$ python2 guessing_fix.py
Please enter your guess (between1 and 10): 2
('Entered value is', 2)
You won!

$ python2 guessing_fix.py
Please enter your guess (between 1 and 10): passwords
(Wrong type entered, try again =>, passwords)

$ python2 guessing_fix.py
Please enter your guess (between 1 and 10): globals()
(Wrong type entered, try again =>, globals())
```

The new program is now much more secure than the first version.

This problem is not there in Python 3.x as the following illustration shows. (We are using the original version to run this.)

```
$ python3 guessing.py
Please enter your guess (between 1 and 10): passwords
Entered value is passwords
Try again

$ python3 guessing.py
Please enter your guess (between 1 and 10): globals()
Entered value is globals()
Try again
```

Evaluating arbitrary input

The `eval` function in Python is very powerful, but it is also dangerous, since it allows one to pass arbitrary strings to it, which can evaluate to potentially dangerous code or commands.

Let's look at this rather silly piece of code as a test program to see what `eval` can do:

```
# test_eval.py
import sys
import os

def run_code(string):
    """ Evaluate the passed string as code """

    try:
        eval(string, {})
    except Exception as e:
        print(repr(e))

if __name__ == "__main__":
     run_code(sys.argv[1])
```

Let's assume a scenario where an attacker is trying to exploit this piece of code to find out the contents of the directory where the application is running. (For the time being, you can assume the attacker can run this code via a web application, but hasn't got direct access to the machine itself.)

Let's assume the attacker tries to list the contents of the current folder:

```
$ python2 test_eval.py "os.system('ls -a')"
NameError("name 'os' is not defined",)
```

This preceding attack doesn't work, because `eval` takes a second argument, which provides the global values to use during evaluation. Since, in our code, we are passing this second argument as an empty dictionary, we get the error, as Python is unable to resolve the `os` name.

So does this mean, `eval` is safe? No it's not. Let's see why.

What happens when we pass the following input to the code?

```
$ python2 test_eval.py "__import__('os').system('ls -a')"
.   guessing_fix.py  test_eval.py    test_input.py
..  guessing.py      test_format.py  test_io.py
```

We can see that we are still able to coax `eval` to do our bidding by using the built-in function `__import__`.

The reason why this works is because names such as `__import__` are available in the default built-in `__builtins__` global. We can deny `eval` this by specifically passing this as an empty dictionary via the second argument. Here is the modified version:

```
# test_eval.py
import sys
import os

def run_code(string):
    """ Evaluate the passed string as code """

    try:
        # Pass __builtins__ dictionary as empty
        eval(string,  {'__builtins__':{}})
    except Exception as e:
        print(repr(e))

if __name__ == "__main__":
  run_code(sys.argv[1])
```

Now the attacker is not able to exploit via the built-in `__import__`:

```
$ python2 test_eval.py "__import__('os').system('ls -a')"
NameError("name '__import__' is not defined",)
```

However, this doesn't still make `eval` any safer, as it is open to slightly longer, but clever attacks. Here is one such attack:

```
$ python2 test_eval.py "(lambda f=(lambda x: [c for c in [].__
class__.__bases__[0].__subclasses__() if c.__name__ == x][0]):
f('function')(f('code')(0,0,0,0,'BOOM',(), (),(),'','',0,''),{})())()"
Segmentation fault (core dumped)
```

We are able to core dump the Python interpreter with a rather obscure looking piece of malicious code. How did this happen ?

Here is a somewhat detailed explanation of the steps.

First, let's consider this:

```
>>> [].__class__.__bases__[0]
<type 'object'>
```

This is nothing but the base-class `object`. Since we don't have access to the built-ins, this is an indirect way to get access to it.

Next, the following line of code loads all the sub-classes of `object` currently loaded in the Python interpreter:

```
>>> [c for c in [].__class__.__bases__[0].__subclasses__()]
```

Among them, what we want is the `code` object type. This can be accessed by checking the name of the item via the `__name__` attribute:

```
>>> [c for c in [].__class__.__bases__[0].__subclasses__() if c.__
name__ == 'code']
```

Here is the same achieved by using an anonymous `lambda` function:

```
>>> (lambda x: [c for c in [].__class__.__bases__[0].__subclasses__()
if c.__name__ == x])('code')
[<type 'code'>]
```

Next, we want to execute this code object. However, `code` objects cannot be called directly. They need to be tied to a function in order for them to be called. This is achieved by wrapping the preceding `lambda` function in an outer `lambda` function:

```
>>> (lambda f: (lambda x: [c for c in [].__class__.__bases__[0].__
subclasses__() if c.__name__ == x])('code'))
<function <lambda> at 0x7f8b16a89668
```

Now our inner `lambda` function can be called in two steps:

```
>>> (lambda f=(lambda x: [c for c in [].__class__.__bases__[0].__
subclasses__() if c.__name__ == x][0]): f('function')(f('code')))
<function <lambda> at 0x7fd35e0db7d0>
```

We finally invoke the `code` object via this outer `lambda` function by passing mostly default arguments. The code-string is passed as the string `BOOM`, which is, of course, a bogus code-string that causes the Python interpreter to segfault, producing a core-dump:

```
>>> (lambda f=(lambda x:
[c for c in [].__class__.__bases__[0].__subclasses__() if c.__name__
== x][0]):
f('function')(f('code')(0,0,0,0,'BOOM',(), (),(),'','',0,''),{})())()
Segmentation fault (core dumped)
```

This shows that `eval` in any context, even bereft of built-in module support, is unsafe, and can be exploited by a clever and malicious hacker to crash the Python interpreter, and thereby, possibly gain control over the system.

Note that the same exploit works in Python 3 as well, but we need some modification in the arguments to the `code` object, as in Python 3, `code` objects takes an extra argument. Also, the code-string and some arguments must be the `byte` type.

The following is the exploit running on Python 3. The end result is the same:

```
$ python3 test_eval.py
"(lambda f=(lambda x: [c for c in ().__class__.__bases__[0].__
  subclasses__()
  if c.__name__ == x][0]): f('function')(f('code')(0,0,0,0,0,b't\x00\
  x00j\x01\x00d\x01\x00\x83\x01\x00\x01d\x00\x00S',(),
  (),(),'','',0,b''),{})())()"
Segmentation fault (core dumped)
```

Overflow errors

In Python 2, the `xrange()` function produces an overflow error if the range cannot fit into the integer range of Python:

```
>>> print xrange(2**63)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
OverflowError: Python int too large to convert to C long
```

The `range()` function also overflows with a slightly different error:

```
>>> print range(2**63)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
OverflowError: range() result has too many items
```

The problem is that `xrange()` and `range()` use plain integer objects (type `<int>`) instead of automatically getting converted to the `long` type, which is limited only by the system memory.

However, this problem is fixed in the Python 3.x versions, as types `int` and `long` are unified into one (`int` type), and the `range()` objects manage the memory internally. Also, there is no longer a separate `xrange()` object:

```
>>> range(2**63)
range(0, 9223372036854775808)
```

Here is another example of integer overflow errors in Python, this time for the `len` function.

In the following examples, we try the `len` function on instances of two classes A and B, whose magic method `__len__` has been over-ridden to provide support for the `len` function. Note that A is a new-style class, inheriting from `object` and B is an old-style class:

```
# len_overflow.py

class A(object):
    def __len__(self):
        return 100 ** 100

class B:
    def __len__(self):
        return 100 ** 100


try:
    len(A())
    print("OK: 'class A(object)' with 'return 100 ** 100' - len
         calculated")
except Exception as e:
    print("Not OK: 'class A(object)' with 'return 100 ** 100' - len
         raise Error: " + repr(e))

try:
    len(B())
    print("OK: 'class B' with 'return 100 ** 100' - len calculated")
except Exception as e:
    print("Not OK: 'class B' with 'return 100 ** 100' - len raise
         Error: " + repr(e))
```

Here is the output of the code when executed with Python2:

```
$ python2 len_overflow.py
Not OK: 'class A(object)' with 'return 100 ** 100' - len raise Error:
OverflowError('long int too large to convert to int',)
Not OK: 'class B' with 'return 100 ** 100' - len raise Error:
TypeError('__len__() should return an int',)
```

The same code is executed in Python 3 as follows:

```
$ python3 len_overflow.py
Not OK: 'class A(object)' with 'return 100 ** 100' - len raise Error:
OverflowError("cannot fit 'int' into an index-sized integer",)
Not OK: 'class B' with 'return 100 ** 100' - len raise Error:
OverflowError("cannot fit 'int' into an index-sized integer",)
```

The problem in the preceding code is that `len` returns `integer` objects, and in this case, the actual value is too large to fit inside an `int`, so Python raises an overflow error. In Python 2, however, for the case when the class is not derived from `object`, the code executed is slightly different, which anticipates an `int` object, but gets `long` and throws a `TypeError` instead. In Python 3, both examples return overflow errors.

Is there a security issue with integer overflow errors such as this?

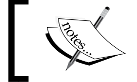
On the ground, it depends on the application code and the dependent module code used, and how they are able to deal with or mask the overflow errors.

However, since Python is written in C, any overflow errors which are not correctly handled in the underlying C code can lead to buffer overflow exceptions, where an attacker can write to the overflow buffer and hijack the underlying process, thereby gaining control over the application.

Typically, if a module or data structure is able to handle the overflow error and raise exceptions preventing further code execution, the chances of code exploitation are reduced.

Serializing objects

It is very common for Python developers to use the `pickle` module and its C implementation cousin `cPickle` for serializing objects in Python. However, both these modules allow unchecked execution of code, as they don't enforce any kind of type check or rules on the objects being serialized to verify whether it is a benign Python object or a potential command that can exploit the system.



NOTE: In Python3, both the `cPickle` and `pickle` modules are merged into a single `pickle` module.

Here is an illustration via a shell exploit, which lists the contents of the root folder (/) in a Linux/POSIX system:

```
# test_serialize.py
import os
import pickle

class ShellExploit(object):
    """ A shell exploit class """
```

```
    def __reduce__(self):
        # this will list contents of root / folder.
        return (os.system, ('ls -al /',)

def serialize():
    shellcode = pickle.dumps(ShellExploit())
    return shellcode

def deserialize(exploit_code):
    pickle.loads(exploit_code)

if __name__ == '__main__':
    shellcode = serialize()
    deserialize(shellcode)
```

The previous code simply packages a `ShellExploit` class, which, upon pickling, returns the command for listing the contents of the root filesystem / by way of the `os.system()` method. The `Exploit` class thus masquerades malicious code into a `pickle` object, which, upon unpickling, executes the code, and exposes the contents of the root folder of the machine to the attacker. The output of the preceding code is shown here:

```
$ python3 test_serialize.py
total 112
drwxr-xr-x  23 root root  4096 Mar 11 22:05 .
drwxr-xr-x  23 root root  4096 Mar 11 22:05 ..
drwxr-xr-x   2 root root 12288 Mar 11 22:56 bin
drwxr-xr-x   5 root root  4096 Mar 11 23:22 boot
drwxr-xr-x   2 root root  4096 Mar 11 22:02 cdrom
drwxr-xr-x  21 root root  4840 Apr 15 10:46 dev
drwxr-xr-x 160 root root 12288 Apr 14 19:26 etc
drwxr-xr-x   6 root root  4096 Mar 13 18:45 home
lrwxrwxrwx   1 root root    32 Mar 11 22:05 initrd.img -> boot/initrd.img-4.4.0-53-generic
drwxr-xr-x  26 root root  4096 Mar 11 22:56 lib
drwxr-xr-x   2 root root  4096 Dec 13 15:36 lib64
drwx------   2 root root 16384 Mar 11 21:57 lost+found
drwxr-xr-x   3 root root  4096 Apr  7 16:47 media
drwxr-xr-x   2 root root  4096 Dec 13 15:32 mnt
drwxr-xr-x   3 root root  4096 Apr  5 07:55 opt
dr-xr-xr-x 242 root root     0 Apr 14 08:33 proc
drwx------   9 root root  4096 Apr  2 17:08 root
drwxr-xr-x  41 root root  1260 Apr 15 10:52 run
drwxr-xr-x   2 root root 12288 Mar 11 22:56 sbin
drwxr-xr-x   2 root root  4096 Dec 13 15:32 srv
dr-xr-xr-x  13 root root     0 Apr 15 11:57 sys
drwxrwxrwt  20 root root  4096 Apr 15 11:57 tmp
drwxr-xr-x  10 root root  4096 Dec 13 15:32 usr
drwxr-xr-x  12 root root  4096 Dec 13 16:17 var
lrwxrwxrwx   1 root root    29 Mar 11 22:05 vmlinuz -> boot/vmlinuz-4.4.0-53-generic
$
```

Output of the shell exploit code for serializing using pickle, exposing contents of / folder.

As you can see, the output clearly lists the contents of the root folder.

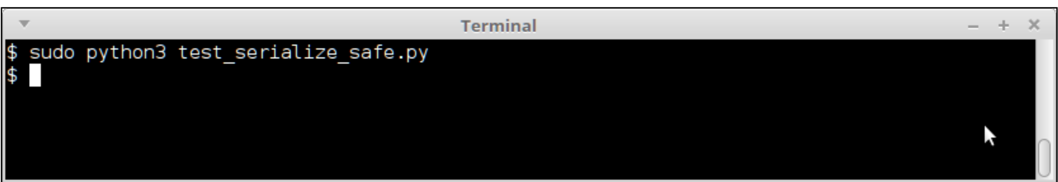
What is the work-around to prevent such exploits?

First of all, don't use an unsafe module like `pickle` for serialization in your applications. Instead, rely on a safer alternative like `json` or `yaml`. If your application really is dependent on using the `pickle` module for some reason, then use sand-boxing software or code jails to create safe environments that prevent execution of malicious code on the system.

For example, here is a slight modification of the earlier code, now with a simple `chroot` jail, which prevents code execution on the actual root folder. It uses a local `safe_root/` subfolder as the new root via a context-manager hook. Note that this is a simple minded example. An actual jail would be much more elaborate than this:

```
# test_serialize_safe.py
import os
import pickle
from contextlib import contextmanager

class ShellExploit(object):
    def __reduce__(self):
        # this will list contents of root / folder.
        return (os.system, ('ls -al /',))

@contextmanager
def system_jail():
    """ A simple chroot jail """

    os.chroot('safe_root/')
    yield
    os.chroot('/')

def serialize():
    with system_jail():
        shellcode = pickle.dumps(ShellExploit())
        return shellcode


def deserialize(exploit_code):
    with system_jail():
        pickle.loads(exploit_code)


if __name__ == '__main__':
    shellcode = serialize()
    deserialize(shellcode)
```

With this jail in place, the code executes as follows:



Output of the shell exploit code for serializing using pickle, with a simple chroot jail.

No output is produced now, because this is a fake jail, and Python cannot find the `ls` command in the new root. Of course, in order to make this work in a production system, a proper jail should be set up, which allows programs to execute, but at the same time, prevents or limits malicious program execution.

How about other serialization formats like JSON ? Can such exploits work with them? Let's see using an example.

Here is the same serialization code written using the `json` module:

```
# test_serialize_json.py
import os
import json
import datetime

class ExploitEncoder(json.JSONEncoder):
    def default(self, obj):
        if any(isinstance(obj, x) for x in (datetime.datetime,
                                            datetime.date)):
            return str(obj)

        # this will list contents of root / folder.
        return (os.system, ('ls -al /',))

def serialize():
    shellcode = json.dumps([range(10),
                            datetime.datetime.now()],
                           cls=ExploitEncoder)
    print(shellcode)
    return shellcode
```

```
def deserialize(exploit_code):
    print(json.loads(exploit_code))


if __name__ == '__main__':
    shellcode = serialize()
    deserialize(shellcode)
```

Note how the default JSON encoder has been overridden using a custom encoder named `ExploitEncoder`. However, as the JSON format doesn't support such serializations, it returns the correct serialization of the list passed as input:

```
$ python2 test_serialize_json.py
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], "2017-04-15 12:27:09.549154"]
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], u'2017-04-15 12:27:09.549154']
```

With Python3, the exploit fails as Python3 raises an exception.

```
Terminal
$ python3 test_serialize_json.py
Traceback (most recent call last):
  File "test_serialize_json.py", line 27, in <module>
    shellcode = serialize()
  File "test_serialize_json.py", line 17, in serialize
    cls=ExploitEncoder)
  File "/usr/lib/python3.5/json/__init__.py", line 237, in dumps
    **kw).encode(obj)
  File "/usr/lib/python3.5/json/encoder.py", line 198, in encode
    chunks = self.iterencode(o, _one_shot=True)
  File "/usr/lib/python3.5/json/encoder.py", line 256, in iterencode
    return _iterencode(o, 0)
ValueError: Circular reference detected
$
```

Output of the shell exploit code for serializing using json, with Python3

Security issues with web applications

So far, we have seen four types of security issues with Python, namely, those with reading input, evaluating expressions, overflow errors, and serialization issues. All our examples so far have been with Python on the console.

However, almost all of us interact with web applications on a daily basis, many of which are written in Python web frameworks such as Django, Flask, Pyramid, and others. Hence, it is more likely that we are exposed to security issues in such applications. We will look at a few examples here.

Server Side Template Injection

Server Side Template Injection (**SSTI**) is an attack using the server-side templates of common web frameworks as an attack vector. The attack uses weaknesses in the way user input is embedded on the templates. SSTI attacks can be used to figure out internals of a web application, execute shell commands, and even fully compromise the servers.

We will see an example using a very popular web application framework in Python, namely, Flask.

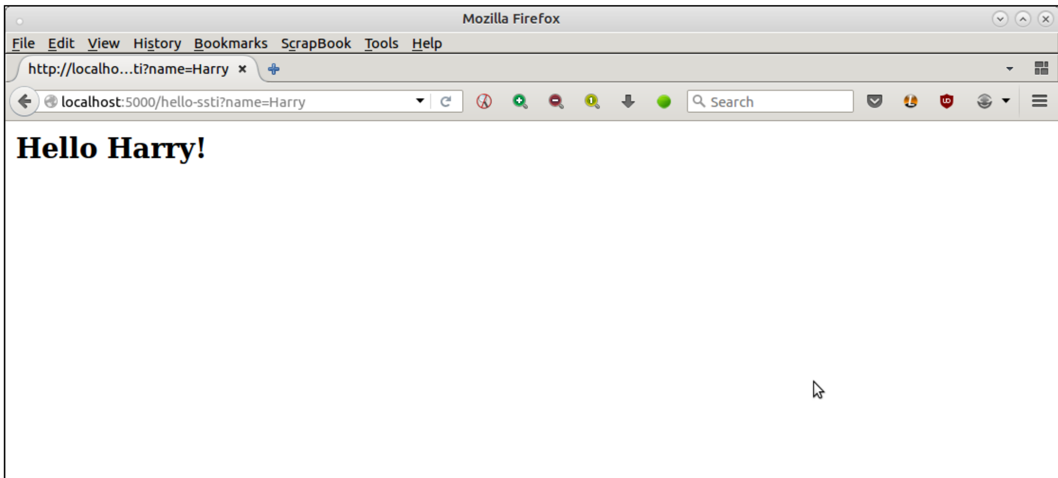
The following is the sample code for a rather simple web application in Flask with an inline template:

```
# ssti-example.py
from flask import Flask
from flask import request, render_template_string, render_template

app = Flask(__name__)

@app.route('/hello-ssti')
defhello_ssti():
    person = {'name':"world", 'secret':
'jo5gmvlligcZ5YZGenWnGcol8JnwhWZd2lJZYo=='}
    if request.args.get('name'):
        person['name'] = request.args.get('name')

    template = '<h2>Hello %s!</h2>' % person['name']
    return render_template_string(template, person=person)

if __name__ == "__main__":
    app.run(debug=True)
```

Running it on the console, and opening it in the browser allows us to play around with the `hello-ssti` route:

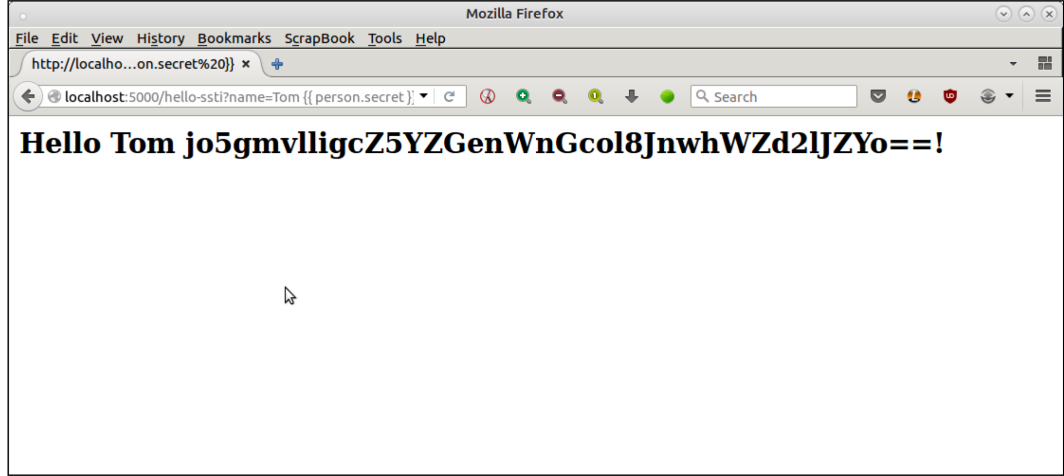
```
$ python3 ssti_example.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 163-936-023
```

First, let's try some benign inputs:



Here is another example.

Next, let's try with some crafty inputs which an attacker may use.



What is happening here?

Since the template uses unsafe `%s` string templates, it evaluates anything that is passed to it into Python expressions. We passed `{{ person.secret }}`, which, in the Flask templating language (Flask uses Jinja2 templating), got evaluated to the value of the key secret in the dictionary `person`, effectively exposing the secret key of the app!

We can perform even more ambitious attacks, as this hole in the code allows an attacker to try the full power of Jinja templates, including `for` loops. Here is an example:

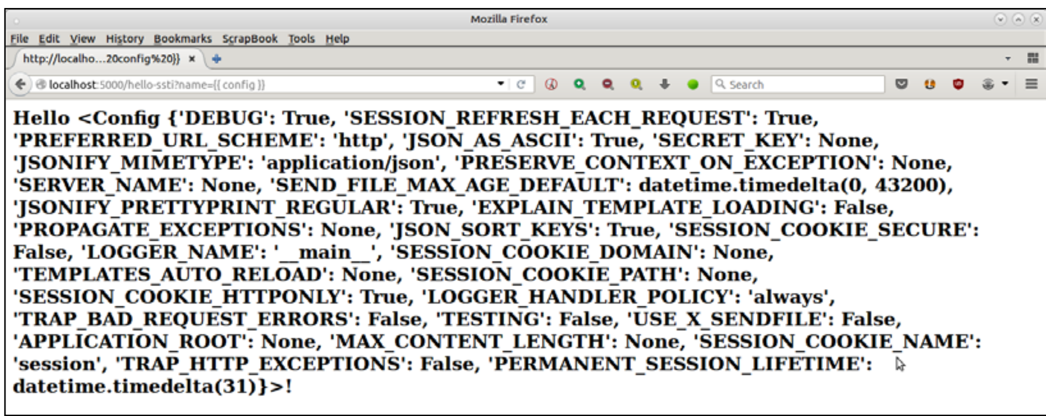
The URL used for the attack is as follows:

```
http://localhost:5000/hello-ssti?name={% for item in person %}<p>{{
item, person[item] }}</p>{% endfor %}
```

This goes through a for loop, and tries to print all contents of the `person` dictionary.

This also allows an attacker easy access to the sensitive server-side configuration parameters. For example, he can print out the Flask configuration by passing the name parameter as `{{ config }}`.

Here is the screenshot of the browser, printing the server configuration using this attack.



Server-Side Template Injection – Mitigation

We saw in the previous section some examples of using server side templates as an attack vector to expose sensitive information of the web application/server. In this section, we will see how the programmer can safeguard his code against such attacks.

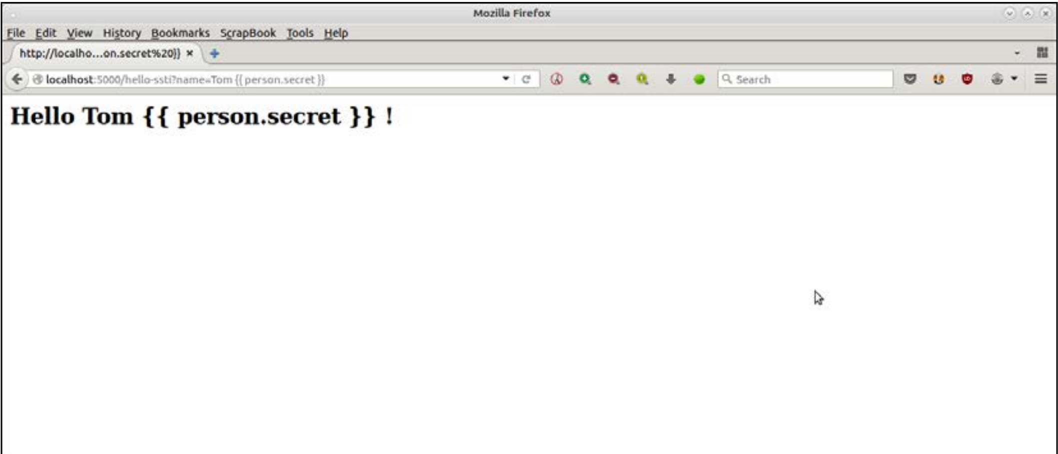
In this specific case, the fix for this is to use the specific variable that we want in the template, rather than the dangerous, allow-all `%s` string. Here is the modified code with the fix:

```
# ssti-example-fixed.py
from flask import Flask
from flask import request, render_template_string, render_template

app = Flask(__name__)

@app.route('/hello-ssti')
defhello_ssti():
    person = {'name':"world", 'secret':
```

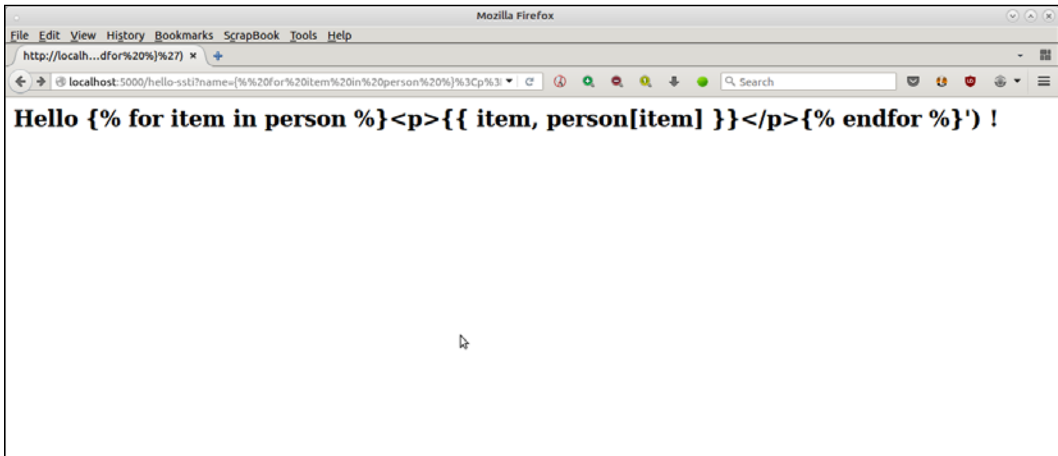
```
jo5gmvlligcZ5YZGenWnGcol8JnwhWZd2lJZYo=='}
    if request.args.get('name'):
        person['name'] = request.args.get('name')

    template = '<h2>Hello {{ person.name }} !</h2>'
    return render_template_string(template, person=person)

if __name__ == "__main__":
    app.run(debug=True)
```

Now the earlier attacks all fizzle off.

Here is the browser screenshot for the first attack:



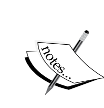
Here is the browser screenshot for the next attack.



Denial of Service

Now let's look at another attack that is commonly used by malicious hackers, namely, **Denial of Service** (**DoS**).

DoS attacks target vulnerable routes or URLs in a web application, and sends them crafty packets or URLs, which either force the server to perform infinite loops or CPU-intensive computations, or force it to load huge amounts of data from databases, which puts a lot of load on the server CPU, preventing the server from executing other requests.



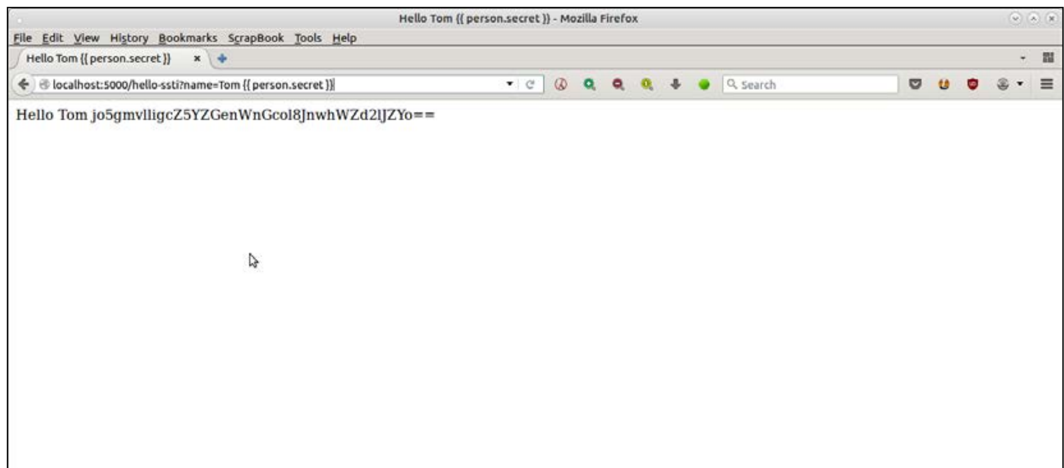
A DDoS or distributed DoS attack is when the DoS attack is performed in a choreographed way using multiple systems targeting a single domain. Usually thousands of IP addresses are used, which are managed via botnets.

We will see a minimal example of a DoS attack using a variation of our previous example:

```
# ssti-example-dos.py
from flask import Flask
from flask import request, render_template_string, render_template

app = Flask(__name__)

TEMPLATE = '''
<html>
 <head><title> Hello {{ person.name }} </title></head>
 <body> Hello FOO </body>
</html>
'''

@app.route('/hello-ssti')
defhello_ssti():
    person = {'name':"world", 'secret':
'jo5gmvlligcZ5YZGenWnGcol8JnwhWZd2lJZYo=='}
    if request.args.get('name'):
        person['name'] = request.args.get('name')

    # Replace FOO with person's name
    template = TEMPLATE.replace("FOO", person['name'])
    return render_template_string(template, person=person)

if __name__ == "__main__":
    app.run(debug=True)
```

In the preceding code, we use a global template variable named `TEMPLATE`, and use the `safer {{ person.name }}` template variable as the one used with the SSTI fix. However, the additional code here is a replacement of the holding name `FOO` with the name value.

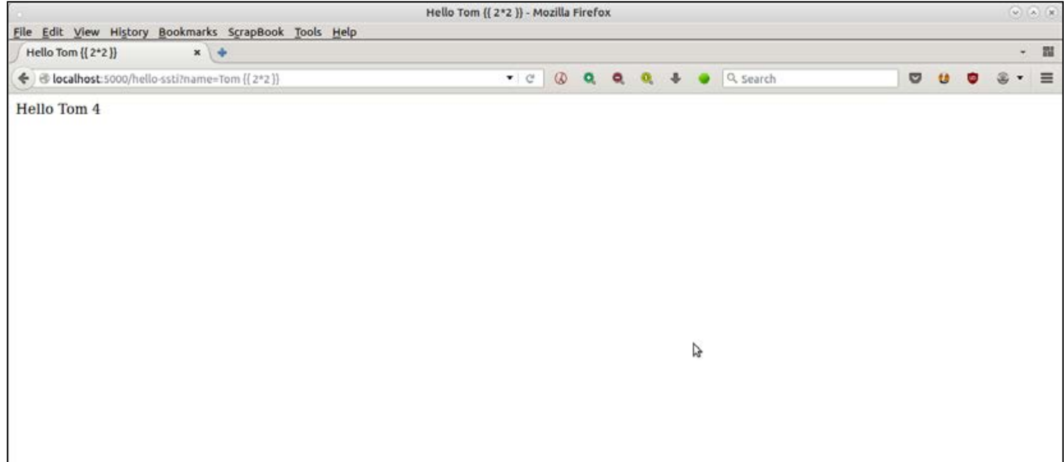
This version has all the vulnerabilities of the original code, even with the `%s` code removed. For example, take a look at the following screenshot of the browser exposing the `{{ person.secret }}` variable value in the body, but not in the title of the page.



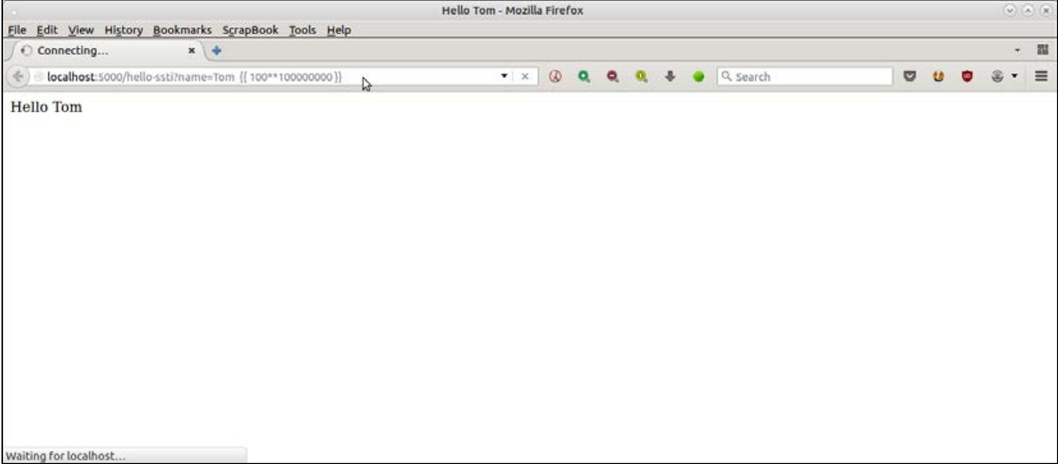
This is due to this following line of code that we added shown as follows:

```
# Replace FOO with person's name
template = TEMPLATE.replace("FOO", person['name'])
```

Any expression passed is evaluated, including the arithmetic ones. For example:



This opens up pathways to simple DoS attacks by passing in CPU-intensive computations that the server cannot handle. For example, in the following attack, we pass in a very large computation of a number, which occupies the CPU of the system, slows the system down and makes the application non-responsive:

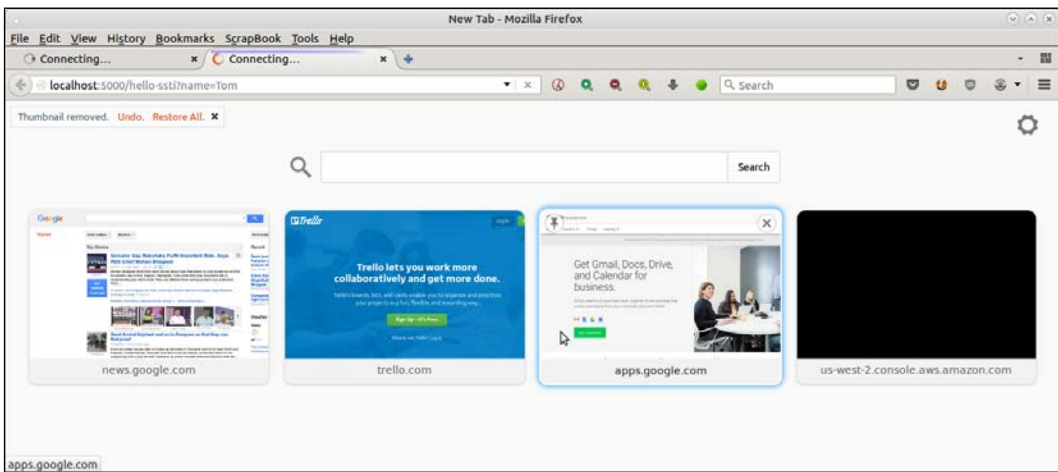


An example demonstrating a DoS style attack using computationally intensive code.

The URL used for this attack is `http://localhost:5000/hello-ssti?name=Tom {{ 100*100000000 }}`.

By passing in the arithmetical expression `{{ 100**100000000 }}`, which is computationally intensive, the server is overloaded and cannot handle other requests.

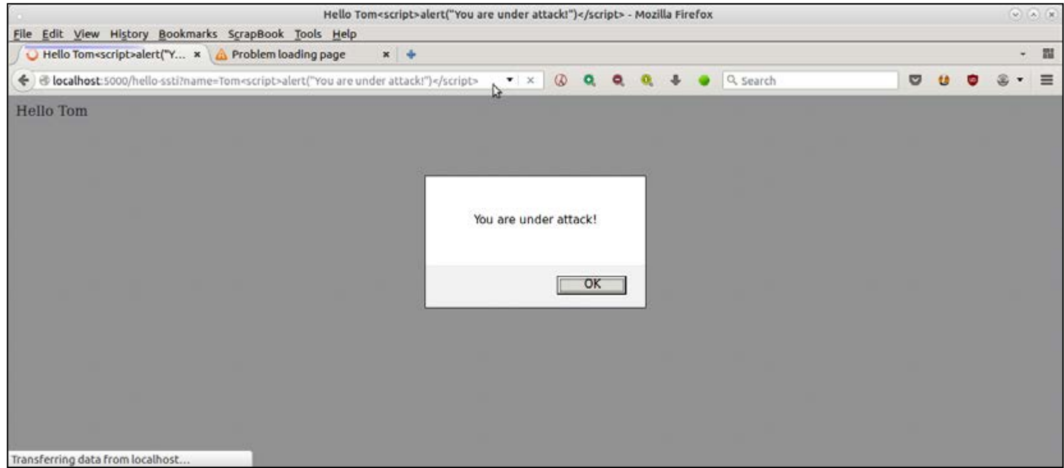
As you can see in the previous screenshot, the request never completes, and also prevents the server from responding to other requests; as you can see from how a normal request to the same application on a new tab opened on the right side is also held up causing the effect of a DoS style attack:



A new tab opened on the right side of the tab with attack vector shows that the application has become unresponsive

Cross-Site Scripting (XSS)

The code that we used in the earlier section to demonstrate a minimalistic DOS attack is also vulnerable to script injection. Here is an illustration:



A simple demonstration of XSS scripting using server side templates and JavaScript injection

The URL used for this attack is as follows:

```
http://localhost:5000/hello-ssti?name=Tom<script>alert("You are under
attack!")</script>
```

These kinds of script injection vulnerabilities can lead to XSS, a common form of web exploit where attackers are able to inject malicious scripts into your server's code, which are loaded from other websites, and take control of it.

Mitigation – DoS and XSS

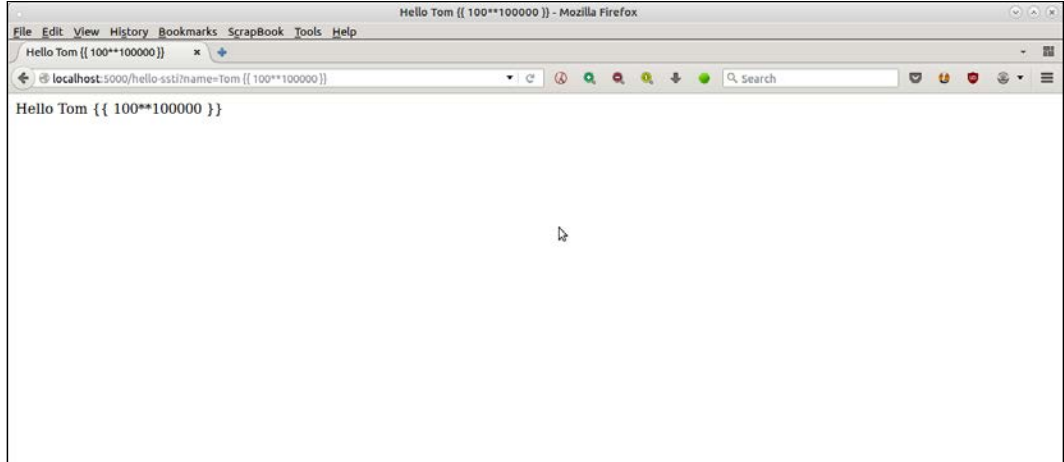
We saw a few examples of DoS attacks and simple XSS attacks in the previous section. Now let's look at how the programmer can take steps in his code to mitigate such attacks.

In the previous specific example that we have used for illustration, the fix is to remove the line that replaces the string `FOO` with the name value, and to replace it with the parameter template itself. For good measure, we also make sure that the output is properly escaped by using the escape filter, `|e`, of Jinja 2. Here is the rewritten code:

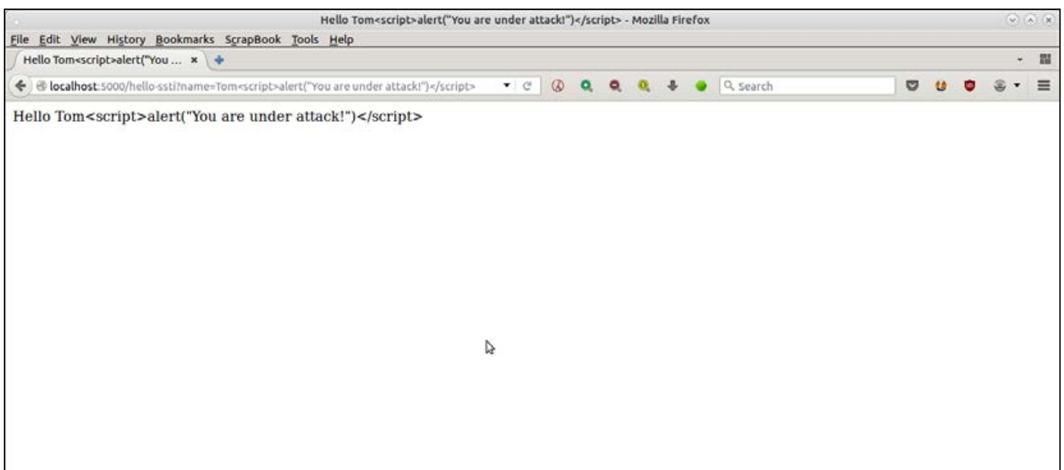
```
# ssti-example-dos-fix.py
from flask import Flask
from flask import request, render_template_string, render_template

app = Flask(__name__)

TEMPLATE = '''
<html>
 <head><title> Hello {{ person.name | e }} </title></head>
 <body> Hello {{ person.name | e }} </body>
</html>
'''

@app.route('/hello-ssti')
defhello_ssti():
    person = {'name':"world", 'secret':
'jo5gmvlligcZ5YZGenWnGcol8JnwhWZd2lJZYo=='}
    if request.args.get('name'):
        person['name'] = request.args.get('name')
    return render_template_string(TEMPLATE, person=person)

if __name__ == "__main__":
    app.run(debug=True)
```

Now that both of the vulnerabilities are mitigated, the attacks have no effect, and fail harmlessly.

Here is an screenshot demonstrating the DoS attack .



Here is the one, demonstrating the XSS attack.



Similar vulnerabilities due to bad code in server side templates exist in other Python web frameworks such as Django, Pyramid, Tornado, and others. However, a step-by-step discussion on each of these is beyond the scope of this chapter. The interested reader is directed to security resources on the web discussing such issues.

Strategies for security – Python

We have discussed quite a few vulnerabilities that exist in the core Python programming language, and also taken a look at some of the common security issues affecting Python web applications.

The time is ripe now to go through strategies—tips and techniques that a security architect can use so that their team can apply secure coding principles to mitigate security issues right from the stage of program design and development:

- **Reading input**: While reading console input, prefer raw input over `input`, as the former doesn't evaluate Python expressions, but returns input as plain strings. Any type conversions or validations should be done manually, and exceptions are thrown or errors returned if types don't match. For reading passwords, use libraries such as `getpass`, and also perform validations on the returned data. Any evaluation of the data can be safely done once the validations succeed.
- **Evaluating expressions**: As we've seen in our examples, `eval` always has loopholes whichever way it is used. Hence, the best strategy with Python is to avoid using eval and its cousin `exec`. If you have to use `eval`, make it a point to never use it with user input strings, or data read from third-party libraries, or APIs on which you have no control. Use `eval` only with input sources and return values from functions that you have control of and that you trust.
- **Serialization**: Don't use `pickle` or `cPickle` for serialization. Favor other modules such JSON or YAML. If you absolutely have to use `pickle`/ `cPickle`, use mitigation strategies such as a chroot jail or sandbox to avoid the bad effects of malicious code execution, if any.
- **Overflow errors**: Guard against integer overflows by using exception handlers. Python doesn't suffer from pure buffer overflow errors, as it always checks its containers for read/write access beyond the bounds and throws exceptions. For overridden `__len__` methods on classes, catch the overflow or `TypeError` exceptions as required.
- **String formatting**: Prefer the newer and safer format method of template strings over the older and unsafe `%s` interpolation.

For example:

```
def display_safe(employee):
    """ Display details of the employee instance """

    print("Employee: {name}, Age: {age},
             profession: {job}".format(**employee))

def display_unsafe(employee):
    """ Display details of employee instance """

    print ("Employee: %s, Age: %d,
              profession: %s" % (employee['name'],
                                               employee['age'],
                                               employee['job']))

>>> employee={'age': 25, 'job': 'software engineer', 'name':
'Jack'}
>>> display_safe(employee)
Employee: Jack, Age: 25, profession: software engineer
>>> display_unsafe(employee)
Employee: Jack, Age: 25, profession: software engineer
```

- **Files**: When working with files, it is a good idea to use the with context managers to make sure that the file descriptors are closed after the operation.

  For example, favor this approach:

  ```
  with open'somefile.txt','w') as fp:
  fp.write(buffer)
  ```

  And avoid the following:

  ```
  fp = open('somefile.txt','w')
  fp.write(buffer)
  ```

  This will also ensure that the file descriptor is closed if any exception occurs during file read or write instead of keeping open file handles in the system.

- **Handling passwords and sensitive information**: When validating sensitive information like passwords, it is a good idea to compare cryptographic hashes rather than comparing the original data in memory:
    - This way, even if an attacker is able to pry out sensitive data from the program by exploits such as shell execution exploits or due to weaknesses in input data evaluation, the actual sensitive data is protected from immediate breach. Here is a simple approach for this:

```
# compare_passwords.py - basic
import hashlib
import sqlite3
import getpass

def read_password(user):
    """ Read password from a password DB """
    # Using an sqlite db for demo purpose

    db = sqlite3.connect('passwd.db')
    cursor = db.cursor()
    try:
        passwd=cursor.execute("select password from passwds
where user='%(user)s'" % locals()).fetchone()[0]
        return hashlib.sha1(passwd.encode('utf-8')).
hexdigest()
    except TypeError:
        pass

def verify_password(user):
    """ Verify password for user """

    hash_pass = hashlib.sha1(getpass.getpass("Password:
").encode('utf-8')).hexdigest()
    print(hash_pass)
    if hash_pass==read_password(user):
        print('Password accepted')
    else:
        print('Wrong password, Try again')

if __name__ == "__main__":
    import sys
    verify_password(sys.argv[1])
```

A more cryptographically correct technique is to use strong password-hashing libraries with built-in `salt` and a fixed number of hashing rounds.

Here is an example using the `passlib` library in Python:

```
# crypto_password_compare.py
import sqlite3
import getpass
from passlib.hash import bcrypt

def read_passwords():
    """ Read passwords for all users from a password DB """
    # Using an sqlite db for demo purpose

    db = sqlite3.connect('passwd.db')
    cursor = db.cursor()
    hashes = {}

    for user,passwd in cursor.execute("select user,password from
passwds"):
        hashes[user] = bcrypt.encrypt(passwd, rounds=8)

    return hashes

def verify_password(user):
    """ Verify password for user """

    passwds = read_passwords()
    # get the cipher
    cipher = passwds.get(user)
    if bcrypt.verify(getpass.getpass("Password: "), cipher):
        print('Password accepted')
    else:
        print('Wrong password, Try again')

if __name__ == "__main__":
    import sys
    verify_password(sys.argv[1])
```

For the purpose of illustration, a `passwd.db` sqlite database has been created with two users and their passwords, as seen in the following screenshot:

```
(env) $ sqlite3 passwd.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> select * from passwds;
jack|reacher123
frodo|ring123
sqlite>
```

Here is the code in action:

Note that for purposes of clarity, the typed password is shown here—it won't be shown in the actual program, since it uses the `getpass` library.

Here is the code in action:

```
$ python3 crytpo_password_compare.py jack
Password: test
Wrong password, Try again

$ python3 crytpo_password_compare.py jack
Password: reacher123
Password accepted
```

- **Local data**: Wherever possible, avoid storing sensitive data local to functions. Any input validation or evaluation loophole in the functions can be exploited to gain access to the local stack, and hence, to the local data. Always store sensitive data encrypted or hashed separate modules.

  The following is a simple illustration:

```
def func(input):
  secret='e4fe5775c1834cc8bd6abb712e79d058'
  verify_secret(input, secret)
  # Do other things
```

The above function is unsafe for the secret key 'secret', as any attacker gaining access to the function's stack can gain access to the secret as well.

Such secrets are better kept in a separate module. If you are using the secret for hashing and verification, the following code is much safer than the first, since it does not expose the original value of the 'secret':

```
 # This is the 'secret' encrypted via bcrypt with eight rounds.
 secret_hash='$2a$08$Q/lrMAMe14vETxJC1kmxp./JtvF4vI7/b/
VnddtUIbIzgCwA07Hty'
 def func(input):
    verify_secret(input, secret_hash)
```

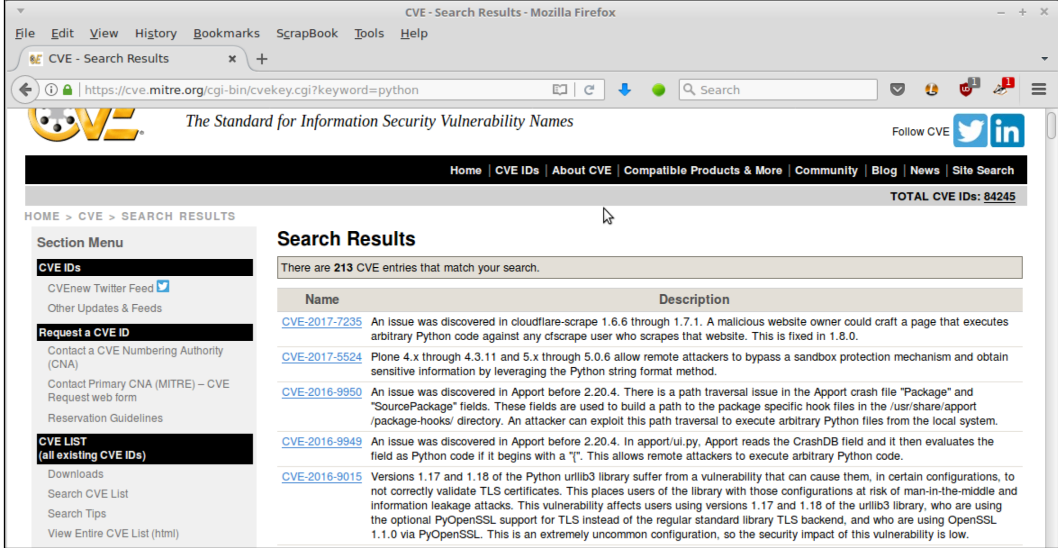
- **Race conditions**: Python provides an excellent set of threading primitives. If your program uses multiple threads and shared resources, follow these guidelines to synchronize access to resources to avoid race conditions and deadlocks:
    - Protect resources that can be writeable concurrently by a mutex (`threading.Lock`)
    - Protect resources that need to be serialized with respect to multiple, but limited, concurrent accesses by a semaphore (`threading.BoundedSemaphore`)
    - Use condition objects to wake up synchronize multiple threads waiting on a programmable condition or function (`threading.Condition`)

For programs using multiple processes, similar counterparts provided by the `multiprocessing` library should be used to manage concurrent access to resources.

- **Keep your system up to date**: Though this may sound clichéd, keeping up to date with respect to security updates of packages in your system and with security news in general, especially on packages that impact your application, is a simple way to keep your system and application secure. A number of websites provide constant updates on the state of security of a number of opensource projects including Python and its standard library modules.

  These reports usually go by the name of **Common Vulnerabilities and Exposures** (**CVEs**)—and sites such as Mitre (`http://cve.mitre.org`) provide a constant stream of updates.

  A search for Python on this sites shows 213 results:



Results for 'python' keyword search on Mitre CVE list

Architects, DevOps engineers, and webmasters can also tune in to their system package updates, and keep security updates always enabled by default. For remote severs, upgrading to the latest security patches every two to three months is highly recommended.

- Similarly, the Python **Open Web Application Security Project** (**OWASP**) project is a free, third-party project aimed at creating a hardened version of Python more resilient to security threats than the standard CPython. It is part of the larger OWASP initiative.
- The Python OWASP project makes available its Python bug-reports, tools, and other artifacts via the website and associated GitHub projects. The main website for this is, and most of the code is available from, the GitHub project page at: `https://github.com/ebranca/owasp-pysec/`.

Home page of the OWASP Python security project

It is a good idea for the stakeholders to keep track of this project, run their tests, and read their reports to keep up to date on Python security aspects.

Secure coding strategies

We are coming towards the end of our discussion on the security aspects of software architecture. It is a good time to summarize the strategies that one should try and impart to a software development team from a security architect's point of view. The following is a table summarizing the top 10 of these.

SL	Strategy	How it helps
1	Validate inputs	Validate inputs from all untrusted data sources. Proper input validation can eliminate a vast majority of software vulnerabilities.
2	Keep it simple	Keep program design as simple as possible. Complex designs increase the chances of security errors being made in their implementation, configuration, and deployment.
3	Principle of least privilege	Every process should execute with the least set of system privileges necessary to complete the work. For example, to read data from `/tmp`, one doesn't need root permission, but any unprivileged user is fine.
4	Sanitize data	Sanitize data read from and sent to all third-party systems such as databases, command shells, COTs components, third-party middlewares, and so on. This lessens the chances of SQL injection, shell exploit, or other similar attacks.
5	Authorize access	Separate parts of your application by roles that need specific authentication via login or other privileges. Don't mix different parts of applications together in the same code that requires different levels of access. Employ proper routing to make sure that no sensitive data is exposed via unprotected routes.
6	Perform effective QA	Good security testing techniques are effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should be performed as part of the program.
7	Practice defense in layers	Mitigate risks with multiple layers of security. For example, combining secure programming techniques with secure runtime configuration will reduce the chances of any remaining code vulnerabilities being exposed in the runtime environment.
8	Define security requirements	Identify and document the security constraints in the early lifecycle of the system, and keep updating them, making sure that any further features down the line keep up with these requirements.
9	Model threats	Use threat modeling to anticipate the threats to which the software will be subjected.
10	Architect and design for security policies	Create and maintain a software architecture that enforces a pattern of consistent security policies across your system and its subsystems.

Summary

In this chapter, we started by looking at the details of a system architecture that has information security built-in. We went on to define **secure coding**, and looked at the philosophies and principles behind the practice of secure coding.

We then studied the different types of security vulnerabilities encountered in software systems, such as buffer overflows, input validation issues, access control issues, cryptographic weaknesses, information leaks, insecure file operations, and so on.

We then went on to a detailed discussion on Python security issues with a lot of examples. We looked in detail at reading and evaluating input, overflow errors, and serialization issues. We then went on to look at the common vulnerabilities in Python web application frameworks by using Flask as the web application server for illustration. We saw how one can exploit the weaknesses on web application templates, and perform attacks such as SSTI, XSS, and DoS. We also saw few examples of how to mitigate these attacks.

We then went on to list specific techniques in Python for writing secure code. We looked in detail at managing cryptographic hashes of passwords and other sensitive data in code, and discussed a couple of examples of doing this the right way. The importance of keeping oneself updated with security news and projects, and keeping the system updated with security patches was also mentioned.

Finally, we summarized the top 10 secure coding strategies that a security architect can impart to their team in order to create secure code and systems.

In the next chapter, we take a look at one of the most interesting aspects of software engineering and design, namely that of Design Patterns.

7

Design Patterns in Python

Design patterns simplify building software by reusing successful designs and architectures. Patterns build on the collective experience of software engineers and architects. When faced with a problem that needs new code to be written, an experienced software architect tends to make use of the rich ecosystem of available design/architecture patterns.

Patterns evolve when a specific design proves successful in solving certain classes of problem repeatedly. When experts find that a specific design or architecture helps them to solve classes of related problems consistently, they tend to apply it more and more, codifying the structure of the solution into a pattern.

Python (given that it's a language which supports dynamic types and high-level object oriented structures such as classes and metaclasses, first-class functions, co-routines, callable objects, and so on) is a very rich playground for constructing reusable design and architecture patterns. In fact, as opposed to languages such as C++ or Java, you often find there are multiple ways of implementing a specific design pattern in Python. Also, more often than not, you find that the Pythonic way of implementing a pattern is more intuitive and illustrative than, say, copying a standard implementation from C++/Java into Python.

This chapter's focus is mostly on this latter aspect—illustrating how one can build design patterns which are more Pythonic than those in the usual books and literature on this topic. It doesn't aim to be a comprehensive guide to design patterns, though we would be covering most of the usual aspects as we head into the content.

The topics we plan to cover in this chapter are as follows:

- Design patterns elements
- Categories of design patterns
- Pluggable hashing algorithms
- Summing up pluggable hashing algorithms

- Patterns in Python – Creational
    - The Singleton pattern
    - The Borg pattern
    - The Factory pattern
    - The Prototype pattern
    - The Builder pattern
- Patterns in Python – Structural
    - The Adapter pattern
    - The Facade pattern
    - The Proxy pattern
- Patterns in Python – Behavioral
    - The Iterator pattern
    - The Observer pattern
    - The State pattern

Design patterns – elements

A design pattern attempts to record those aspects of a recurring design in object-oriented systems that solve a problem or a class of problems.

When we inspect design patterns, we find that almost all of them have the following elements:

- **Name**: A well-known handle or title, which is commonly used to describe the pattern. Having standard names for design patterns aids communication and increases our design vocabulary.
- **Context**: This is the situation in which the problem arises. A context can be generic such as *Develop a web application software*, or specific such as *Implementing resource-change notification in a shared memory implementation of the publisher-subscriber system*.

- **Problem**: Describes the actual problem that the pattern is applied to. A problem can be described in terms of its forces, which are as follows:
    - **Requirements**: The requirements that the solution should fulfill, for example, the *publisher-subscriber pattern implementation must support HTTP*.
    - **Constraints**: The constraints to the solution, if any, for example, the *Scalable peer-to-peer publisher pattern should not exchange more than three messages for publishing a notification*.
    - **Properties**: The properties of the solution which are desirable to have, for example, *The solution should work equally well on the Windows and Linux platforms*.
- **Solution**: Shows the actual solution to the problem. It describes the structure and responsibilities, the static relationships, and the runtime interactions (collaborations) of the elements making up the solution. A solution should also discuss which *forces* of the problem it solves and doesn't solve. A solution should also try to mention its consequences, that is, the results and trade-offs of applying a pattern.

A design pattern solution almost never resolves all the forces of the problem leading to it, but leaves some of them open to related or alternate implementations.

Categories of design patterns

Design patterns can be categorized in different ways according to the criteria chosen. A commonly accepted way of categorizing patterns is based on their purpose. In other words, we ask the pattern what class of problem the pattern solves.

This kind of categorization gives us three neat varieties of pattern classes. These are as follows:

- **Creational**: These patterns solve the problems associated with object creation and initialization. These are problems that occur the earliest in the life cycle of problem solving with objects and classes. Take a look at the following examples:
    - **The Factory pattern**: The "How do I make sure I can create related class instances in a repeatable and predictable fashion?" question is solved by the Factory class of patterns.

- **The Prototype pattern**: The "What is a smart approach to instantiate an object, and then create hundreds of similar objects by just copying across this one object ?" question is solved by Prototype patterns.
- **Singleton and related patterns**: The "How do I make sure that any instance of a class I create is created and initialized just once" or "How do I make sure that any instances of a class share the same initial state ?" questions are solved by the Singleton and related patterns.

- **Structural**: These patterns concern themselves with the composition and assembling of objects into meaningful structures, which provides the architect and developer with reusable behaviors, where "the whole is more than the sum of its parts". Naturally, they occur in the next step of problem solving with objects, once they are created. Examples of such problems are as follows:
    - **The Proxy pattern**: "How do I control access to an object and its methods via a wrapper, behavior on top?"
    - **The Composite pattern**: "How can I represent an object which is made of many components at the same time using the same class for representing the part and the whole—for example, a Widget tree ?"
- **Behavioral**: These patterns solve the problems originating with runtime interactions of objects, and how they distribute responsibilities. Naturally, they occur at a later stage, once the classes are created, and then combined into larger structures. The following are a couple of examples:
    - **Using the Median pattern in the following case**: "Ensure that all the objects use loose coupling to refer to each other at runtime to promote run-time dynamism for interactions"
    - **Using the Observer pattern in the following case**: "An object wants to be notified when the state of a resource changes, but it does not want to keep polling the resource to find this out. There may be many such instances of objects in the system"

The order of Creational, Structural, and Behavioral patterns implicitly embeds the life cycle of objects in a system at runtime. Objects are first created (Creational), then combined into useful structures (Structural), and then they interact (Behavioral).

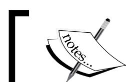
Let's now turn our attention to the subject of this chapter, namely, implementing patterns in Python in Python's own inimitable way. We will look at an illustrative example to get started.

Pluggable hashing algorithms

Let's look at the following problem.

You want to read data from an input stream—a file or network socket—and hash the contents in a chunked manner. You write some code as follows:

```
# hash_stream.py
from hashlib import md5

def hash_stream(stream, chunk_size=4096):
    """ Hash a stream of data using md5 """

    shash = md5()

    for chunk in iter(lambda: stream.read(chunk_size), ''):
        shash.update(chunk)

    return shash.hexdigest()
```



All code is in Python3, unless explicitly mentioned otherwise.

```
>>> import hash_stream
>>> hash_stream.hash_stream(open('hash_stream.py'))
'e51e8ddf511d64aeb460ef12a43ce480'
```

So that works, as expected.

Now let's say you want a more reusable and versatile implementation, one that will work with multiple hashing algorithms. You first attempt to modify the previous code, but quickly realize that this means rewriting a lot of code, which is not a very smart way of doing it:

```
# hash_stream.py
from hashlib import sha1
from hashlib import md5

def hash_stream_sha1(stream, chunk_size=4096):
```

```
        """ Hash a stream of data using sha1 """

        shash = sha1()

        for chunk in iter(lambda: stream.read(chunk_size), ''):
            shash.update(chunk.encode('utf-8'))

        return shash.hexdigest()

    def hash_stream_md5(stream, chunk_size=4096):
        """ Hash a stream of data using md5 """

        shash = md5()

        for chunk in iter(lambda: stream.read(chunk_size), ''):
            shash.update(chunk.encode('utf-8'))

        return shash.hexdigest()
```

```
>>> import hash_stream
>>> hash_stream.hash_stream_md5(open('hash_stream.py'))
'e752a82db93e145fcb315277f3045f8d'
>>> hash_stream.hash_stream_sha1(open('hash_stream.py'))
'360e3bd56f788ee1a2d8c7eeb3e2a5a34cca1710'
```

You realize that you can reuse a lot of code by using a class. Being an experienced programmer, you may end up with something like the following after a few iterations:

```
# hasher.py
class StreamHasher(object):
    """ Stream hasher class with configurable algorithm """

    def __init__(self, algorithm, chunk_size=4096):
        self.chunk_size = chunk_size
        self.hash = algorithm()

    def get_hash(self, stream):

        for chunk in iter(lambda: stream.read(self.chunk_size), ''):
            self.hash.update(chunk.encode('utf-8'))

        return self.hash.hexdigest()
```

First let's try this with `md5`, as follows:

```
>>> import hasher
>>> from hashlib import md5
>>> md5h = hasher.StreamHasher(algorithm=md5)
>>> md5h.get_hash(open('hasher.py'))
'7d89cdc1f11ec62ec918e0c6e5ea550d'
```

Now let's use `sha1`:

```
>>> from hashlib import sha1
>>> shah_h = hasher.StreamHasher(algorithm=sha1)
>>> shah_h.get_hash(open('hasher.py'))
'1f0976e070b3320b60819c6aef5bd6b0486389dd'
```

As must be evident by now, you can build different hasher objects, each with a specific algorithm, which will return the corresponding hash digest of the stream (in this case, a file).

Now lets summarize what we just did here.

We first developed a function, `hash_stream`, which took in a stream object, and hashed it chunk-wise using the `md5` algorithm. We then developed a class named `StreamHasher`, which allowed us to configure it using one algorithm at a time, thereby making the code more reusable. We obtained the hash digest by way of `get_hash`, which accepts the stream object as argument.

Now let's turn our attention to what else Python can do for us.

Our class is versatile with respect to different hashing algorithms, and is definitely more reusable, but is there a way to call it as if it were a function? That would be rather neat, wouldn't it?

The following is a slight reimplementation of our `StreamHasher` class, which does just that:

```
# hasher.py
class StreamHasher(object):
    """ Stream hasher class with configurable algorithm """

    def __init__(self, algorithm, chunk_size=4096):
        self.chunk_size = chunk_size
        self.hash = algorithm()

    def __call__(self, stream):
```

```
            for chunk in iter(lambda: stream.read(self.chunk_size), ''):
                self.hash.update(chunk.encode('utf-8'))

            return self.hash.hexdigest()
```

What did we do in the last code ? We simply renamed the `get_hash` function to `Get_Call`. Let's see what effect this has:

```
>>> from hashlib import md5, sha1
>>> md5_h = hasher.StreamHasher(md5)
>>> md5_h(open('hasher.py'))
'ad5d5673a3c9a4f421240c4dbc139b22'
>>> sha_h = hasher.StreamHasher(sha1)
>>> sha_h(open('hasher.py'))
'd174e2fae1d6e1605146ca9d7ca6ee927a74d6f2'
```

We are able to call the instance of the class as if it were a function by simply passing the file object to it.

So our class not only gives us reusable and versatile code, but also acts as if it were a function. This is done by making our class a callable type in Python by simply implementing the magic method `__call__` .

Callables in Python are any object that can be called. In other words, `x` is a callable if we can perform `x()` —with or without params, depending upon how the `__call__` method is overridden. Functions are the simplest and most familiar callables.

In Python, `foo(args)` is syntactic sugar for `foo.__call__(args)`.

Summing up pluggable hashing algorithm

So what does the previous example illustrate? It illustrates the power of Python in dealing with an existing problem, which would be solved traditionally in other programming languages, in a more exotic and powerful way due to the power of Python and the way it does things—in this case, by making any object callable by overriding a special method.

But what is the pattern we have achieved here? We said at the start of the chapter that something is a pattern only if it solves a class of problems. Is there a pattern hidden in this particular illustration?

Yes there is—this is an implementation of the Strategy behavioral pattern:

The Strategy pattern is used when we need different behaviors from a class and we should be able to configure a class with one of many available behaviors or algorithms.

In this particular case, we needed a class which supports different algorithms to perform the same thing—hashing data from a stream using chunks, and returning the digest. The class accepted the algorithm as a parameter, and since all algorithms support the same method for returning data (the `hexdigest` method), we were able to implement the class in a very simple way.

Let's continue our journey to discover some other interesting patterns we can write using Python, and its unique way of solving problems. We will follow the order of the Creational, Structural, and Behavioral patterns in this journey.

Our approach to the discussion on patterns that follows is very pragmatic. It may not use the formal language used by the popular **Gang-of-Four** (**G4**) patterns—the most elemental approach to design patterns. Our focus is on demonstrating the power of Python in building patterns rather than getting the formalisms right.

Patterns in Python – creational

In this section, we will take a look at a few of the common creational patterns. We will start with Singleton, and then go on to Prototype, Builder, and Factory, in that order.

The Singleton pattern

The Singleton pattern is one of the most well-known and easily understood patterns in the entire pantheon of design patterns. It is usually defined as:

A Singleton is a class which has only one instance and a well-defined point of access to it.

The requirements of a Singleton can be summarized as follows:

- A class must have only one instance accessible via a well-known access point.
- The class must be extensible by inheritance without breaking the pattern.
- The simplest Singleton implementation in Python is shown next. It is done by overriding the `__new__` method of the base `object` type:

```
# singleton.py
class Singleton(object):
```

```
    """ Singleton in Python """

    _instance = None

    def __new__(cls):
        if cls._instance == None:
            cls._instance = object.__new__(cls)
        return cls._instance
```

```
>>> from singleton import Singleton
>>> s1 = Singleton()
>>> s2 = Singleton()
>>> s1==s2
True
```

- Since we would be requiring this check for a while, let's define a function for the same:

```
def test_single(cls):
    """ Test if passed class is a singleton """
    return cls() == cls()
```

- Now let's see if our Singleton implementation satisfies the second requirement. We will define a simple subclass to test this:

```
class SingletonA(Singleton):
    pass

>>> test_single(SingletonA)
True
```

Cool! So our simple implementation passes the test. Are we done here now?

Well, the point with Python, as we discussed before, is that it provides a number of ways to implement patterns due to its dynamism and flexibility. So, let's stay with with Singleton for a while, and see if we can get some illustrative examples which would give us insights into the power of Python:

```
class MetaSingleton(type):
    """ A type for Singleton classes (overrides __call__) """

    def __init__(cls, *args):
        print(cls,"__init__ method called with args", args)
        type.__init__(cls, *args)
```

```
        cls.instance = None

    def __call__(cls, *args, **kwargs):
        if not cls.instance:
            print(cls,"creating instance", args, kwargs)
            cls.instance = type.__call__(cls, *args, **kwargs)
        return cls.instance

class SingletonM(metaclass=MetaSingleton):
    pass
```

The preceding implementation moves the logic of creating a Singleton to the type of the class, namely, its metaclass.

We first create a type for Singletons, named `MetaSingleton`, by extending the type and overriding the `__init__` and `__call__` methods on the metaclass. Then we declare that the `SingletonM` class, `SingletonM`, uses the metaclass.

```
>>> from singleton import *
<class 'singleton.SingletonM'> __init__ method called with args
('SingletonM', (), {'__module__': 'singleton', '__qualname__':
'SingletonM'})
>>> test_single(SingletonM)
<class 'singleton.SingletonM'> creating instance ()
True
```

The following is a peep into what is happening behind the scenes in the new implementation of the Singleton:

- **Initializing a class variable**: We can either do it at the class level (just after the class declaration) as we saw in the previous implementation, or we can put it in the metaclass `__init__` method. This is what we are doing here for the `_instance` class variable, which will hold the single instance of the class.
- **Overriding class creation**: One can either do it at the class level by overriding the `__new__` method of class as we saw in previous implementation, or, equivalently, we can do it in the metaclass by overriding its `__call__` method. This is what the new implementation does.

When we override a class's `__call__` method, it affects its instance, and instances become callable. Similarly, when we override a metaclass's `_call_` method, it affects its classes, and modifies the way the classes are called—in other words, the way the class creates its instances.

Let's take a look at the pros and cons in the metaclass approach over the class approach:

- One benefit is that we can create any number of new top-level classes which get the Singleton behavior via the metaclass. Using the default implementation, every class has to inherit the top-level class Singleton or its subclasses to obtain the Singleton behavior. The metaclass approach provides more flexibility with respect to class hierarchies.
- However, the metaclass approach can be interpreted as creating slightly obscure and difficult-to-maintain code as opposed to the class approach. This is because fewer Python programmers understand metaclasses and metaprogramming when compared to those who understand classes. This may be a disadvantage with the metaclass solution.

Now let's think out of the box, and see if we can solve the Singleton problem in a slightly different way.

The Singleton – do we need a Singleton?

Let's paraphrase the first requirement of a Singleton in a slightly different way:

A class must provide a way for all its instances to share the same initial state.

To explain that, let's briefly look at what a Singleton pattern actually tries to achieve.

When a Singleton ensures it has only one instance, what it guarantees is that the class provides one single state when it is created and initialized. In other words, what a Singleton actually gives is a way for a class to ensure a single shared state across all its instances.

In other words, the first requirement of the Singleton pattern can be paraphrased in a slightly different form, which has the same end result as the first form.

A class must provide a way for all its instances to share the same initial state.

The technique of ensuring just a single actual instance at a specific memory location is just one way of achieving this.

Ah! So what has been happening so far is that we have been expressing the pattern in terms of the implementation details of less flexible and versatile programming languages. With a language such as Python, we need not stick pedantically to this original definition.

Let's look at the following class:

```
class Borg(object):
    """ I am not a Singleton """

    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
```

This pattern ensures that when you create a class, you specifically initialize all of its instances with a shared state which belongs to the class (since it is declared at the class level).

What we really care about in a Singleton is actually this shared state, so `Borg` works without worrying about all instances being exactly the same.

Since this is Python, it does this by initializing a shared state dictionary on the class, and then instantiating the instance's dictionary to this value, thereby ensuring that all instances share the same state.

The following is a specific example of `Borg` in action:

```
class IBorg(Borg):
    """ I am a Borg """

    def __init__(self):
        Borg.__init__(self)
        self.state = 'init'

    def __str__(self):
        return self.state

>>> i1 = IBorg()
>>> i2 = IBorg()
>>> print(i1)
init
>>> print(i2)
init
>>> i1.state='running'
>>> print(i2)
running
>>> print(i1)
running
>>> i1==i2
False
```

By using `Borg`, we managed to create a class whose instances share the same state, even though the instances are actually not the same. And the state change was propagated across the instances; as the preceding example shows, when we change the value of state in `i1`, it also changes in `i2`.

What about dynamic values? We know they will work in a Singleton, since it's the same object always, but what about the Borg?

```
>>> i1.x='test'
>>> i2.x
'test'
```

So we attached a dynamic attribute `x` to instance `i1`, and it appeared in instance `i2` as well. Neat!

So let's see if `Borg` offers any benefits over Singleton:

- In a complex system where we may have multiple classes inheriting from a root Singleton class, it may be difficult to impose the requirement of a single instance due to import issues or race conditions—for example, if a system is using threads. The Borg pattern circumvents these problems neatly by doing away with the requirement for a single instance in memory.
- The Borg pattern also allows for simple sharing of state across the Borg class and all its subclasses. This is not the case for a Singleton, since each subclass creates its own state. We will see an example illustrating this next.

State sharing – Borg versus Singleton

A Borg pattern always shares the same state from the top class (Borg) down to all the subclasses. This is not the case with a Singleton. Let's see an illustration.

For this exercise, we will create two subclasses of our original Singleton class, namely, `SingletonA` and `SingletonB`:

```
>>> class SingletonA(Singleton): pass
...
>>> class SingletonB(Singleton): pass
...
```

Let's create a subclass of `SingletonA`, namely, `SingletonA1`:

```
>>> class SingletonA1(SingletonA): pass
...
```

Now let's create instances:

```
>>> a = SingletonA()
>>> a1 = SingletonA1()
>>> b = SingletonB()
```

Let's attach a dynamic property, `x`, with a value `100` to `a`:

```
>>> a.x = 100
>>> print(a.x)
100
```

Let's check if this is available on the `a1` instance of the `SingletonA1` subclass:

```
>>> a1.x
100
```

Good! Now let's check if it is available on the `b` instance:

```
>>> b.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SingletonB' object has no attribute 'x'
```

Oops! So, it appears that `SingletonA` and `SingletonB` don't share the same state. This is why a dynamic attribute that is attached to an instance of `SingletonA` appears in the instance of its sub-classes, but doesn't appear on the instance of a sibling or peer subclass namely `SingletonB` – because it is a different branch of the class hierarchy from the top-level `Singleton` class.

Let's see if Borgs can do any better.

First, let's create the classes and their instances:

```
>>> class ABorg(Borg):pass
...
>>> class BBorg(Borg):pass
...
>>> class A1Borg(ABorg):pass
...
>>> a = ABorg()
>>> a1 = A1Borg()
>>> b = BBorg()
```

Now let's attach a dynamic attribute x to a with value 100:

```
>>> a.x = 100
>>> a.x
100
>>> a1.x
100
```

Let's check if the instance of the sibling class Borg also gets it:

```
>>> b.x
100
```

This proves that the Borg pattern is much better at state sharing across classes and sub classes than the Singleton pattern, and it does so without a lot of fuss or the overhead of ensuring a single instance.

Let's now move on to other creational patterns.

The Factory pattern

The Factory pattern solves the problem of creating instances of related classes to another class, which usually implements instance creation via a single method, usually defined on a parent Factory class and overridden by subclasses (as needed).

The Factory pattern provides a convenient way for the client (user) of a class to provide a single entry point to create instances of classes and subclasses, usually, by passing in parameters to a specific method of the `Factory` class: the factory method.

Let's look at a specific example:

```
from abc import ABCMeta, abstractmethod

class Employee(metaclass=ABCMeta):
    """ An Employee class """

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    @abstractmethod
```

```
    def get_role(self):
        pass

    def __str__(self):
        return "{} - {}, {} years old {}".format(self.__class__.
          __name__,
                                                 self.name,
                                                 self.age,
                                                 self.gender)

class Engineer(Employee):
    """ An Engineer Employee """

    def get_role(self):
        return "engineering"

class Accountant(Employee):
    """ An Accountant Employee """

    def get_role(self):
        return "accountant"

class Admin(Employee):
    """ An Admin Employee """

    def get_role(self):
        return "administration"
```

We have created a general `Employee` class with some attributes and three subclasses, namely, `Engineer`, `Accountant`, and `Admin`.

Since all of them are related classes, a `Factory` class is useful to abstract away the creation of instances of these classes.

The following is our `EmployeeFactory` class:

```
class EmployeeFactory(object):
    """ An Employee factory class """

    @classmethod
    def create(cls, name, *args):
```

```
        """ Factory method for creating an Employee instance """

        name = name.lower().strip()

        if name == 'engineer':
            return Engineer(*args)
        elif name == 'accountant':
            return Accountant(*args)
        elif name == 'admin':
            return Admin(*args)
```

The class provides a single `create` factory method that accepts a `name` parameter, which is matched to the class's name and instance created accordingly. The rest of the arguments are parameters required for instantiating the class's instance, which is passed unchanged to its constructor.

Let's see our `Factory` class in action:

```
>>> factory = EmployeeFactory()
>>> print(factory.create('engineer','Sam',25,'M'))
Engineer - Sam, 25 years old M
>>> print(factory.create('engineer','Tracy',28,'F'))
Engineer - Tracy, 28 years old F

>>> accountant = factory.create('accountant','Hema',39,'F')
>>> print(accountant)

Accountant - Hema, 39 years old F
>>> accountant.get_role()

accounting
>>> admin = factory.create('Admin','Supritha',32,'F')
>>> admin.get_role()
'administration'
```

The following are a few interesting notes about our `Factory` class:

- A single factory class can create instances of any class in the Employee hierarchy.
- In the Factory pattern, it is conventional to use one `Factory` class associated to a class family (a class and its subclass hierarchy). For example, a `Person` class could use a `PersonFactory`, an automobile class could use `AutomobileFactory`, and so on.

- The factory method is usually decorated as a `classmethod` in Python. This way it can be called directly via the class namespace. For example:

  ```
  >>> print(EmployeeFactory.create('engineer','Vishal',24,'M'))
  Engineer - Vishal, 24 years old M
  ```

In other words, an instance of the `Factory` class is really not required for this pattern.

The Prototype pattern

The Prototype design pattern allows a programmer to create an instance of a class as a template instance, and then create new instances by copying or cloning this Prototype.

A Prototype is most useful in the following cases:

- When the classes instantiated in a system are dynamic, that is, they are specified as part of a configuration, or can otherwise change at runtime.
- When the instances only have a few combinations of initial state. Rather than keeping track of the state and instantiating an instance each time, it is more convenient to create prototypes matching each state and clone them.

A Prototype object usually supports copying itself via the `clone` method.

The following is a simple implementation of the Prototype in Python:

```
import copy

class Prototype(object):
    """ A prototype base class """

    def clone(self):
        """ Return a clone of self """
        return copy.deepcopy(self)
```

The `clone` method is implemented using the `copy` module, which performs a deepcopy on the object? and returns a clone.

Let's see how this works. For that, we need to create a meaningful subclass:

```
class Register(Prototype):
    """ A student Register class  """

    def __init__(self, names=[]):
        self.names = names


>>> r1=Register(names=['amy','stu','jack'])
>>> r2=r1.clone()
>>> print(r1)
<prototype.Register object at 0x7f42894e0128>
>>> print(r2)
<prototype.Register object at 0x7f428b7b89b0>


>>> r2.__class__
<class 'prototype.Register'>
```

Prototype – deep versus shallow copy

Now let's take a deeper look at the implementation details of our Prototype class.

You may notice that we use the `deepcopy` method of the `copy` module to implement our object cloning. This module also has a `copy` method, which implements shallow copying.

If you implement shallow copying, you will find that all objects are copied via a reference. This is fine for immutable objects such as strings or tuples, as they can't be changed.

However, for mutables such as lists or dictionaries, this is a problem since the state of the instance is shared instead of being wholly owned by the instance, and any modification of a mutable in one instance will modify the same object in the cloned instances as well!

Let's see an example. We will use a modified implementation of our Prototype class, which uses shallow copying, to demonstrate this:

```
class SPrototype(object):
    """ A prototype base class using shallow copy """

    def clone(self):
        """ Return a clone of self """
        return copy.copy(self)
```

The `SRegister` class inherits from the new prototype class:

```
class SRegister(SPrototype):
    """ Sub-class of SPrototype """

    def __init__(self, names=[]):
        self.names = names


>>> r1=SRegister(names=['amy','stu','jack'])
>>> r2=r1.clone()
```

Let's add a name to the names register of instance `r1`:

```
>>> r1.names.append('bob')
```

Now let's check `r2.names`:

```
>>> r2.names
['amy', 'stu', 'jack', 'bob']
```

Oops! This is not what we wanted, but due to the shallow copy, both `r1` and `r2` end up sharing the same `names` list, as only the reference is copied over, not the entire object. This can be verified by a simple inspection:

```
>>> r1.names is r2.names
True
```

A deep copy, on the other hand, calls `copy` recursively for all objects contained in the cloned (copied) object, so nothing is shared, but each clone will end up having its own copy of all the referenced objects.

Prototype using metaclasses

We've seen how to build the Prototype pattern using classes. Since we've already seen a bit of meta-programming in Python in the Singleton pattern example, let's find out whether we can do the same in Prototype.

What we need to do is attach a `clone` method to all the Prototype classes. Dynamically attaching a method to a class like this can be done in its metaclass via the `__init__` method of the metaclass.

This provides a simple implementation of Prototype using metaclasses:

```
import copy

class MetaPrototype(type):

    """ A metaclass for Prototypes """

    def __init__(cls, *args):
        type.__init__(cls, *args)
        cls.clone = lambda self: copy.deepcopy(self)

class PrototypeM(metaclass=MetaPrototype):
    pass
```

The `PrototypeM` class now implements a Prototype pattern. Let's see an illustration by using a subclass:

```
class ItemCollection(PrototypeM):
    """ An item collection class """

    def __init__(self, items=[]):
        self.items = items
```

First we create an `ItemCollection` object:

```
>>> i1=ItemCollection(items=['apples','grapes','oranges'])
>>> i1
<prototype.ItemCollection object at 0x7fd4ba6d3da0>
```

Now we clone it as follows:

```
>>> i2 = i1.clone()
```

The clone is clearly a different object:

```
>>> i2
<prototype.ItemCollection object at 0x7fd4ba6aceb8>
```

And it has its own copy of the attributes:

```
>>> i2.items is i1.items
False
```

Combining patterns using metaclasses

It is possible to create interesting and customized patterns by using the power of metaclasses. The following example illustrates a type which is both a Singleton as well as a Prototype:

```
class MetaSingletonPrototype(type):
    """ A metaclass for Singleton & Prototype patterns """

    def __init__(cls, *args):
        print(cls,"__init__ method called with args", args)
        type.__init__(cls, *args)
        cls.instance = None
        cls.clone = lambda self: copy.deepcopy(cls.instance)

    def __call__(cls, *args, **kwargs):
        if not cls.instance:
            print(cls,"creating prototypical instance", args, kwargs)
            cls.instance = type.__call__(cls,*args, **kwargs)
        return cls.instance
```

Any class using this metaclass as its type would show both Singleton and Prototype behavior.

It may look a bit strange to have a single class combine what look like conflicting behaviors into one, since a Singleton allows only one instance and a Prototype allows cloning to derive multiple instances, but if we think of patterns in terms of their APIs then it begins to feel a bit more natural:

- Calling the class using the constructor would always return the same instance – it behaves like the Singleton pattern.
- Calling `clone` on the class's instance would always return cloned instances. The instances are always cloned using the Singleton instance as the source – it behaves like the Prototype pattern.

Here, we have modified our `PrototypeM` class to now use the new metaclass:

```
class PrototypeM(metaclass=MetaSingletonPrototype):
    pass
```

Since `ItemCollection` continues to subclass `PrototypeM`, it automatically gets the new behavior.

Take a look at the following code:

```
>>> i1=ItemCollection(items=['apples','grapes','oranges'])
<class 'prototype.ItemCollection'> creating prototypical instance ()
{'items': ['apples'
, 'grapes', 'oranges']}
>>> i1
<prototype.ItemCollection object at 0x7fbfc033b048>
>>> i2=i1.clone()
```

The `clone` method works as expected, and produces a clone:

```
>>> i2
<prototype.ItemCollection object at 0x7fbfc033b080>
>>> i2.items is i1.items
False
```

However, building an instance via the constructor always returns the Singleton (Prototype) instance only as it invokes the Singleton API:

```
>>> i3=ItemCollection(items=['apples','grapes','mangoes'])
>>> i3 is i1
True
```

Metaclasses allow powerful customization of class creation. In this specific example, we created a combination of behaviors which included both Singleton and Prototype patterns into one class via a metaclass. The power of Python using metaclasses allows the programmer to go beyond traditional patterns and come up with creative techniques.

The Prototype factory

A prototype class can be enhanced with a helper **Prototype factory** or **registry class**, which can provide factory functions for creating prototypical instances of a configured family or group of products. Think of this as a variation on our previous Factory pattern.

The following is the code for this class. Notice that we inherit it from `Borg` to share state automatically from the top of the hierarchy:

```
class PrototypeFactory(Borg):
    """ A Prototype factory/registry class """

    def __init__(self):
```

```
        """ Initializer """

        self._registry = {}

    def register(self, instance):
        """ Register a given instance """

        self._registry[instance.__class__] = instance

    def clone(self, klass):
        """  Return cloned instance of given class """

        instance = self._registry.get(klass)
        if instance == None:
            print('Error:',klass,'not registered')
        else:
            return instance.clone()
```

Let's create a few subclasses of Prototype, whose instances we can register on the factory:

```
class Name(SPrototype):
    """ A class representing a person's name """

    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __str__(self):
        return ' '.join((self.first, self.second))


class Animal(SPrototype):
    """ A class representing an animal """

    def __init__(self, name, type='Wild'):
        self.name = name
        self.type = type

    def __str__(self):
        return ' '.join((str(self.type), self.name))
```

We have two classes: one, a `Name` class another, an animal class, both of which inherit from `SPrototype`.

First create a name and animal object:

```
>>> name = Name('Bill', 'Bryson')
>>> animal = Animal('Elephant')
>>> print(name)
Bill Bryson
>>> print(animal)
Wild Elephant
```

Now, let's create an instance of `PrototypeFactory`:

```
>>> factory = PrototypeFactory()
```

Now let's register the two instances on the factory:

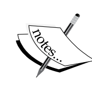
```
>>> factory.register(animal)
>>> factory.register(name)
```

Now the factory is ready to clone any number of instances from the configured instances:

```
>>> factory.clone(Name)
<prototype.Name object at 0x7ffb552f9c50>

>> factory.clone(Animal)
<prototype.Animal object at 0x7ffb55321a58>
```

The factory, rightfully, complains if we try to clone a class whose instance is not registered:

```
>>> class C(object): pass
...
>>> factory.clone(C)
Error: <class '__main__.C'> not registered
```



The factory class shown here could be enhanced with a check for the existence of the `clone` method on the registered class to make sure any class that is registered is obeying the API of the Prototype class. This is left as an exercise to the reader.

It is instructive to discuss a few aspects of the specific example we have chosen if the reader hasn't observed them already:

- The `PrototypeFactory` class is a Factory class, so it is usually a Singleton. In this case, we have made it a Borg, as we've seen that `Borgs` make a better fist of state sharing across class hierarchies.
- The `Name` class and `Animal` class inherit from `SPrototype`, since their attributes are integers and strings which are immutable; so, a shallow copy is fine here. This is unlike our first Prototype subclass.
- Prototypes preserve the class creation signature in the prototypical instance, namely the `clone` method. This makes it easy for the programmer, as he/she does not to have to worry about the class creation signature, the order and type of parameters to `__new__`, and hence, the `__init__` methods, but only has to call `clone` on an existing instance.

The Builder pattern

A Builder pattern separates out the construction of an object from its representation (assembly) so that the same construction process can be used to build different representations.

In other words, using a Builder pattern one can conveniently create different types or representative instances of the same class, each using a slightly different building or assembling process.

Formally, the Builder pattern uses a `Director` class, which instructs the `Builder` object to build instances of the target class. Different types (classes) of builders help to build slightly different variations on the same class.

Let's look at an example:

```
class Room(object):
    """ A class representing a Room in a house """

    def __init__(self, nwindows=2, doors=1, direction='S'):
        self.nwindows = nwindows
        self.doors = doors
        self.direction = direction

    def __str__(self):
        return "Room <facing:%s, windows=#%d>" % (self.direction,
                                                  self.nwindows)
class Porch(object):
```

```
    """ A class representing a Porch in a house """

    def __init__(self, ndoors=2, direction='W'):
        self.ndoors = ndoors
        self.direction = direction

    def __str__(self):
        return "Porch <facing:%s, doors=#%d>" % (self.direction,
                                                  self.ndoors)

class LegoHouse(object):
    """ A lego house class """

    def __init__(self, nrooms=0, nwindows=0,nporches=0):
        # windows per room
        self.nwindows = nwindows
        self.nporches = nporches
        self.nrooms = nrooms
        self.rooms = []
        self.porches = []

    def __str__(self):
        msg="LegoHouse<rooms=#%d, porches=#%d>" % (self.nrooms,
                                                    self.nporches)

        for i in self.rooms:
            msg += str(i)

        for i in self.porches:
            msg += str(i)

        return msg

    def add_room(self,room):
        """ Add a room to the house """

        self.rooms.append(room)

    def add_porch(self,porch):
        """ Add a porch to the house """

        self.porches.append(porch)
```

Our example shows three classes, which are as follows:

- A `Room` and `Porch` class each representing a room and porch of a house—a room has windows and doors, and a porch has doors.
- A `LegoHouse` class representing a toy example for an actual house (We are imagining a kid building a house with lego blocks here, with rooms and porches.) The Lego house will consist of any number of rooms and porches.

Let's try and create a simple `LegoHouse` instance with one room and one porch, each with the default configuration:

```
>>> house = LegoHouse(nrooms=1,nporches=1)
>>> print(house)
LegoHouse<rooms=#1, porches=#1>
```

Are we done ? No! Notice that our `LegoHouse` is a class that doesn't fully construct itself in its constructor. The rooms and porches are not really built yet, only their counters are initialized.

So we need to build the rooms and porches separately, and add them to the house. Let's do that:

```
>>> room = Room(nwindows=1)
>>> house.add_room(room)
>>> porch = Porch()
>>> house.add_porch(porch)
>>> print(house)
LegoHouse<rooms=#1, porches=#1>
Room <facing:S, windows=#1>
Porch <facing:W, doors=#1>
```

Now you see that our house is fully built. Printing it displays not only the number of rooms and porches, but also details about them. All good!

Now, imagine that you need to build 100 such different house instances, each with different configurations of rooms and porches, and often the rooms themselves have varying numbers of windows and directions!

(Maybe you are building a mobile game which uses Lego Houses where cute little characters such as Trolls or Minions stay and do interesting things.)

It is pretty clear from the example that writing code like the last will not scale to solve the problem.

This is where the Builder pattern can help you. Let's start with a simple `LegoHouse` builder.

```
class LegoHouseBuilder(object):
    """ Lego house builder class """

    def __init__(self, *args, **kwargs):
        self.house = LegoHouse(*args, **kwargs)

    def build(self):
        """ Build a lego house instance and return it """

        self.build_rooms()
        self.build_porches()
        return self.house

    def build_rooms(self):
        """ Method to build rooms """

        for i in range(self.house.nrooms):
            room = Room(self.house.nwindows)
            self.house.add_room(room)

    def build_porches(self):
        """ Method to build porches """

        for i in range(self.house.nporches):
            porch = Porch(1)
            self.house.add_porch(porch)
```

The following are the main aspects of this class:

- You configure the Builder class with the target class configuration—the number of rooms and porches in this case.
- It provides a `build` method, which constructs and assembles (builds) the components of the house—in this case, `Rooms` and `Porches`, according to the specified configuration.
- The `build` method returns the constructed and assembled house.

Now building different types of Lego Houses with different designs of rooms and porches is just two lines of code:

```
>>> builder=LegoHouseBuilder(nrooms=2,nporches=1,nwindows=1)
>>> print(builder.build())
LegoHouse<rooms=#2, porches=#1>
Room <facing:S, windows=#1>
Room <facing:S, windows=#1>
Porch <facing:W, doors=#1>
```

We will now build a similar house, but with rooms that have two windows each:

```
>>> builder=LegoHouseBuilder(nrooms=2,nporches=1,nwindows=2)
>>> print(builder.build())
LegoHouse<rooms=#2, porches=#1>
Room <facing:S, windows=#2>
Room <facing:S, windows=#2>
Porch <facing:W, doors=#1>
```

Let's say you find you are continuing to build a lot of Lego Houses with this configuration. You can encapsulate it in a subclass of the Builder so that the preceding code itself is not duplicated a lot:

```
class SmallLegoHouseBuilder(LegoHouseBuilder):
""" Builder sub-class building small lego house with 1 room and 1
     porch and rooms having 2 windows """

    def __init__(self):
        self.house = LegoHouse(nrooms=2, nporches=1, nwindows=2)
```

Now, the house configuration is *burned into* the new builder class, and building one is as simple as this:

```
>>> small_house=SmallLegoHouseBuilder().build()
>>> print(small_house)
LegoHouse<rooms=#2, porches=#1>
Room <facing:S, windows=#2>
Room <facing:S, windows=#2>
Porch <facing:W, doors=#1>
```

You can also build many of them (say `100`, `50` for the Trolls and `50` for the Minions) as follows:

```
>>> houses=list(map(lambda x: SmallLegoHouseBuilder().build(),
range(100)))
>>> print(houses[0])
LegoHouse<rooms=#2, porches=#1>
```

```
Room <facing:S, windows=#2>
Room <facing:S, windows=#2>
Porch <facing:W, doors=#1>

>>> len(houses)
100
```

One can also create more exotic builder classes which do some very specific things. For example, the following is a builder class which creates houses with rooms and porches always facing north:

```
class NorthFacingHouseBuilder(LegoHouseBuilder):
    """ Builder building all rooms and porches facing North """

    def build_rooms(self):

        for i in range(self.house.nrooms):
            room = Room(self.house.nwindows, direction='N')
            self.house.add_room(room)

    def build_porches(self):

        for i in range(self.house.nporches):
            porch = Porch(1, direction='N')
            self.house.add_porch(porch)


>>> print(NorthFacingHouseBuilder(nrooms=2, nporches=1, nwindows=1).
build())
LegoHouse<rooms=#2, porches=#1>
Room <facing:N, windows=#1>
Room <facing:N, windows=#1>
Porch <facing:N, doors=#1>
```

And, by using Python's multiple inheritance power, one can combine any such builders into new and interesting subclasses. The following, for example, is a builder that produces north-facing small houses:

```
class NorthFacingSmallHouseBuilder(NorthFacingHouseBuilder,
SmallLegoHouseBuilder):
    pass
```

As expected, it always produces North-facing, small houses with 2 windowed rooms repeatedly. Not very interesting maybe, but very reliable indeed:

```
>>> print(NorthFacingSmallHouseBuilder().build())
LegoHouse<rooms=#2, porches=#1>
Room <facing:N, windows=#2>
Room <facing:N, windows=#2>
Porch <facing:N, doors=#1>
```

Before we conclude our discussion on Creational Patterns, let's summarize some interesting aspects of these creational patterns and their interplay, as follows:

- **Builder and Factory**: The Builder pattern separates out the assembling process of a class's instance from its creation. A Factory on the other hand is concerned with creating instances of different sub-classes belonging to the same hierarchy using a unified interface. A builder also returns the built instance as a final step, whereas a Factory returns the instance immediately, as there is no separate building step.
- **Builder and Prototype**: A Builder can, internally, use a prototype for creating its instances. Further instances from the same builder can then be cloned from this instance. For example, it is instructive to build a Builder class which uses one of our Prototype metaclasses to always clone a prototypical instance.
- **Prototype and Factory**: A Prototype factory can, internally, make use of a Factory pattern to build the initial instances of the classes in question.
- **Factory and Singleton**: A Factory class is usually a Singleton in traditional programming languages. The other option is to make its methods a class or static method so there is no need to create an instance of the Factory itself. In our examples, we made it a Borg instead.

We will now move on to the next class of patterns: Structural Patterns.

Patterns in Python – structural

Structural patterns concern themselves with the intricacies of combining classes or objects to form larger structures that are more than the sum of their parts.

Structural patterns implement this in these two distinct ways:

- By using class Inheritance to compose classes into one. This is the static approach.
- By using object composition at runtime to achieve combined functionality. This approach is more dynamic and flexible.

Python, by virtue of supporting multiple inheritance, can implement both of these very well. Being a language with dynamic attributes and using the power of magic methods, Python can also do object composition and the resultant method wrapping pretty well also. So, with Python, a programmer is indeed in a good place with respect to implementing structural patterns.

We will be discussing the following structural patterns in this section: Adapter, Facade, and Proxy.

The Adapter pattern

As the name implies, the Adapter pattern wraps or adapts an existing implementation of a specific interface into another interface which a client expects. The Adapter is also called a **Wrapper**.

You very often adapt objects into interfaces or types you want when you program, most often without realizing this.

Example:

Look at the following list containing two instances of a fruit and detailing how many:

```
>>> fruits=[('apples',2), ('grapes',40)]
```

Let's say you want to quickly find the number of fruits, given a fruit name. The list doesn't allow you to use the fruit as a key, which is a more suitable interface for the operation.

What do you do ? Well, you simply convert the list to a dictionary:

```
>>> fruits_d=dict(fruits)
>>> fruits_d['apples']
2
```

Voilà! You got the object in a form that is more convenient for you, adapted to your programming needs. This is a kind of data or object adaptation.

Programmers do such data or object adaptation almost continuously in their code without realizing it. Adaptation of code or data is more common than you think.

Let's consider a class Polygon, representing a regular or irregular Polygon of any shape:

```
class Polygon(object):
    """ A polygon class """

    def __init__(self, *sides):
        """ Initializer - accepts length of sides """
        self.sides = sides

    def perimeter(self):
        """ Return perimeter """

        return sum(self.sides)

    def is_valid(self):
        """ Is this a valid polygon """

        # Do some complex stuff - not implemented in base class
        raise NotImplementedError

    def is_regular(self):
        """ Is a regular polygon ? """

        # True: if all sides are equal
        side = self.sides[0]
        return all([x==side for x in self.sides[1:]])

    def area(self):
        """ Calculate and return area """

        # Not implemented in base class
        raise NotImplementedError
```

This preceding class describes a generic, closed Polygon geometric figure in geometry.

We have implemented some basic methods such as `perimeter` and `is_regular`, the latter returning whether the Polygon is a regular one such as a hexagon or pentagon.

Let's say we want to implement specific classes for a few regular geometric shapes such as a triangle or rectangle. We can implement these from scratch, of course. However, since a Polygon class is available, we can try to reuse it, and adapt it to our needs.

Let's say the `Triangle` class requires the following methods:

- `is_equilateral`: Returns whether the triangle is an equilateral one
- `is_isosceles`: Returns whether the triangle is an isosceles triangle
- `is_valid`: Implements the `is_valid` method for a triangle
- `area`: Implements the area method for a triangle

Similarly the `Rectangle` class, needs the following methods:

- `is_square`: Returns whether the rectangle is a square
- `is_valid`: Implements the `is_valid` method for a rectangle
- `area`: Implements the area method for a rectangle

The following is the code for an adapter pattern, reusing the `Polygon` class for the `Triangle` and `Rectangle` classes.

The following is the code for the `Triangle` class:

```
import itertools

class InvalidPolygonError(Exception):
    pass

class Triangle(Polygon):
    """ Triangle class from Polygon using class adapter """

    def is_equilateral(self):
        """ Is this an equilateral triangle ? """

        if self.is_valid():
            return super(Triangle, self).is_regular()

    def is_isosceles(self):
        """ Is the triangle isosceles """
```

```
        if self.is_valid():
            # Check if any 2 sides are equal
            for a,b in itertools.combinations(self.sides, 2):
                if a == b:
                    return True
        return False

    def area(self):
        """ Calculate area """

        # Using Heron's formula
        p = self.perimeter()/2.0
        total = p
        for side in self.sides:
            total *= abs(p-side)

        return pow(total, 0.5)

    def is_valid(self):
        """ Is the triangle valid """

        # Sum of 2 sides should be > 3rd side
        perimeter = self.perimeter()
        for side in self.sides:
            sum_two = perimeter - side
            if sum_two <= side:
                raise InvalidPolygonError(str(self.__class__) + "is
invalid!")

        return True
```

Take a look at the following `Rectangle` class:

```
class Rectangle(Polygon):
    """ Rectangle class from Polygon using class adapter """

    def is_square(self):
        """ Return if I am a square """

        if self.is_valid():
            # Defaults to is_regular
            return self.is_regular()

    def is_valid(self):
```

```
        """ Is the rectangle valid """

        # Should have 4 sides
        if len(self.sides) != 4:
            return False

        # Opposite sides should be same
        for a,b in [(0,2),(1,3)]:
            if self.sides[a] != self.sides[b]:
                return False

        return True

    def area(self):
        """ Return area of rectangle """

        # Length x breadth
        if self.is_valid():
            return self.sides[0]*self.sides[1]
```

Now let's see classes in action.

Let's create an equilateral triangle for the first test:

```
>>> t1 = Triangle(20,20,20)
>>> t1.is_valid()
True
```

An equilateral triangle is also isosceles:

```
>>> t1.is_equilateral()
True
>>> t1.is_isosceles()
True
```

Let's calculate the area:

```
>>> t1.area()
173.20508075688772
```

Let's try a triangle which is not valid:

```
>>> t2 = Triangle(10, 20, 30)
>>> t2.is_valid()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "/home/anand/Documents/ArchitectureBook/code/chap7/adapter.py",
line 75, in is_valid
    raise InvalidPolygonError(str(self.__class__) + "is invalid!")
adapter.InvalidPolygonError: <class 'adapter.Triangle'>is invalid!
```

Its dimensions show it is a straight line, not a triangle. The `is_valid` method is not implemented in the base class, hence the subclasses need to override it to provide a proper implementation. In this case, we raise an exception if the triangle is invalid.

The following is an illustration of the `Rectangle` class in action:

```
>>> r1 = Rectangle(10,20,10,20)
>>> r1.is_valid()
True
>>> r1.area()
200
>>> r1.is_square()
False
>>> r1.perimeter()
60
```

Let's create a square:

```
>>> r2 = Rectangle(10,10,10,10)
>>> r2.is_square()
True
```

The `Rectangle`/`Triangle` classes shown here are examples of `class adapters`. This is because they inherit the class that they want to adapt, and provide the methods expected by the client, often delegating the computation to the base-class's methods. This is evident in the `is_equilateral` and `is_square` methods of the `Triangle` and `Rectangle` classes respectively.

Let's look at an alternative implementation of the same classes—this time, via object composition, in other words, `object adapters`:

```
import itertools

class Triangle (object) :
    """ Triangle class from Polygon using class adapter """

    def __init__(self, *sides):
        # Compose a polygon
        self.polygon = Polygon(*sides)
```

```
    def perimeter(self):
        return self.polygon.perimeter()

    def is_valid(f):
        """ Is the triangle valid """

        def inner(self, *args):
            # Sum of 2 sides should be > 3rd side
            perimeter = self.polygon.perimeter()
            sides = self.polygon.sides

            for side in sides:
                sum_two = perimeter - side
                if sum_two <= side:
                    raise InvalidPolygonError(str(self.__class__) +
                                              "is invalid!")

            result = f(self, *args)
            return result

        return inner

    @is_valid
    def is_equilateral(self):
        """ Is this equilateral triangle ? """

        return self.polygon.is_regular()

    @is_valid
    def is_isosceles(self):
        """ Is the triangle isoscles """

        # Check if any 2 sides are equal
        for a,b in itertools.combinations(self.polygon.sides, 2):
            if a == b:
                return True
        return False

    def area(self):
        """ Calculate area """

        # Using Heron's formula
        p = self.polygon.perimeter()/2.0
        total = p
```

```
        for side in self.polygon.sides:
            total *= abs(p-side)

        return pow(total, 0.5)
```

This class works similarly to the other one, even though the internal details are implemented via object composition rather than class inheritance:

```
>>> t1=Triangle(2,2,2)
>>> t1.is_equilateral()
True
>>> t2 = Triangle(4,4,5)
>>> t2.is_equilateral()
False
>>> t2.is_isosceles()
True
```

The main differences between this implementation and the class adapter are as follows:

- The object adapter class doesn't inherit from the class we want to adapt from. Instead, it composes an instance of the class.
- Any wrapper methods are forwarded to the composed instance, for example, the `perimeter` method.
- All attribute access to the wrapped instance has to be specified explicitly in this implementation. Nothing comes for free since we are not inheriting the class. (For example, inspect the way we access the `sides` attribute of the enclosed `polygon` instance.)

Observe how we converted the previous `is_valid` method to a decorator in this implementation. This is because many methods carry out a first check on `is_valid`, and then perform their actions, so it is an ideal candidate for a decorator. This also aids rewriting this implementation to a more convenient form, which is discussed next.

One problem with the object adapter implementation, as shown in the preceding implementation, is that any attribute reference to the enclosed adapted instance has to be made explicitly. For example, had we forgotten to implement the `perimeter` method for the `Triangle` class here, there would have been no method at all to call, as we aren't inheriting from the `Adapter` class.

The following is an alternate implementation, which makes use of the power of one of Python's magic methods, namely `__getattr__`, to simplify this. We are demonstrating this implementation on the `Rectangle` class:

```
class Rectangle(object):
    """ Rectangle class from Polygon using object adapter """


    method_mapper = {'is_square': 'is_regular'}

    def __init__(self, *sides):
        # Compose a polygon
        self.polygon = Polygon(*sides)

    def is_valid(f):
        def inner(self, *args):
            """ Is the rectangle valid """

            sides = self.sides
            # Should have 4 sides
            if len(sides) != 4:
                return False

            # Opposite sides should be same
            for a,b in [(0,2),(1,3)]:
                if sides[a] != sides[b]:
                    return False

            result = f(self, *args)
            return result

        return inner

    def __getattr__(self, name):
        """ Overloaded __getattr__ to forward methods to wrapped
            instance """

        if name in self.method_mapper:
            # Wrapped name
            w_name = self.method_mapper[name]
            print('Forwarding to method',w_name)
            # Map the method to correct one on the instance
            return getattr(self.polygon, w_name)
        else:
```

```
            # Assume method is the same
            return getattr(self.polygon, name)

    @is_valid
    def area(self):
        """ Return area of rectangle """

        # Length x breadth
        sides = self.sides
        return sides[0]*sides[1]
```

Let's look at examples using this class:

```
>>> r1=Rectangle(10,20,10,20)
>>> r1.perimeter()
60
>>> r1.is_square()
Forwarding to method is_regular
False
```

You can see that we are able to call the method `is_perimeter` on the `Rectangle` instance even though no such method is actually defined on the class. Similarly, `is_square` seems to work magically. What is happening here?

The magic method `__getattr__` is invoked by Python on an object if it cannot find an attribute in the usual ways – by first looking up the object's dictionary, then its class's dictionary, and so on. It takes a name, and hence provides a hook on a class, to implement a way to provide method lookups by routing them to other objects.

In this case, the `__getattr__` method does the following:

- Checks for the attribute name in the `method_mapper` dictionary. This is a dictionary we have created on the class, which maps a method name that we want to call on the class (as a key) to the actual method name on the wrapped instance (as a value). If an entry is found, it is returned.
- If no entry is found on the `method_mapper` dictionary, the entry is passed as such to the wrapped instance to be looked up by the same name.
- We use `getattr` in both cases to look up and return the attribute from the wrapped instance.

- Attributes can be anything—data attributes or methods. For example, see how we refer to the `sides` attribute of the wrapped `polygon` instance as if it belonged to the `Rectangle` class in the method `area` and the `is_valid` decorator.
- If an attribute is not present on the wrapped instance, it raises an `AttributeError`:

```
    >>> r1.convert_to_parallelogram(angle=30)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
     File "adapter_o.py", line 133, in __getattr__
        return getattr(self.polygon, name)
    AttributeError: 'Polygon' object has no attribute 'convert_to_
parallelogram'
```

Object adapters implemented using this technique are much more versatile, and lead to less code than regular object adapters where every method has to be explicitly written and forwarded to the wrapped instance.

The Facade pattern

A facade is a structural pattern that provides a unified interface to multiple interfaces in a subsystem. The Facade pattern is useful where a system consists of multiple subsystems, each with its own interfaces, but presents some high-level functionality, which needs to be captured, as a general top-level interface to the client.

A classic example of an object in everyday life which is a Facade is an automobile.

For example, a car consists of an engine, power train, axle and wheel assembly, electronics, steering systems, brake systems, and other such components.

However, usually, you don't have to bother whether the brake in your car is a disc-brake, or whether its suspension is coil-spring or McPherson struts, do you?

This is because the car manufacturer has provided a Facade for you to operate and maintain the car which reduces the complexity and provides you with simpler sub-systems which are easy to operate by themselves, such as the following:

- The ignition system to start the car
- The steering system to maneuver it
- The clutch-accelerator-brake system to control it
- The gear and transmission system to manage the power and speed

A lot of complex systems around us are Facades. Like the car example, a computer is a Facade, an Industrial Robot is another. All factory control systems are facades, supplying a few dashboards and controls for the engineer to tweak the complex systems behind it, and keep them running.

Facades in Python

The Python standard library contains a lot of modules which are good examples of Facades. The `compiler` module, which provides hooks to parse and compile Python source code, is a Facade to the lexer, parser, AST tree generator, and the like.

The following shows the help contents of this module:

```
anand@ubuntu-pro-book: ~/Documents/ArchitectureBook/code/chap7
File  Edit  View  Search  Terminal  Help
Help on package compiler:

NAME
    compiler - Package for parsing and compiling Python source code

FILE
    /usr/lib/python2.7/compiler/__init__.py

DESCRIPTION
    There are several functions defined at the top level that are imported
    from modules contained in the package.

    parse(buf, mode="exec") -> AST
        Converts a string containing Python source code to an abstract
        syntax tree (AST).  The AST is defined in compiler.ast.

    parseFile(path) -> AST
        The same as parse(open(path))

    walk(ast, visitor, verbose=None)
        Does a pre-order walk over the ast using the visitor instance.
        See compiler.visitor for details.

    compile(source, filename, mode, flags=None, dont_inherit=None)
        Returns a code object.  A replacement for the builtin compile() function.

    compileFile(filename)
        Generates a .pyc file by compiling filename.

PACKAGE CONTENTS
:
```

In the next page of the help contents, you can see how this module acts as a facade to other modules which are used to implement the functions defined in this package. (Look at `PACKAGE CONTENTS` at the bottom of the screenshot):

```
anand@ubuntu-pro-book: ~/Documents/ArchitectureBook/code/chap7
File  Edit  View  Search  Terminal  Help

    parse(buf, mode="exec") -> AST
        Converts a string containing Python source code to an abstract
        syntax tree (AST).  The AST is defined in compiler.ast.

    parseFile(path) -> AST
        The same as parse(open(path))

    walk(ast, visitor, verbose=None)
        Does a pre-order walk over the ast using the visitor instance.
        See compiler.visitor for details.

    compile(source, filename, mode, flags=None, dont_inherit=None)
        Returns a code object.  A replacement for the builtin compile() function.

    compileFile(filename)
        Generates a .pyc file by compiling filename.

PACKAGE CONTENTS
    ast
    consts
    future
    misc
    pyassem
    pycodegen
    symbols
    syntax
    transformer
    visitor

(END)
```

Let's look at sample code for a Facade pattern. In this example, we will model a Car with a few of its multiple subsystems.

The following is the code for all the subsystems:

```
class Engine(object):
    """ An Engine class """

    def __init__(self, name, bhp, rpm, volume, cylinders=4,
      type='petrol'):
        self.name = name
        self.bhp = bhp
        self.rpm = rpm
        self.volume = volume
        self.cylinders = cylinders
        self.type = type

    def start(self):
        """ Fire the engine """
```

```
        print('Engine started')

    def stop(self):
        """ Stop the engine """
        print('Engine stopped')

class Transmission(object):
    """ Transmission class """

    def __init__(self, gears, torque):
        self.gears = gears
        self.torque = torque
        # Start with neutral
        self.gear_pos = 0

    def shift_up(self):
        """ Shift up gears """

        if self.gear_pos == self.gears:
            print('Cannot shift up anymore')
        else:
            self.gear_pos += 1
            print('Shifted up to gear',self.gear_pos)

    def shift_down(self):
        """ Shift down gears """

        if self.gear_pos == -1:
            print("In reverse, can't shift down")
        else:
            self.gear_pos -= 1
            print('Shifted down to gear',self.gear_pos)

    def shift_reverse(self):
        """ Shift in reverse """

        print('Reverse shifting')
        self.gear_pos = -1

    def shift_to(self, gear):
        """ Shift to a gear position """

        self.gear_pos = gear
```

```
        print('Shifted to gear',self.gear_pos)


class Brake(object):
    """ A brake class """

    def __init__(self, number, type='disc'):
        self.type = type
        self.number = number

    def engage(self):
        """ Engage the break """

        print('%s %d engaged' % (self.__class__.__name__,
                                 self.number))

    def release(self):
        """ Release the break """

        print('%s %d released' % (self.__class__.__name__,
                                  self.number))

class ParkingBrake(Brake):
    """ A parking brake class """

    def __init__(self, type='drum'):
        super(ParkingBrake, self).__init__(type=type, number=1)


class Suspension(object):
    """ A suspension class """

    def __init__(self, load, type='mcpherson'):
        self.type = type
        self.load = load

class Wheel(object):
    """ A wheel class """

    def __init__(self, material, diameter, pitch):
        self.material = material
        self.diameter = diameter
```

```
        self.pitch = pitch

class WheelAssembly(object):
    """ A wheel assembly class """

    def __init__(self, brake, suspension):
        self.brake = brake
        self.suspension = suspension
        self.wheels = Wheel('alloy', 'M12',1.25)

    def apply_brakes(self):
        """ Apply brakes """

        print('Applying brakes')
        self.brake.engage()

class Frame(object):
    """ A frame class for an automobile """

    def __init__(self, length, width):
        self.length = length
        self.width = width
```

As you can see, we have covered a good number of the subsystems in a car, or those which are essential, at least.

The following code for the `Car` class combines them as a Facade with two methods, to `start` and `stop` the car:

```
class Car(object):
    """ A car class - Facade pattern """

    def __init__(self, model, manufacturer):
        self.engine = Engine('K-series',85,5000, 1.3)
        self.frame = Frame(385, 170)
        self.wheel_assemblies = []
        for i in range(4):
            self.wheel_assemblies.append(WheelAssembly(Brake(i+1),
                                         Suspension(1000)))

        self.transmission = Transmission(5, 115)
        self.model = model
        self.manufacturer = manufacturer
        self.park_brake = ParkingBrake()
        # Ignition engaged
```

```
        self.ignition = False

    def start(self):
        """ Start the car """

        print('Starting the car')
        self.ignition = True
        self.park_brake.release()
        self.engine.start()
        self.transmission.shift_up()
        print('Car started.')

    def stop(self):
        """ Stop the car """

        print('Stopping the car')
        # Apply brakes to reduce speed
        for wheel_a in self.wheel_assemblies:
            wheel_a.apply_brakes()

        # Move to 2nd gear and then 1st
        self.transmission.shift_to(2)
        self.transmission.shift_to(1)
        self.engine.stop()
        # Shift to neutral
        self.transmission.shift_to(0)
        # Engage parking brake
        self.park_brake.engage()
        print('Car stopped.')
```

Let's build an instance of the `Car` first:

```
>>> car = Car('Swift','Suzuki')
>>> car
<facade.Car object at 0x7f0c9e29afd0>
```

Let's now take the car out of the garage and go for a spin:

```
>>> car.start()
Starting the car
ParkingBrake 1 released
Engine started
Shifted up to gear 1
```

From the preceding output you can see that our car has started.

Now that we have driven it for a while, we can stop the car. As you may have guessed, stopping is more involved than starting!

```
>>> car.stop()
Stopping the car
Shifted to gear 2
Shifted to gear 1
Applying brakes
Brake 1 engaged
Applying brakes
Brake 2 engaged
Applying brakes
Brake 3 engaged
Applying brakes
Brake 4 engaged
Engine stopped
Shifted to gear 0
ParkingBrake 1 engaged
Car stopped.
>>>
```

Facades are useful for taking the complexity out of systems so that working with them becomes easier. As the preceding example shows, it would've been awfully difficult if we hadn't built the `start` and `stop` methods the way we did in this example. These methods hide the complexity behind the actions involved with subsystems in starting and stopping a `Car`.

This is what a Facade does best.

The proxy pattern

A proxy pattern wraps another object to control access to it. Some usage scenarios are as follows:

- We need a virtual resource closer to the client, which acts in place of the real resource in another network, for example, a remote proxy.
- We need to control/monitor access to a resource, for example, a network proxy and an instance counting proxy.
- We need to protect a resource or object (protection proxy) because direct access to it would cause security issues or compromise it, for example, a reverse proxy server.

- We need to optimize access to results from a costly computation or network operation so that the computation is not performed every time, for example, a caching proxy

A proxy always implements the interface of the object it is proxying to, its target in other words. This can be either via inheritance or via composition. In Python, the latter can be done more powerfully by overriding the `__getattr__` method, as we've seen in the Adapter example.

An instance-counting proxy

We will start with an example that demonstrates using the proxy pattern to keep track of instances of a class. We will reuse our `Employee` class and its subclasses from the Factory pattern here:

```
class EmployeeProxy(object):
    """ Counting proxy class for Employees """

    # Count of employees
    count = 0

    def __new__(cls, *args):
        """ Overloaded __new__ """
        # To keep track of counts
        instance = object.__new__(cls)
        cls.incr_count()
        return instance

    def __init__(self, employee):
        self.employee = employee

    @classmethod
    def incr_count(cls):
        """ Increment employee count """
        cls.count += 1

    @classmethod
    def decr_count(cls):
        """ Decrement employee count """
        cls.count -= 1

    @classmethod
    def get_count(cls):
```

```
        """ Get employee count """
        return cls.count

    def __str__(self):
        return str(self.employee)

    def __getattr__(self, name):
        """ Redirect attributes to employee instance """

        return getattr(self.employee, name)

    def __del__(self):
        """ Overloaded __del__ method """
        # Decrement employee count
        self.decr_count()

class EmployeeProxyFactory(object):
    """ An Employee factory class returning proxy objects """

    @classmethod
    def create(cls, name, *args):
        """ Factory method for creating an Employee instance """

        name = name.lower().strip()

        if name == 'engineer':
            return EmployeeProxy(Engineer(*args))
        elif name == 'accountant':
            return EmployeeProxy(Accountant(*args))
        elif name == 'admin':
            return EmployeeProxy(Admin(*args))
```

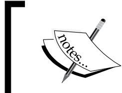
We haven't duplicated the code for the employee subclasses, as these are already available in the Factory pattern discussion.

We have two classes here: the `EmployeeProxy` and the original `factory` class modified to return instances of `EmployeeProxy` instead of employee. The modified factory class makes it easy for us to create proxy instances instead of having to do it ourselves.

The proxy, as implemented here, is a composition or object proxy, as it wraps around the target object (employee) and overloads `__getattr__` to redirect attribute access to it. It keeps track of the count of instances by overriding the `__new__` and `__del__` methods for instance creation and instance deletion respectively.

Let's see an example of using the Proxy:

```
>>> factory = EmployeeProxyFactory()
>>> engineer = factory.create('engineer','Sam',25,'M')
>>> print(engineer)
Engineer - Sam, 25 years old M
```



This prints details of the engineer via proxy, since we have overridden the `__str__` method in the proxy class, which calls the same method of the employee instance.

```
>>> admin = factory.create('admin','Tracy',32,'F')
>>> print(admin)
Admin - Tracy, 32 years old F
```

Let's check the instance count now. This can be done either via the instances or via the class, since anyway it references a class variable:

```
>>> admin.get_count()
2
>>> EmployeeProxy.get_count()
2
```

Let's delete the instances, and see what happens!

```
>>> del engineer
>>> EmployeeProxy.get_count()
1
>>> del admin
>>> EmployeeProxy.get_count()
0
```

The weak reference module in Python provides a proxy object which performs something very similar to what we have implemented, by proxying access to class instances.

The following is an example:

```
>>> import weakref
>>> import gc
>>> engineer=Engineer('Sam',25,'M')
```

Let's check the reference count of the new object:

```
>>> len(gc.get_referrers(engineer))
1
```

Now create a weak reference to it:

```
>>> engineer_proxy=weakref.proxy(engineer)
```

The `weakref` object acts in all respects like the object it's proxying for:

```
>>> print(engineer_proxy)
Engineer - Sam, 25 years old M
>>> engineer_proxy.get_role()
'engineering'
```

However, note that a `weakref` proxy doesn't increase the reference count of the proxied object:

```
>>> len(gc.get_referrers(engineer))
      1
```

Patterns in Python – behavioral

Behavioral patterns are the last stage in the complexity and functionality of patterns. They also come last chronologically in the object life cycle in a system since objects are first created then built into larger structures, before they interact with each other.

These patterns encapsulate models of communication and interaction between objects. These patterns allow us to describe complex workflows that may be difficult to follow at runtime.

Typically, Behavioral patterns favor object composition over inheritance as usually, the interacting objects in a system would be from separate class hierarchies.

In this brief discussion, we will look at the following patterns: **Iterator**, **Observer**, and **State**.

The Iterator pattern

An iterator provides a way to access elements of a container object sequentially without exposing the underlying object itself. In other words, an iterator is a proxy that provides a single method of iterating over a container object.

Iterators are everywhere in Python, so there is no special need to introduce them.

All container/sequence types in Python, that is, list, tuple, str, and set, implement their own iterators. Dictionaries also implement iterators over their keys.

In Python, an iterator is any object that implements the magic method `__iter__`, and also responds to the `iter` function returning the iterator instance.

Usually, the iterator object that is created is hidden behind the scenes in Python.

For example, we iterate through a list as follows:

```
>>> for i in range(5):
...         print(i)
...
0
1
2
3
4
```

Internally, something very similar to the following happens:

```
>>> I = iter(range(5))
>>> for i in I:
...         print(i)
...
0
1
2
3
4
```

Every sequence type implements its own iterator type as well in Python. Examples for this are given as follows:

- **Lists**:

```
>>> fruits = ['apple','oranges','grapes']
>>> iter(fruits)
<list_iterator object at 0x7fd626bedba8>
```

- **Tuples**:

```
>>> prices_per_kg = (('apple', 350), ('oranges', 80), ('grapes',
120))
>>> iter(prices_per_kg)
<tuple_iterator object at 0x7fd626b86fd0>
```

- **Sets**:

```
>>> subjects = {'Maths','Chemistry','Biology','Physics'}
>>> iter(subjects)
<set_iterator object at 0x7fd626b91558>
```

Even dictionaries come with their own special key iterator type in Python3:

```
>>> iter(dict(prices_per_kg))
<dict_keyiterator object at 0x7fd626c35ae8>
```

We will explore a small example of implementing your own iterator class/type in Python now:

```
class Prime(object):
    """ An iterator for prime numbers """

    def __init__(self, initial, final=0):
        """ Initializer - accepts a number """
        # This may or may not be prime
        self.current = initial
        self.final = final

    def __iter__(self):
        return self

    def __next__(self):
        """ Return next item in iterator """
        return self._compute()

    def _compute(self):
```

```
        """ Compute the next prime number """

        num = self.current

        while True:
            is_prime = True

            # Check this number
            for x in range(2, int(pow(self.current, 0.5)+1)):
                if self.current%x==0:
                    is_prime = False
                    break


            num = self.current
            self.current += 1

            if is_prime:
                return num

            # If there is an end range, look for it
            if self.final > 0 and self.current>self.final:
                raise StopIteration
```

This preceding class is a prime number iterator, which returns prime numbers between two limits:

```
>>> p=Prime(2,10)
>>> for num in p:
... print(num)
...
2
3
5
7
>>> list(Prime(2,50))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

The prime number iterator without the end limit is an infinite iterator. For example, the following iterator will return all prime numbers starting from `2` and will never stop:

```
>>> p = Prime(2)
```

However by combining this with the itertools module, one can extract specific data that one wants from such infinite iterators.

For example here, we use it with the `islice` method of `itertools` to compute the first 100 prime numbers:

```
>>> import itertools
>>> list(itertools.islice(Prime(2), 100))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
```

Similarly, the following are the first 10 prime numbers ending with 1 in the unit's place using the `filterfalse` method:

```
>>> list(itertools.islice(itertools.filterfalse(lambda x: x % 10 != 1,
Prime(2)), 10))
[11, 31, 41, 61, 71, 101, 131, 151, 181, 191]
```

In a similar way, the following are the first 10 palindromic primes:

```
>>> list(itertools.islice(itertools.filterfalse(lambda x:
str(x)!=str(x)[-1::-1], Prime(2)), 10))
[2, 3, 5, 7, 11, 101, 131, 151, 181, 191]
```

Interested readers are referred to the documentation on the `itertools` module and its methods to find fun and interesting ways to use and manipulate data for such infinite generators.

The Observer pattern

The Observer pattern decouples objects, but at the same time allows one set of objects (Subscribers) to keep track of the changes in another object (the Publisher). This avoids one-to-many dependency and references while keeping their interaction alive.

This pattern is also called **Publish-Subscribe**.

The following is a rather simple example using an `Alarm` class, which runs in its own thread and generates periodic alarms every second (by default). It also works as a `Publisher` class, notifying its subscribers whenever the alarm happens.

```
import threading
import time

from datetime import datetime

class Alarm(threading.Thread):
    """ A class which generates periodic alarms """

    def __init__(self, duration=1):
        self.duration = duration
        # Subscribers
        self.subscribers = []
        self.flag = True
        threading.Thread.__init__(self, None, None)

    def register(self, subscriber):
        """ Register a subscriber for alarm notifications """

        self.subscribers.append(subscriber)

    def notify(self):
        """ Notify all the subscribers """

        for subscriber in self.subscribers:
            subscriber.update(self.duration)

    def stop(self):
        """ Stop the thread """

        self.flag = False

    def run(self):
        """ Run the alarm generator """

        while self.flag:
            time.sleep(self.duration)
            # Notify
            self.notify()
```

Our subscriber is a simple `DumbClock` class, which subscribes to the `Alarm` object for its notifications and, using that, updates its time:

```
class DumbClock(object):
    """ A dumb clock class using an Alarm object """

    def __init__(self):
        # Start time
        self.current = time.time()

    def update(self, *args):
        """ Callback method from publisher """

        self.current += args[0]

    def __str__(self):
        """ Display local time """

        return datetime.fromtimestamp(self.current).
               strftime('%H:%M:%S')
```

Let's get these objects ticking:

1. First create the alarm with a notification period of 1 second. This allows:

   ```
   >>> alarm=Alarm(duration=1)
   ```

2. Next create the `DumbClock` object:

   ```
   >>> clock=DumbClock()
   ```

3. Finally, register the clock object on the alarm object as an observer so that it can receive notifications:

   ```
   >>> alarm.register(clock)
   ```

4. Now the clock will keep receiving updates from the alarm. Every time you print the clock, it will show the current time correct to the second:

   ```
   >>> print(clock)
   10:04:27
   ```

   After a while, it will show you the following:

   ```
   >>> print(clock)
   10:08:20
   ```

5. Then it will sleep for a while and print:

```
>>> print(clock);time.sleep(20);print(clock)
10:08:23
10:08:43
```

The following are some aspects to keep in mind when implementing observers:

- **References to subscribers**: Publishers can choose to keep a reference to subscribers or use a Mediator pattern to get a reference when required. A Mediator pattern decouples many objects in a system from strongly referencing each other. In Python, for example, this could be a collection of weak references or proxies or an object managing such a collection if both publisher and subscriber objects are in the same Python runtime. For remote references, one can use a remote proxy.
- **Implementing Callbacks**: In this example, the `Alarm` class directly updates the state of the subscriber by calling its `update` method. An alternate implementation is for the publisher to simply notify the subscribers, at which point they query the state of the Publisher using a `get_state` type of method to implement their own state change:

  This is the preferred option for a Publisher which may be interacting with subscribers of different types/classes. This also allows for decoupling code from the Publisher to the Subscriber as the publisher doesn't have to change its code if the `update` or `notify` method of the Subscriber changes.

- **Synchronous versus Asynchronous**: In this example, the notify is called in the same thread as the Publisher when the state is changed since the clock needs reliable and immediate notifications to be accurate. In an asynchronous implementation, this could be done asynchronously so that the main thread of the Publisher continues running; for example this may be the preferred approach in systems using asynchronous execution, which returns a future object upon notification, but the actual notification may occur sometime later.

Since we've already encountered asynchronous processing in *Chapter 5*, *Writing Applications That Scale*, we will conclude our discussion on the Observer pattern with one more example, showing an asynchronous example where the Publisher and Subscriber interact asynchronously. We will be using the asyncio module in Python for this.

For this example, we will be using the domain of news publishing. Our publisher gets news stories from various sources as news URLs which are tagged to certain specific news channels. Examples of such channels could be—"sports", "international", "technology", "India", and so on.

News subscribers register for news channels they're interested in, consuming news stories as URLs. Once they get a URL they fetch the data of the URL asynchronously. The publisher-to-subscriber notification also happens asynchronously.

The following is the source code for our publisher:

```
import weakref
import asyncio

from collections import defaultdict, deque

class NewsPublisher(object):
  """ A news publisher class with asynchronous notifications """

  def __init__(self):
      # News channels
      self.channels = defaultdict(deque)
      self.subscribers = defaultdict(list)
      self.flag = True

  def add_news(self, channel, url):
      """ Add a news story """

      self.channels[channel].append(url)

  def register(self, subscriber, channel):
      """ Register a subscriber for a news channel """

      self.subscribers[channel].append(weakref.proxy(subscriber))

  def stop(self):
      """ Stop the publisher """

      self.flag = False

  async def notify(self):
      """ Notify subscribers """

      self.data_null_count = 0

      while self.flag:
          # Subscribers who were notified
          subs = []

          for channel in self.channels:
              try:
```

```
                data = self.channels[channel].popleft()
            except IndexError:
                self.data_null_count += 1
                continue

            subscribers = self.subscribers[channel]
            for sub in subscribers:
                print('Notifying',sub,'on channel',channel,'with
                       data=>',data)
                response = await sub.callback(channel, data)
                print('Response from',sub,'for
                       channel',channel,'=>',response)
                subs.append(sub)

        await asyncio.sleep(2.0)
```

The publisher's `notify` method is asynchronous. It goes through list of channels, finds the subscribers to each of them, and calls back to the subscriber using its `callback` method, supplying it with the most recent data from the channel.

The `callback` method itself being asynchronous, it returns a future and no final processed result. Further processing of this future occurs asynchronously inside the `fetch_urls` method of the subscriber.

The following is the source code for the subscriber:

```
import aiohttp

class NewsSubscriber(object):
    """ A news subscriber class with asynchronous callbacks """

    def __init__(self):
        self.stories = {}
        self.futures = []
        self.future_status = {}
        self.flag = True

    async def callback(self, channel, data):
        """ Callback method """

        # The data is a URL
        url = data
        # We return the response immediately
        print('Fetching URL',url,'...')
        future = aiohttp.request('GET', url)
```

```
            self.futures.append(future)

            return future

        async def fetch_urls(self):

            while self.flag:

                for future in self.futures:
                    # Skip processed futures
                    if self.future_status.get(future):
                        continue

                    response = await future

                    # Read data
                    data = await response.read()

                    print('\t',self,'Got data for URL',response.
                           url,'length:',len(data))
                    self.stories[response.url] = data
                    # Mark as such
                    self.future_status[future] = 1

                await asyncio.sleep(2.0)
```

Notice how both the `callback` and `fetch_urls` methods are both declared as asynchronous. The `callback` method passes the URL from the publisher to the `aiohttp` module's `GET` method, which simply returns a future.

The future is appended as a local list of futures, which is processed again asynchronously by the `fetch_urls` method to get the URL data, which is then appended to the local stories dictionary with the URL as the key.

The following is the asynchronous loop part of the code.

Take a look at the following steps:

1. To get things started, we create a publisher and add some news stories via specific URLs to couple of channels on the publisher:

```
publisher = NewsPublisher()

# Append some stories to the 'sports' and 'india' channel

publisher.add_news('sports', 'http://www.cricbuzz.com/
cricket-news/94018/collective-dd-show-hands-massive-loss-to-
kings-xi-punjab')

publisher.add_news('sports', 'https://sports.ndtv.com/
indian-premier-league-2017/ipl-2017-this-is-how-virat-kohli-
recovered-from-the-loss-against-mumbai-indians-1681955')

publisher.add_news('india','http://www.business-standard.com/
article/current-affairs/mumbai-chennai-and-hyderabad-airports-put-
on-hijack-alert-report-117041600183_1.html')
    publisher.add_news('india','http://timesofindia.indiatimes.
com/india/pakistan-to-submit-new-dossier-on-jadhav-to-un-report/
articleshow/58204955.cms')
```

2. We then create two subscribers, one listening to the `sports` channel and the other to the `india` channel:

```
subscriber1 = NewsSubscriber()
subscriber2 = NewsSubscriber()
publisher.register(subscriber1, 'sports')
publisher.register(subscriber2, 'india')
```

3. Now we create the asynchronous event loop:

```
loop = asyncio.get_event_loop()
```

4. Next, we add the tasks as co-routines to the loop to get the asynchronous loop to start its processing. We need to add the following three tasks:
   - `publisher.notify()`:
   - `subscriber.fetch_urls()`: (one for each of the two subscribers)

5. Since both the publisher and subscriber processing loops never exit, we add a timeout to processing via its `wait` method:

```
    tasks = map(lambda x: x.fetch_urls(), (subscriber1,
subscriber2))
    loop.run_until_complete(asyncio.wait([publisher.notify(), *tas
ks],                                    timeout=120))

    print('Ending loop')
    loop.close()
```

   The following is our asynchronous Publisher and Subscriber(s) in action, on the console.

```
Terminal
(env) $ python3 observer_async.py
Notifying <__main__.NewsSubscriber object at 0x7efbf5153f98> on channel india with data=> http://www.busin
ess-standard.com/article/current-affairs/mumbai-chennai-and-hyderabad-airports-put-on-hijack-alert-report-
117041600183_1.html
Fetching URL http://www.business-standard.com/article/current-affairs/mumbai-chennai-and-hyderabad-airport
s-put-on-hijack-alert-report-117041600183_1.html ...
Response from <__main__.NewsSubscriber object at 0x7efbf5153f98> for channel india => <aiohttp.client._Ses
sionRequestContextManager object at 0x7efbf53ded38>
Notifying <__main__.NewsSubscriber object at 0x7efbf691d2e8> on channel sports with data=> http://www.cric
buzz.com/cricket-news/94018/collective-dd-show-hands-massive-loss-to-kings-xi-punjab
Fetching URL http://www.cricbuzz.com/cricket-news/94018/collective-dd-show-hands-massive-loss-to-kings-xi-
punjab ...
Response from <__main__.NewsSubscriber object at 0x7efbf691d2e8> for channel sports => <aiohttp.client._Se
ssionRequestContextManager object at 0x7efbf4c68318>
Notifying <__main__.NewsSubscriber object at 0x7efbf5153f98> on channel india with data=> http://timesofin
dia.indiatimes.com/india/pakistan-to-submit-new-dossier-on-jadhav-to-un-report/articleshow/58204955.cms
Fetching URL http://timesofindia.indiatimes.com/india/pakistan-to-submit-new-dossier-on-jadhav-to-un-repor
t/articleshow/58204955.cms ...
Response from <__main__.NewsSubscriber object at 0x7efbf5153f98> for channel india => <aiohttp.client._Ses
sionRequestContextManager object at 0x7efbf39e0ca8>
Notifying <__main__.NewsSubscriber object at 0x7efbf691d2e8> on channel sports with data=> https://sports.
ndtv.com/indian-premier-league-2017/ipl-2017-this-is-how-virat-kohli-recovered-from-the-loss-against-mumba
i-indians-1681955
Fetching URL https://sports.ndtv.com/indian-premier-league-2017/ipl-2017-this-is-how-virat-kohli-recovered
-from-the-loss-against-mumbai-indians-1681955 ...
Response from <__main__.NewsSubscriber object at 0x7efbf691d2e8> for channel sports => <aiohttp.client._Se
ssionRequestContextManager object at 0x7efbf39e05e8>
        <__main__.NewsSubscriber object at 0x7efbf691d2e8> Got data for URL http://www.cricbuzz.com/crick
et-news/94018/collective-dd-show-hands-massive-loss-to-kings-xi-punjab length: 66230
        <__main__.NewsSubscriber object at 0x7efbf691d2e8> Got data for URL https://sports.ndtv.com/india
n-premier-league-2017/ipl-2017-this-is-how-virat-kohli-recovered-from-the-loss-against-mumbai-indians-1681
```

We now move on to the last pattern in our discussion of design patterns, namely the State pattern.

The State pattern

A State pattern encapsulates the internal state of an object in another class (**state object**). The object changes its state by switching the internally encapsulated state object to different values.

A State object and its related cousin, **Finite State Machine** (**FSM**) allow a programmer to implement state transitions seamlessly across different states for the object without requiring complex code.

In Python, the State pattern can be implemented easily, since Python has a magic attribute for an object's class: the `__class__` attribute.

It may sound a bit strange, but in Python this attribute can be modified on the dictionary of the instance! This allows the instance to dynamically change its class, something which we can take advantage of to implement this pattern in Python.

The following is a simple example showing this:

```
>>> class C(object):
...     def f(self): return 'hi'
...
>>> class D(object): pass
...
>>> c = C()
>>> c
<__main__.C object at 0x7fa026ac94e0>
>>> c.f()
'hi'
>>> c.__class__=D
>>> c
<__main__.D object at 0x7fa026ac94e0>
>>> c.f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'D' object has no attribute 'f'
```

We were able to change the class of the object `c` at runtime. In this example, this proved dangerous, since `C` and `D` are unrelated classes, so this is never a smart thing to do in such cases. This is evident in the way c forgot its `f` method when it changed to an instance of class `D` (`D` has no `f` method).

However, for related classes, and more specifically, subclasses of a parent class implementing the same interface, this gives a lot of power, and can be used to implement patterns such as State.

In the following example, we have used this technique to implement the State pattern. It shows a computer which can switch from one state to another.

Notice how we are using an iterator to define this class since an iterator defines movement to the next position naturally according to its nature. We are taking advantage of this fact to implement our State pattern:

```
import random

class ComputerState(object):
    """ Base class for state of a computer """

    # This is an iterator
    name = "state"
    next_states = []
    random_states = []

    def __init__(self):
        self.index = 0

    def __str__(self):
        return self.__class__.__name__

    def __iter__(self):
        return self

    def change(self):
        return self.__next__()

    def set(self, state):
        """ Set a state """

        if self.index < len(self.next_states):
            if state in self.next_states:
                # Set index
                self.index = self.next_states.index(state)
                self.__class__ = eval(state)
                return self.__class__
            else:
                # Raise an exception for invalid state change
              current = self.__class__
                new = eval(state)
                raise Exception('Illegal transition from %s to %s' %
(current, new))
        else:
            self.index = 0
            if state in self.random_states:
```

```
                self.__class__ = eval(state)
                return self.__class__

    def __next__(self):
        """ Switch to next state """

        if self.index < len(self.next_states):
            # Always move to next state first
            self.__class__ = eval(self.next_states[self.index])
            # Keep track of the iterator position
            self.index += 1
            return self.__class__
        else:
             # Can switch to a random state once it completes
            # list of mandatory next states.
            # Reset index
            self.index = 0
            if len(self.random_states):
                state = random.choice(self.random_states)
                self.__class__ = eval(state)
                return self.__class__
            else:
                raise StopIteration
```

Now let's define some concrete subclasses of the `ComputerState` class.

Each class can define a list of `next_states` which is a set of legal states the current state can switch to. It can also define a list of random states which are random legal states it can switch to once it has switched to the next state.

For example, the following is the first state: the `off` state of the computer. The next compulsory state is of course the `on` state. Once the computer is on, this state can move off to any of the other random states.

Hence the definition is as follows:

```
class ComputerOff(ComputerState):
    next_states = ['ComputerOn']
    random_states = ['ComputerSuspend', 'ComputerHibernate',
'ComputerOff']
```

Similarly, the following are the definitions of the other state classes:

```
class ComputerOn(ComputerState):
    # No compulsory next state
    random_states = ['ComputerSuspend', 'ComputerHibernate',
'ComputerOff']

class ComputerWakeUp(ComputerState):
    # No compulsory next state
    random_states = ['ComputerSuspend', 'ComputerHibernate',
'ComputerOff']

class ComputerSuspend(ComputerState):
    next_states = ['ComputerWakeUp']
    random_states = ['ComputerSuspend', 'ComputerHibernate',
'ComputerOff']

class ComputerHibernate(ComputerState):
    next_states = ['ComputerOn']
    random_states = ['ComputerSuspend', 'ComputerHibernate',
'ComputerOff']
```

Finally, the following is the class for the Computer which uses the state classes to set its internal state.

```
class Computer(object):
    """ A class representing a computer """

    def __init__(self, model):
        self.model = model
        # State of the computer - default is off.
        self.state = ComputerOff()

    def change(self, state=None):
        """ Change state """

        if state==None:
            return self.state.change()
        else:
            return self.state.set(state)

    def __str__(self):
        """ Return state """
        return str(self.state)
```

The following are some interesting aspects of this implementation:

- **State as an iterator**: We have implemented the `ComputerState` class as an iterator. This is because a state has, naturally, a list of immediate future states it can switch to and nothing else. For example, a computer in an `Off` state can move only to the `On` state next. Defining it as an iterator allows us to take advantage of the natural progression of an iterator from one state to next.
- **Random States**: We have implemented the concept of random states in this example. Once a computer moves from one state to its mandatory next state (On to Off, Suspend to WakeUp), it has a list of random states available to move on to. A computer that is On need not always be switched off. It can also go to Sleep (Suspend) or Hibernate.
- **Manual Change**: The computer can move to a specific state via the second optional argument of the `change` method. However, this is possible only if the state change is valid; otherwise an exception is raised.

We will now see our State pattern in action.

The computer is off to start with, of course:

```
>>> c = Computer('ASUS')
>>> print(c)
ComputerOff
```

Let's see some automatic state changes:

```
>>> c.change()
<class 'state.ComputerOn'>
```

And now, let the state machine decide its next states—note these are random states till the computer enters a state where it has to mandatorily move on to the next state:

```
>>> c.change()
<class 'state.ComputerHibernate'>
```

Now the state is Hibernate, which means the next state has to be On as it is a compulsory next state:

```
>>> c.change()
<class 'state.ComputerOn'>
>>> c.change()
<class 'state.ComputerOff'>
```

Now the state is Off, which means the next state has to be On:

```
>>> c.change()
<class 'state.ComputerOn'>
```

The following are all random state changes:

```
>>> c.change()
<class 'state.ComputerSuspend'>
>>> c.change()
<class 'state.ComputerWakeUp'>
>> c.change()
<class 'state.ComputerHibernate'>
```

Now, since the underlying state is an iterator, one can even iterate on the state using a module such as itertools.

The following is an example of this – iterating on the next five states of the computer:

```
>>> import itertools
>>> for s in itertools.islice(c.state, 5):
... print (s)
...
<class 'state.ComputerOn'>
<class 'state.ComputerOff'>
<class 'state.ComputerOn'>
<class 'state.ComputerOff'>
<class 'state.ComputerOn'>
```

Now let's try some manual state changes:

```
>>> c.change('ComputerOn')
<class 'state.ComputerOn'>
>>> c.change('ComputerSuspend')
<class 'state.ComputerSuspend'>

>>> c.change('ComputerHibernate')
Traceback (most recent call last):
  File "state.py", line 133, in <module>
      print(c.change('ComputerHibernate'))
  File "state.py", line 108, in change
      return self.state.set(state)
  File "state.py", line 45, in set
      raise Exception('Illegal transition from %s to %s' %
          (current, new))
Exception: Illegal transition from <class '__main__.ComputerSuspend'>
to <class '__main__.ComputerHibernate'>
```

We get an exception when we try an invalid state transition, as the computer cannot go directly from Suspend to Hibernate. It has to wake up first!

```
>>> c.change('ComputerWakeUp')
<class 'state.ComputerWakeUp'>
>>> c.change('ComputerHibernate')
<class 'state.ComputerHibernate'>
```

All good now.

We have completed our discussion of design patterns in Python, so it is time to summarize what we've learned so far.

Summary

In this chapter, we took a detailed tour of object-oriented design patterns, and found out new and different ways to implement them in Python. We started with an overview of design patterns and their classification into Creational, Structural, and Behavioral patterns.

We went on to see an example of a Strategy design pattern, and saw how to implement this in a Pythonic manner. We then began our formal discussion of patterns in Python.

In Creational patterns, we covered the Singleton, Borg, Prototype, Factory, and Builder patterns. We saw why Borg is usually a better approach than Singleton in Python due to its ability to keep state across class hierarchies. We saw the interplay between the Builder, Prototype, and Factory patterns, and saw a few examples. Everywhere possible, metaclass discussions were introduced, and pattern implementations were done using metaclasses.

In Structural patterns, our focus was on the Adapter, Facade, and Proxy patterns. We saw detailed examples using the Adapter pattern, and discussed approaches via inheritance and object composition. We saw the power of magic methods in Python when we implemented the Adapter and Proxy patterns via the `__getattr__` technique.

In Facade, using a `Car` class, we saw a detailed example on how Facade helps programmers conquer complexity and provide generic interfaces over the subsystems. We also saw that many Python standard library modules are themselves facades.

In the Behavioral section, we discussed the Iterator, Observer, and State patterns. We saw how iterators are part and parcel of Python. We implemented an iterator as a generator for building Prime numbers.

We saw a simple example of the Observer pattern by using an `Alarm` class as a Publisher and a clock class as Subscriber. We also saw an example of an asynchronous observer pattern using the asyncio module in Python.

Finally, we ended our discussion of patterns with the State pattern. We discussed a detailed example, switching the states of a computer through allowable state changes, and how one can use Python's `__class__` as a dynamic attribute to change the class of an instance. In the implementation of State, we borrowed techniques from the Iterator pattern, and implemented the State example class as an Iterator.

In our next chapter, we move on from design to the next-higher paradigm of patterns in software architectures: architectural patterns.

8
Python – Architectural Patterns

Architectural patterns are the highest level of patterns in the pantheon of patterns in software. Architectural patterns allow the architects to specify the fundamental structure of an application. The architectural pattern chosen for a given software problem governs the rest of its activities, such as the design of systems involved, communication between different parts of the system, and so on.

There are a number of architectural patterns to choose from depending upon the problem at hand. Different patterns solve different classes or families of problems, creating their own style or class of architecture. For example, a certain class of patterns solves the architecture of client/server systems, another helps to build distributed systems, and a third helps to design highly decoupled peer-to-peer systems.

In this chapter, we will discuss and focus on a few architectural patterns that are encountered often in the Python world. Our pattern of discussion in the chapter will be to take a well-known architectural pattern, and explore one or two popular software applications or frameworks that implement it, or a variation of it.

We will not discuss a lot of code in this chapter—the usage of code will be limited to those patterns where an illustration using a program is absolutely essential. On the other hand, most of the discussion will be on the architectural details, participating subsystems, variations in the architecture implemented by the chosen application/framework, and the like.

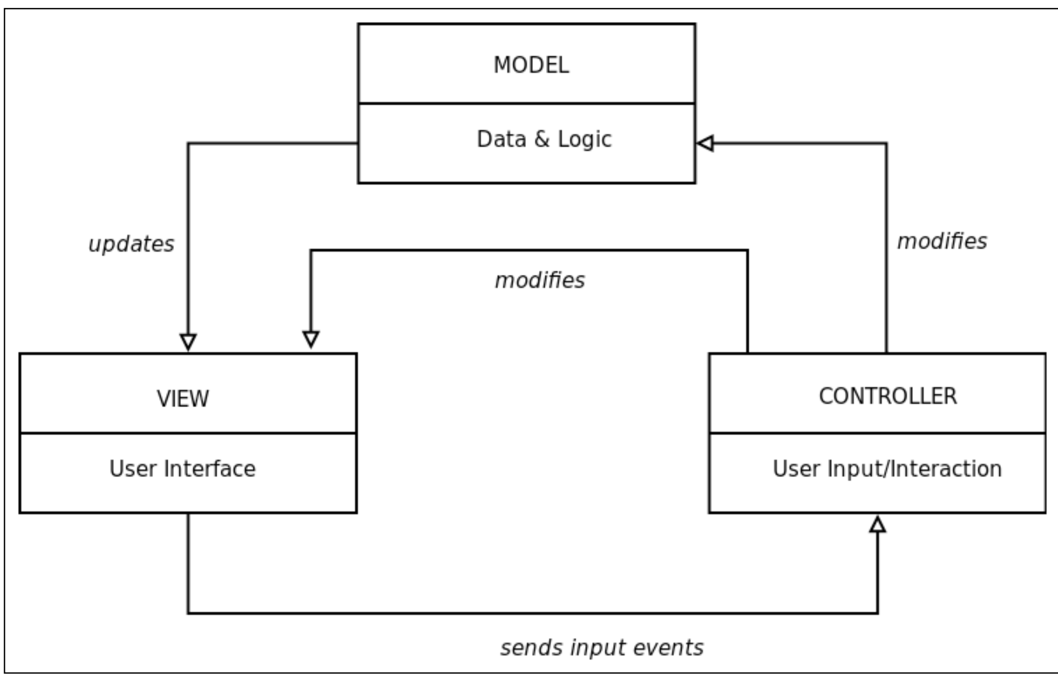
There are any number of architecture patterns that we can look at. In this chapter, we will focus on MVC and its related patterns, event-driven programming architectures, microservices architectures, and pipes and filters.

We will be covering the following topics in this chapter:

- Introducing MVC:
    - Model View Template—Django
    - Flask microframework
- Event-driven programming:
    - Chat server and client using select
    - Event-driven versus concurrent programming
    - Twisted
        - Twisted chat server and client
    - Eventlet
        - Eventlet chat server
    - Greenlets and gevent
        - Gevent chat server
- Microservices architecture:
    - Microservice frameworks in Python
    - Microservice example
    - Microservice advantages
- Pipe and filter architecture:
    - Pipe and filter in Python—examples

Introducing MVC

Model View Controller (**MVC**) is a well-known and popular architectural pattern for building interactive applications. MVC splits the application into three components: the Model, the View, and the Controller.



MVC architecture

The three components perform the following responsibilities:

- **Model**: The model contains the core data and logic of the application.
- **View**: The view(s) form the output of the application to the user. They display information to the user. Multiple views of the same data are possible.
- **Controller**: The controller receives and processes user inputs such as keyboard clicks or mouse clicks/movements, and converts them into change requests for the model or the view.

Separation of concerns using these three components avoids tight coupling between the data of the application and its representation. It allows for multiple representations (views) of the same data (model), which can be computed and presented according to user input received via the controller.

The MVC pattern allows the following interactions:

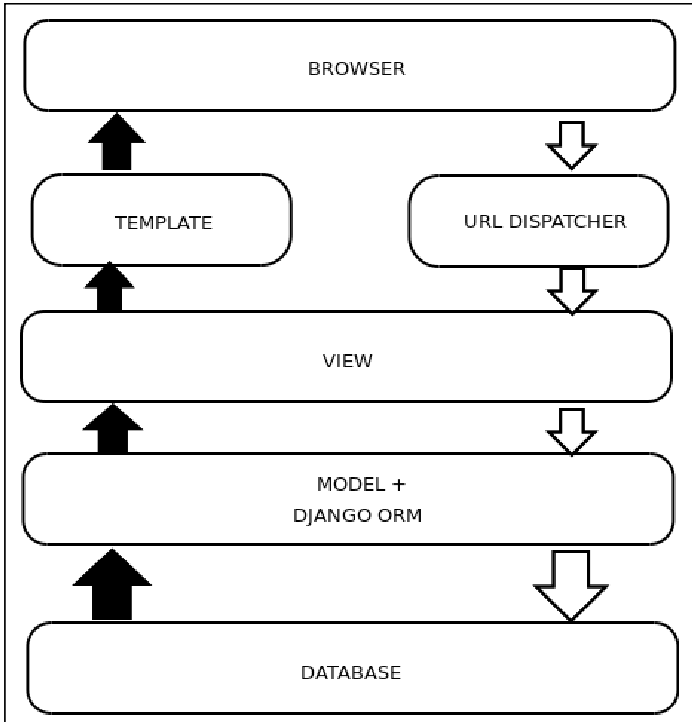
- A model can change its data depending upon inputs received from the controller.
- The changed data is reflected on the views, which are subscribed to changes in the model.

- Controllers can send commands to update the model's state, such as when making changes to a document. Controllers can also send commands to modify the presentation of a view without any change to the model, such as zooming in on a graph or chart.
- The MVC pattern implicitly includes a change propagation mechanism to notify each component of changes on the other dependent components.
- A number of web applications in the Python world implement MVC or a variation of it. We will look at a couple of them, namely Django and Flask, in the coming sections.

Model Template View (MTV) – Django

The Django project is one of the most popular web application frameworks in the Python world. Django implements something like an MVC pattern, but with some subtle differences.

The Django (core) component architecture is illustrated in the following diagram:



Django core component architecture

The core components of the Django framework are as follows:

- An **Object Relational Mapper** (**ORM**), which acts as a mediator between data models (Python) and the database (RDBMS)—this can be thought of as the **Model** layer.
- A set of callback functions in Python, which renders the data to the user interface for a specific URL—this can be thought of as the **VIEW** layer. The view focuses on building and transforming the content rather than on its actual presentation.
- A set of HTML templates to render content in different presentations. The view delegates to a specific template, which is responsible for how the data is presented.
- A regular expression-based **URL DISPATCHER**, which connects relative paths on the server to specific views and their variable arguments. This can be thought of as a rudimentary **Controller**.
- In Django, since the presentation is performed by the **TEMPLATE** layer and only the content mapping done by the **VIEW** layer, Django is often described as implementing the **MTV** framework.
- The Controller in Django is not very well defined—it can be thought of as the entire framework itself—or limited to the **URL DISPATCHER** layer.

Django admin – automated model-centric views

One of the most powerful components of the Django framework is its automatic admin system, which reads metadata from the Django models, and generates quick, model-centric admin views, where administrators of the system can view and edit data models via simple HTML forms.

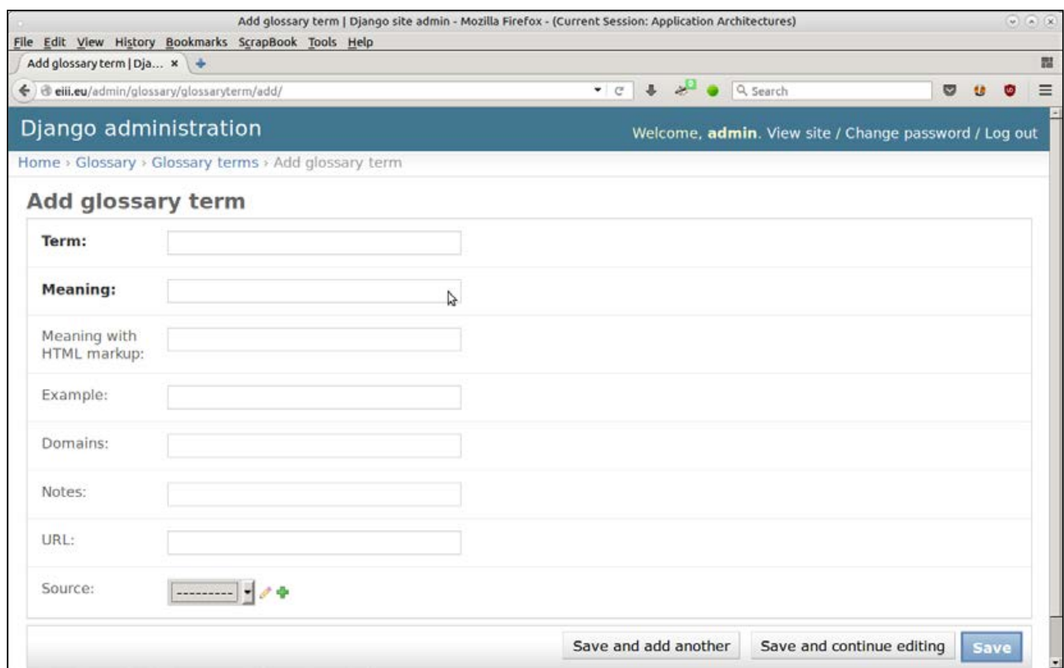
For illustration, the following is an example of a Django model that describes a term that is added to a website as a `glossary` term (a glossary is a list or index of words that describes the meaning of words related to a specific subject, text, or dialect):

```
from django.db import models

class GlossaryTerm(models.Model):
    """ Model for describing a glossary word (term) """

    term = models.CharField(max_length=1024)
    meaning = models.CharField(max_length=1024)
    meaning_html = models.CharField('Meaning with HTML markup',
                    max_length=4096, null=True, blank=True)
    example = models.CharField(max_length=4096, null=True, blank=True)

    # can be a ManyToManyField?
    domains = models.CharField(max_length=128, null=True, blank=True)

    notes = models.CharField(max_length=2048, null=True, blank=True)
    url = models.CharField('URL', max_length=2048, null=True,
blank=True)
    name = models.ForeignKey('GlossarySource', verbose_name='Source',
blank=True)

    def __unicode__(self):
        return self.term

    class Meta:
        unique_together = ('term', 'meaning', 'url')
```

This is combined with an admin system that registers a model for an automated admin view:

```
from django.contrib import admin

admin.site.register(GlossaryTerm)
admin.site.register(GlossarySource)
```

The following is a screenshot of the automated admin view (HTML form) for adding a glossary term via the Django admin interface:



Django automated admin view (HTML form) for adding a glossary term

A quick observation tells you how the Django admin is able to generate the correct field type for the different data fields in the model, and generate a form for adding the data. This is a powerful pattern present in Django that allows one to generate automated admin views for adding/editing models with almost no coding effort.

Let's now look at another popular Python web application framework, namely Flask.

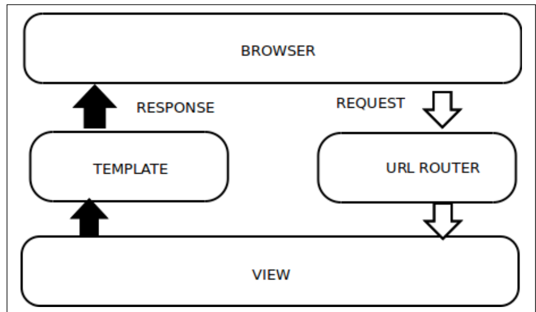
Flexible Microframework – Flask

Flask is a micro web framework that uses a minimalistic philosophy for building web applications. Flask relies on just two libraries: the Werkzeug (`http://werkzeug.pocoo.org/`) WSGI toolkit and the Jinja2 templating framework.

Flask comes with simple URL routing via decorators. The *micro* word in Flask indicates that the core of the framework is small. Support for databases, forms, and others is provided by multiple extensions that the Python community has built around Flask.

The core Flask can thus be thought of as an MTV framework minus the M (View Template), since the core does not implement support for models.

Here is an approximate schematic diagram of the Flask component architecture:



Schematic diagram of Flask components

A simple Flask application using templates looks something like this:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    data = 'some data'
    return render_template('index.html', **locals())
```

We can find a few components of the MVC pattern right here:

- The `@app.route` decorator routes requests from the browser to the `index` function. The application router can be thought of as the controller.
- The `index` function returns the data, and renders it using a template. The `index` function can be thought of as generating the view or the view component.

- Flask uses templates like Django to keep the content separate from the presentation. This can be thought of as the template component.
- There is no specific model component in Flask core. However, this can be added on with the help of additional plugins.
- Flask uses a plugin architecture to support additional features. For example, models can be added on by using Flask-SQLAlchemy, RESTful API support using Flask-RESTful, serialization using Flask-marshmallow, and others.

Event-driven programming

Event-driven programming is a paradigm of system architecture where the logic flow within the program is driven by events such as user actions, messages from other programs, or hardware (sensor) inputs.

In event-driven architectures, there is usually a main event loop, which listens for events and then triggers callback functions with specific arguments when an event is detected.

In modern operating systems such as Linux, support for events on input file descriptors such as sockets or opened files is implemented by system calls such as `select`, `poll`, and `epoll`.

Python provides wrappers to these system calls via its `select` module. It is not very difficult to write a simple event-driven program using the `select` module in Python.

The following set of programs together implement a basic chat server and client in Python using the power of the select module.

Chat server and client using I/O multiplexing with the select module

Our chat server uses the `select` system call via the `select` module to create channels where clients can connect to and talk with each other. It handles the events (sockets) that are input ready–if the event is a client connecting to the server, it connects and performs a handshake; if the event is data to be read from standard input, the server reads the data, or else it passes the data received from one client to the others.

Here is our chat server:

```
# chatserver.py

import socket
import select
import signal
import sys
from communication import send, receive

class ChatServer(object):
    """ Simple chat server using select """


    def serve(self):
        inputs = [self.server,sys.stdin]
        self.outputs = []


        while True:

                inputready,outputready,exceptready = select.
select(inputs, self.outputs, [])





            for s in inputready:

                if s == self.server:
                    # handle the server socket
                    client, address = self.server.accept()

                    # Read the login name
                    cname = receive(client).split('NAME: ')[1]

                    # Compute client name and send back
                    self.clients += 1
                    send(client, 'CLIENT: ' + str(address[0]))
                    inputs.append(client)

                    self.clientmap[client] = (address, cname)
```

```
                    self.outputs.append(client)

                elif s == sys.stdin:
                    # handle standard input - the server exits
                    junk = sys.stdin.readline()
            break
                else:
                    # handle all other sockets
                    try:
                        data = receive(s)
                        if data:
                            # Send as new client's message...
                            msg = '\n#[' + self.get_name(s) + ']>> ' +
data
                            # Send data to all except ourselves
                            for o in self.outputs:
                                if o != s:
                                    send(o, msg)
                        else:
                            print('chatserver: %d hung up' %
s.fileno())
                            self.clients -= 1
                            s.close()
                            inputs.remove(s)
                            self.outputs.remove(s)

                    except socket.error as e:
                        # Remove
                        inputs.remove(s)
                        self.outputs.remove(s)

        self.server.close()

if __name__ == "__main__":
    ChatServer().serve()
```

Since the code of the chat server is big, we are only including the main function, namely the `serve` function here showing how the server uses select-based I/O multiplexing. A lot of code in the `serve` function has also been trimmed to keep the printed code small.

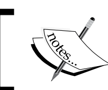
The complete source code can be downloaded from the code archive of this book from the book's website.

The chat server can be stopped by sending a single line of empty input.

The chat client also uses the `select` system call. It uses a socket to connect to the server, and then waits for events on the socket plus the standard input. If the event is from the standard input, it reads the data. Otherwise, it sends the data to the server via the socket:

```
# chatclient.py
import socket
import select
import sys
from communication import send, receive

class ChatClient(object):
    """ A simple command line chat client using select """

    def __init__(self, name, host='127.0.0.1', port=3490):
        self.name = name
        # Quit flag
        self.flag = False
        self.port = int(port)
        self.host = host
        # Initial prompt
        self.prompt='[' + '@'.join((name, socket.gethostname().
split('.')[0])) + ']> '
        # Connect to server at port
        try:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_
STREAM)
            self.sock.connect((host, self.port))
            print('Connected to chat server@%d' % self.port)
            # Send my name...
            send(self.sock,'NAME: ' + self.name)
            data = receive(self.sock)
            # Contains client address, set it
            addr = data.split('CLIENT: ')[1]
            self.prompt = '[' + '@'.join((self.name, addr)) + ']> '
        except socket.error as e:
            print('Could not connect to chat server @%d' % self.port)
            sys.exit(1)

    def chat(self):
```

```
        """ Main chat method """

        while not self.flag:
            try:
                sys.stdout.write(self.prompt)
                sys.stdout.flush()

                # Wait for input from stdin & socket
                inputready, outputready,exceptrdy = select.select([0, 
self.sock], [],[])

                for i in inputready:
                    if i == 0:
                        data = sys.stdin.readline().strip()
                        if data: send(self.sock, data)
                    elif i == self.sock:
                        data = receive(self.sock)
                        if not data:
                            print('Shutting down.')
                            self.flag = True
                            break
                        else:
                            sys.stdout.write(data + '\n')
                            sys.stdout.flush()

            except KeyboardInterrupt:
                print('Interrupted.')
                self.sock.close()
                break

if __name__ == "__main__":
    if len(sys.argv)<3:
        sys.exit('Usage: %s chatid host portno' % sys.argv[0])

    client = ChatClient(sys.argv[1],sys.argv[2], int(sys.argv[3]))
    client.chat()
```



The chat client can be stopped by pressing *Ctrl* + *C* on the Terminal.

In order to send data to and fro via sockets, both these scripts use a third module, named `communication`, which has a `send` and a `receive` function. This module uses pickle to serialize and deserialize data in the `send` and `receive` functions, respectively:

```
# communication.py
import pickle
import socket
import struct

def send(channel, *args):
    """ Send a message to a channel """

    buf = pickle.dumps(args)
    value = socket.htonl(len(buf))
    size = struct.pack("L",value)
    channel.send(size)
    channel.send(buf)

def receive(channel):
    """ Receive a message from a channel """

    size = struct.calcsize("L")
    size = channel.recv(size)
    try:
        size = socket.ntohl(struct.unpack("L", size)[0])
    except struct.error as e:
        return ''

    buf = ""

    while len(buf) < size:
        buf = channel.recv(size - len(buf))

    return pickle.loads(buf)[0]
```

The following are some screenshots of the server running and two clients that are connected to each other via the chat server:

Here is the screenshot of client #1, named `andy`, connected to the chat server:

```
$ python3 chatclient.py andy localhost 3490
Connected to chat server@3490
[andy@127.0.0.1]>
(Connected: New client (2) from betty@127.0.0.1)
[andy@127.0.0.1]> hi betty
[andy@127.0.0.1]>
#[betty@127.0.0.1]>> hello andy!
[andy@127.0.0.1]> how are you ?
[andy@127.0.0.1]>
#[betty@127.0.0.1]>> I am good, and you ?
[andy@127.0.0.1]> I am fine.
[andy@127.0.0.1]>
#[betty@127.0.0.1]>> Thats nice
[andy@127.0.0.1]>
#[betty@127.0.0.1]>> Ok, see you. Bye
[andy@127.0.0.1]> Bbye.
[andy@127.0.0.1]>
[andy@127.0.0.1]> ^CInterrupted.
$
```

Chat session of chat client #1 (client name: andy)

Similarly, here is a client named `betty` who is connected to the chat server and is talking to `andy`:

```
$ python3 chatclient.py betty localhost 3490
Connected to chat server@3490
[betty@127.0.0.1]>
#[andy@127.0.0.1]>> hi betty
[betty@127.0.0.1]> hello andy!
[betty@127.0.0.1]>
#[andy@127.0.0.1]>> how are you ?
[betty@127.0.0.1]> I am good, and you ?
[betty@127.0.0.1]>
#[andy@127.0.0.1]>> I am fine.
[betty@127.0.0.1]> Thats nice
[betty@127.0.0.1]> Ok, see you. Bye
[betty@127.0.0.1]>
#[andy@127.0.0.1]>> Bbye.
[betty@127.0.0.1]>
(Hung up: Client from andy@127.0.0.1)
[betty@127.0.0.1]> ^CInterrupted.
$
```

Chat session of chat client #2 (client name: betty)

Some interesting points of the program are as follows:

- See how the clients are able to see each other's messages. This happens because the server sends the data sent by one client to all the other connected clients. Our chat server prefixes the messages with a hash (`#`) to indicate that this message is from another client.
- See how the server sends connection and disconnection information of a client to all other clients. This informs the clients when another client is connected to or disconnected from the session.
- The server echoes messages when a client disconnects saying that the client *hung up*.

The preceding chat server and client example is a minor variation of the author's own Python recipe in the ASPN Cookbook at `https://code.activestate.com/recipes/531824`.

The simple select-based multiplexing is taken to the next level by libraries such as Twisted, Eventlet, and Gevent in order to build systems that provide high-level event-based programming routines to the programmer, typically based on a core event loop very similar to the loop of our chat server example.

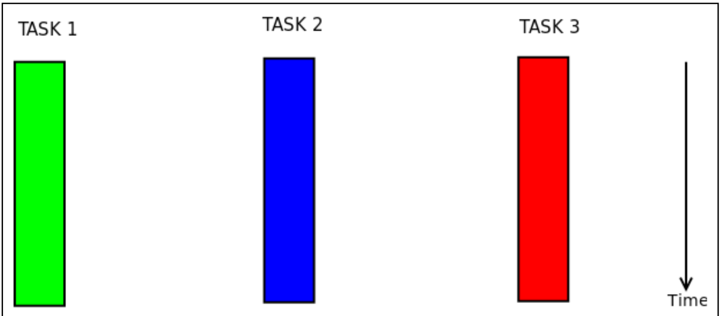
We will discuss the architecture of these frameworks in the following sections.

Event-driven programming versus concurrent programming

The example we saw in the previous section uses the technique of asynchronous events as we saw in the chapter on concurrency. This is different from true concurrent or parallel programming.

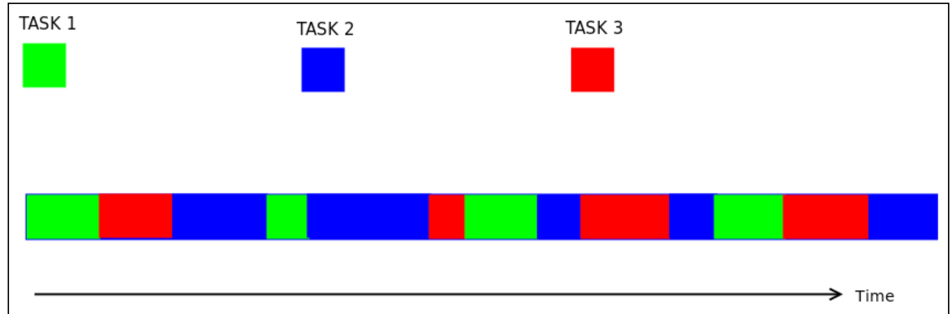
Event programming libraries also work on the technique of asynchronous events. There is only a single thread of execution in which tasks are interleaved one after another based on the events received.

In the following diagram, consider a truly parallel execution of three tasks by three threads or processes:



Parallel execution of three tasks using three threads

Contrast this with what happens when the tasks are executed via event-driven programming as depicted in the following diagram:



Asynchronous execution of three tasks in a single thread

In the asynchronous model, there is only one single thread of execution with tasks executing in an interleaved fashion. Each task gets its own slot of processing time in the event loop of the asynchronous processing server, but only one task executes at a given time. Tasks yield control back to the loop so that it can schedule a different task in the next time slice from the task that is being executed currently. As we have seen in *Chapter 5*, *Writing Applications that Scale*, this is a kind of cooperative multitasking.

Twisted

Twisted is an event-driven networking engine with support for multiple protocols, such as DNS, SMTP, POP3, IMAP, and so on. It also comes with support for writing SSH clients and servers, and to build messaging and IRC clients and servers.

Twisted also provides a set of patterns (styles) to write common servers and clients, such as web server/client (HTTP), publish/subscribe patterns, messaging clients and servers (SOAP/XML-RPC), and others.

It uses the Reactor design pattern, which multiplexes and dispatches events from multiple sources to their event handlers in a single thread.

It receives messages, requests, and connections coming from multiple concurrent clients, and processes these posts sequentially using event handlers without requiring concurrent threads or processes.

The reactor pseudo-code looks, approximately, as follows:

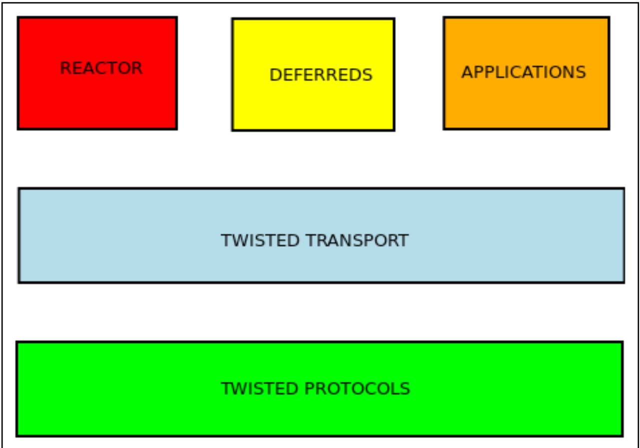
```
while True:
    timeout = time_until_next_timed_event()
    events = wait_for_events(timeout)
    events += timed_events_until(now())
    for event in events:
        event.process()
```

Twisted uses callbacks to call event handlers as and when an event happens. To handle a specific event, a callback is registered for that event. Callbacks can be used for regular processing, and also for managing exceptions (errbacks).

Like the `asyncio` module, Twisted uses an object such as futures in order to wrap the results of a task execution, whose actual results are still not available. In Twisted, these objects are called **Deferreds**.

Deferred objects have a pair of callback chains: one for processing results (callbacks) and one for managing errors (errbacks). When the result of an execution is obtained, a Deferred object is created, and its callbacks and/or errbacks are called in the order in which they were added.

Here is an architecture diagram of Twisted, showing the high-level components:



Twisted – Core Components

Twisted – a simple web client

The following is a simple example of a web HTTP client using Twisted, fetching a given URL and saving its contents to a specific filename:

```
# twisted_fetch_url.py
from twisted.internet import reactor
from twisted.web.client import getPage
import sys

def save_page(page, filename='content.html'):
    print type(page)
    open(filename,'w').write(page)
    print 'Length of data',len(page)
    print 'Data saved to',filename

def handle_error(error):
    print error

def finish_processing(value):
    print "Shutting down..."
    reactor.stop()

if __name__ == "__main__":
    url = sys.argv[1]
```

```
    deferred = getPage(url)
    deferred.addCallbacks(save_page, handle_error)
    deferred.addBoth(finish_processing)

    reactor.run()
```

As you can see in the preceding code, the `getPage` method returns a deferred, and not the data of the URL. To the deferred, we add two callbacks: one for processing the data (the `save_page` function) and another for handling errors (the `handle_error` function). The `addBoth` method of the deferred adds a single function as both callback and errback.

The event processing is started by running the reactor. In the `finish_processing` callback, which is called at the end, the reactor is stopped. Since event handlers are called in the order that they are added, this function will be called only at the very end.

When the reactor is run, the following events happen:

- The page is fetched and the deferred is created.
- The callbacks are called in order on the deferred. First the `save_page` function is called, which saves contents of the page to the `content.html` file. Then a `handle_error` event handler is called, which prints any error string.
- Finally, `finish_processing` is called, which stops the reactor, and the event processing ends, exiting the program.

At the time of writing, Twisted is not yet available for Python3, so the preceding code is written for Python2.

- When you run the code, you will see that the following output is produced:

```
$ python2 twisted_fetch_url.py http://www.google.com
Length of data 13280
Data saved to content.html
Shutting down...
```

Chat server using Twisted

Let's now see how we can write a simple chat server in Twisted on lines similar to our chat server using the `select` module.

In Twisted, servers are built by implementing protocols and protocol factories. A protocol class typically inherits from the Twisted `Protocol` class.

A factory is nothing but a class that serves as a factory pattern for protocol objects.

Using this, here is our chat server using Twisted:

```
from twisted.internet import protocol, reactor

class Chat(protocol.Protocol):
    """ Chat protocol """

    transports = {}
    peers = {}

    def connectionMade(self):
        self._peer = self.transport.getPeer()
        print 'Connected',self._peer

    def connectionLost(self, reason):
        self._peer = self.transport.getPeer()
        # Find out and inform other clients
        user = self.peers.get((self._peer.host, self._peer.port))
        if user != None:
            self.broadcast('(User %s disconnected)\n' % user, user)
            print 'User %s disconnected from %s' % (user, self._peer)

    def broadcast(self, msg, user):
        """ Broadcast chat message to all connected users except
        'user' """

        for key in self.transports.keys():
            if key != user:
                if msg != "<handshake>":
                    self.transports[key].write('#[' + user + "]>>> " +
msg)
                else:
                    # Inform other clients of connection
                    self.transports[key].write('(User %s connected
from %s)\n' % (user, self._peer))

    def dataReceived(self, data):
        """ Callback when data is ready to be read from the socket """

        user, msg = data.split(":")
        print "Got data=>",msg,"from",user
        self.transports[user] = self.transport
        # Make an entry in the peers dictionary
```

```
        self.peers[(self._peer.host, self._peer.port)] = user
        self.broadcast(msg, user)

class ChatFactory(protocol.Factory):
    """ Chat protocol factory """

    def buildProtocol(self, addr):
        return Chat()

if __name__ == "__main__":
    reactor.listenTCP(3490, ChatFactory())
    reactor.run()
```

Our chat server is a bit more sophisticated than the one before as it performs the following additional steps:

1. It has a separate handshake protocol using the special `<handshake>` message.
2. When a client connects, it is broadcast to other clients, informing them of the client's name and connection details.
3. When a client disconnects, other clients are informed about this.

The chat client also uses Twisted and uses two protocols—namely a `ChatClientProtocol` for communication with the server and a `StdioClientProtocol` for reading data from standard input and echoing data received from the server to the standard output.

The latter protocol also connects the former one to its input, so that any data that is received on the standard input is sent to the server as a chat message.

Take a look at the following code:

```
import sys
import socket
from twisted.internet import stdio, reactor, protocol


class ChatProtocol(protocol.Protocol):
    """ Base protocol for chat """

    def __init__(self, client):
        self.output = None
        # Client name: E.g: andy
        self.client = client
```

```
        self.prompt='[' + '@'.join((self.client, socket.gethostname().
split('.')[0])) + ']> '

    def input_prompt(self):
        """ The input prefix for client """
        sys.stdout.write(self.prompt)
        sys.stdout.flush()

    def dataReceived(self, data):
        self.processData(data)

class ChatClientProtocol(ChatProtocol):
    """ Chat client protocol """

    def connectionMade(self):
        print 'Connection made'
        self.output.write(self.client + ":<handshake>")

    def processData(self, data):
        """ Process data received """

        if not len(data.strip()):
            return

        self.input_prompt()

        if self.output:
            # Send data in this form to server
            self.output.write(self.client + ":" + data)


class StdioClientProtocol(ChatProtocol):
    """ Protocol which reads data from input and echoes
    data to standard output """

    def connectionMade(self):
        # Create chat client protocol
        chat = ChatClientProtocol(client=sys.argv[1])
        chat.output = self.transport

        # Create stdio wrapper
        stdio_wrapper = stdio.StandardIO(chat)
        # Connect to output
```

```
        self.output = stdio_wrapper
        print "Connected to server"
        self.input_prompt()

    def input_prompt(self):
        # Since the output is directly connected
        # to stdout, use that to write.
        self.output.write(self.prompt)

    def processData(self, data):
        """ Process data received """

        if self.output:
            self.output.write('\n' + data)
            self.input_prompt()


class StdioClientFactory(protocol.ClientFactory):

    def buildProtocol(self, addr):
        return StdioClientProtocol(sys.argv[1])


def main():
    reactor.connectTCP("localhost", 3490, StdioClientFactory())
    reactor.run()


if __name__ == '__main__':
    main()
```

Here are some screenshots of the two clients `andy` and `betty` communicating using this chat server and client:

```
Terminal
anand@mangu-probook:/home/user/programs/chap8$ python2 twisted_chat_client.py andy
Connection made
Connected to server
[andy@mangu-probook]>
(User betty connected from IPv4Address(TCP, '127.0.0.1', 39252))
[andy@mangu-probook]> hello betty
[andy@mangu-probook]>
#[betty]>>> hi andy
[andy@mangu-probook]> how are you ?
[andy@mangu-probook]>
#[betty]>>> I am good, and you ?
[andy@mangu-probook]> I am pretty fine
[andy@mangu-probook]>
#[betty]>>> Ok
[andy@mangu-probook]>
#[betty]>>> How about the dinner plan today ?
[andy@mangu-probook]> All good
[andy@mangu-probook]>
#[betty]>>> See you at 8 then
[andy@mangu-probook]> Sure, see you then
[andy@mangu-probook]>
#[betty]>>> bye andy
[andy@mangu-probook]> bye betty
anand@mangu-probook:/home/user/programs/chap8$
anand@mangu-probook:/home/user/programs/chap8$
```

Chat client using Twisted chat server—session for client #1 (andy)

Here is the second session, for the client `betty`:

```
Terminal
anand@mangu-probook:/home/user/programs/chap8$ python twisted_chat_client.py betty
Connection made
Connected to server
[betty@mangu-probook]>
#[andy]>>> hello betty
[betty@mangu-probook]> hi andy
[betty@mangu-probook]>
#[andy]>>> how are you ?
[betty@mangu-probook]> I am good, and you ?
[betty@mangu-probook]>
#[andy]>>> I am pretty fine
[betty@mangu-probook]> Ok
[betty@mangu-probook]> How about the dinner plan today ?
[betty@mangu-probook]>
#[andy]>>> All good
[betty@mangu-probook]> See you at 8 then
[betty@mangu-probook]>
#[andy]>>> Sure, see you then
[betty@mangu-probook]> bye andy
[betty@mangu-probook]>
#[andy]>>> bye betty
[betty@mangu-probook]>
#[andy]>>> (User andy disconnected)
anand@mangu-probook:/home/user/programs/chap8$
anand@mangu-probook:/home/user/programs/chap8$
```

Chat client using Twisted chat server—session for client #2 (betty)

You can follow the flow of the conversation by alternately looking at the screenshots.

Note the connection and disconnection messages are sent by the server when user `betty` connects and user andy disconnects respectively.

Eventlet

Eventlet is another well-known networking library in the Python world that allows one to write event-driven programs using the same concept of asynchronous execution.

Eventlet uses co-routines for this purpose with the help of a set of so-called *green threads*, which are lightweight user-space threads that perform cooperative multitasking.

Eventlet uses an abstraction over a set of green threads, the `Greenpool` class, in order to perform its tasks.

The `Greenpool` class runs a predefined set of `Greenpool` threads (default is `1000`), and provides ways to map functions and callables to the threads in different ways.

Here is the multiuser chat server rewritten using Eventlet:

```
# eventlet_chat.py

import eventlet
from eventlet.green import socket

participants = set()

def new_chat_channel(conn):
    """ New chat channel for a given connection """

    data = conn.recv(1024)
    user = ''

    while data:
        print("Chat:", data.strip())
        for p in participants:
            try:
                if p is not conn:
                    data = data.decode('utf-8')
```

```
                    user, msg = data.split(':')
                    if msg != '<handshake>':
                        data_s = '\n#[' + user + ']>>> says ' + msg
                    else:
                        data_s = '(User %s connected)\n' % user

                    p.send(bytearray(data_s, 'utf-8'))
            except socket.error as e:
                # ignore broken pipes, they just mean the participant
                # closed its connection already
                if e[0] != 32:
                    raise
        data = conn.recv(1024)

    participants.remove(conn)
    print("Participant %s left chat." % user)

if __name__ == "__main__":
    port = 3490
    try:
        print("ChatServer starting up on port", port)
        server = eventlet.listen(('0.0.0.0', port))

        while True:
            new_connection, address = server.accept()
            print("Participant joined chat.")
            participants.add(new_connection)
            print(eventlet.spawn(new_chat_channel,
                                 new_connection))

    except (KeyboardInterrupt, SystemExit):
        print("ChatServer exiting.")
```

This server can be used with the Twisted chat client that we've seen in the previous example, and behaves in exactly the same way. Hence, we will not show running examples of this server.

The Eventlet library internally uses `greenlets`, a package that provides green threads on Python runtime. We will see greenlet and a related library, Gevent, in the following section.

Greenlets and Gevent

Greenlet is a package that provides a version of green or microthreads on top of the Python interpreter. It is inspired by Stackless, a version of CPython that supports microthreads called stacklets. However, greenlets are able to run on the standard CPython runtime.

Gevent is a Python networking library providing a high-level synchronous API on top of `libev`, the event library written in C.

Gevent is inspired by gevent, but it features a more consistent API and better performance.

Like Eventlet, gevent does a lot of monkey patching on system libraries to provide support for cooperative multitasking. For example, gevent comes with its own sockets, just like Eventlet does.

Unlike Eventlet, gevent also requires explicit monkey patching to be done by the programmer. It provides a method to do this on the module itself.

Without further ado, let's look at the multiuser chat server using gevent:

```
# gevent_chat_server.py

import gevent
from gevent import monkey
from gevent import socket
from gevent.server import StreamServer

monkey.patch_all()

participants = set()

def new_chat_channel(conn, address):
    """ New chat channel for a given connection """

    participants.add(conn)
    data = conn.recv(1024)
    user = ''

    while data:
        print("Chat:", data.strip())
        for p in participants:
            try:
```

```
                if p is not conn:
                    data = data.decode('utf-8')
                    user, msg = data.split(':')
                    if msg != '<handshake>':
                        data_s = '\n#[' + user + ']>>> says ' + msg
                    else:
                        data_s = '(User %s connected)\n' % user

                    p.send(bytearray(data_s, 'utf-8'))
            except socket.error as e:
                # ignore broken pipes, they just mean the participant
                # closed its connection already
                if e[0] != 32:
                    raise
        data = conn.recv(1024)

    participants.remove(conn)
    print("Participant %s left chat." % user)

if __name__ == "__main__":
    port = 3490
    try:
        print("ChatServer starting up on port", port)
        server = StreamServer(('0.0.0.0', port), new_chat_channel)
        server.serve_forever()
    except (KeyboardInterrupt, SystemExit):
        print("ChatServer exiting.")
```

The code for the gevent-based chat server is almost the same as the one using Eventlet. The reason for this is that they work in very similar ways, by handling control to a callback function when a new connection is made. In both cases the callback function is named `new_chat_channel`, which has the same functionality and hence very similar code.

The differences between the two are as follows:

- gevent provides its own TCP server class—`StreamingServer`–so we use that instead of listening on the module directly.
- In the gevent server, for every connection the `new_chat_channel` handler is invoked, hence the participant set is managed there.
- Since the gevent server has its own event loop, there is no need to create a while loop for listening for incoming connections as we had to do with Eventlet.

This example works exactly the same as the previous ones and works with the Twisted chat client.

Microservice architecture

Microservice architecture is an architectural style of developing a single application as a suite of small independent services, each running in its own process and communicating via lightweight mechanisms—typically, using HTTP protocol.

Microservices are independently deployable components, and usually have zero or minimalistic central management or configuration.

Microservices can be thought of as a specific implementation style for **Service Oriented Architectures** (**SOA**), where, instead of building a monolith application top-down, the application is built as a dynamic group of mutually interacting, independent services.

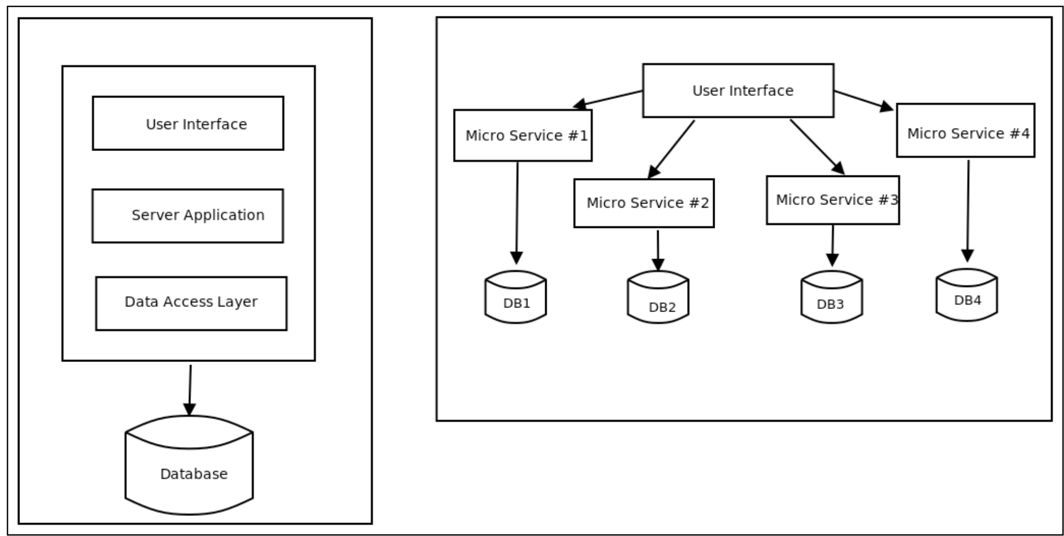
Traditionally, enterprise applications were built in a monolithic pattern, typically consisting of these three layers:

1. A client-side **user interface** (**UI**) layer consisting of HTML and JavaScript.
2. A server-side application consisting of the business logic.
3. A database and data access layer, which holds the business data.

On the other hand, a microservices architecture will split this layer into multiple services. For example, the business logic, instead of being in a single application, will be split into multiple component services, whose interactions define the logic flow inside the application. The services might query a single database or independent local databases, with the latter configuration being more common.

Data in microservices architectures are usually processed and returned in the form of document objects—typically encoded in JSON.

The following schematic diagram illustrates the difference of a monolithic architecture from a microservices one:



Monolithic (left) versus microservices (right) Architecture

Microservice frameworks in Python

With microservices being more of a philosophy or style of architecture, there are no distinct classes of software frameworks that one can say is the right fit for them. However, one can still make a few educated projections for the properties that a framework should have for it being a good choice for building a microservices architecture for your web application in Python. These properties include the following:

- The component architecture should be flexible. The framework should not be rigid in the component choices that it stipulates to make the different parts of the system work.
- The core of the framework should be lightweight. This makes sense, since if we start off with, say, a lot of dependencies for the microservices framework itself, the software starts feeling heavy right in the beginning. This may cause issues in deployment, testing, and so on.

- The framework should support zero or minimalistic configuration. Microservices architectures are usually configured automatically (zero configuration) or with a minimal set of configuration inputs that are available at one place. Usually the configuration is itself available as a microservice for other services to query and make the sharing of configuration easy, consistent, and scalable.
- It should make it very easy to take an existing piece of business logic, say, coded as a class or a function, and turn it into an HTTP or RCP service. This allows reuse and smart refactoring of code.

If you use these principles and look around in the Python software ecosystem, you will figure out that a few web application frameworks fit the bill, whereas a few don't.

For example, Flask and its single-file counterpart Bottle are good candidates for a microservices framework due to their minimal footprint, small core, and simple configuration.

A framework such as Pyramid can also be used for a microservices architecture since it promotes flexibility of choice of components and eschews tight integration.

A more sophisticated web framework such as Django makes a poor choice for a microservices framework due to exactly the opposite reasons–tight vertical integration of components, lack of flexibility in choosing components, complex configuration, and so on.

Another framework that is written specifically for implementing microservices in Python is Nameko. Nameko is geared toward testability of the application, and it provides support for different protocols for communication such as HTTP, RPC (over AMQP)—a Pub-Sub system, and a Timer service.

We will not be going into details of these frameworks. On the other hand, we will take a look at architecting and designing a real-life example of a web application using microservices.

Microservices example – restaurant reservation

Let's take a real-life example for a Python web application, and try to design it as a set of microservices.

Our application is a restaurant reservation app that helps users make a reservation for a certain number of people at a specific time in a restaurant close to their current location. Assume that reservations are only done for the same day.

The application needs to do the following:

1. Return a list of restaurants open for business at the time for which the user wants to make the reservation.
2. For a given restaurant, return enough meta information, such as cuisine choices, rating, pricing, and so on, and allow the user to filter the restaurants based on their criteria.
3. Once the user has made a choice, allow them to make a reservation at the selected restaurant for a certain number of people for a given time.

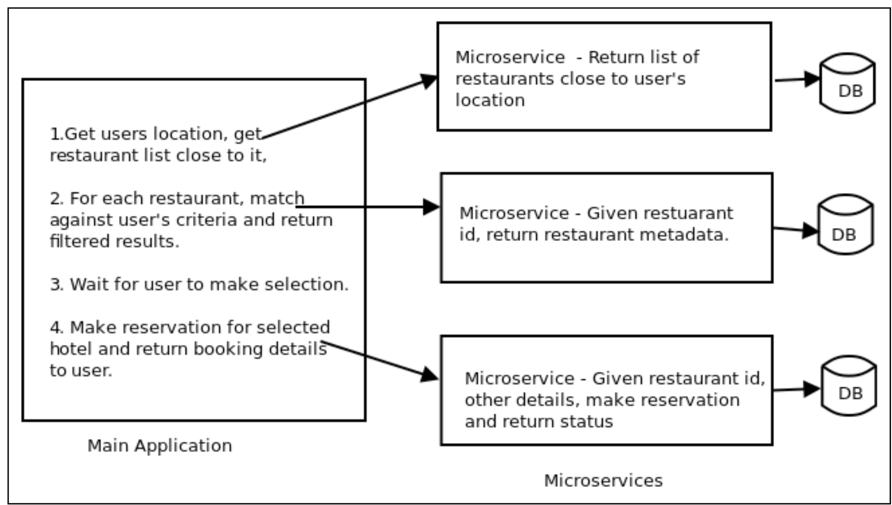
Each of these requirements is granular enough to have its own microservice.

Hence, our application will be designed with the following set of microservices:

- A service that uses the user's location, and returns a list of restaurants that are open for business and that support the online reservation API.
- A second service that retrieves metadata for a given hotel, given the restaurant ID. The application can use this metadata to compare against the user's criteria to see if it's a match.
- A third service, which, given a restaurant ID, the user's information, the number of seats required, and the time of reservation, uses the reservation API to make a reservation for seats, and returns the status.

The core parts of the application logic now fit these three microservices. Once they are implemented, the plumbing—in terms of calling these services and performing a reservation—will happen in the application logic directly.

We will not be showing any code for this application as that is a project on its own, but we will show the reader how the microservices look like in terms of their APIs and return data:



Architecture of restaurant reservation application using microservices

A microservice usually returns data in the form of JSON. For example, our first service, which returns a list of restaurants, would return a JSON similar to the one that follows:

```
GET /restaurants?geohash=tdr1y1g1zgzc

{
    "8f95e6ad-17a7-48a9-9f82-07972d2bc660": {
        "name": "Tandoor",
        "address": "Centenary building, #28, MG Road b-01"
        "hours": "12.00 - 23.30"
   },
  "4307a4b1-6f35-481b-915b-c57d2d625e93": {
        "name": "Karavalli",
        "address": "The Gateway Hotel, 66, Ground Floor"
        "hours": "12.30 - 01:00"
   },
   ...
}
```

The second service, which returns restaurant metadata, would mostly return a JSON like this one:

```
GET /restaurants/8f95e6ad-17a7-48a9-9f82-07972d2bc660

{

   "name": "Tandoor",
   "address": "Centenary building, #28, MG Road b-01"
   "hours": "12.00 – 23.30",
   "rating": 4.5,
   "cuisine": "north indian",
   "lunch buffet": "no",
   "dinner buffet": "no",
   "price": 800

}
```

Next is the interaction for the third one, which does a booking given the restaurant ID.

Since this service needs the user to provide information for the reservation, it needs a JSON payload with the details of booking. Hence, this is best done as an HTTP POST call:

```
POST  /restaurants/reserve
```

The service in this case will use the following given payload as the POST data:

```
{
   "name": "Anand B Pillai",
   "phone": 9880078014,
   "time": "2017-04-14 20:40:00",
   "seats": 3,
   "id": "8f95e6ad-17a7-48a9-9f82-07972d2bc660"
}
```

It will return a JSON like the following as a response:

```
{
   "status": "confirmed",
   "code": "WJ7D2B",
   "time": "2017-04-14 20:40:00",
   "seats": 3
}
```

With this design in place, it is not very difficult to implement the application in a framework of your choice, be it Flask, Bottle, Nameko, or anything else.

Microservices – advantages

So what are the advantages of using microservices over a monolithic application? Let's take a look at some of the important ones:

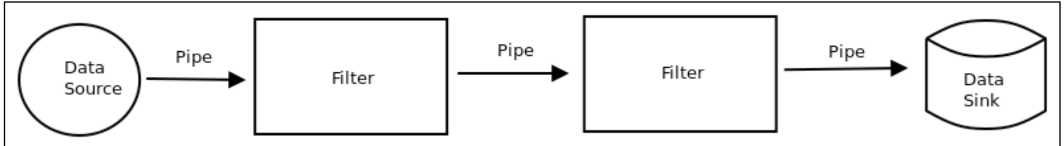
- Microservices enhance separation of concern by splitting the application logic into multiple services. This improves cohesion, and decreases coupling. There is no need for a top-down, upfront design of the system, since the business logic is not in a single place. Instead, the architect can focus on the interplay and communication between the microservices and the application, and let the design and architecture of the microservices itself emerge iteratively through refactoring.
- Microservices improve testability, since now each part of the logic is independently testable as a separate service, and hence is easy to isolate from other parts and test.
- Teams can be organized around the business capabilities rather than around tiers of the application or technology layers. Since each microservice includes logic, data, and deployment, companies using microservices encourage cross-functional roles. This helps to build a more agile organization.
- Microservices encourage decentralized data. Usually, each service will have its own local database or data store instead of the central database that is preferred by monolithic applications.
- Microservices facilitate continuous delivery and integration, and fast deployments. Since a change to business logic might often need only a small change in one or a few services, testing and redeployment can be often done in tight cycles, and in most cases, can be fully automated.

Pipe and Filter architectures

Pipe and Filter is a simple architectural style that connects a number of components that process a stream of data, each connected to the next component in the processing pipeline via a **Pipe**.

The Pipe and Filter architecture is inspired by the Unix technique of connecting the output of an application to the input of another via pipes on the shell.

The pipe and filter architecture consists of one or more data sources. The data source is connected to data filters via pipes. Filters process the data they receive, passing them to other filters in the pipeline. The final data is received at a **Data Sink**:



Pipe and Filter Architecture

Pipe and filter are used commonly for applications that perform a lot of data processing such as data analytics, data transformation, metadata extraction, and so on.

The filters can be running on the same machine, and they use actual Unix pipes or shared memory for communication. However, in large systems, these usually run on separate machines, and the pipes need not be actual pipes, but any kind of data channel such as sockets, shared memory, queues, and the like.

Multiple filter pipelines can be connected together to perform complex data processing and data staging.

A very good example of a Linux application that works using this architecture is `gstreamer`—the multimedia processing library that can perform a number of tasks on multimedia video and audio including play, record, edit, and stream.

Pipe and filter in Python

In Python, we encounter pipes in their most pure form in the multiprocessing module. The multiprocessing module provides pipes as a way to communicate from one process to another.

A pipe is created as a pair of parent and child connections. What is written on one side of the connection can be read on the other side and vice versa.

This allows us to build very simple pipelines of data processing.

For example, on Linux, the number of words in a file can be computed by this series of commands:

```
$ cat filename | wc -w
```

We will write a simple program that mimics this pipeline using the multiprocessing module:

```
# pipe_words.py
from multiprocessing import Process, Pipe
import sys

def read(filename, conn):
    """ Read data from a file and send it to a pipe """

    conn.send(open(filename).read())

def words(conn):
    """ Read data from a connection and print number of words """

    data = conn.recv()
    print('Words',len(data.split()))

if __name__ == "__main__":
    parent, child = Pipe()
    p1 = Process(target=read, args=(sys.argv[1], child))
    p1.start()
    p2 = Process(target=words, args=(parent,))
    p2.start()
    p1.join();p2.join()
```

Here is an analysis of the workflow:

1. A pipe is created, and two connections are obtained.
2. The `read` function is executed as a process, passing one end of the pipe (child) and the filename to be read.
3. This process reads the file, writing the data to the connection.
4. The `words` function is executed as a second process, passing the other end of the pipe to it.
5. When this function executes as a process, it reads the data from the connection, and prints the number of words.

The following screenshot shows the output of both the shell command and the preceding program on the same file:

```
$ cat pipe_words.py | wc -w
64
$ python pipe_words.py pipe_words.py
('Words', 64)
$
```

Output of a shell command using pipes and its equivalent Python program

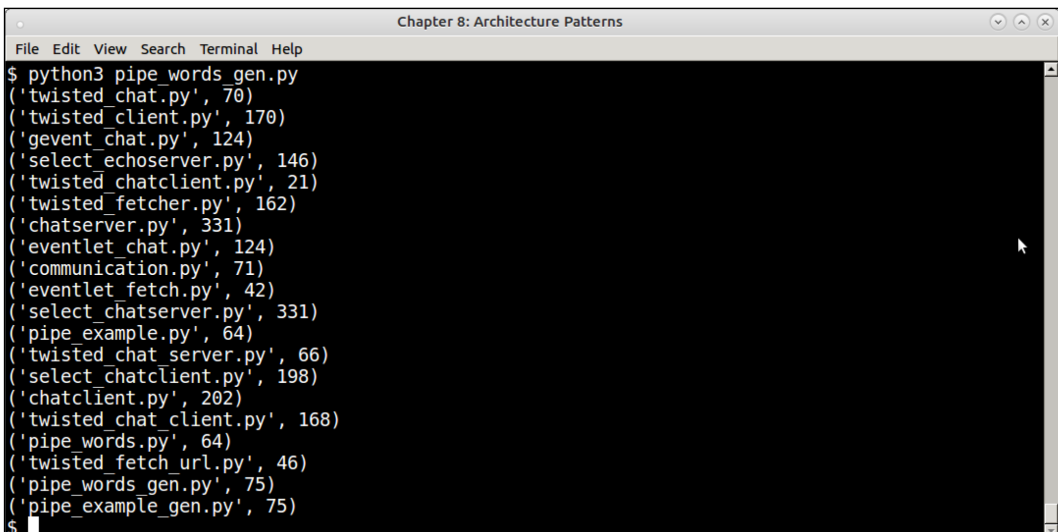
You do not need to use an object that looks like an actual pipe in order to create pipelines. On the other hand, generators in Python provide an excellent way to create a set of callables, which call each other, and consume and process each other's data, producing a pipeline of data processing.

Here is the same example as the previous one, rewritten to use generators, and this time, to process all the files in the folder matching a particular pattern:

```
# pipe_words_gen.py

# A simple data processing pipeline using generators
# to print count of words in files matching a pattern.
import os

def read(filenames):
    """ Generator that yields data from filenames as (filename, data)
tuple """

    for filename in filenames:
        yield filename, open(filename).read()

def words(input):
    """ Generator that calculates words in its input """

    for filename, data in input:
        yield filename, len(data.split())
```

```
def filter(input, pattern):
    """ Filter input stream according to a pattern """

    for item in input:
        if item.endswith(pattern):
            yield item

if __name__ == "__main__":
    # Source
    stream1 = filter(os.listdir('.'), '.py')
    # Piped to next filter
    stream2 = read(stream1)
    # Piped to last filter (sink)
    stream3 = words(stream2)

    for item in stream3:
        print(item)
```

Here is a screenshot of the output:



Output of a pipeline using generators that print the word count of Python programs

One can verify the output of a program such as the preceding one using this command:

`$ wc -w *.py`

Here is another program that uses another couple of data filtering generators to build a program, which watches files matching a specific pattern and prints information about the most recent file—something similar to what is done by the watch program on Linux:

```
# pipe_recent_gen.py
# Using generators, print details of the most recently modified file
# matching a pattern.

import glob
import os
from time import sleep

def watch(pattern):
    """ Watch a folder for modified files matching a pattern """

    while True:
        files = glob.glob(pattern)
        # sort by modified time
        files = sorted(files, key=os.path.getmtime)
        recent = files[-1]
        yield recent
        # Sleep a bit
        sleep(1)

def get(input):
    """ For a given file input, print its meta data """
    for item in input:
        data = os.popen("ls -lh " + item).read()
        # Clear screen
        os.system("clear")
        yield data

if __name__ == "__main__":
    import sys

    # Source + Filter #1
    stream1 = watch('*.' + sys.argv[1])

    while True:
        # Filter #2 + sink
        stream2 = get(stream1)
        print(stream2.__next__())
        sleep(2)
```

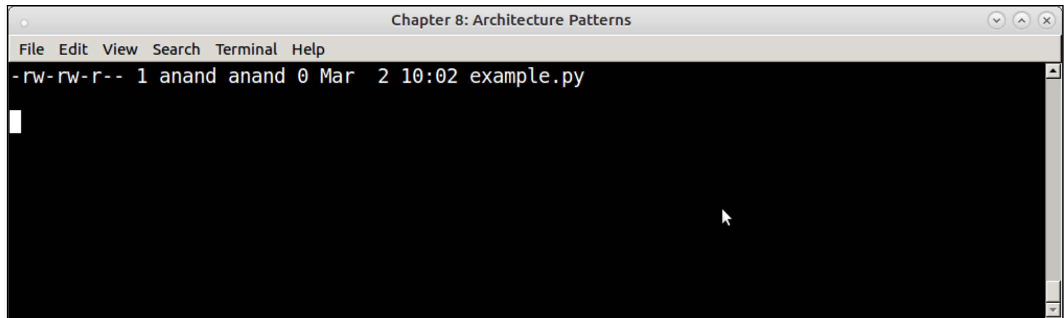
The details of this last program should be self-explanatory to the reader.

Here is the output of our program on the console, watching over Python source files:

```
Chapter 8: Architecture Patterns
File Edit View Search Terminal Help
-rw-rw-r-- 1 anand anand 870 Mar  2 09:55 pipe_recent_gen.py
```

Output of the program that watches over recently modified Python source files

If we create an empty Python source file, say `example.py`, the output changes in two seconds:



Output of the watch program changes, always showing the most recently modified file

The underlying technique of using generators (co-routines) to build such pipelines is to connect the output of one generator to the input of the next. By connecting many such generators in a series, one can build data processing pipelines that vary in complexity from simple to complex.

Of course, one can use a number of techniques for building pipelines apart from this. Some common choices are producer-consumer tasks connected using queues, which can use threads or processes. We have seen examples of this in the chapter on scalability.

Microservices can also build simple processing pipelines by connecting the input of one microservice to the output of another.

In the Python third-party software ecosystem, there are a number of modules and frameworks that allow you to build complex data pipelines. Celery, though a task queue, can be used to build simple batch processing workflows with limited pipeline support. Pipelining is not the main feature of Celery, but it has limited support for chaining tasks that can be used for this purpose.

Luigi is another robust framework that is written for complex, long-running batch processing jobs that require a pipe and filter architecture. Luigi comes with built-in support for Hadoop jobs, so it makes it a good choice for building data analytics pipelines.

Summary

In this chapter, we looked at some common architectural patterns of building software. We started with the Model View Controller architecture, and looked at examples in Django and Flask. You learned about the components of an MVC architecture, and learned that Django implements a variant of MVC using templates.

We looked at Flask as an example of a micro framework that implements the minimal footprint of a web application by using a plugin architecture with additional services that can be added on.

We went on to discuss the event-driven programming architecture, which is a kind of asynchronous programming using co-routines and events. We started with a multiuser chat example using the `select` module in Python. From there, we went on to discuss larger frameworks and libraries.

We discussed the architecture of Twisted and its components. We also discussed Eventlet and its close cousin gevent. For each of these frameworks, we saw an implementation of the multiuser chat server.

Next, we took up microservices as an architecture, which builds scalable services and deployments by splitting the core business logic across multiple services. We designed an example of a restaurant reservation application using microservices, and briefly looked at the landscape of Python web frameworks, which can be used to build microservices.

Toward the end of the chapter, we saw the architecture of using pipes and Filters for serial and scalable data processing. We built a simple example of actual pipes using the multiprocessing module in Python, which mimicked a Unix pipe command. We then looked at the technique of building pipelines using generators, and saw a couple of examples. We summarized techniques for building pipelines and frameworks available in the Python third-party software ecosystem.

This brings us to the end of the chapter on application architectures. In the next chapter, we will look at deployability, namely the aspect of deploying software to environments such as production systems.

9
Deploying Python Applications

Pushing code to production is often the last step in taking an application from development to the customer. Though this is an important activity, it often gets overlooked in the scheme of importance in a software architect's checklist.

It is a pretty common and fatal mistake to assume that, if a system works in the development environment, it will also work dutifully in production. For one thing, the configuration of a production system is often very different from that of a development environment. Many optimizations and debugging that are available and taken for granted in a developer's box, are often not available in the production setup.

Deployment to production is an art rather than an exact science. The complexity of deployment of a system depends on a number of factors, such as the language the system is developed in, its runtime portability and performance, the number of configuration parameters, whether the system is deployed in a homogeneous or heterogeneous environment, binary dependencies, geographic distribution of the deployments, deployment automation tooling, and a host of other factors.

In recent years, Python, as an open source language, has matured in the level of automation and support it provides for deploying packages to production systems. With its rich availability of built-in and third-party support tools, the pain and hassle for production deployments and maintaining deployment systems up to date has decreased.

In this chapter, we will discuss, briefly, deployable systems and the concept of deployability. We'll take some time to understand the deployment of Python applications, and the tools and processes that the architect can add to his repertoire in order to ease the deploying and maintenance of his production systems' running applications, written using Python. We will also look at techniques and best practices that an architect can adopt to keep his production systems chugging along healthily and securely, without frequent downtimes.

Here are the list of topics we will be talking about in this chapter.

- Deployability
    - Factors affecting deployability
    - Tiers of software deployment architecture
- Software Deployment in Python
    - Packaging Python code
        - PIP
        - Virtualenv
        - Virtualenv and PIP
        - PyPI—the Python Package Index
        - Packaging and submission of an application
        - PyPA
    - Remote deployments using Fabric
    - Remote deployments using Ansible
    - Managing remote daemons using Supervisor
- Deployment—patterns and best practices

Deployability

The deployability of a software system is the ease with which it can be taken from development to production. It can be measured in terms of the effort—in terms of man-hours, or complexity—in terms of the number of disparate steps required for deploying code from a development to production environment.

It is a common mistake to assume that a code that runs well in a development or staging system would behave in a similar way in a production system. It is not often the case due to the vastly dissimilar requirements that a production system has when compared to a development one.

Factors affecting deployability

Here is a brief look at some of the factors that differentiate a production system from a development one, which can often give rise to unexpected issues in deployment leading to *Production Gotchas*:

- **Optimizations and debugging**: It is very common for development systems to turn off optimizations in code.

  If your code is running in an interpreted runtime like Python, it is common to turn on debug configurations, which allows the programmer to generate generous tracebacks when an exception occurs. Also any Python interpreter optimizations are usually turned off.

  On the other hand, in production systems, the reverse is true – as optimizations are turned on and debugging is turned off. This usually requires additional configuration to be enabled for the code to work in a similar way. It is also possible (though rare) that the program gives a different behavior upon optimization under certain circumstances than it does when running unoptimized.

- **Dependencies and versions**: A development environment, usually, has a rich installation of development and support libraries for running multiple applications that a developer may be working on. Quite often, these may be dependencies which are themselves not stale, since developers often work on bleeding edge code.

  Production systems, on the other hand, need to be carefully prepared using a precompiled list of dependencies and their versions. It is quite common to specify only mature or stable versions for deployment on production systems. Hence, if a developer had relied on a feature or bug-fix which was available on an unstable (alpha, beta or release-candidate) version of a downstream dependency, one may find—too late—that the feature doesn't work in production as intended.

  Another common problem is undocumented dependencies or dependencies that need to be compiled from source code—this is often a problem with first-time deployments.

- **Resource configuration and access privileges**: Development systems and production systems often differ in level, privilege, and details of access of resources locally and in the network. A development system may have a local database, whereas, production systems tend to use separate hosting for application and database systems. A development system may use a standard configuration file, while, in production, the configuration may have to be generated specifically for a host or an environment using specific scripts. Similarly, in production, the application may be required to run with lesser privileges as a specific user/group, whereas, in development, it may be common to run the program as the root or superuser. Such disparities in user privileges and configuration may affect resource access and might cause software to fail in production, when it runs fine on the development environment.
- **Heterogeneous production environments**: Code is usually developed in development environments, which are usually homogeneous. But it may often be required to be deployed on heterogeneous systems in production. For example, software may be developed on Linux, but there may be a requirement for a customer deployment on Windows.

  The complexity of deployments increases proportionally to heterogeneity in environments. Well-managed staging and testing environments are required before such code is taken to production. Also, heterogeneous systems make dependency management more complex, as a separate list of dependencies needs to be maintained for each target system architecture.

- **Security**: In development and testing environments, it is somewhat common to give a wide berth to security aspects to save time and to reduce the configuration complexity for testing. For example, in a web application, routes which need logins may be disabled by using special development environment flags to facilitate quick programming and testing.

  Similarly, systems used in development environments may often use easy-to-guess passwords, such as database systems, web application logins, and others, to make routine recall and usage easy. Also, role-based authorization may be ignored to facilitate testing.

  However, security is critical in production, so these aspects require the opposite treatment. Routes which need logins should be enforced as such. Strong passwords should be used. Role-based authentication needs to be enforced. These can often cause subtle bugs in production where a feature which works in the development environment fails in production.

Since these and other similar problems are the bane of deploying code in production, standard practices have been defined to make the life of the DevOps practitioner a bit easier. Most companies follow the practice of using isolated environments to develop, test, and validate code and applications before pushing them to production. Let us take a look at this.

Tiers of software deployment architecture

To avoid complexities in taking the code from development to testing, and further to production, it is common to use a multitiered architecture for each stage of the life cycle of the application before deployment to production.

Let's take a look at some of the following common deployment tiers:

- **Development/Test/Stage/Production**: This is the traditional four-tiered architecture.
    - The developers push their code to a development environment, where unit tests and developer tests are run. This environment will always be on the latest trunk or bleeding edge of the code. Frequently, this environment is skipped and replaced with the local setup on developer's laptops.
    - The software is then tested by QA or testing engineers on a test environment using black-box techniques. They may also run performance tests on this environment. This environment is always behind the development environment in terms of code updates. Usually, internal releases, tags, or **code dumps** are used to sync the QA environment from the development environment.
    - The staging environment tries to mirror the production environment as closely as possible. It is the *pre-production* stage, where the software is tested on an environment as close as possible to the deployment one to identify issues that may occur in production in advance. This is the environment where usually stress or load tests are run. It also allows the DevOps engineer to test out his deployment automation scripts, cron jobs, and verify system configuration.

    - Production is, of course, the final tier where software that is tested from staging is pushed and deployed. A number of deployments often use identical staging/production tiers, and simply switch from one to the other.

- **Development and Test/Stage/Production**: This is a variation of the previous tier, where the development environment also performs the double duty of a testing environment. This system is used in companies with agile software development practices, where code is pushed at least once a week to production, and there is no space or time to keep and manage a separate testing environment. When there is no separate development environment – that is when developers use their laptops for programming – the testing environment is also a local one.
- **Development and Test/ Stage and Production**: In this setup, staging and production environments are exactly the same with multiple servers used. Once a system is tested and verified in staging, it is *pushed* to production by simply switching the hosts—the current production system switches to staging, and staging switches to production.

Apart from these, it is possible to have more elaborate architectures where a separate **Integration** environment is used for integration testing, a **Sandbox** environment for testing experimental features, and so on.

Using a staging system is important to ensure that software is well tested and orchestrated in a production-like environment, before pushing the code to production.

Software deployment in Python

As mentioned earlier, Python developers are richly blessed in the various tools offered by Python, and its third-party ecosystem in easing and automating the deployment of applications and code written using Python.

In this section, we will briefly take a look at some of these tools.

Packaging Python code

Python comes with built in support for packaging applications for a variety of distributions—source, binary, and specific OS-level packaging.

The primary way of packaging source code in Python is to write a `setup.py` file. The source can then be packaged with the help of the in-built `distutils` library, or the more sophisticated and rich `setuptools` framework.

Before we get introduced to the guts of Python packaging, let us get familiar with a couple of closely related tools, namely, `pip` and `virtualenv`.

PIP

PIP stands for the recursive acronym **PIP installs packages**. Pip is the standard and suggested tool to install packages in Python.

We've seen PIP in action throughout this book, but so far, we've never seen pip itself getting installed, have we?

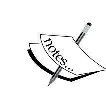
Let's see this in the following screenshot:

```
Chap 9 - Deployment
File  Edit  View  Search  Terminal  Help
$ wget https://bootstrap.pypa.io/get-pip.py
--2017-01-29 20:20:38--  https://bootstrap.pypa.io/get-pip.py
Resolving bootstrap.pypa.io (bootstrap.pypa.io)... 151.101.120.175
Connecting to bootstrap.pypa.io (bootstrap.pypa.io)|151.101.120.175|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1595408 (1.5M) [text/x-python]
Saving to: 'get-pip.py'

get-pip.py           100%[=========================>]   1.52M   642KB/s    in 2.4s

2017-01-29 20:20:41 (642 KB/s) - 'get-pip.py' saved [1595408/1595408]

$ sudo python3 get-pip.py
The directory '/home/anand/.cache/pip/http' or its parent directory is not owned by the
current user and the cache has been disabled. Please check the permissions and owner of
that directory. If executing pip with sudo, you may want sudo's -H flag.
The directory '/home/anand/.cache/pip' or its parent directory is not owned by the curre
nt user and caching wheels has been disabled. check the permissions and owner of that di
rectory. If executing pip with sudo, you may want sudo's -H flag.
Collecting pip
  Downloading pip-9.0.1-py2.py3-none-any.whl (1.3MB)
    100% |████████████████████████████████| 1.3MB 646kB/s
Installing collected packages: pip
  Found existing installation: pip 8.1.1
    Uninstalling pip-8.1.1:
      Successfully uninstalled pip-8.1.1
Successfully installed pip-9.0.1
$
```

Downloading and installing pip for Python3

The `pip` installation script is available at `https://bootstrap.pypa.io/get-pip.py`.

The steps should be self-explanatory.



In the preceding example, there was already a pip version, so the action upgraded the existing version instead of doing a fresh install. We can see the version details by trying the program with the –`version` option, as follows:

Take a look at the following screenshot:

```
Chap 9 - Deployment
File Edit View Search Terminal Help
$ pip3 --version
pip 9.0.1 from /usr/local/lib/python3.5/dist-packages (python 3.5)
$
```

Printing the current version of pip (pip3)

See how `pip` clearly prints its version number along with the directory location of the installation, plus the Python version for which it is installed.

To distinguish between pip for the Python2 and Python3 versions, remember that the version installed for Python3 is always named `pip3`. The Python2 version is `pip2` , or just `pip`.

To install a package using PIP, simply provide the package name via the command `install`. For example, the following screenshot shows installing the `numpy` package using `pip`:

```
Chap 9 - Deployment
File Edit View Search Terminal Help
$ sudo -H pip3 install numpy
Collecting numpy
  Downloading numpy-1.12.0-cp35-cp35m-manylinux1_x86_64.whl (16.8MB)
    100% |████████████████████████████████| 16.8MB 38kB/s
Installing collected packages: numpy
Successfully installed numpy-1.12.0
$
```

We will not go into further details of using pip here. Instead, let's take a look at another tool that works closely with pip in installing the Python software.

Virtualenv

Virtualenv is a tool that allows developers to create sand-boxed Python environments for local development. Let's say that you want to maintain two different versions of a particular library or framework for two different applications you are developing side by side.

If you are going to install everything to the system Python, then you can keep only one version at a given time. The other option is to create different system Python installations in different root folders—say, `/opt` instead of `/usr`. However, this creates additional overhead and management headaches of paths. Also, it wouldn't be possible to get write permission to these folders if you want the version dependency to be maintained on a shared host where you don't have superuser permissions.

Virtualenv solves the problems of permissions and versions in one go. It creates a local installation directory with its own Python executable standard library and installer (defaults to pip).

Once the developer has activated the virtual environment thus created, any further installations goes to this environment instead of the system Python environment.

Virtualenv can be installed using pip.

The following screenshot shows creating a virtualenv named `appvenv` using the `virtualenv` command, and activating the environment along with installing a package to the environment.

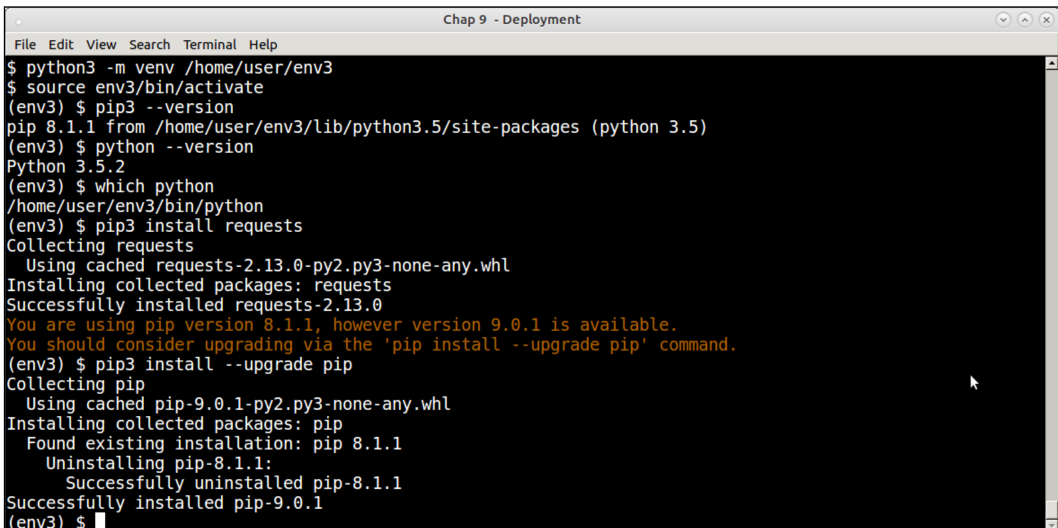
The installation also installs PIP, setuptools, and other dependencies.

```
$ virtualenv appenv
Running virtualenv with interpreter /usr/bin/python2
New python executable in /home/user/appenv/bin/python2
Also creating executable in /home/user/appenv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
$ source appenv/bin/activate
(appenv) $ which python
/home/user/appenv/bin/python
(appenv) $ which pip
/home/user/appenv/bin/pip
(appenv) $ pip install numpy
Collecting numpy
  Downloading numpy-1.12.0-cp27-cp27mu-manylinux1_x86_64.whl (16.5MB)
    100% |████████████████████████████████| 16.5MB 24kB/s
Installing collected packages: numpy
Successfully installed numpy-1.12.0
(appenv) $ pip --version
pip 9.0.1 from /home/user/appenv/local/lib/python2.7/site-packages (python 2.7)
(appenv) $ python --version
Python 2.7.12
(appenv) $
```

See how the `python` and `pip` commands point to the ones inside the virtual environment. The `pip -version` command clearly shows the path of `pip` inside the virtual environment folder.

From Python 3.3 onwards, support for virtual environments is built into the Python installation via the new `venv` library.

The following screenshot shows installing a virtual environment in Python 3.5 using this library, and installing some packages into it. As usual, take a look at Python and pip executable paths:



The preceding screenshot also shows how to upgrade pip itself via the `pip` command.

Virtualenv and pip

Once you've set up a virtual environment for your application(s) and installed the required packages, it is a good idea to generate the dependencies and their versions. This can be easily done via the following command using pip:

```
$ pip freeze
```

This command asks pip to output a list of all the installed Python packages along with their versions. This can be saved to a requirements file, and the setup duplicated on the server for mirroring deployments:

```
Chap 9 - Deployment
File Edit View Search Terminal Help
(appenv) $ pip freeze | tee requirements.txt
appdirs==1.4.0
backports-abc==0.5
certifi==2017.1.23
numpy==1.12.0
packaging==16.8
pkg-resources==0.0.0
pyparsing==2.1.10
requests==2.13.0
singledispatch==3.4.0.3
six==1.10.0
tornado==4.4.2
(appenv) $
```

The following screenshot shows recreating the same setup in another virtual environment via the `-r` option of the `pip install` command, which accepts such a file as input:

```
Chap 9 - Deployment
File Edit View Search Terminal Help
(env3) $ pip3 install -r requirements.txt
Collecting appdirs==1.4.0 (from -r requirements.txt (line 1))
  Using cached appdirs-1.4.0-py2.py3-none-any.whl
Collecting backports-abc==0.5 (from -r requirements.txt (line 2))
  Using cached backports_abc-0.5-py2.py3-none-any.whl
Collecting certifi==2017.1.23 (from -r requirements.txt (line 3))
  Using cached certifi-2017.1.23-py2.py3-none-any.whl
Collecting numpy==1.12.0 (from -r requirements.txt (line 4))
  Downloading numpy-1.12.0-cp35-cp35m-manylinux1_x86_64.whl (16.8MB)
    100% |████████████████████████████████| 16.8MB 25kB/s
Collecting packaging==16.8 (from -r requirements.txt (line 5))
  Using cached packaging-16.8-py2.py3-none-any.whl
Requirement already satisfied: pkg-resources==0.0.0 in ./env3/lib/python3.5/site-packages (from -r requir
ements.txt (line 6))
Collecting pyparsing==2.1.10 (from -r requirements.txt (line 7))
  Using cached pyparsing-2.1.10-py2.py3-none-any.whl
Requirement already satisfied: requests==2.13.0 in ./env3/lib/python3.5/site-packages (from -r requiremen
ts.txt (line 8))
Collecting singledispatch==3.4.0.3 (from -r requirements.txt (line 9))
  Using cached singledispatch-3.4.0.3-py2.py3-none-any.whl
Collecting six==1.10.0 (from -r requirements.txt (line 10))
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting tornado==4.4.2 (from -r requirements.txt (line 11))
  Using cached tornado-4.4.2.tar.gz
Installing collected packages: appdirs, backports-abc, certifi, numpy, pyparsing, six, packaging, singled
ispatch, tornado
  Running setup.py install for tornado ... done
Successfully installed appdirs-1.4.0 backports-abc-0.5 certifi-2017.1.23 numpy-1.12.0 packaging-16.8 pypa
rsing-2.1.10 singledispatch-3.4.0.3 six-1.10.0 tornado-4.4.2
(env3) $
```

Our source virtual environment was in Python2, and the target was in Python3. However, pip was able to install the dependencies from the `requirements.txt` file without any issues whatsoever.

Relocatable virtual environments

The suggested way to copy package dependencies from one virtual environment to another is to perform a freeze, and install via `pip` as illustrated in the previous section. For example, this is the most common way to freeze Python package requirements from a development environment, and recreate it successfully on a production server.

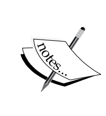
One can also try and make a virtual environment relocatable so that it can be archived and moved to a compatible system:

```
Chap 9 - Deployment
File  Edit  View  Search  Terminal  Help
$ virtualenv lenv
Running virtualenv with interpreter /usr/bin/python2
New python executable in /home/user/lenv/bin/python2
Also creating executable in /home/user/lenv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
$ virtualenv --relocatable lenv
Running virtualenv with interpreter /usr/bin/python2
Making script /home/user/lenv/bin/python-config relative
Making script /home/user/lenv/bin/pip2.7 relative
Making script /home/user/lenv/bin/easy_install relative
Making script /home/user/lenv/bin/pip relative
Making script /home/user/lenv/bin/wheel relative
Making script /home/user/lenv/bin/pip2 relative
Making script /home/user/lenv/bin/easy_install-2.7 relative
$ cd lenv
$ ls
bin  include  lib  local  pip-selfcheck.json  share
$ source bin/activate
```

Creating a relocatable virtual environment

Here is how it works:

1. First, the virtual environment is created as usual.
2. It is then made relocatable by running `virtualenv -relocatable lenv` on it.
3. This changes some of the paths used by setuptools as relative paths, and sets up the system to be relocatable.
4. Such a virtual environment is relocatable to another folder in the same machine, or to a folder in a *remote and similar machine*.



A relocatable virtual environment doesn't guarantee that it will work if the remote environment differs from the machine environment. For example, if your remote machine is a different architecture, or even uses a different Linux distribution with another type of packaging, the relocation will fail to work. This is what is meant by the words *similar machine*.

PyPI

We learned that PIP is the standardized tool to do package installations in Python. It is able to pick up any package by name as long as it exists. It is also able to install packages by version, as we saw with the example of the requirements file.

But where does PIP fetch its packages from?

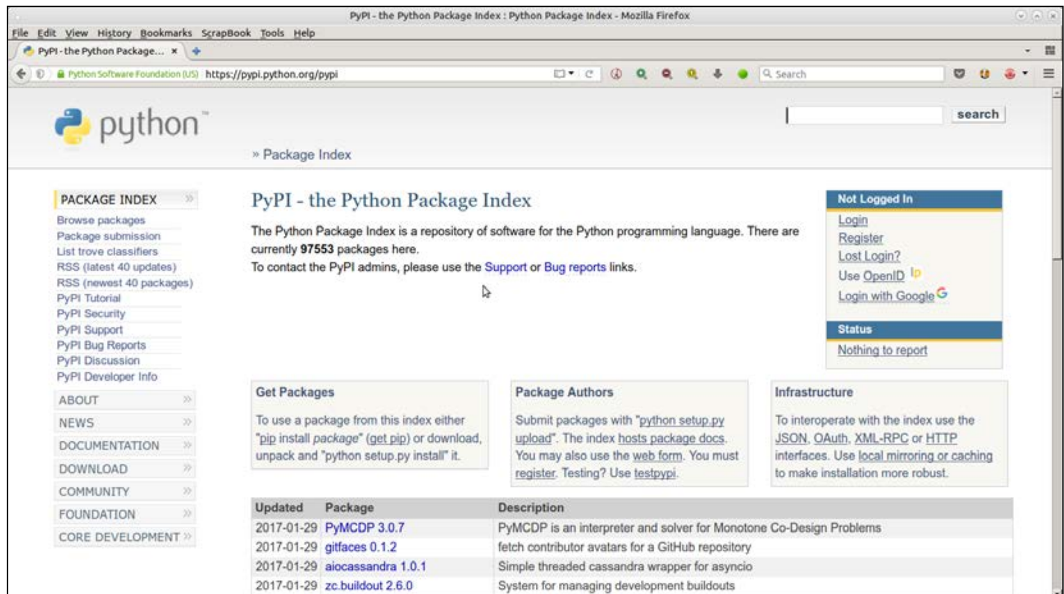
To answer this, we turn to the Python Package Index, more commonly known as PyPI.

Python Package Index (PyPI) is the official repository for hosting metadata for third-party Python packages on the Web. As the name implies, it is an index to the Python packages on the Web whose metadata is published and indexed on a server. PyPI is hosted at the URL `http://pypi.python.org`.

PyPI hosts close to a million packages at present. The packages are submitted to PyPI using Python's packaging and distribution tools, distutils, and setuptools, which have hooks for publishing package metadata to PyPI. A number of packages also host the actual package data in PyPI, although PyPI can be used to point to package data sitting in a URL on another server.

When you install a package using pip, it actually performs the search for the package on PyPI, and downloads the metadata. It uses the metadata to find out the package's download URL and other information, such as further downstream dependencies, which it uses to fetch and install the package for you.

Here is a screenshot of PyPI, which shows the actual count of the packages at the time of writing this:



A developer can do quite a few things directly on the PyPI site:

1. Register using e-mail address and log in to the site.
2. After logging in, submit your package directly on the site.
3. Search for packages via keywords.
4. Browse for packages via a number of top-level *trove* classifiers, such as Topics, Platforms/Operating Systems, Development Status, Licenses, and so on.

Now that we are familiar with the suite of all Python packaging and installation tools and their relationships, let us try out a small example of packaging a trivial Python module and submitting it to PyPI.

Packaging and submission of an application

Remember that we had developed a mandelbrot program, which uses pymp to scale, in *Chapter 5*, *Writing Applications that Scale*. We will use it as an example of a program to develop a package, and a `setup.py` file, which we will use to submit the application to PyPI.

We will package the mandelbrot application in a main package consisting of two sub-packages as follows:

- `mandelbrot.simple`: The sub-package (sub-module) consisting of the basic implementation of mandelbrot
- `mandelbrot`.mp: The sub package (sub-module) having the PyMP implementation of mandelbrot

Here is our folder structure for the package:

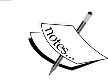
```
Chap 9: Deployment
File  Edit  View  Search  Terminal  Help
$ pwd
/home/user/programs/chap9/mandelbrot
$ tree .
.
├── mandelbrot
│   ├── __init__.py
│   ├── mp
│   │   ├── __init__.py
│   │   └── mandelbrot.py
│   └── simple
│       ├── __init__.py
│       └── mandelbrot.py
├── README
└── setup.py

3 directories, 7 files
$
```

Folder layout of the mandelbrot package

Let us quickly analyze the folder structure of the application which we will be packaging:

- The top directory is named `mandelbrot`. It has an `__init__.py`, a `README`, and a `setup.py` file.
- This directory has two sub directories—`mp` and `simple`.

- Each of these subfolders consists of two files, namely, `__init__.py` and `mandelbrot.py`. These subfolders will form our sub-modules, each containing the respective implementation of the mandelbrot set.



For the purpose of installing the mandelbrot modules as executable scripts, the code has been changed to add a `main` method to each of our `mandelbrot.py` modules.

The __init__.py files

The `__init__.py` files allow to convert a folder inside a Python application as a package. Our folder structure has three of them: the first one is for the top-level package `mandelbrot`, and the rest two for each of the sub-packages, namely, `mandelbrot.simple` and `mandelbrot.mp`.

The top-level `__init__.py` is empty. The other two have the following single line:

```
from . import mandelbrot
```

The relative imports are to make sure that the sub-packages are importing the local `mandelbrot.py` module instead of the top-level `mandelbrot` package.

The setup.py file

The `setup.py` file is the central point of the entire package. Let us take a look at it:

```
from setuptools import setup, find_packages
setup(
    name = "mandelbrot",
    version = "0.1",
    author = "Anand B Pillai",
    author_email = "abpillai@gmail.com",
    description = ("A program for generating Mandelbrot fractal
images"),
    license = "BSD",
    keywords = "fractal mandelbrot example chaos",
    url = "http://packages.python.org/mandelbrot",
    packages = find_packages(),
    long_description=open('README').read(),
    classifiers=[
        "Development Status :: 4 - Beta",
        "Topic :: Scientific/Engineering :: Visualization",
```

```
            "License :: OSI Approved :: BSD License",
        ],
        install_requires = [
            'Pillow>=3.1.2',
            'pymp-pypi>=0.3.1'
            ],
        entry_points = {
            'console_scripts': [
                'mandelbrot = mandelbrot.simple.mandelbrot:main',
                'mandelbrot_mp = mandelbrot.mp.mandelbrot:main'
                ]
            }
)
```

A full discussion of the `setup.py` file is outside the scope of this chapter, but do note these few key points:

- The `setup.py` file allows the author to create a lot of package metadata such as name, author name, e-mail, package keywords, and others. These are useful in creating the package meta information, which helps people to search for the package in PyPI once it's submitted.
- One of the main fields in this file is packages, which is the list of packages (and sub-packages) that is created by this `setup.py` file. We make use of the `find_packages` helper function provided by the setuptools module to do this.
- We provide the installment requirements in the `install-requires` key, which lists the dependencies one by one in a PIP-like format.
- The `entry_points` key is used to configure the console scripts (executable programs) that this package installs. Let us look at one of them:

  ```
  mandelbrot = mandelbrot.simple.mandelbrot:main
  ```

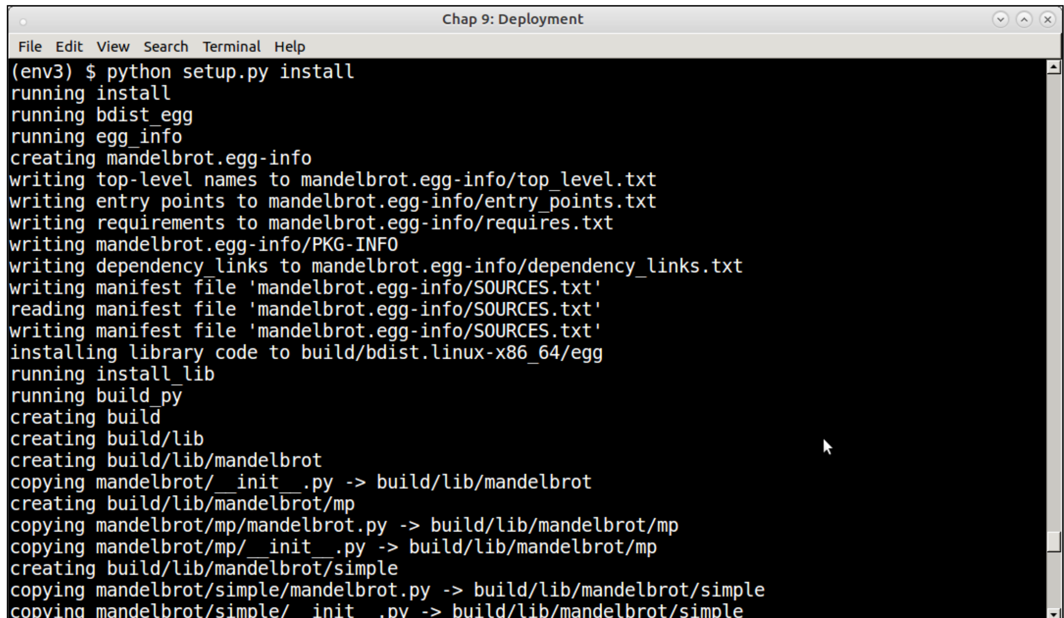
  This tells the package resource loader to load the module named `mandelbrot.simple.mandelbrot`, and execute its function `main` when the script `mandelbrot` is invoked.

Installing the package

The package can be now installed using this command:

```
$ python setup.py install
```

The following screenshot of the installation shows a few of the initial steps:



We have installed this package to a virtual environment named `env3`.

Submitting the package to PyPI

The `setup.py` file plus setuptools/distutils ecosystem in Python is useful, not just to install and package code, but also to submit code to the Python package index.

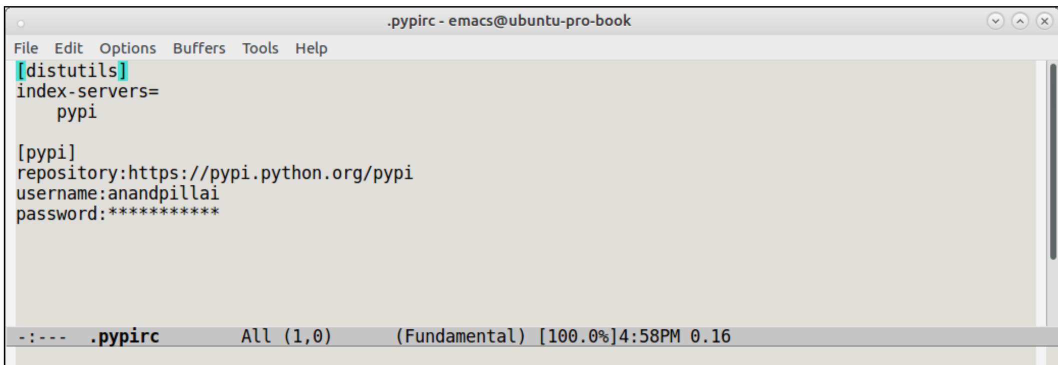
It is very easy to register your package to PyPI. There are just the following two requirements:

- A package with a proper setup.py file
- An account on the PyPI website

We will now submit our new mandelbrot package to PyPI by performing the following steps:

1. First, one needs to create a `.pypirc` file in one's home directory containing some details—mainly the authentication details for the PyPI account.

   Here is the author's `.pypirc` file with the password obscured:



2. Once this is done, registration is as simple as running `setup.py` with the `register` command:

   $ python setup.py register

   The next screenshot shows the actual command in action on the console:

```
Chap 9: Deployment
File Edit View Search Terminal Help
(env3) $ python setup.py register
running register
running egg_info
writing entry points to mandelbrot.egg-info/entry_points.txt
writing top-level names to mandelbrot.egg-info/top_level.txt
writing mandelbrot.egg-info/PKG-INFO
writing requirements to mandelbrot.egg-info/requires.txt
writing dependency_links to mandelbrot.egg-info/dependency_links.txt
reading manifest file 'mandelbrot.egg-info/SOURCES.txt'
writing manifest file 'mandelbrot.egg-info/SOURCES.txt'
running check
Registering mandelbrot to https://pypi.python.org/pypi
Server response (200): OK
(env3) $
```

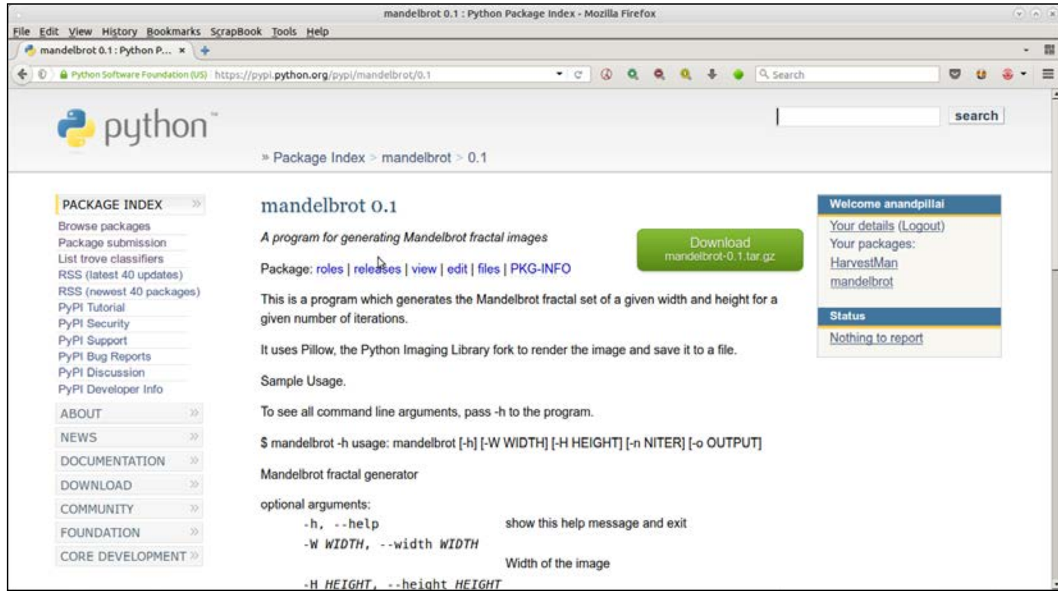
   However, this last step has only registered the package by submitting its metadata. No package data, as in the source code data, has been submitted as part of this step.

3. To submit the source code also to PyPI, the following command should be run:

   $ python setup.py sdist upload

```
Chap 9: Deployment
File  Edit  View  Search  Terminal  Help
(env3) $ python setup.py sdist upload
running sdist
running egg_info
writing mandelbrot.egg-info/PKG-INFO
writing dependency_links to mandelbrot.egg-info/dependency_links.txt
writing entry points to mandelbrot.egg-info/entry_points.txt
writing top-level names to mandelbrot.egg-info/top_level.txt
writing requirements to mandelbrot.egg-info/requires.txt
reading manifest file 'mandelbrot.egg-info/SOURCES.txt'
writing manifest file 'mandelbrot.egg-info/SOURCES.txt'
running check
creating mandelbrot-0.1
creating mandelbrot-0.1/mandelbrot
creating mandelbrot-0.1/mandelbrot.egg-info
creating mandelbrot-0.1/mandelbrot/mp
creating mandelbrot-0.1/mandelbrot/simple
making hard links in mandelbrot-0.1...
hard linking README -> mandelbrot-0.1
hard linking setup.py -> mandelbrot-0.1
hard linking mandelbrot/__init__.py -> mandelbrot-0.1/mandelbrot
hard linking mandelbrot.egg-info/PKG-INFO -> mandelbrot-0.1/mandelbrot.egg-info
hard linking mandelbrot.egg-info/SOURCES.txt -> mandelbrot-0.1/mandelbrot.egg-info
```

Here's a view of our new package on the PyPI server:



Now the package is installable via pip, completing the cycle of software development: that is, first packaging, then deployment, and finally, installation.

PyPA

Python Packaging Authority (**PyPA**) is a working group of Python developers who maintain the standards and the relevant applications related to packaging in Python.

PyPA has their website at `https://www.pypa.io/`, and they maintain the application on GitHub at `https://github.com/pypa/`.

The following table lists the projects that are maintained by PyPA. You've already seen some of these, such as pip, virtualenv, and setuptools; others may be new:

Project	**Description**
`setuptools`	A collection of enhancements to Python distutils
`virtualenv`	A tool for creating sandbox Python environments
`pip`	A tool for installing Python packages
`packaging`	Core Python utilities for packaging used by pip and setuptools
`wheel`	An extension to setuptools for creating wheel distributions, which are an alternative to Python eggs (ZIP files) and specified in PEP 427
`twine`	A secure replacement for `setup.py` upload
`warehouse`	The new PyPI application, which can be seen at `https://pypi.org`
`distlib`	A low-level library implementing functions relating to packaging and distribution of Python code
`bandersnatch`	A PyPI mirroring client to mirror the contents of PyPI

Interested developers can go visit the PyPA site and sign up for one of the projects - and contribute to them in terms of testing, submitting patches and so on by visiting the github repository of PyPA.

Remote deployments using Fabric

Fabric is a command-line tool and library written in Python, which helps to automate remote deployments on servers via a set of well-defined wrappers over the SSH protocol. It uses the `ssh-wrapper` library, `paramiko`, behind the scenes.

Fabric works with Python 2.x versions only. However, there is a fork Fabric3 which works for both the Python 2.x and 3.x versions.

When using fabric, a DevOps user usually deploys his remote system administrator commands as Python functions in a `fabfile` named as `fabfile.py`.

Fabric works best when the remote systems are already configured with the ssh public keys of the user's machine from where he performs deployments, so there is no need to supply a username and password.

Here is an example of remote deployment on a server. In this case, we are installing our mandelbrot application on a remote server.

The fabfile looks as follows. See that it is written for Python3:

```
from fabric.api import run

def remote_install(application):

    print ('Installing',application)
    run('sudo pip install ' + application)
```

Here is an example of running this, installing it on a remote server:

```
Chap 9: Deployment
File Edit View Search Terminal Help
(env3) $ fab remote_install:mandelbrot -H mylinode
[mylinode] Executing task 'remote_install'
Installing mandelbrot
[mylinode] run: sudo pip install mandelbrot
[mylinode] out: Downloading/unpacking mandelbrot
[mylinode] out:   Downloading mandelbrot-0.1.tar.gz
[mylinode] out:   Running setup.py (path:/tmp/pip-build-3RuBh1/mandelbrot/setup.py) egg_info for package m
[mylinode] out: Requirement already satisfied (use --upgrade to upgrade): Pillow>=3.1.2 in /usr/local/lib/
dist-packages (from mandelbrot)
[mylinode] out: Requirement already satisfied (use --upgrade to upgrade): pymp-pypi>=0.3.1 in /usr/local/l
.7/dist-packages (from mandelbrot)
[mylinode] out: Requirement already satisfied (use --upgrade to upgrade): olefile in /usr/local/lib/python
ackages (from Pillow>=3.1.2->mandelbrot)
[mylinode] out: Installing collected packages: mandelbrot
[mylinode] out:   Running setup.py install for mandelbrot
[mylinode] out:     Installing mandelbrot_mp script to /usr/local/bin
[mylinode] out:     Installing mandelbrot script to /usr/local/bin
[mylinode] out:   Could not find .egg-info directory in install record for mandelbrot
[mylinode] out: Successfully installed mandelbrot
[mylinode] out: Cleaning up...
[mylinode] out:


Done.
Disconnecting from mylinode... done.
(env3) $
```

DevOps engineers and system administrators can use a predefined set of fabfiles for automating different system and application deployment tasks across multiple servers.

Though it is written in Python, Fabric can be used to automate deployment of any kind of remote server administration and configuration tasks.

Remote deployments using Ansible

Ansible is a configuration management and deployment tool written in Python. Ansible can be thought of as a wrapper over SSH with scripts with support for orchestration via tasks which can be assembled in easy-to-manage units called *playbooks* which map a group of hosts to a set of roles.

Ansible uses "facts" which are system and environment information it gathers before it runs tasks. It uses the facts to check if there is any need to change any state before running a task to get the desired outcome.

This makes it safe for Ansible tasks to be run on a server in a repeated fashion. Well-written Ansible tasks are *idempotent* in that they have zero to few side effects on the remote system.

Ansible is written in Python and can be installed using `pip`.

It uses its own hosts file, namely `/etc/ansible/hosts` to keep the host information against which it runs its tasks.

A typical ansible host file may look as follows:

```
[local]
127.0.0.1

[webkaffe]
139.162.58.8
```

The following is a snippet from an Ansible playbook named `dependencies.yaml` which installs a few Python packages via `pip` on a remote host named `webkaffeStyle`:

```
---
- hosts: webkaffe
  tasks:
    - name: Pip - Install Python Dependencies
      pip:
          name="{{ python_packages_to_install | join(' ') }}"

      vars:
          python_packages_to_install:
          - Flask
          - Bottle
          - bokeh
```

Here is an image of running this playbook on the command line using ansible-playbook:

```
(env) $ ansible-playbook --user=webkaffe -s dependencies.yaml

PLAY [webkaffe] ***************************************************************************

TASK [Gathering Facts] ********************************************************************
ok: [139.162.58.8]

TASK [Pip - Install Python Dependencies] **************************************************
ok: [139.162.58.8]

PLAY RECAP ********************************************************************************
139.162.58.8               : ok=2    changed=0    unreachable=0    failed=0

(env) $
```

Ansible is an easy and efficient way of managing remote dependencies and, due to its idempotent playbooks, is much better than Fabric at the task.

Managing remote daemons using Supervisor

Supervisor is a client/server system, which is useful to control processes on Unix and Unix-like systems. It consists mainly of a server daemon process named **supervisord** and a command-line client, which interacts with the server named **supervisorctl**.

Supervisor also comes with a basic webserver, which can be accessed via port 9001. It is possible to view the state of running processes, and also to start/stop them via this interface. Supervisor doesn't run on any version of Windows.

Supervisor is an application written using Python, and hence, is installable via pip. It runs only on Python 2.x versions.

Applications to be managed via supervisor should be configured via the supervisor daemons configuration file. By default, such files sit in the `/etc/supervisor.d/conf` folder.

However, it is possible to run Supervisor locally by installing it to a virtual environment, and keeping the configuration local to the virtual environment. In fact, this is a common way to run multiple supervisor daemons, each managing processes specific to the virtual environment.

We won't go into details or examples of using Supervisor, but here are some benefits of using Supervisor vs a traditional approach like system `rc.d` scripts:

- Decoupling process creation/management and process control can be achieved by using a client/server system. The `supervisor.d` file manages the processes via subprocesses. The user can get the process state information via supervisorctl, the client. Also, whereas most traditional rc.d processes require root or sudo access, supervisor processes can be controlled by normal users of the system via the client or through the web UI.
- Since supervisord starts processes via subprocesses, they can be configured to automatically restart upon crash. It is also easier to get a more accurate status of the subprocesses rather than relying on PID files.
- Supervisor supports process groups allowing users to define processes in a priority order. Processes can be started and stopped in a specific order as a group. This allows implementation of a fine-grained process control when there is a temporal dependency between creation of processes in an application. (Process B requires A to be running, C requires B to be running, and the like.)

We will complete the discussion in this chapter with an overview of the common deployment patterns, which an architect can choose from to solve common issues with deployability.

Deployment – patterns and best practices

There are different deployment approaches or patterns that can be used to address issues like down-times, reduce risks with deployment, and for a seamless development and deployment of software.

- **Continuous deployment**: Continuous deployment is a deployment model where software is ready to go live at any time. Continuous delivery is possible only if tiers, including development, testing, and staging, are integrated continuously. In a continuous deployment model, multiple production deployments can occur in a day, and automatically, via a deployment pipeline. Since one is constantly deploying incremental changes, the continuous deployment mode minimizes deployment risks. In agile software development houses, it also helps the customer to track progress directly by seeing live code in production almost as soon as it leaves development and testing. There is also the added advantage of getting user feedback faster allowing faster iterations to the code and features.

- **BlueGreen deployment**: We already discussed this in *Chapter 5*, *Writing Applications that Scale*. Blue green deployments keep two production environments, closely identical to each other. At a given instance, one environment is live (Blue). You prepare your new deployment changes to the other environment (Green), and, once tested and ready to go live, switch your systems—Green becomes active and Blue becomes the backup. BlueGreen deployments reduce deployment risks considerably, since, for anything that goes wrong with the new deployment, you just need to switch your router or load-balancer to the new environment. Usually, in typical BlueGreen systems, one system is the production (live) and other the staging, and you switch the roles between them.
- **Canary releases**: If you want to test the changes in your software on a subset of users before deploying it for the entire audience of your customers, you can use this approach. In canary release, the changes are rolled out to a small subset of users first. A simple approach is dogfooding, where the changes are rolled out internally to the employees first. Another approach is beta-testing, where a select group of audience is invited to test out your early features. Other involved approaches include selecting users based on their geographic location, demographics, and profiles. Canary releases, apart from insulating the company from sudden user reaction to badly managed features, also allow you to manage load and capacity scaling in an incremental way. For example, if a particular feature becomes popular, and starts driving, say, 100X users to your servers than before, a traditional deployment may cause server failures and availability issues as opposed to a gradual deployment using a Canary release. Geographical routing is a technique that can be used to select a subset of users if you don't want to do complex user profiling and analysis. This is where the load is sent more to nodes deployed in a particular geography or data center as opposed to other nodes. Canary release is also related to the concept of incremental rollout or phased rollout.
- **Bucket testing (A/B testing)**: This is the technique of deploying two dissimilar versions of an application or a webpage to production to test out which version is more popular and/or has more engagement. In production, a subset of your audience sees the A version of the app (or page)—the control or basic version—and the other subset sees the B version or the modified (variant) version. Usually, this is a 50-50 split, though, as with Canary releases, user profiles, geo locations, or other complex models can be used. User experience and engagement is collected using an analytics dashboard, and then it is determined whether the change had a positive, negative, or neutral response.
- **Induced chaos**: This is a technique of purposely introducing errors or disabling part of a production deployment system to test its resilience to failures and/or level of availability.

Production servers have the problem of drift—unless you use continuous deployment or similar approaches for sync, production servers, usually, tend to drift away from the standard configuration. One way to test your system is to intentionally disable part of the production system—this can be done, for example, by disabling a random 50% of the nodes in a load-balancer configuration, and see how the rest of the system performs.

A similar approach in finding out and weeding unused parts of code is to inject random secrets in parts of the configuration using, say, an API that you suspect is redundant and no longer required. You then observe how the application performs in production. Since a random secret will fail the API, if there is an active part of the application which still uses the dependent code, it will fail in production. Otherwise, it is an indication that the code can be safely removed.

Netflix has a tool called **Chaos Monkey**, which automatically injects failures in production systems, and then measures the impact.

Induced Chaos allows the DevOps engineer and architect to understand weak points in the system, learn about systems which are undergoing configuration drift, and find and weed out unnecessary or unused parts of an application.

Summary

This chapter was about deploying your Python code to production. We looked at the different factors that affect the deployability of a system. We went on to discuss the tiers in deployment architecture, such as the traditional four-tiered and the three- and two-tiered architectures, including combinations of development, testing, staging/QA, and production tiers.

We then went on to discuss the details of packaging Python code. We discussed the tools of PIP and virtualenv in detail. We looked at how pip and virtualenv can work together, and how to install a set of requirements using pip, and set up similar virtual environments using it. We also took a quick look at relocatable virtual environments.

We then went to discuss PyPI—the Python Package Index which hosts Python third-party packages on the web. We then went through a detailed example of setting up a Python package using setuptools and the `setup.py` file. We used the mandelbrot application as an example in this case.

We ended that discussion by showing how to register the package to PyPI using its metadata, and also how to upload the package data including its code. We also took a brief look at PyPA, the Python Packaging Authority and their projects.

After that, two tools, both developed in Python, were discussed—Fabric for remote automated deployments, and Supervisor for remote management of processes on Unix systems. We finished the chapter with an overview of the common deployment patterns that one can use to solve deployment problems.

In the final chapter of this book, we talk about a variety of techniques of Debugging your code to identify potential issues.

10
Techniques for Debugging

Debugging a program can often be as hard or, sometimes, even more difficult than writing it. Quite often, programmers seem to spend an awful amount of time hunting for that elusive bug, the reason for which may be staring them in the face, yet not revealing itself.

Many developers, even the good ones, find troubleshooting a difficult art. Most often, programmers resort to complicated debugging techniques when simple approaches such as properly placed print statements and strategically commented code would do the trick.

Python has its own set of problems when it comes to debugging code. Being a dynamically typed language, type-related exceptions, which happen due to the programmer assuming a type to be something (when it's something else), are pretty common in Python. Name errors and attribute errors fall in a similar category too.

In this chapter, we will exclusively focus on this lesser discussed aspect of software.

Here is a topic-wise listing of what we are going to encounter in this chapter:

- Maximum subarray problem:
    - The power of "print"
    - Analysis and rewrite
    - Timing and optimizing the code
- Simple debugging tricks and techniques:
    - Word searcher program
    - Word searcher program—debugging step 1
    - Word searcher program—debugging step 2
    - Word searcher program—final code

    - Skipping blocks of code
    - Stopping execution
    - External dependencies—using wrappers
    - Replacing functions with their return value/data (mocking)
    - Saving to/loading data from files as cache
    - Saving to/loading data from memory as cache
    - Returning random/mock data
    - Generating random patient data

- Logging as a debugging technique:
    - Simple application logging
    - Advanced logging—logger objects
    - Advanced logging—custom formatting and loggers
    - Advanced logging—writing to syslog

- Debugging tools—using debuggers:
    - A debugging session with pdb
    - Pdb—similar tools
    - iPdb
    - Pdb++
- Advanced debugging—tracing:
    - The `trace` module
    - The `iptrace` program
    - System call tracing using `strace`

Okay, so let's debug it!

Maximum subarray problem

For starters, let's look at an interesting problem. In this problem, the goal is to find the maximum contiguous subarray of an array (sequence) of integers having mixed negative and positive numbers.

For example, say we have the following array:

```
>>> a  = [-5, 20, -10, 30, 15]
```

It is pretty obvious with a quick scan that the maximum sum is for the subarray `[20, -10, 30, 15]`, giving a sum of `55`.

Let's say, as a first cut, you write this piece of code:

```
import itertools

# max_subarray: v1
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    sums = []

    for i in range(len(sequence)):
        # Create all sub-sequences in given size
        for sub_seq in itertools.combinations(sequence, i):
            # Append sum
            sums.append(sum(sub_seq))

    return max(sums)
```

Now let's try it out:

```
>>>  max_subarray([-5, 20, -10, 30, 15])
65
```

This output seems clearly wrong, as any manual addition of any subarray in the array doesn't seem to yield a number more than 55. We need to debug the code.

The power of "print"

In order to debug the preceding example, a simple, strategically placed `print` statement does the trick. Let's print out the subsequences in the inner `for` loop:

The function is modified as follows:

max_subarray: v1

```
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    sums = []
    for i in range(len(sequence)):
        for sub_seq in itertools.combinations(sequence, i):
            sub_seq_sum = sum(sub_seq)
            print(sub_seq,'=>',sub_seq_sum)
            sums.append(sub_seq_sum)

    return max(sums)
```

Now the code executes and prints this output:

```
>>> max_subarray([-5, 20, -10, 30, 15])
((), '=>', 0)
((-5,), '=>', -5)
((20,), '=>', 20)
((-10,), '=>', -10)
((30,), '=>', 30)
((15,), '=>', 15)
((-5, 20), '=>', 15)
((-5, -10), '=>', -15)
((-5, 30), '=>', 25)
((-5, 15), '=>', 10)
((20, -10), '=>', 10)
((20, 30), '=>', 50)
((20, 15), '=>', 35)
((-10, 30), '=>', 20)
((-10, 15), '=>', 5)
((30, 15), '=>', 45)
((-5, 20, -10), '=>', 5)
((-5, 20, 30), '=>', 45)
((-5, 20, 15), '=>', 30)
((-5, -10, 30), '=>', 15)
((-5, -10, 15), '=>', 0)
((-5, 30, 15), '=>', 40)
((20, -10, 30), '=>', 40)
((20, -10, 15), '=>', 25)
((20, 30, 15), '=>', 65)
((-10, 30, 15), '=>', 35)
((-5, 20, -10, 30), '=>', 35)
((-5, 20, -10, 15), '=>', 20)
((-5, 20, 30, 15), '=>', 60)
((-5, -10, 30, 15), '=>', 30)
((20, -10, 30, 15), '=>', 55)
65
```

The problem is clear now by looking at the output of the print statements.

There is a subarray `[20, 30, 15]` (highlighted in bold in the preceding output), which produces the sum `65`. However, this is *not a valid subarray*, as the elements are not contiguous in the original array.

Clearly, the program is wrong and needs a fix.

Analysis and rewrite

A quick analysis tells us that the use of `itertools.combinations` is the culprit here. We used it as a way to quickly generate all the subarrays of different lengths from the array, but using combinations *does not* respect the order of items, and generates *all* combinations producing subarrays that are not contiguous.

Clearly, we need to rewrite this. Here is a first attempt at the rewrite:

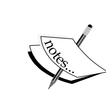
max_subarray: v2

```
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    sums = []

    for i in range(len(sequence)):
        for j in range(i+1, len(sequence)):
            sub_seq = sequence[i:j]
            sub_seq_sum = sum(sub_seq)
            print(sub_seq,'=>',sub_seq_sum)
            sums.append(sum(sub_seq))

    return max(sums)
```

Now the output is as follows:

```
>>> max_subarray([-5, 20, -10, 30, 15])
([-5], '=>', -5)
([-5, 20], '=>', 15)
([-5, 20, -10], '=>', 5)
([-5, 20, -10, 30], '=>', 35)
([20], '=>', 20)
([20, -10], '=>', 10)
([20, -10, 30], '=>', 40)
([-10], '=>', -10)
([-10, 30], '=>', 20)
([30], '=>', 30)
40
```

The answer is not correct again, as it gives the suboptimal answer `40`, not the correct one, which is, *55*. Again, the print statement comes to the rescue, as it tells us clearly that the main array itself is not being considered—we have an *off-by-one* bug.



An off-by-one or one-off error occurs in programming when an array index used to iterate over a sequence (array) is off either by *one less* or *one more* than the correct value. This is often found in languages where the index for sequences starts from zero, such as C/C++, Java, or Python.

In this case, the *off-by-one* error is in this line:

```
"sub_seq = sequence[i:j]"
```

The correct code should, instead, be as follows:

```
"sub_seq = sequence[i:j+1]"
```

With this fix, our code produces the output as expected:

max_subarray: v2

```
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    sums = []

    for i in range(len(sequence)):
        for j in range(i+1, len(sequence)):
            sub_seq = sequence[i:j+1]
            sub_seq_sum = sum(sub_seq)
          print(sub_seq,'=>',sub_seq_sum)
            sums.append(sub_seq_sum)

    return max(sums)
```

Here is the output:

```
>>> max_subarray([-5, 20, -10, 30, 15])
([-5, 20], '=>', 15)
([-5, 20, -10], '=>', 5)
([-5, 20, -10, 30], '=>', 35)
([-5, 20, -10, 30, 15], '=>', 50)
([20, -10], '=>', 10)
([20, -10, 30], '=>', 40)
([20, -10, 30, 15], '=>', 55)
```

```
([-10, 30], '=>', 20)
([-10, 30, 15], '=>', 35)
([30, 15], '=>', 45)
55
```

Let us assume at this point that you consider the code to be complete.

You pass the code on to a reviewer, and they mention that your code, though called `max_subarray`, actually forgets to return the subarray itself, instead returning only the sum. There is also the feedback that you don't need to maintain an array of sums.

You combine this feedback and produce a version 3.0 of the code, which fixes both the issues:

max_subarray: v3

```
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    # Trackers for max sum and max sub-array
    max_sum, max_sub = 0, []

    for i in range(len(sequence)):
        for j in range(i+1, len(sequence)):
            sub_seq = sequence[i:j+1]
            sum_s = sum(sub_seq)
            if sum_s > max_sum:
                # If current sum > max sum so far, replace the values
                max_sum, max_sub = sum_s, sub_seq

    return max_sum, max_sub

>>>  max_subarray([-5, 20, -10, 30, 15])
(55, [20, -10, 30, 15])
```

Note that we removed the print statement in this last version, as the logic was already correct, and so there was no need for debugging.

All good.

Timing and optimizing the code

If you analyze the code a bit, you'll find that the code performs two passes through the full sequence, one outer and one inner. So if the sequence contains *n* items, the code performs *n*n* passes.

We know from *Chapter 4*, *Good Performance is Rewarding!*, on performance that such a piece of code performs at the order of *O(n2)*. We can measure the real time spent on the code by using simple `context-manager` using the `with` operator.

Our context manager looks as follows:

```
import time
from contextlib import contextmanager

@contextmanager
def timer():
    """ Measure real-time execution of a block of code """

    try:
        start = time.time()
        yield
    finally:
        end = (time.time() - start)*1000
        print 'time taken=> %.2f ms' % end
```

Let's modify the code to create an array of random numbers of different sizes to measure the time taken. We will write a function for this:

```
import random

def num_array(size):
    """ Return a list of numbers in a fixed random range
    of given size """

    nums = []
    for i in range(size):
        nums.append(random.randrange(-25, 30))
    return nums
```

Let's time our logic for various sizes of arrays, beginning with 100:

```
>>> with timer():
... max_subarray(num_array(100))
... (121, [7, 10, -17, 3, 21, 26, -2, 5, 14, 2, -19, -18, 23, 12, 8,
     -12, -23, 28, -16, -19, -3, 14, 16, -25, 26, -16, 4, 12, -23, 26,
      22, 12, 23])
time taken=> 16.45 ms
```

For an array of 1,000, the code will be as follows:

```
>>> with timer():
... max_subarray(num_array(100))
... (121, [7, 10, -17, 3, 21, 26, -2, 5, 14, 2, -19, -18, 23, 12, 8,
     -12, -23, 28, -16, -19, -3, 14, 16, -25, 26, -16, 4, 12, -23, 26,
      22, 12, 23])
time taken=> 16.45 ms
```

So this takes about 3.3 seconds.

It can be shown that with an input size of 10,000, the code will take around 2 to 3 hours to run.

Is there a way to optimize the code? Yes, there is an *O(n)* version of the same code, which looks like this:

```
def max_subarray(sequence):
    """ Maximum subarray - optimized version """

    max_ending_here = max_so_far = 0

    for x in sequence:
        max_ending_here = max(0, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far
```

With this version, the time taken is much better:

```
>>> with timer():
... max_subarray(num_array(100))
... 240
time taken=> 0.77 ms
```

For an array of 1,000, the time taken is as follows:

```
>>> with timer():
... max_subarray(num_array(1000))
... 2272
time taken=> 6.05 ms
```

For an array of 10,000, the time is around 44 milliseconds:

```
>>> with timer():
... max_subarray(num_array(10000))
... 19362
time taken=> 43.89 ms
```

Simple debugging tricks and techniques

We saw the power of the simple `print` statement in the previous example. In a similar way, other simple techniques can be used to debug programs without requiring to resort to a debugger.

Debugging can be thought of as a step-wise process of exclusion until the programmer arrives at the truth—the cause of the bug. It essentially involves the following steps:

- Analyze the code and come up with a set of probable assumptions (causes) that may be the source of the bug.
- Test out each of the assumptions one by one by using appropriate debugging techniques.
- At every step of the test, you either arrive at the source of the bug—as the test succeeds telling you the problem was with the specific cause you were testing for; or the test fails and you move on to test the next assumption.
- You repeat the last step until you either arrive at the cause or you discard the current set of probable assumptions. Then you restart the entire cycle until you (hopefully) find the cause.

Word searcher program

In this section, we will look at some simple debugging techniques one by one using examples. We will start with the example of a word searcher program that looks for lines containing a specific word in a list of files—and appends and returns the lines in a list.

Here is the listing of the code for the word searcher program:

```
import os
import glob

def grep_word(word, filenames):
    """ Open the given files and look for a specific word.
    Append lines containing word to a list and
    return it """

    lines, words = [], []

    for filename in filenames:
        print('Processing',filename)
        lines += open(filename).readlines()


    word = word.lower()
    for line in lines:
        if word in line.lower():
            lines.append(line.strip())

    # Now sort the list according to length of lines
    return sorted(words, key=len)
```

You may have noticed a subtle bug in the preceding code—it appends to the wrong list. It reads from the list "lines," and appends to the same list, which will cause the list to grow forever; the program will go into an infinite loop when it encounters even a single line containing the given word.

Let's run the program on the current directory:

```
>>> parse_filename('lines', glob.glob('*.py'))
(hangs)
```

On any day, you may find this bug easily. On a bad day, you may be stuck on this for a while, not noticing that the same list being read from is being appended to.

Here are a few things that you can do:

- As the code is hanging and there are two loops, find out the loop that causes the problem. To do this, either put a print statement between the two loops, or put a `sys.exit` function, which will cause the interpreter to exit at that point.
- A print statement can be missed by a developer, especially if the code has many other print statements, but `sys.exit` can never be missed of course.

Word searcher program – debugging step 1

The code is rewritten as follows to insert a specific `sys.exit(…)` call between the two loops:

```
import os
import glob

def grep_word(word, filenames):
    """ Open the given files and look for a specific word.
    Append lines containing word to a list and
    return it """

    lines, words = [], []

    for filename in filenames:
        print('Processing',filename)
        lines += open(filename).readlines()

    sys.exit('Exiting after first loop')

    word = word.lower()
    for line in lines:
        if word in line.lower():
            lines.append(line.strip())

    # Now sort the list according to length of lines
    return sorted(words, key=len)
```

When trying it out a second time, we get this output:

```
>>> grep_word('lines', glob.glob('*.py'))
Exiting after first loop
```

Now it's pretty clear that the problem is not in the first loop. You can now proceed to debug the second loop (we are assuming that you are totally blind to the wrong variable usage, so you are figuring out the issue the hard way, by debugging).

Word searcher program – debugging step 2

Whenever you suspect a block of code inside a loop to be causing a bug, there are a few tricks to debug this, and confirm your suspicion. These include the following:

- Put a strategic continue just preceding the block of code. If the problem disappears, then you've confirmed that the specific block or any next block is the issue. You can continue to move down your `continue` statement until you identify the specific block of code that is causing the issue.
- Make Python skip the code block by prefixing it with `if 0:`. This is more useful if the block is a line of code or a few lines of code.
- If there is a lot of code inside a loop, and the loop executes many times, print statements may not help you much, as a ton of data will be printed, and it would be difficult to sift and scan through it and find out where the problem is.

In this case, we will use the first trick to figure out the issue. Here is the modified code:

```
def grep_word(word, filenames):
    """ Open the given files and look for a specific word.
    Append lines containing word to a list and
    return it """

    lines, words = [], []

    for filename in filenames:
        print('Processing',filename)
        lines += open(filename).readlines()


    # Debugging steps
    # 1. sys.exit
    # sys.exit('Exiting after first loop')

    word = word.lower()
    for line in lines:
        if word in line.lower():
```

```
            words.append(line.strip())
            continue

    # Now sort the list according to length of lines
    return sorted(words, key=len)


>>> grep_word('lines', glob.glob('*.py'))
[]
```

Now the code executes, making it pretty clear that the problem is in the processing step. Hopefully, from there it is just one step to figure out the bug, as the programmer has finally got his eye on the line causing the issue by way of the process of debugging.

Word searcher program – final code

We have spent some time figuring out issues in the program by following a couple of debugging steps documented in the previous sections. With this, our hypothetical programmer was able to find the issue in the code and solve it.

Here is the final code with the bug fixed:

```
def grep_word(word, filenames):
    """ Open the given files and look for a specific word.
    Append lines containing word to a list and
    return it """

    lines, words = [], []

    for filename in filenames:
        print('Processing',filename)
        lines += open(filename).readlines()


    word = word.lower()
    for line in lines:
        if word in line.lower():
            words.append(line.strip())

    # Now sort the list according to length of lines
    return sorted(words, key=len)
```

The output is as follows:

```
>>> grep_word('lines', glob.glob('*.py'))
['for line in lines:', 'lines, words = [], []',
  '#lines.append(line.strip())',
  'lines += open(filename).readlines()',
  'Append lines containing word to a list and',
  'and return list of lines containing the word.',
  '# Now sort the list according to length of lines',
  "print('Lines => ', grep_word('lines', glob.glob('*.py')))"]
```

Let's summarize the simple debugging tricks that we've learned so far in this section, and also look at a few related tricks and techniques.

Skipping blocks of code

A programmer can skip code blocks that they suspect of causing a bug during debugging. If the block is inside a loop, this can be done by skipping execution with a `continue` statement. We've seen an example of this already.

If the block is outside of a loop, this can be done by using `if 0`, and moving the suspect code to the dependent block, as follows:

```
if 0:
     # Suspected code block
     perform_suspect_operation1(args1, args2, ...)
     perform_suspect_operation2(…)
```

If the bug disappears after this, then you're sure that the problem lies in the suspected block of code.

This trick has its own deficiency, in that it requires indenting large blocks of code to the right, which once the debugging is finished, should be indented back. Hence it is not advised for anything more than five or six lines of code.

Stopping execution

If you're in the middle of a hectic programming session, and you're trying to figure out an elusive bug, having already tried print statements, using the debugger, and other approaches, a rather drastic, but often fantastically useful, approach is to stop the execution just before or at the suspected code path using a function, `sys.exit` expression.

A sys.exit(<strategic message>) expression stops the program dead in its tracks, so this *can't be missed* by the programmer. This is often very useful in the following scenarios:

- A complex piece of code has an elusive bug depending upon specific values or ranges of input, which causes an exception that is caught and ignored, but later causes an issue in the program.
- In this case, checking for the specific value or range and then exiting the code using the right message in the exception handler via `sys.exit` will allow you to pinpoint the problem. The programmer can then decide to fix the issue by correcting the input or variable processing code.

  When writing concurrent programs, wrong usage of resource locking or other issues can make it difficult to track bugs like deadlocks, race conditions, and others. Since debugging multithreaded or multiple process programs via the debugger is very difficult, a simple technique is to put `sys.exit` in the suspect function after implementing the correct exception-handling code.
- When your code has a serious memory leak or an infinite loop, then it becomes difficult to debug after a while, and you're not able to pinpoint the problem otherwise. Moving a sys.exit(<message>) line from one line of code to the next until you identify the problem can be used as a last resort.

External dependencies – using wrappers

In cases where you suspect the problem is not inside your function, but in a function that you are calling from your code, this approach can be used.

Since the function is outside of your control, you can try and replace it with a wrapper function in a module where you have control.

For example, the following is generic code for processing serial JSON data. Let's assume that the programmer finds a bug with processing of certain data (maybe having a certain key-value pair), and suspects the external API to be the source of the bug. The bug may be that the API times out, returns a corrupt response, or in the worst case, causes a crash:

```
import external_api
def process_data(data):
    """ Process data using external API """

    # Clean up data—local function
    data = clean_up(data)
```

```
    # Drop duplicates from data—local function
    data = drop_duplicates(data)

    # Process line by line JSON
    for json_elem in data:
        # Bug ?
        external_api.process(json_elem)
```

One way to verify this is to *dummy* or *fake* the API for the specific ranges or values of the data. In this case, it can be done by creating a wrapper function as follows:

```
def process(json_data, skey='suspect_key',svalue='suspect_value'):
    """ Fake the external API except for the suspect key & value """

    # Assume each JSON element maps to a Python dictionary

    for json_elem in json_data:
        skip = False

        for key in json_elem:
            if key == skey:
                if json_elem[key] == svalue:
                    # Suspect key,value combination - dont process
                    # this JSON element
                    skip = True
                    break

        # Pass on to the API
        if not skip:
            external_api.process(json_elem)

def process_data(data):
    """ Process data using external API """

    # Clean up data—local function
    data = clean_up(data)
    # Drop duplicates from data—local function
    data = drop_duplicates(data)

    # Process line by line JSON using local wrapper
    process(data)
```

If your suspicion is indeed correct, this will cause the problem to disappear. You can then use this as a test code, and communicate with the stakeholders of the external API to get the problem fixed, or write code to make sure that the problem key-value pair is skipped in data sent to the API.

Replacing functions with their return value/ data (mocking)

In modern web application programming, you are never too far away from a blocking I/O call in your program. This can be a simple URL request, a slightly involved external API request, or maybe a costly database query and such calls can be the sources of bugs.

You may find either of the following situations:

- The return data from such a call could be the cause of an issue.
- The call itself is the cause of an issue, such as I/O or network errors, timeouts, or resource contentions.

When you encounter problems with costly I/O, replicating them can often be a problem. This is because of the following reasons:

- The I/O calls take time, so debugging this costs you a lot of wasted time, not allowing you to focus on the real issue.
- Subsequent calls may not be repeatable with respect to the issue, as external requests may return slightly different data every time.
- If you are using an external paid API, the calls may actually cost you money, so you cannot exhaust a lot of such calls on debugging and testing.

A common technique that is very useful in these cases is to save the return data of these APIs/functions, and then mock the functions by using their return data to replace the functions/APIs themselves. This is an approach similar to mock testing, but it is used in the context of debugging.

Let's look at an example of an API that returns *business listings* on websites, given a business address including details like its name, street address, city, and so on. The code looks like this:

```
import config

search_api = 'http://api.%(site)s/listings/search'

def get_api_key(site):
```

```
    """ Return API key for a site """

    # Assumes the configuration is available via a config module
    return config.get_key(site)

def api_search(address, site='yellowpages.com'):
    """ API to search for a given business address
    on a site and return results """

    req_params = {}
    req_params.update({
        'key': get_api_key(site),
        'term': address['name'],
        'searchloc': '{0}, {1}, {1}'.format(address['street'],
                                            address['city'],
                                            address['state'])})
    return requests.post(search_api % locals(),
                         params=req_params)

def parse_listings(addresses, sites):
    """ Given a list of addresses, fetch their listings
    for a given set of sites, process them """

    for site in sites:
        for address in addresses:
            listing = api_search(address, site)
            # Process the listing
            process_listing(listing, site)

def process_listings(listing, site):
    """ Process a listing and analzye it """

     # Some heavy computational code
     # whose details we are not interested.
```

> The code makes a few assumptions, one of which is that every site has the same API URL and parameters. Note that this is only for illustration purposes. In reality, each site will have very different API formats including its URL and the parameters it accepts.

Note that in this last piece of code, the actual work is being done in the `process_listings` function, the code for which is not shown, as the example is illustrative.

Let's say you are trying to debug this function. However, due to a delay or error in the API calls, you find you are wasting a lot of valuable time in fetching the listings themselves. What are some of the techniques that you can use to avoid this dependency? Here are a few things that you can do:

- Instead of fetching listings via API, save them to files, to a database, or an in-memory store, and load them on demand.
- Cache the return value of the `api_search` function via a caching or memoize patterns so that further calls after the first call, return data from memory.
- Mock the data, and return random data that has the same characteristics as the original data.

We will look at each of these in turn.

Saving to / loading data from files as cache

In this technique, you construct a filename using unique keys from the input data. If a matching file exists on disk, it is opened and the data is returned; otherwise, the call is made and the data is written. This can be achieved by using a *file caching* decorator as the following code illustrates:

```
import hashlib
import json
import os

def unique_key(address, site):
    """ Return a unique key for the given arguments """

    return hashlib.md5(''.join((address['name'],
                               address['street'],
                               address['city'],
                               site)).encode('utf-8')).hexdigest()

def filecache(func):
    """ A file caching decorator """

    def wrapper(*args, **kwargs):
        # Construct a unique cache filename
```

```
        filename = unique_key(args[0], args[1]) + '.data'

        if os.path.isfile(filename):
            print('=>from file<=')
            # Return cached data from file
            return json.load(open(filename))

        # Else compute and write into file
        result = func(*args, **kwargs)
        json.dump(result, open(filename,'w'))

        return result

    return wrapper

@filecache
def api_search(address, site='yellowpages.com'):
    """ API to search for a given business address
    on a site and return results """

    req_params = {}
    req_params.update({
        'key': get_api_key(site),
        'term': address['name'],
        'searchloc': '{0}, {1}, {1}'.format(address['street'],
                                            address['city'],
                                            address['state'])})
    return requests.post(search_api % locals(),
                         params=req_params)
```

Here's how this preceding code works:

1. The `api_search` function is decorated with `filecache` as a decorator.
2. Then `filecache` uses `unique_key` as the function to calculate the unique filename for storing the results of an API call. In this case, the `unique_key` function uses the hash of a combination of the business name, street, and city, plus the site queried for in order to build the unique value.
3. The first time the function is called, the data is fetched via API and stored in the file. During further invocations, the data is returned directly from the file.

This works pretty well in most cases. Most data is loaded just once, and on further calls, returned from the file cache. However, this suffers from the problem of *stale data*, as once the file is created, the data is always returned from it. Meanwhile, the data on the server may have changed.

This can be solved by using an in-memory key-value store and saving the data there instead of in files on disk. One can use well-known key-value stores such as **Memcached**, **MongoDB**, or **Redis** for this purpose. In the following example, we'll show you how to replace the `filecache` decorator with a memory cached decorator using Redis.

Saving to / loading data from memory as cache

In this technique, a unique in-memory cache key is constructed using unique values from the input arguments. If the cache is found on the cache store by querying using the key, its value is returned from the store; or else the call is made and the cache is written. To ensure that data is not too stale, a fixed **time-to-live** (**TTL**) is used. We use Redis as the cache store engine:

```
from redis import StrictRedis

def memoize(func, ttl=86400):
    """ A memory caching decorator """

    # Local redis as in-memory cache
    cache = StrictRedis(host='localhost', port=6379)

    def wrapper(*args, **kwargs):
        # Construct a unique key

        key = unique_key(args[0], args[1])
        # Check if its in redis
        cached_data = cache.get(key)
        if cached_data != None:
             print('=>from cache<=')
             return json.loads(cached_data)
         # Else calculate and store while putting a TTL
         result = func(*args, **kwargs)
         cache.set(key, json.dumps(result), ttl)

         return result

    return wrapper
```

Note that we are reusing the definition of `unique_key` from the previous code example.

The only thing that changes in the rest of the code is that we replace the `filecache` decorator with the `memoize` one:

```
@memoize
def api_search(address, site='yellowpages.com'):
    """ API to search for a given business address
    on a site and return results """

    req_params = {}
    req_params.update({
        'key': get_api_key(site),
        'term': address['name'],
        'searchloc': '{0}, {1}, {1}'.format(address['street'],
                                            address['city'],
                                            address['state'])})
    return requests.post(search_api % locals(),
                         params=req_params)
```

The advantages of this version over the previous one are as follows:

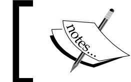
- The cache is stored in memory. No additional files are created.
- The cache is created with a TTL, beyond which it expires. So the problem of stale data is circumvented. The TTL is customizable, and defaults to a day (86,400 seconds) in this example.

There are a few other techniques for mocking external API calls and similar dependencies. Some of these are listed as follows:

- Using a `StringIO` object in Python to read/write data, instead of using a file. For example, the `filecache` or `memoize` decorators can be easily modified to use a `StringIO` object.
- Using a mutable default argument, such as a dictionary or a list, as a cache and writing results to it. Since a mutable argument in Python holds its state after repeated calls, it effectively works as an in-memory cache.
- Replacing an external API with a call to a replacement/dummy API call to a service on the local machine (`127.0.0.1` IP address) by editing the system's host file, adding an entry for the host in question, and putting its IP as `127.0.0.1`. The call to localhost can always return a standard (canned) response.

For example, on Linux and other POSIX systems, you can add a line like this in the `/etc/hosts` file:

```
# Only for testing—comment out after that!
127.0.0.1 api.website.com
```



Note that this technique is a very useful and clever approach as long as you remember to comment out such lines after testing!

Returning random/mock data

Another technique, which is mostly useful for performance testing and debugging, is to feed functions with data that is *similar*, but *not the same* as the original data.

Let's say, for example, that you are working on an application that works with patient/doctor data for patients under a specific insurance scheme (say Medicare/Medicaid in the US, ESI in India) to analyze and find out patterns such as common ailments, top 10 health issues in terms of government expenses, and so on.

Let's say that your application is expected to load and analyze tens of thousands of rows of patient data from a database at one time, which is expected to scale to 1-2 million under peak load. You want to debug the application and find out performance characteristics under such a load, but you don't have any real data, as the data is in the collection stage.

In such scenarios, libraries or functions that generate and return mock data are very useful. In this section, we will use a third-party Python library to accomplish this.

Generating random patient data

Let's assume that for a patient we need the following basic fields:

- Name
- Age
- Gender
- Health issue
- Doctor's name
- Blood group
- Insured or not
- Date of last visit to doctor

The `schematics` library in Python provides a way to generate such data structures using simple types, which can then be validated, transformed, and also mocked.

The `schematics` library is installable via `pip` using the following command:

```
$ pip install schematics
```

To generate a model of a person with just their name and age is as simple as writing a class in `schematics`:

```
from schematics import Model
from schematics.types import StringType, DecimalType

class Person(Model):
    name = StringType()
    age = DecimalType()
```

To generate mock data, a mock object is returned, and a *primitive* is created using this:

```
>>> Person.get_mock_object().to_primitive()
{'age': u'12', 'name': u'Y7bnqRt'}
>>> Person.get_mock_object().to_primitive()
{'age': u'1', 'name': u'xyrh40EO3'}
```

One can create custom types using `schematics`. For the `Patient` model, for example, let's say that we are only interested in the age group 18-80, so we need to return age data in that range.

The following custom type does that for us:

```
from schematics.types import IntType

class AgeType(IntType):
    """ An age type for schematics """

    def __init__(self, **kwargs):
        kwargs['default'] = 18
        IntType.__init__(self, **kwargs)

    def to_primitive(self, value, context=None):
        return random.randrange(18, 80)
```

Also, since the names returned by the `schematics` library are just random strings, they have some room for improvement. The following `NameType` class improves upon it by returning names containing a clever mix of vowels and consonants:

```
import string
import random

class NameType(StringType):
    """ A schematics custom name type """

    vowels='aeiou'
    consonants = ''.join(set(string.ascii_lowercase) - set(vowels))

    def __init__(self, **kwargs):
        kwargs['default'] = ''
        StringType.__init__(self, **kwargs)

   def get_name(self):
        """ A random name generator which generates
        names by clever placing of vowels and consontants """

        items = ['']*4

        items[0] = random.choice(self.consonants)
        items[2] = random.choice(self.consonants)

        for i in (1, 3):
            items[i] = random.choice(self.vowels)

        return ''.join(items).capitalize()

    def to_primitive(self, value, context=None):
        return self.get_name()
```

When combining both of these new types, our `Person` class looks much better when returning mock data:

```
class Person(Model):
    name = NameType()
    age = AgeType()
```

```
>>> Person.get_mock_object().to_primitive()
{'age': 36, 'name': 'Qixi'}
>>> Person.get_mock_object().to_primitive()
{'age': 58, 'name': 'Ziru'}
>>> Person.get_mock_object().to_primitive()
{'age': 32, 'name': 'Zanu'}
```

In a similar way, it is rather easy to come up with a set of custom types and standard types to satisfy all the fields required for a `Patient` model:

```
class GenderType(BaseType):
    """A gender type for schematics """

    def __init__(self, **kwargs):
        kwargs['choices'] = ['male','female']
        kwargs['default'] = 'male'
        BaseType.__init__(self, **kwargs)


class ConditionType(StringType):
    """ A gender type for a health condition """

    def __init__(self, **kwargs):
        kwargs['default'] = 'cardiac'
        StringType.__init__(self, **kwargs)

    def to_primitive(self, value, context=None):
        return random.choice(('cardiac',
                              'respiratory',
                              'nasal',
                              'gynec',
                              'urinal',
                              'lungs',
                              'thyroid',
                              'tumour'))

import itertools

class BloodGroupType(StringType):
    """ A blood group type for schematics  """

    def __init__(self, **kwargs):
        kwargs['default'] = 'AB+'
```

```
        StringType.__init__(self, **kwargs)

    def to_primitive(self, value, context=None):
        return ''.join(random.choice(list(itertools.product(['AB','A',
'O','B'],['+','-']))))
```

Now, combining all these with some standard types and default values into a `Patient` model, we get the following code:

```
class Patient(Model):
    """ A model class for patients """

    name = NameType()
    age = AgeType()
    gender = GenderType()
    condition = ConditionType()
    doctor = NameType()
    blood_group = BloodGroupType()
    insured = BooleanType(default=True)
    last_visit = DateTimeType(default='2000-01-01T13:30:30')
```

Now, creating random data of any size is as easy as invoking the `get_mock_object` method on the `Patient` class for any number *n*:

```
patients = map(lambda x: Patient.get_mock_object().to_primitive(),
range(n))
```

For example, to create 10,000 random sets of patient data, we use the following:

```
>>> patients = map(lambda x: Patient.get_mock_object().to_primitive(),
range(1000))
```

This data can be input to the processing functions as mock data until the real data is made available.

Note: The Faker library in Python is also useful for generating a wide variety of fake data such as names, addresses, URIs, random text, and the like.

Let's now move on from these simple tricks and techniques to something more involved, mainly configuring logging in your applications.

Logging as a debugging technique

Python comes with standard library support for logging via the aptly named `logging` module. Though print statements can be used as a quick and rudimentary tool for debugging, real-life debugging mostly requires that the system or application generate some logs. Logging is useful because of the following reasons:

- Logs are usually saved to specific log files, typically, with timestamps, and remain at the server for a while until they are rotated out. This makes debugging easy even if the programmer is debugging the issue some time after it happened.
- Logging can be done at different levels—from the basic INFO to the verbose DEBUG levels—changing the amount of information output by the application. This allows the programmer to debug at different levels of logging to extract the information they want, and figure out the problem.
- Custom loggers can be written, which can perform logging to various outputs. At its most basic, logging is done to log files, but one can also write loggers that write to sockets, HTTP streams, databases, and the like.

Simple application logging

To configure simple logging in Python is rather easy and is shown as follows:

```
>>> import logging
>>> logging.warning('I will be back!')
WARNING:root:I will be back!

>>> logging.info('Hello World')
>>>
```

Nothing happens on executing the preceding code, because, by default, `logging` is configured at the **WARNING** level. However, it is pretty easy to configure logging to change its level.

The following code changes logging to log at the `info` level, and also adds a target file to save the log:

```
>>> logging.basicConfig(filename='application.log', level=logging.DEBUG)
>>> logging.info('Hello World')
```

If we inspect the `application.log` file, we will find that it contains the following lines:

```
INFO:root:Hello World
```

In order to add timestamps to the log lines, we need to configure the logging format. This can be done as follows:

```
>>> logging.basicConfig(format='%(asctime)s %(message)s')
```

Combining this, we get the final logging configuration as follows:

```
>>> logging.basicConfig(format='%(asctime)s %(message)s',
filename='application.log', level=logging.DEBUG)
>>> logging.info('Hello World!')
```

Now, the contents of `application.log` look something like the following:

```
INFO:root:Hello World
2016-12-26 19:10:37,236 Hello World!
```

Logging supports variable arguments, which are used to supply arguments to a template string supplied as the first argument.

Direct logging of arguments separated by commas doesn't work. For example:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> x,y=10,20
>>> logging.info('Addition of',x,'and',y,'produces',x+y)
--- Logging error ---
Traceback (most recent call last):
  File "/usr/lib/python3.5/logging/__init__.py", line 980, in emit
    msg = self.format(record)
  File "/usr/lib/python3.5/logging/__init__.py", line 830, in format
    return fmt.format(record)
  File "/usr/lib/python3.5/logging/__init__.py", line 567, in format
    record.message = record.getMessage()
  File "/usr/lib/python3.5/logging/__init__.py", line 330, in getMessage
    msg = msg % self.args
TypeError: not all arguments converted during string formatting
Call stack:
  File "<stdin>", line 1, in <module>
Message: 'Addition of'
Arguments: (10, 'and', 20, 'produces', 30)
```

However, we can use the following:

```
>>> logging.info('Addition of %s and %s produces %s',x,y,x+y)
INFO:root:Addition of 10 and 20 produces 30
```

The earlier example works nicely.

Advanced logging – logger objects

Logging using the `logging` module directly works in most simple situations. However, in order to extract the maximum value out of the `logging` module, we should work with logger objects. It also allows us to perform a lot of customizations such as custom formatters, custom handlers, and so on.

Let's write a function that returns such a custom logger. It accepts the application name, the logging level, and two more options—the log filename, and whether to turn console logging on or not:

```
import logging
def create_logger(app_name, logfilename=None,
                             level=logging.INFO, console=False):

    """ Build and return a custom logger. Accepts the application
name,
    log filename, loglevel and console logging toggle """

    log=logging.getLogger(app_name)
    log.setLevel(logging.DEBUG)
    # Add file handler
    if logfilename != None:
        log.addHandler(logging.FileHandler(logfilename))

    if console:
        log.addHandler(logging.StreamHandler())

    # Add formatter
    for handle in log.handlers:
        formatter = logging.Formatter('%(asctime)s : %(levelname)-8s -
%(message)s', datefmt='%Y-%m-%d %H:%M:%S')

        handle.setFormatter(formatter)

    return log
```

Let's inspect the function:

1. Instead of using `logging` directly, it creates a `logger` object using the `logging.getLogger` factory function.
2. By default, the `logger` object is useless as it has not been configured with any handlers. Handlers are stream wrappers that take care of logging to a specific stream, such as the console, files, sockets, and so on.
3. The configuration is done on this logger object, such as setting the level (via the `setLevel` method) and adding handlers such as `FileHandler` for logging to a file and `StreamHandler` for logging to the console.
4. Formatting of the log message is done on the handlers, and not on the logger object per se. We use a standard format of `<timestamp>: <level>—<message>` using the date format for the timestamp of `YY-mm-dd HH:MM:SS`.

Let's see this in action:

```
>>> log=create_logger('myapp',logfilename='app.log', console=True)
>>> log
<logging.Logger object at 0x7fc09afa55c0>
>>> log.info('Started application')
2016-12-26 19:38:12 : INFO     - Started application
>>> log.info('Initializing objects...')
2016-12-26 19:38:25 : INFO     - Initializing objects…
```

Inspecting the app.log file in the same directory reveals the following contents:

```
2016-12-26 19:38:12 : INFO    —Started application
2016-12-26 19:38:25 : INFO    —Initializing objects…
```

Advanced logging – custom formatting and loggers

We looked at how we can create and configure logger objects according to our requirements. Sometimes, one needs to go over and above, and print extra data in the log lines, which helps debugging.

A common problem that arises in debugging applications, especially those that are performance critical, is to find out how much time each function or method takes. Now, though this can be found out by methods such as profiling the application using profilers and by using some techniques discussed previously like timer context managers, quite often, a custom logger can be written to do the trick.

Let's assume that your application is a business listing API server, which responds to listing API requests like the one we discussed in an earlier section. When it starts off, it needs to initialize a number of objects and load some data from the DB.

Assume that as part of performance optimization, you have tuned these routines, and would like to record how much time these take. We'll see if we can write a custom logger to do it for us:

```
import logging
import time
from functools import partial

class LoggerWrapper(object):
    """ A wrapper class for logger objects with
    calculation of time spent in each step """

    def __init__(self, app_name, filename=None,
                       level=logging.INFO, console=False):
        self.log = logging.getLogger(app_name)
        self.log.setLevel(level)

        # Add handlers
        if console:
            self.log.addHandler(logging.StreamHandler())

        if filename != None:
            self.log.addHandler(logging.FileHandler(filename))

        # Set formatting
        for handle in self.log.handlers:

          formatter = logging.Formatter('%(asctime)s [%(timespent)s]:
%(levelname)-8s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S')
             handle.setFormatter(formatter)


        for name in ('debug','info','warning','error','critical'):
            # Creating convenient wrappers by using functools
            func = partial(self._dolog, name)
            # Set on this class as methods
            setattr(self, name, func)

        # Mark timestamp
```

```
        self._markt = time.time()

    def _calc_time(self):
        """ Calculate time spent so far """

        tnow = time.time()
        tdiff = int(round(tnow - self._markt))

        hr, rem = divmod(tdiff, 3600)
        mins, sec = divmod(rem, 60)
        # Reset mark
        self._markt = tnow
        return '%.2d:%.2d:%.2d' % (hr, mins, sec)

    def _dolog(self, levelname, msg, *args, **kwargs):
        """ Generic method for logging at different levels """

        logfunc = getattr(self.log, levelname)
        return logfunc(msg, *args, extra={'timespent': self._calc_
time()})
```

We have built a custom class named `LoggerWrapper`. Let's analyze the code and see what it does:

1. The `__init__` method of this class is very similar to our `create_logger` function written before. It takes the same argument, constructs handler objects, and configures `logger`. However, this time, the `logger` object is part of the outer `LoggerWrapper` instance.
2. The formatter takes an additional variable template named `timespent`.
3. No direct logging methods seem to be defined. However, using the partial functions technique, we wrap the `_dolog` method at the different levels of logging, and set them on the class as `logging` methods, dynamically, by using `setattr`.
4. The `_dolog` method calculates the time spent in each routine by using a marker timestamp—initialized the first time, and then reset in every call. The time spent is sent to the logging methods using a dictionary argument named extra.

Let's see how the application can use this logger wrapper to measure the time spent in critical routines. Here is an example that assumes a Flask web application:

```
# Application code
log=LoggerWrapper('myapp', filename='myapp.log',console=True)

app = Flask(__name__)
log.info("Starting application...")
log.info("Initializing objects.")
init()
log.info("Initialization complete.")
log.info("Loading configuration and data …")
load_objects()
log.info('Loading complete. Listening for connections …')
mainloop()
```

Note that the time spent is logged inside square brackets just after the timestamp.

Let's say that this last code produces an output like the following:

```
2016-12-26 20:08:28 [00:00:00]: INFO    —Starting application...
2016-12-26 20:08:28 [00:00:00]: INFO     - Initializing objects.
2016-12-26 20:08:42 [00:00:14]: INFO     - Initialization complete.
2016-12-26 20:08:42 [00:00:00]: INFO     - Loading configuration and data
...
2016-12-26 20:10:37 [00:01:55]: INFO     - Loading complete. Listening
for connections
```

From the log lines, it's evident that the initialization took 14 seconds, whereas the loading of configuration and data took 1 minute and 55 seconds.

By adding similar log lines, you can get a quick and reasonably accurate estimate of the time spent on critical pieces of the application. Being saved in log files, another added advantage is that you don't need to specially calculate and save it anywhere else.

Using this custom logger, note that the time shown as time spent for a given log line is the time spent in the routine of the previous line.

Advanced logging – writing to syslog

POSIX systems such as Linux and Mac OS X have a system log file, which the application can write to. Typically, this file is present as `/var/log/syslog`. Let's see how Python logging can be configured to write to the system log file.

The main change that you need to make is to add a system log handler to the logger object like this:

```
log.addHandler(logging.handlers.SysLogHandler(address='/dev/log'))
```

Let's modify our `create_logger` function to enable it to write to `syslog`, and see the complete code in action:

```
import logging
import logging.handlers

def create_logger(app_name, logfilename=None, level=logging.INFO,
                             console=False, syslog=False):
    """ Build and return a custom logger. Accepts the application
        name,
    log filename, loglevel and console logging toggle and syslog
toggle """

    log=logging.getLogger(app_name)
    log.setLevel(logging.DEBUG)
    # Add file handler
    if logfilename != None:
        log.addHandler(logging.FileHandler(logfilename))

    if syslog:
        log.addHandler(logging.handlers.SysLogHandler(address='/dev/
                       log'))

    if console:
        log.addHandler(logging.StreamHandler())

    # Add formatter
    for handle in log.handlers:
        formatter = logging.Formatter('%(asctime)s : %(levelname)-8s
                    - %(message)s',  datefmt='%Y-%m-%d %H:%M:%S')
        handle.setFormatter(formatter)



    return log
```

Now let's try to create a logger while logging to `syslog`:

```
>>> create_logger('myapp',console=True, syslog=True)
>>> log.info('Myapp - starting up…')
```

Let's inspect syslog to see if it actually got logged:

```
$ tail -3 /var/log/syslog
Dec 26 20:39:54 ubuntu-pro-book kernel: [36696.308437] psmouse serio1:
TouchPad at isa0060/serio1/input0 - driver resynced.
Dec 26 20:44:39 ubuntu-pro-book 2016-12-26 20:44:39 : INFO     - Myapp -
starting up...
Dec 26 20:45:01 ubuntu-pro-book CRON[11522]: (root) CMD (command -v
debian-sa1 > /dev/null && debian-sa1 1 1)
```

The output shows that it did.

Debugging tools – using debuggers

Most programmers tend to think of *debugging* as something that they ought to do with a debugger. In this chapter, we have so far seen that, more than an exact science, debugging is an art, which can be done using a lot of tricks and techniques rather than directly jumping to a debugger. However, sooner or later, we expected to encounter the debugger in this chapter—and here we are!

The Python Debugger, or pdb as it is known, is part of the Python runtime.

Pdb can be invoked when running a script from the beginning as follows:

```
$ python3 -m pdb script.py
```

However, the most common way in which programmers invoke pdb is to insert the following line at a place in the code where you want to enter the debugger:

```
import pdb; pdb.set_trace()
```

Let's use this, and try to debug an instance of the first example in this chapter, that is, the sum of the max subarray. We will debug the `O(n)` version of the code as an example:

```
def max_subarray(sequence):
    """ Maximum subarray - optimized version """

    max_ending_here = max_so_far = 0
    for x in sequence:
```

```
            # Enter the debugger
            import pdb; pdb.set_trace()
            max_ending_here = max(0, max_ending_here + x)
            max_so_far = max(max_so_far, max_ending_here)

        return max_so_far
```

A debugging session with pdb

The debugger is entered in the very first loop immediately after the program is run:

```
>>> max_subarray([20, -5, -10, 30, 10])
> /home/user/programs/maxsubarray.py(8)max_subarray()
-> max_ending_here = max(0, max_ending_here + x)
-> for x in sequence:
(Pdb) max_so_far
20
```

You can stop the execution using `s`. `Pdb` will execute the current line, and stop:

```
> /home/user/programs/maxsubarray.py(7)max_subarray()
-> max_ending_here = max(0, max_ending_here + x)
```

You can inspect the variables by simply typing them and pressing *Enter*:

```
(Pdb) max_so_far
20
```

The current stack trace can be printed using `w` or where. An arrow (→) indicates the current stack frame:

```
(Pdb) w
  <stdin>(1)<module>()
> /home/user/programs/maxsubarray.py(7)max_subarray()
-> max_ending_here = max(0, max_ending_here + x)
```

The execution can be continued until the next breakpoint by using `c` or continue:

```
   > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb) max_so_far
20
(Pdb) c
```

```
> /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb) max_so_far
20
(Pdb) c
> /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb) max_so_far
35
(Pdb) max_ending_here
35
```

In the preceding code, we continued three iterations of the `for` loop until the max value changed from 20 to 35. Let's inspect where we are in the sequence:

```
(Pdb) x
30
```

We have one more item to go in the list, namely, the last one. Let's inspect the source code at this point using the `l` or the `list` command:

```
(Pdb) l
  1
  2     def max_subarray(sequence):
  3         """ Maximum subarray - optimized version """
  4
  5         max_ending_here = max_so_far = 0
  6  ->     for x in sequence:
  7             max_ending_here = max(0, max_ending_here + x)
  8             max_so_far = max(max_so_far, max_ending_here)
  9             import pdb; pdb.set_trace()
 10
 11         return max_so_far
```

One can traverse up and down the stack frames by using the `u` or `up` and `d` or *down* commands, respectively:

```
(Pdb) up
> <stdin>(1)<module>()
(Pdb) up
```

```
*** Oldest frame
(Pdb) list
[EOF]
(Pdb) d
> /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
```

Let's now return from the function:

```
(Pdb) r
> /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb) r
--Return--
> /home/user/programs/maxsubarray.py(11)max_subarray()->45
-> return max_so_far
```

The return value of the function is `45`.

Pdb has a lot of other commands than what we covered here. However, we don't intend for this session to be a fully fledged pdb tutorial. Interested programmers can refer to the documentation on the web to learn more.

Pdb – similar tools

The Python community has built a number of useful tools that build on top of `Pdb`, but add more useful functionality, developer's ease-of-use, or both.

iPdb

Basically, iPdb is iPython-enabled pdb. It exports functions to access the iPython debugger. It also has tab completion, syntax highlighting, and better traceback, and introspection methods.

iPdb can be installed with `pip`.

The following screenshot shows a session of debugging using iPdb, the same function as we did with pdb before. Observe the syntax highlighting that **iPdb** provides:

```
IPython: user/programs
File  Edit  View  Search  Terminal  Help
 warn("Attempting to work in a virtualenv. If you encounter problems, please "
> /home/user/programs/maxsubarray.py(6)max_subarray()
      4
      5     max_ending_here = max_so_far = 0
----> 6     for x in sequence:
      7         max_ending_here = max(0, max_ending_here + x)
      8         max_so_far = max(max_so_far, max_ending_here)

ipdb> max_so_far
20
ipdb> r
> /home/user/programs/maxsubarray.py(6)max_subarray()
      4
      5     max_ending_here = max_so_far = 0
----> 6     for x in sequence:
      7         max_ending_here = max(0, max_ending_here + x)
      8         max_so_far = max(max_so_far, max_ending_here)

ipdb> max_so_far
20
ipdb> l
      1
      2 def max_subarray(sequence):
      3     """ Maximum subarray - optimized version """
      4
      5     max_ending_here = max_so_far = 0
----> 6     for x in sequence:
      7         max_ending_here = max(0, max_ending_here + x)
      8         max_so_far = max(max_so_far, max_ending_here)
```

iPdb in action, showing syntax highlighting

Also note that iPdb provides a fuller stack trace as opposed to `Pdb`:

```
IPython: user/programs
File  Edit  View  Search  Terminal  Help
      1
      2 def max_subarray(sequence):
      3     """ Maximum subarray - optimized version """
      4
      5     max_ending_here = max_so_far = 0
----> 6     for x in sequence:
      7         max_ending_here = max(0, max_ending_here + x)
      8         max_so_far = max(max_so_far, max_ending_here)
      9         import ipdb; ipdb.set_trace()
     10
     11     return max_so_far

ipdb> w
  /home/user/programs/maxsubarray.py(14)<module>()
     10
     11     return max_so_far
     12
     13 if __name__ == "__main__":
---> 14     print(max_subarray([20, -5, -10, 30, 10]))

> /home/user/programs/maxsubarray.py(6)max_subarray()
      4
      5     max_ending_here = max_so_far = 0
----> 6     for x in sequence:
      7         max_ending_here = max(0, max_ending_here + x)
      8         max_so_far = max(max_so_far, max_ending_here)

ipdb>
```

iPdb in action, showing a fuller stack trace than pdb

Note that iPdb uses iPython as the default runtime instead of Python.

Pdb++

Pdb++ is a drop-in replacement for pdb with features similar to iPdb, but it works on the default Python runtime instead of requiring iPython. Pdb++ is also installable via pip.

Once pdb++ is installed, it takes over at all places that import `Pdb`, so no code change is required at all.

Pdb++ does smart command parsing. For example, if there are variable names conflicting with the standard Pdb commands, `Pdb` will give preference to the command over displaying the variable contents. Pdb++ figures this out intelligently.

Here is a screenshot showing Pdb++ in action, including syntax highlighting, tab completion, and smart command parsing:

```
IPython: user/programs
File  Edit  View  Search  Terminal  Help
-> print(max_subarray([20, -5, -10, 30, 10]))
[1] > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb++) max
                 max              max_ending_here  max_so_far       max_subarray
(Pdb++) max_so_far
20
(Pdb++) c
[1] > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb++) c
[1] > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb++) x
-10
(Pdb++) c
[1] > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb++) max_s
              max_so_far    max_subarray
(Pdb++) max_so_far
35
(Pdb++) c
[1] > /home/user/programs/maxsubarray.py(6)max_subarray()
-> for x in sequence:
(Pdb++) c=x
(Pdb++) c
10
(Pdb++)
```

Pdb++ in action—Note the smart command parsing, where the `c` variable is interpreted correctly

Advanced debugging – tracing

Tracing of a program right from the beginning can often be used as an advanced debugging technique. Tracing allows a developer to trace program execution, find caller/callee relationships, and figure out all functions executed during the run of a program.

The trace module

Python comes with a default `trace` module as part of its standard library.

The trace module takes one of the `-trace`, `--count`, or `-listfuncs` options. The first option traces and prints all the source lines as they are executed. The second option produces an annotated list of files, which shows how many times a statement was executed. The latter simply displays all the functions executed by running of the program.

The following is a screenshot of the subarray problem being invoked by the `-trace` option of the `trace` module:

```
IPython: user/programs
File  Edit  View  Search  Terminal  Help
$ python3 -m trace --trace maxsubarray.py
 --- modulename: maxsubarray, funcname: <module>
maxsubarray.py(2): def max_subarray(sequence):
maxsubarray.py(12): if __name__ == "__main__":
maxsubarray.py(13):     print(max_subarray([20, -5, -10, 30, 10]))
 --- modulename: maxsubarray, funcname: max_subarray
maxsubarray.py(5):     max_ending_here = max_so_far = 0
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(7):         max_ending_here = max(0, max_ending_here + x)
maxsubarray.py(8):         max_so_far = max(max_so_far, max_ending_here)
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(7):         max_ending_here = max(0, max_ending_here + x)
maxsubarray.py(8):         max_so_far = max(max_so_far, max_ending_here)
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(7):         max_ending_here = max(0, max_ending_here + x)
maxsubarray.py(8):         max_so_far = max(max_so_far, max_ending_here)
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(7):         max_ending_here = max(0, max_ending_here + x)
maxsubarray.py(8):         max_so_far = max(max_so_far, max_ending_here)
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(7):         max_ending_here = max(0, max_ending_here + x)
maxsubarray.py(8):         max_so_far = max(max_so_far, max_ending_here)
maxsubarray.py(6):     for x in sequence:
maxsubarray.py(10):     return max_so_far
45
 --- modulename: trace, funcname: _unsettrace
trace.py(77):         sys.settrace(None)
$
```

Tracing program execution using the trace module by using its `-trace` option.

As you can see, the `trace` module traced the entire program execution, printing the lines of code one by one. Since most of this code is a `for` loop, you actually see the lines of code in the loop getting printed the number of times the loop was executed (five times).

The `-trackcalls` option traces and prints the relationships between the caller and callee functions.

There are many other options to the `trace` module such as tracking calls, generating annotated file listings, reports, and so on. We won't be having an exhaustive discussion regarding these, as the reader can refer to the documentation of this module on the web to read more about it.

The lptrace program

When debugging servers and trying to find out performance or other issues on production environments, what a programmer needs is not often the Python system or stack trace as given by the `trace` module, but to attach to a process in real time and see which functions are getting executed.

`iptrace` can be installed using pip. Note that it doesn't work with **Python3**.

The `lptrace` package allows you to do this. Instead of giving a script to run, it attaches to an existing process running a Python program via its process ID, such as running servers, applications, and the like.

In the following screenshot, you can see `iptrace` debugging the Twisted chat server that we developed in *Chapter 8*, *Architectural Patterns – The Pythonic Approach*, live. The session shows the activity when the client andy has connected:

```
IPython: user/programs
File  Edit  View  Search  Terminal  Help
$ python3 -m trace --trackcalls maxsubarray.py
45

calling relationships:

*** /usr/lib/python3.5/trace.py ***
    trace.Trace.runctx -> trace._unsettrace
  --> maxsubarray.py
    trace.Trace.runctx -> maxsubarray.<module>

*** maxsubarray.py ***
    maxsubarray.<module> -> maxsubarray.max_subarray
$
```

The `iptrace` command debugging a chat server in Twisted

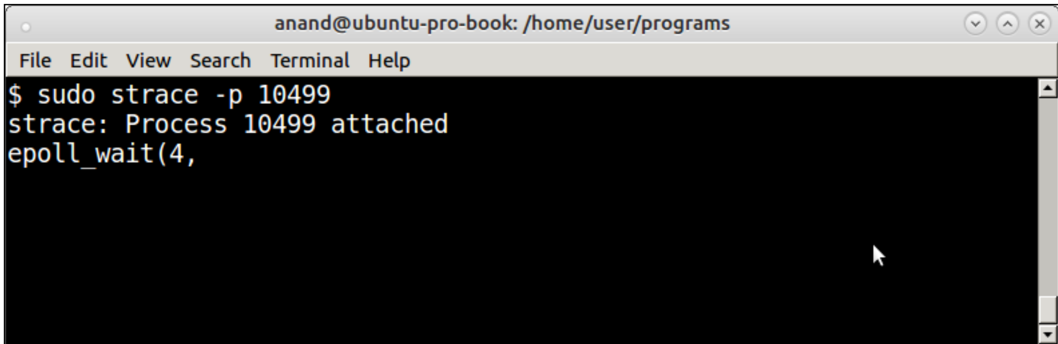
There are lots of log lines, but you can observe how some well-known methods of the Twisted protocol are being logged such as **connectionMade** when the client has connected. Socket calls such as *accept* can also be seen as part of accepting the connection from the client.

System call tracing using strace

`strace` is a Linux command, which allows a user to trace system calls and signals invoked by a running program. It is not exclusive to Python, but it can be used to debug any program. `strace` can be used in combination with `iptrace` to troubleshoot programs with respect to their system calls.

`strace` is similar to `iptrace` in that it can be made to attach to a running process. It can also be invoked to run a process from the command line, but it is more useful when running attached to a process such as a server.

For example, this screenshot shows the `strace` output when running attached to our chat server:



The `strace` command attached to the Twisted chat server

The `strace` command corroborates the conclusion of the `lptrace` command of the server waiting on an epoll handle for incoming connections.

This is what happens when a client connects:

```
anand@ubuntu-pro-book: /home/user/programs
File  Edit  View  Search  Terminal  Help

[{EPOLLIN, {u32=3, u64=14887175234342879235}}], 4, -1) = 1
accept(3, {sa_family=AF_INET, sin_port=htons(44874), sin_addr=inet_addr("127.0.0.1")}, [16
]) = 10
fcntl(10, F_GETFD)                      = 0
fcntl(10, F_SETFD, FD_CLOEXEC)          = 0
fcntl(10, F_GETFL)                      = 0x2 (flags O_RDWR)
fcntl(10, F_SETFL, O_RDWR|O_NONBLOCK)   = 0
epoll_ctl(4, EPOLL_CTL_ADD, 10, {EPOLLIN, {u32=10, u64=14887175234342879242}}) = 0
accept(3, 0x7ffd17f2acd0, 0x7ffd17f2accc) = -1 EAGAIN (Resource temporarily unavailable)
epoll_wait(4, [{EPOLLIN, {u32=11, u64=14887175234342879243}}], 5, -1) = 1
recvfrom(11, "", 65536, 0, NULL, NULL)  = 0
epoll_ctl(4, EPOLL_CTL_DEL, 11, 0x7ffd17f2aa60) = 0
shutdown(11, SHUT_RDWR)                 = 0
close(11)                               = 0
epoll_wait(4,
```

The strace command showing system calls for a client connecting to the Twisted chat server

`strace` is a very powerful tool, which can be combined with tools specific for the runtime (such as lptrace for Python) in order to do advanced debugging in production environments.

Summary

In this chapter, we learned about different debugging techniques with Python. We started with the simple `print` statement and followed it with simple tricks to debug a Python program such as using the `continue` statement in a loop, strategically placing the `sys.exit` calls between code blocks, and so on.

We then looked at debugging techniques in some detail, especially on mocking and randomizing data. Techniques such as caching in files and in-memory databases such as Redis were discussed with examples.

An example using Python schematics library showed generating random data for a hypothetical application in the healthcare domain.

The next section was about logging and using it as a debugging technique. We discussed simple logging using the `logging` module, advanced logging using the `logger` object, and wrapped up the discussion by creating a logger wrapper with its custom formatting for logging the time taken inside functions. We also studied an example of writing to `syslog`.

The end of the chapter was devoted to a discussion on debugging tools. You learned the basic commands of pdb, the Python debugger, and took a quick look at similar tools that provide a better experience, namely, iPdb and Pdb++. We ended the chapter with a brief discussion on tracing tools such as lptrace and the ubiquitous `strace` program on Linux.

Index

Symbols

__init__.py files 463

A

accessibility testing 89
Advanced Message Queuing Protocol (AMQP) 263
antipatterns
about 39
functional constructs, overusing 39
mixed Indentation 39
string literal types, mixing 39
arbitrary input
evaluating 294, 295, 297
objects, serializing 300-304
overflow errors 298, 300
asymptotic notation 136
async
using 244-246
asynchronous execution 236
asynchronous processing 196
asyncio module 197, 241, 242
availability, quality attributes
about 26
fault detection 26
fault prevention 27
fault recovery 27
await
using 244, 246, 247

B

Bachmann-Landau notation 136
behavorial pattern
about 381
Iterator pattern 382-385
Observer pattern 385-393
State pattern 393-400
Black-box testing 87
bloom filters 184-188
BlueGreen deployment 473
Borg
versus Singleton 340, 341
Bucket testing (A/B testing) 473
Builder pattern 353, 355, 357, 358

C

canary releases 473
Celery
about 264
Gunicorn 274
serving, with Python on web 269-271
uWSGI 272, 273
with Mandelbrot set 264-269
ChainMap 181, 182
Chaos Monkey 474
code
cohesion and coupling, measuring 50-53
commenting 49
string and text, processing 54
code coverage
about 105
measuring, coverage.py used 105, 106
measuring, nose2 used 106, 107
measuring, py.test used 107
code, documenting
about 40
class docstrings 43, 44
code comments 40
docstrings function 41

external documentation 40
inline documentation 40
module docstrings 44
user manuals 40
code or application profiling tools 135
code, refactoring
about 77
code smells, fixing 80, 82
complexity, fixing 78, 79
styling and coding conventions, fixing 82
code smells
about 64
at class level 64
at method/function level 65
collections module
about 175
ChainMap 181, 182
counter 180
defaultdict 176, 177
deque 176
duplicates, dropping from container without losing order 179
least recently used (LRU) cache dictionary, implementing 179
namedtuple 182, 183
OrderedDict 178
comma separated value (CSV) 201
Common Gateway Interface (CGI) 271
Composite pattern 330
concurrency
about 196
asynchronous processing 196
in Python, with multithreading 198
multiprocessing 196
multithreading 196
versus parallelism 197
concurrent futures
concurrency options 252
disk thumbnail generator 249-251
for high-level concurrent processing 247, 249
joblib package 253, 254
Mandelbrot set 255, 256, 257, 261
parallel processing libraries 253
PyMP 254, 255
concurrent.futures module 197
concurrent programming
versus, event-driven programming 418, 419
Confidentiality, Integrity, and Availability (CIA) 283
connectionMade 521
content delivery network (CDN) 277
context manager
used, for managing time 139, 140, 142
continuous deployment 472
cooperative multitasking
about 237
versus pre-emptive multitasking 236-240
co-routines
using 241
counter
about 180
used, for sorting disk files 229, 230, 232
cProfile 153
creational pattern
about 335
Borg, versus Singleton 340, 341
deep, versus shallow copy 346
deep, versuss shallow copy 347
Factory pattern 342-345
factory patterns 329
patterns combining, metaclasses used 349
Prototype pattern 345, 346
prototype patterns 330
Prototype, using metaclasses 347, 348
Singleton and related patterns 330
Singleton pattern 335-340
Cross-Site Scripting (XSS) 314
cyclomatic complexity 65, 66

D

Data Sink 439
debugging, advanced
lptrace program 520, 521
system call tracing, strace used 521, 522
trace module 519
tracing 518
debugging technique, logging as
about 505
custom formatting and loggers 508, 510, 511
logger objects 507, 508

simple application logging 505, 507
syslog, writing to 512, 513
debugging tools
debuggers used 513
debugging session, pdb used 514, 515
iPdb 516, 517
Pdb++ 518
debugging, tricks and techniques
about 486
code blocks, skipping 491
data, loading from files as cache 496-498
data, loading from memory as cache 498, 499
data saving to, from files as cache 496
data saving to, from memory as cache 498, 499
execution, skipping 491, 492
functions, replacing with return value/data (Mocking) 494-496
random/mock data, returning 500
random patient data, generating 500-504
word searcher program 487
word searcher program, debugging step 1 488, 489
word searcher program, debugging step 2 489, 490
word searcher program, final code 490, 491
wrappers, using 492-494
deep
versus shallow copy 346, 347
Default dicts 176, 177
Deferreds 420
Denial of Service (DOS) 310-313
deployability
about 448
factors 449, 450
deployability, quality attributes
about 29
factors 29, 30
deque 176
design patterns
about 328, 329
categories 329, 330
elements 328, 329
pluggable hashing algorithms 331-334
design patterns, categories
about 329
behavioral pattern 330
creational pattern 329
structural pattern 330
deterministic profiling 152
dictionaries 174
disk files
sorting 228, 229
sorting, with counter 229, 230
sorting, with multiprocessing 234
disk thumbnail generator 249-251
doctests 113, 115, 116

E

Event-driven programming
about 411
chat server and client, I/O multiplexing with select module used 411, 414, 416-418
chat server, Twisted used 422, 424, 427, 428
Eventlet 428, 429
Greenlets and Gevent 430, 431
twisted 420
twisted, web client 421, 422
versus, concurrent programming 418, 419
Eventlet 428, 429
executor interface
map method 247
submit method 247

F

Fabric
used, for remote deployments 468, 469
Factory pattern 329, 342, 344, 345
Finite State Machine (FSM) 394
Flake8 64
Flask 409-411
functional testing 87
future
about 236
waiting for, with async and await 244-247

G

Global Interpreter Lock (GIL)
about 224
for multithreading 223, 224

used for multithreading 223
Gevent 430, 431
Greenlets 430, 431
Gunicorn
about 274
versus uWSGI 274

H

horizontal scalability architectures
about 278, 279
active redundancy 275
best practices 277
blue-green deployments 276
hot standby 275
read replicas 276

I

installation testing 88
instrumentation tools 135
integration tests
about 117, 118
writing, approaches 118, 119
iPdb 516, 517
Iterator pattern 382, 383, 385

J

joblib package 253, 254

L

late binding techniques
about 62
brokers/registry lookup services 62
creational patterns,using 63
deployment time binding 62
notification services 62
plugin mechanisms 62
static analysis, tools 63
Lightweight Directory Access Protocol (LDAP) 287
line profiler 159-161
lists 173
locks
used, for implementing resource constraint(s) 207-209
versus semaphores 214, 215
lptrace program 520, 521

M

maintainability 34
Mandelbrot set
about 255-258, 261
Celery, using 264-269
mandelbrot_calc_row function 257
mandelbrot_calc_set function 257
McCabe 64
memory profiler 161-163
message-oriented middleware (MoM) 263
message queues 262
Microservice architecture
about 432, 433
advantages 438
Microservice frameworks 433
restaurant reservation example 435-437
Model Template View (MTV) 406, 407
Model View Controller (MVC)
about 404-406
automated model-centric views 407-409
Flexible Microframework 409-411
modifiability
about 34, 55
Abstract common services 57
aspects 34
code smells 64, 65
cyclomatic complexity 65, 66
explicit interfaces, providing 55
Inheritance techniques, using 58-61
late binding techniques, using 62
metrics, testing for 66
Pylint, running 69-76
two-way dependencies, reducing 56, 57
modifiability, quality attributes
about 18
affecting, factors 21
modularity 34
Monit 277
monitoring tools 135
MTBF 26
MTTR 26
multiprocessing
about 196, 224

disk files, sorting 228, 229
primality checker 224, 225, 227
used, for sorting disk files 234
versus multithreading 235
multiprocessing module 197
multithreading
about 196
for concurrency, in Python 198
versus multiprocessing 235
with Python and GIL 224

N

namedtuple 182, 183
nose2 101, 102
NT LAN Manager (NTLM) 287

O

Object Relational Mapper (ORM) 407
Objgraph (object graph) 16, 80-170
Observer pattern 387-392
OrderedDic 178
order of the function 136

P

parallelism
versus concurrency 197
parallel processing libraries 253
Pdb 516
Pdb++ 518
performance
about 133
code or application profiling tools 135
complexity 136, 137
instrumentation tools 135
measuring 193, 194, 196
monitoring tools 135
software performance engineering 133, 134
stress testing tools 135
testing and measurement tools 135
Performance Engineering Life Cycle (PGLC) 134
performance, measuring
about 138, 139
code timing, timeit module used 142, 143
CPU time measuring, timeit used 151
time complexity, finding 145-150
timeit used 143, 144
time measuring, context manager used 139-142
performance, programming
about 172
bloom filters 184, 185, 187
dictionaries 174
high performance containers 175
immutable containers 175
lists 173
mutable containers 173
Probabilistic data structures 184
sets 174
performance testing
about 88
load testing 88
scalability testing 88
stress testing 88
Perl Webserver Gateway Interface (PSGI) 272
Pip 453, 454
Pipe and Filter architecture
about 438, 439
in Python 439-445
pluggable hashing algorithms
about 331-334
summing up 334, 335
pre-emptive multitasking
about 237
versus cooperative multitasking 236-240
primality checker 224-227
profiling
about 152
deterministic profiling 152
prime number iterator class 156, 157
statistics, collecting 158
statistics, reporting 158
third-party profilers 159
with cProfile and profile 152-156
Prototype pattern
about 330, 345, 346
Builder pattern 353-358
Prototype factory 350, 352
Proxy pattern 330
Publish-Subscribe 385
Pycodestyle 64

Pyflakes 63
Pylint
about 63
running 69-76
PyMP 254, 255
Pympler 170-172
py.test
used, for testing 103, 104
Python
about 35, 36
asyncio module 241, 242
concurrency, with multithreading 198
reading input 291-294
security 290, 291
used for multithreading 223
Python Enhancement Proposal (PEP) 47, 48, 271
Python Imaging Library (PIL) 199, 257
Python Open Web Application Security Project (OWASP) project 324
Python Package Index (PyPI) 460
Python Packaging Authority (PyPA)
about 468
URL 468

Q

quality attributes
about 17
availability 26, 27
deployability 29, 30
modifiability 18-20
performance 25
scalability 23, 24
security 27, 28
testability 21-23
Quality of Service (QoS) 276

R

RabbitMQ 263
readability
about 34, 35
antipatterns 37, 38
code, documenting 39-41
code, refactor and review 48
coding and style guidelines, following 47
techniques 39
Receiving Applications 263
Relational Database Systems (RDBMs) 279
resource constraint(s)
implementing, with locks 207-209
implementing, with semaphores 212, 213
Response For Class (RFC) 90
reusability 34
RLock 198
root-mean-squared (RMS) 49

S

scalability
about 191
measuring 193-195
scalability architectures
about 275
horizontal scalability architectures 275-279
vertical scalability architectures 275
scalability, quality attributes
about 23
horizontal scalability 24
vertical scalability 24
scale horizontally 191
scale out 191
scale up 191
scale vertically 191
scaling
message queues 262, 263
on web 262
task queues 262, 263
workflows 262
secure architecture 282
secure coding
about 284
strategies 284
security issues
Cross-Site Scripting (XSS) 314
Cross-Site Scripting (XSS), mitigation 315, 316
Denial of Service (DOS) 310-313
Denial of Service (DOS), mitigation 315, 316
Server Side Template Injection (SSTI) 305-308
Server Side Template Injection (SSTI), mitigation 308

with web applications 304
security, quality attributes
about 27
authenticity 28
integrity 28
origin 28
security, strategies
about 317
expressions, evaluating 317
files 318
local data 321
overflow errors 317
passwords and sensitive information, handling 319-321
race conditions 322
reading input 317
security updates 323
serialization 317
string formatting 317
security testing 88
security vulnerabilities
about 285-288
cryptography issues 286, 287
improper access control 286
information leak 288, 289
insecure file operations 290
overflow errors 285
race conditions 289
unvalidated input 286
Selenium Web Driver
used, for test automation 120-122
semaphores
used, for implementing resource constraint(s) 212, 213
versus locks 214, 215
Sending Applications 262
Server Side Template Injection (SSTI) 305, 307
Service Oriented Architectures (SOA) 432
sets 174
setup.py file 464
Singleton pattern 335-338
Software Architecture
aspects 4
characteristics 5-9, 11
defining 2
importance 11-13
versus design 3
software deployment
about 452
application, packaging 462
application, submission 462
Fabric, using for remote deployments 468, 469
__init__.py files 463
package, installing 464, 465
package, submitting to PyPI 465, 467
Pip 453, 454
PyPI 460, 461
Python code, packaging 453
setup.py file 463, 464
Supervisor, used for managing remote daemons 471
Virtualenv 455, 456
Virtualenv and pip 457-459
virtual environments, relocatable 459
software deployment architecture, tiers
about 451
Development and Test/ Stage and Production 452
Development and Test/Stage/ Production 452
Development/Test/Stage/ Production 451, 452
software deployment, patterns
BlueGreen deployment 473
Bucket testing (A/B testing) 473
Canary releases 473
Chaos Monkey 474
continuous deployment 472
induced chaos 473
Software Development Life Cycle (SDLC) 133
Software Performance Engineering (SPE) 133
State pattern 398-400
strace
used, for system call tracing 521, 522
stress testing tools 135
structural pattern
about 330, 360
Adapter pattern 360-369
Composite pattern 330
Facade pattern 370, 371

Facades 371
instance-counting proxy 378-380
proxy pattern 330, 377
subarray
analysis and rewrite 481-483
code, optimizing 484, 485
code, timing 484, 485
issues 478, 479
print, power 479, 481
Supervisor
about 277
used, for managing remote daemons 471
symmetric multiprocessing (SMP) 253
System Architecture
versus Enterprise Architecture 13-16

T

task 264
task queues 262
testability
about 86
architectural, aspects 87, 88
control and isolate external dependencies 91-95
predictability, improving 90
reduce system complexity 89
software testability 86
strategies 89
testability, quality attributes 21-23
test automation
about 120
Selenium Web Driver used 120, 121
Test-Driven Development (TDD)
about 122
with palindromes 123-129
third-party profilers
line profiler 159-161
memory profiler 161-163
substring (subsequence) problem 163-168
threading module 197, 198
thumbnail generator
about 199, 200
producer/consumer architecture 201-206
resource constraint(s), implementing with locks 207-209
resource constraint(s), implementing with semaphores 211, 213
URL rate controller, implementing with conditions 215-223
timeit module
used, for code timing 142, 143
used, for measuring CPU time 151
used, for measuring performance 143, 144
trace module 519, 520
tuples 175
twisted
about 420
chat server, using for 422, 424, 427, 428
two-way dependencies
reducing 56, 57

U

unit testing 87, 96
URL rate controller
implementing, with conditions 215-223
usability testing 88
uWSGI
about 272, 273
versus Gunicorn 274

V

vertical scalability architectures
about 275
existing resources, using in system 275
resources, adding to existing system 275
Virtualenv
about 455, 456
and pip 457

W

web
scaling on 262
Web Content Accessibility Guidelines (WCAG) 89
Web Server Gateway Interface (WSGI) 269-271
White-box testing
about 87
code coverage 105

integration tests 117, 119
mocking 108
nose2 101, 102
principles 95
py.test, testing with 102, 104
test automation 120
unit test case 99, 101
unit testing 96
unit testing, in action 97-99

workers 264