

Python Parallel Programming

Cookbook

Second Edition

Over 70 recipes to solve challenges in multithreading and distributed system with Python 3



Packt>

www.packt.com

Giancarlo Zaccone

Python Parallel Programming Cookbook

Second Edition

Over 70 recipes to solve challenges in multithreading and distributed system with Python 3

Giancarlo Zaccone



BIRMINGHAM - MUMBAI

Python Parallel Programming Cookbook

Second Edition

Copyright © 2019 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Commissioning Editor: Richa Tripathi
Acquisition Editor: Chaitanya Nair
Content Development Editor: Ruvika Rao
Senior Editor: Afshaan Khan
Technical Editor: Gaurav Gala
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Joshua Misquitta

First published: August 2015
Second edition: September 2019

Production reference: 1050919

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

ISBN 978-1-78953-373-6

www.packt.com

To my family.



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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.

Contributors

About the author

Giancarlo Zaccone has over fifteen years' experience of managing research projects in the scientific and industrial domains. He is a software and systems engineer at the European Space Agency (ESTEC), where he mainly deals with the cybersecurity of satellite navigation systems.

Giancarlo holds a master's degree in physics and an advanced master's degree in scientific computing.

Giancarlo has already authored the following titles, available from Packt: *Python Parallel Programming Cookbook (First Edition)*, *Getting Started with TensorFlow*, *Deep Learning with TensorFlow (First Edition)*, and *Deep Learning with TensorFlow (Second Edition)*.

About the reviewer

Dr. Michael Galloy is a software developer focusing on high-performance computing and visualization in scientific programming. He works mostly in IDL, but occasionally uses Python, C, and CUDA. Michael currently works for the National Center for Atmospheric Research (NCAR) at the Mauna Loa Solar Observatory. Previously, he worked for Tech-X Corporation, where he was the main developer of GPULib, a library of IDL bindings for GPU-accelerated computation routines. He is the creator and main developer for IDLdoc, mgunit, and rIDL, all of which are open source projects, as well as the author of *Modern IDL*.

Richard Marsden has 25 years of professional software development experience. After starting in the field of geophysical surveying for the oil industry, he has spent the last 15 years running the Winwaed Software Technology LLC, an independent software vendor. Winwaed specializes in geospatial tools and applications including web applications and operates the Mapping-Tools website for tools and add-ins for geospatial applications such as Caliper Maptitude, Microsoft MapPoint, Android, and Ultra Mileage.

Richard has been a technical reviewer for a number of Packt publications, including *Python Geospatial Development* and *Python Geospatial Analysis Essentials*, both by Erik Westra; and *Python Geospatial Analysis Cookbook*, by Michael Diener.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Getting Started with Parallel Computing and Python	8
Why do we need parallel computing?	9
Flynn's taxonomy	10
Single Instruction Single Data (SISD)	10
Multiple Instruction Single Data (MISD)	13
Single Instruction Multiple Data (SIMD)	13
Multiple Instruction Multiple Data (MIMD)	14
Memory organization	15
Shared memory	16
Distributed memory	18
Massively Parallel Processing (MPP)	19
Clusters of workstations	20
Heterogeneous architectures	20
Parallel programming models	22
Shared memory model	22
Multithread model	23
Message passing model	23
Data-parallel model	24
Designing a parallel program	25
Task decomposition	26
Task assignment	26
Agglomeration	26
Mapping	27
Dynamic mapping	27
Evaluating the performance of a parallel program	28
Speedup	29
Efficiency	29
Scaling	30
Amdahl's law	30
Gustafson's law	31
Introducing Python	31
Help functions	32
Syntax	33
Comments	35
Assignments	35
Data types	35
Strings	37
Flow control	37
Functions	39

Classes	40
Exceptions	41
Importing libraries	42
Managing files	42
List comprehensions	43
Running Python scripts	43
Installing Python packages using pip	44
Installing pip	44
Updating pip	44
Using pip	44
Introducing Python parallel programming	45
Processes and threads	45
Chapter 2: Thread-Based Parallelism	50
What is a thread?	51
Python threading module	52
Defining a thread	53
Getting ready	53
How to do it...	54
How it works...	55
There's more...	55
Determining the current thread	55
Getting ready	55
How to do it...	56
How it works...	57
Defining a thread subclass	57
Getting ready	57
How to do it...	58
How it works...	59
There's more...	60
Thread synchronization with a lock	61
Getting ready	61
How to do it...	62
How it works...	63
There's more...	64
Thread synchronization with RLock	65
Getting ready	66
How to do it...	66
How it works...	68
There's more...	68
Thread synchronization with semaphores	69
Getting ready	69
How to do it...	69
How it works...	71
There's more...	72

Thread synchronization with a condition	73
Getting ready	73
How to do it...	73
How it works...	75
There's more...	77
Thread synchronization with an event	77
Getting ready	77
How to do it...	78
How it works...	80
Thread synchronization with a barrier	81
Getting ready	81
How to do it...	81
How it works...	82
Thread communication using a queue	83
Getting ready	83
How to do it...	83
How it works...	84
There's more...	86
Chapter 3: Process-Based Parallelism	87
Understanding Python's multiprocessing module	88
Spawning a process	88
Getting ready	88
How to do it...	89
How it works...	89
There's more...	90
See also	91
Naming a process	91
Getting ready	92
How to do it...	92
How it works...	93
There's more...	93
See also	93
Running processes in the background	93
Getting ready	94
How to do it...	94
How it works...	95
See also	96
Killing a process	96
Getting ready	97
How to do it...	97
How it works...	98
See also	98
Defining processes in a subclass	99
Getting ready	99

How to do it...	99
How it works...	100
There's more...	101
See also	101
Using a queue to exchange data	101
Getting ready	101
How to do it...	102
How it works...	103
There's more...	104
See also	105
Using pipes to exchange objects	105
Getting ready	105
How to do it...	105
How it works...	107
There's more...	108
See also	108
Synchronizing processes	108
Getting ready	109
How to do it...	109
How it works...	110
There's more...	111
See also	112
Using a process pool	112
Getting ready	112
How to do it...	113
How it works...	114
There's more...	114
See also	115
Chapter 4: Message Passing	116
Technical requirements	116
Understanding the MPI structure	117
Using the mpi4py Python module	119
How to do it...	120
How it works...	120
There's more...	121
See also	122
Implementing point-to-point communication	122
How to do it...	122
How it works...	124
There's more...	126
See also	126
Avoiding deadlock problems	126
How to do it...	127
How it works...	127

There's more...	130
See also	130
Collective communication using a broadcast	131
Getting ready	131
How to do it...	132
How it works...	132
There's more...	133
See also	134
Collective communication using the scatter function	134
How to do it...	135
How it works...	135
There's more...	137
See also	137
Collective communication using the gather function	137
Getting ready	137
How to do it...	138
How it works...	139
There's more...	140
See also	140
Collective communication using Alltoall	140
How to do it...	140
How it works...	141
There's more...	142
See also	142
The reduction operation	143
Getting ready	143
How to do it...	143
How it works...	144
There's more...	145
See also	146
Optimizing communication	146
How to do it...	147
How it works...	148
There's more...	150
See also	152
Chapter 5: Asynchronous Programming	153
Using the concurrent.futures Python module	154
Getting ready	155
How to do it...	155
How it works...	157
There's more...	159
See also	160
Managing the event loop with asyncio	160
Understanding event loops	160

How to do it...	162
How it works...	164
There's more...	165
See also	166
Handling coroutines with asyncio	166
Getting ready	167
How to do it...	167
How it works...	171
There's more...	172
See also	172
Manipulating tasks with asyncio	173
How to do it...	174
How it works...	175
There's more...	176
See also	177
Dealing with asyncio and futures	177
Getting ready	177
How to do it...	177
How it works...	179
There's more...	181
See also	181
Chapter 6: Distributed Python	182
Introducing distributed computing	182
Types of distributed applications	183
Client-server applications	183
Client-server architecture	184
Client-server communications	185
TCP/IP client-server architecture	185
Multi-level applications	186
Using the Python socket module	187
Getting ready	187
How to do it...	188
How it works...	190
There's more...	191
Types of sockets	193
Stream sockets	193
See also	194
Distributed task management with Celery	194
Getting ready	196
Windows setup	197
How to do it...	197
How it works...	198
There's more...	200
See also	202
RMI with Pyro4	202

Getting ready	203
How to do it...	204
How it works...	205
There's more...	207
Implementing chain topology	208
See also	210
Chapter 7: Cloud Computing	211
What is cloud computing?	212
Understanding the cloud computing architecture	214
Service models	215
SaaS	215
PaaS	216
IaaS	216
Distribution models	216
Public cloud	217
Private cloud	217
Cloud community	217
Hybrid cloud	217
Cloud computing platforms	218
Developing web applications with PythonAnywhere	218
Getting ready	219
How to do it...	222
How it works...	226
There's more...	228
See also	229
Dockerizing a Python application	230
Getting ready	230
Installing Docker for Windows	231
How to do it...	232
How it works...	233
There's more...	234
See also	236
Introducing serverless computing	236
Getting ready	237
How to do it...	238
How it works...	246
There's more...	246
What is a Lambda function?	247
Why serverless?	247
Possible problems and limitations	248
See also	249
Chapter 8: Heterogeneous Computing	250
Understanding heterogeneous computing	251
Understanding the GPU architecture	251
Understanding GPU programming	252

CUDA	253
OpenCL	253
Dealing with PyCUDA	254
Getting ready	254
How to do it...	254
How it works...	255
There's more...	256
See also	257
Heterogeneous programming with PyCUDA	257
How to do it...	259
How it works...	260
There's more...	262
See also	263
Implementing memory management with PyCUDA	263
Getting ready	264
How to do it...	265
How it works...	268
There's more...	271
See also	272
Introducing PyOpenCL	272
Getting ready	272
How to do it...	273
How it works...	274
There's more...	275
See also	276
Building applications with PyOpenCL	276
How to do it...	278
How it works...	279
There's more...	282
See also	282
Element-wise expressions with PyOpenCL	283
Getting started	283
How to do it...	283
How it works...	284
There's more...	286
See also	287
Evaluating PyOpenCL applications	287
Getting started	287
How to do it...	287
How it works...	290
There's more...	292
Pros of OpenCL and PyOpenCL	292
Cons of OpenCL and PyOpenCL	292
Pros of CUDA and PyCUDA	293
Cons of CUDA and PyCUDA	293

See also	293
GPU programming with Numba	293
Getting ready	294
How to do it...	295
How it works...	297
There's more...	299
See also	301
Chapter 9: Python Debugging and Testing	302
What is debugging?	303
What is software testing?	305
Debugging using Winpdb Reborn	305
Getting ready	306
How to do it...	306
How it works...	312
There's more...	313
See also	314
Interacting with pdb	314
Getting ready	315
Interacting with the command line	315
Using the Python interpreter	316
Inserting a directive in the code to debug	316
How to do it...	317
How it works...	318
There's more...	318
See also	319
Implementing rpdb for debugging	319
Getting ready	319
How to do it...	321
How it works...	323
There's more...	324
See also	325
Dealing with unittest	326
Getting ready	326
How to do it...	327
How it works...	327
There's more...	329
See also	331
Application testing using nose	331
Getting ready	331
How to do it...	332
How it works...	333
There's more...	333
See also	334
Other Books You May Enjoy	335

Index	338
--------------	-----

Preface

The computing industry is characterized by the search for ever-increasing and efficient performance, from high-end applications in the sectors of networking, telecommunications, avionics, to low-power embedded systems in desktop computers, laptops, and video games. This development path has led to multicore systems, where two-, four-, and eight-core processors represent only the beginning of an upcoming expansion to an ever-increasing number of computing cores.

This expansion, however, creates a challenge, not only in the semiconductor industry but also in the development of applications that can be performed with parallel calculations.

Parallel computing, in fact, represents the simultaneous use of multiple computing resources to solve a processing problem, so that it can be executed on multiple CPUs, breaking a problem into discrete parts that can be processed simultaneously, where each is further divided into a series of instructions that can be executed serially on different CPUs.

Computing resources can include a single computer with multiple processors, an arbitrary number of computers connected via a network, or a combination of both approaches. Parallel computing has always been considered the extreme apex or future of computing, and up until a few years ago, it was motivated by numerical simulations of complex systems and situations concerning various sectors: weather and climate forecasts, chemical and nuclear reactions, human genome mapping, seismic and geological activity, the behavior of mechanical devices (from prostheses to space shuttles), electronic circuits, and manufacturing processes.

Today, however, ever more commercial applications are increasingly demanding the development of ever-faster computers to support the processing of large amounts of data in sophisticated ways. Applications for this include data mining and parallel databases, oil exploration, web search engines, and services networked business, computer-aided medical diagnoses, the management of multinational companies, advanced graphics and virtual reality (especially in the video game industry), multimedia and video network technologies, and collaborative work environments.

Last but not least, parallel computing represents an attempt to maximize that infinite, but at the same time, increasingly precious and scarce resource of time. This is why parallel computing is shifting from the world of very expensive supercomputers, reserved for a select few, to more economic and solutions based on multiple processors, **Graphics Processing Units (GPUs)**, or a few interconnected computers that can overcome the constraints of serial computing and the limits of single CPUs.

To introduce the concepts of parallel programming, one of the most popular programming languages has been adopted—**Python**. Python's popularity is partly due to its flexibility since it is a language used regularly by web and desktop developers, sysadmin and code developers, and more recently, by data scientists and machine learning engineers.

From a technological point of view, in Python, there is no separate compilation phase (as happens in C, for example) that generates an executable file starting from the source. The fact that it is pseudo-interpreted makes Python a portable language. Once a source is written, it can be interpreted and executed on most of the platforms currently used, whether they are from Apple (macOS X) or PC (Microsoft Windows and GNU/Linux).

Another strength of Python is its *ease of learning*. Anyone can learn to use it over a couple of days and write their first application. In this context, the open structure of the language plays a fundamental role, without redundant declarations and thus extremely similar to a spoken language. Finally, Python is free software: not only are the Python interpreter and the use of Python in our applications available for free, but Python can also be freely modified and thus redistributed according to the rules of a fully open source license.

Python Parallel Programming Cookbook, Second Edition, contains a wide variety of examples that offer to the reader the opportunity to solve real problems. It examines the principles of software design for parallel architectures, insisting on the importance of clarity of the programs, and avoids the use of complex terminology in favor of clear and direct examples.

Each topic is presented as part of a complete, working Python program, always followed by the output of the program in question. The modular organization of the various chapters provides a proven path along which to move from the simplest arguments to the most advanced, but it is also suitable for those who want to learn only a few specific issues.

Who this book is for

Python Parallel Programming Cookbook, Second Edition, is intended for software developers who want to use parallel programming techniques to write powerful and efficient code. Reading this book will enable you to master both the basics and the advanced aspects of parallel computing.

The Python programming language is easy to use and allows non-experts to tackle and understand the topics outlined in this book with ease.

What this book covers

Chapter 1, *Getting Started with Parallel Computing and Python*, provides an overview of parallel programming architectures and programming models. The chapter introduces the Python programming language, discussing how the characteristics of the language, its ease of learning and use, its extensibility, and the richness of the available software libraries and applications all make Python a valuable tool for any application, and especially, of course, for parallel computing.

Chapter 2, *Thread-Based Parallelism*, discusses thread parallelism using the `threading` Python module. Readers will learn, through full programming examples, how to synchronize and manipulate threads to implement in their multithreading applications.

Chapter 3, *Process-Based Parallelism*, guides the reader through the process-based approach to parallelizing a program. A complete set of examples will show readers how to use the `multiprocessing` Python module.

Chapter 4, *Message Passing*, is focused on message-passing exchange communication systems. In particular, the `mpi4py` library will be described with a lot of application examples.

Chapter 5, *Asynchronous Programming*, explains the asynchronous model for concurrent programming. In some ways, it is simpler than the threaded one because there is a single instruction stream, and tasks explicitly relinquish control instead of being suspended arbitrarily. The chapter shows readers how to use the `asyncio` module to organize each task as a sequence of smaller steps that must be executed in an asynchronous manner.

Chapter 6, *Distributed Python*, introduces the reader to distributed computing, which is the process of aggregating several computing units to collaboratively run a single computational task in a transparent and coherent way. In particular, the example applications provided in the chapter describe the use of the `socket` and `Celery` modules to manage distributed tasks.

Chapter 7, *Cloud Computing*, provides an overview of the main cloud-computing technologies in relation to the Python programming language. The **PythonAnywhere** platform is very useful for deploying Python applications on the cloud, and will be examined in this chapter. This chapter also contains example applications demonstrating the use of **containers** and **serverless** technologies.

Chapter 8, *Heterogeneous Computing*, looks at the modern GPUs that are providing breakthrough performance for numerical computing at the cost of increased programming complexity. In fact, the programming models for GPUs require that the coder manually manage the data transfer between the CPU and GPU. This chapter will teach the reader, using programming examples and use cases, how to exploit the computing power provided by GPU cards using the powerful Python modules of **PyCUDA**, **Numba**, and **PyOpenCL**.

Chapter 9, *Python Debugging and Testing*, is the last chapter that introduces two important topics on software engineering: debugging and testing. In particular, the following Python frameworks will be described: `windbg-reborn` for debugging, and `unittest` and `nose` for software testing.

To get the most out of this book

This book is *self-contained*: the only fundamental requirement before starting to read is a passion for programming and a curiosity for the topics covered in the book.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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
- Zipex/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/Python-Parallel-Programming-Cookbook-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781789533736_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "It's possible to kill a process immediately by using the `terminate` method."

A block of code is set as follows:

```
import socket
port=60000
s =socket.socket()
host=socket.gethostname()
```

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

```
p = multiprocessing.Process(target=foo)
print ('Process before execution:', p, p.is_alive())
p.start()
```

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

```
> python server.py
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Go to **System Properties** | **Environment Variables** | **User or System variables** | **New**."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

Getting Started with Parallel Computing and Python

The *parallel* and *distributed computing* models are based on the simultaneous use of different processing units for program execution. Although the distinction between parallel and distributed computing is very thin, one of the possible definitions associates the parallel calculation model with the shared memory calculation model, and the distributed calculation model with the message passing model.

From this point onward, we will use the term *parallel computing* to refer to both parallel and distributed calculation models.

The next sections provide an overview of parallel programming architectures and programming models. These concepts are useful for inexperienced programmers who are approaching parallel programming techniques for the first time. Moreover, it can be a basic reference for experienced programmers. The dual characterization of parallel systems is also presented. The first characterization is based on the system architecture, while the second characterization is based on parallel programming paradigms.

The chapter ends with a brief introduction to the Python programming language. The characteristics of the language, ease of use and learning, and the extensibility and richness of software libraries and applications make Python a valuable tool for any application, and also for parallel computing. The concepts of threads and processes are introduced in relation to their use in the language.

In this chapter, we will cover the following recipes:

- Why do we need parallel computing?
- Flynn's taxonomy
- Memory organization
- Parallel programming models
- Evaluating performance
- Introducing Python
- Python and parallel programming
- Introducing processes and threads

Why do we need parallel computing?

The growth in computing power made available by modern computers has resulted in us facing computational problems of increasing complexity in relatively short time frames. Until the early 2000s, complexity was dealt with by increasing the number of transistors as well as the clock frequency of single-processor systems, which reached peaks of 3.5-4 GHz. However, the increase in the number of transistors causes the exponential increase of the power dissipated by the processors themselves. In essence, there is, therefore, a physical limitation that prevents further improvement in the performance of single-processor systems.

For this reason, in recent years, microprocessor manufacturers have focused their attention on *multi-core* systems. These are based on a core of several physical processors that share the same memory, thus bypassing the problem of dissipated power described earlier. In recent years, *quad-core* and *octa-core* systems have also become standard on normal desktop and laptop configurations.

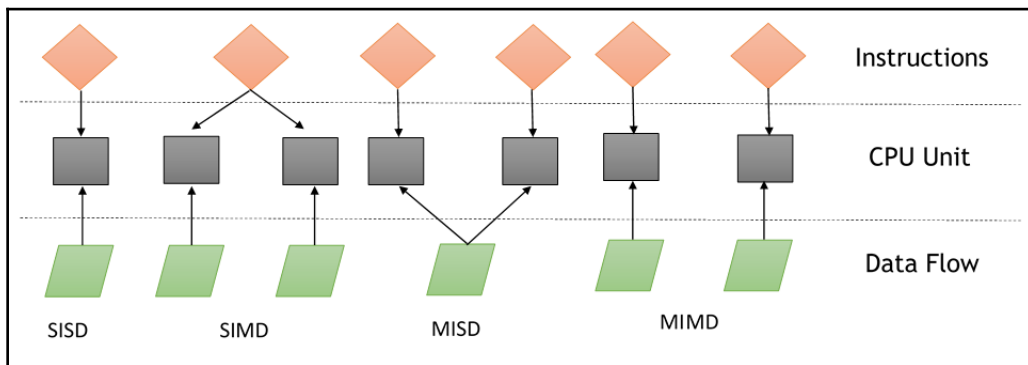
On the other hand, such a significant change in hardware has also resulted in an evolution of software structure, which has always been designed to be executed sequentially on a single processor. To take advantage of the greater computational resources made available by increasing the number of processors, the existing software must be redesigned in a form appropriate to the parallel structure of the CPU, so as to obtain greater efficiency through the simultaneous execution of the single units of several parts of the same program.

Flynn's taxonomy

Flynn's taxonomy is a system for classifying computer architectures. It is based on two main concepts:

- **Instruction flow:** A system with n CPU has n program counters and, therefore, n instructions flows. This corresponds to a program counter.
- **Data flow:** A program that calculates a function on a list of data has a data flow. The program that calculates the same function on several different lists of data has more data flows. This is made up of a set of operands.

As the instruction and data flows are independent, there are four categories of parallel machines: **Single Instruction Single Data (SISD)**, **Single Instruction Multiple Data (SIMD)**, **Multiple Instruction Single Data (MISD)**, and **Multiple Instruction Multiple Data (MIMD)**:



Flynn's taxonomy

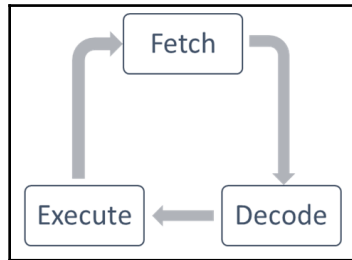
Single Instruction Single Data (SISD)

The SISD computing system is like the von Neumann machine, which is a uniprocessor machine. As you can see in *Flynn's taxonomy* diagram, it executes a single instruction that operates on a single data stream. In SISD, machine instructions are processed sequentially.

In a clock cycle, the CPU executes the following operations:

- **Fetch:** The CPU fetches the data and instructions from a memory area, which is called a *register*.
- **Decode:** The CPU decodes the instructions.
- **Execute:** The instruction is carried out on the data. The result of the operation is stored in another register.

Once the execution stage is complete, the CPU sets itself to begin another CPU cycle:



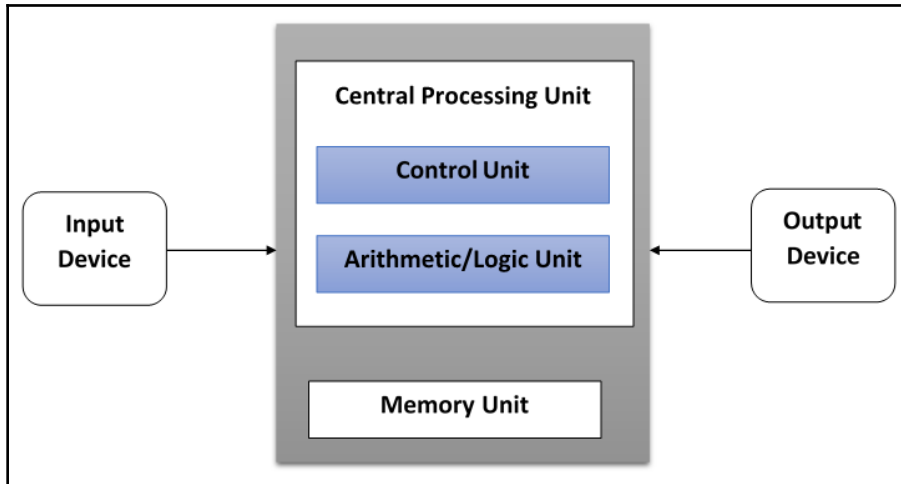
The fetch, decode, and execute cycle

The algorithms that run on this type of computer are sequential (or serial) since they do not contain any parallelism. An example of a SISD computer is a hardware system with a single CPU.

The main elements of these architectures (namely, von Neumann architectures) are as follows:

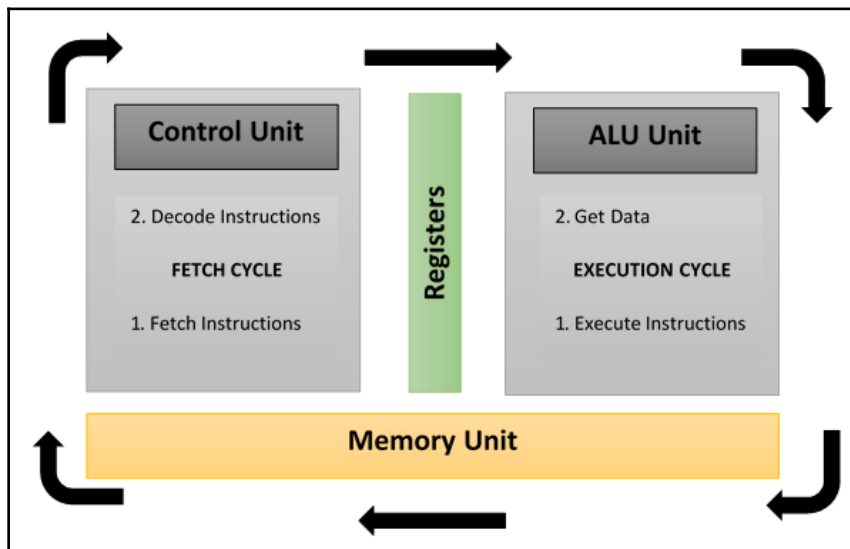
- **Central memory unit:** This is used to store both instructions and program data.
- **CPU:** This is used to get the instruction and/or data from the memory unit, which decodes the instructions and sequentially implements them.
- **The I/O system:** This refers to the input and output data of the program.

Conventional single-processor computers are classified as SISD systems:



The SISD architecture schema

The following diagram specifically shows which areas of a CPU are used in the stages of fetch, decode, and execute:



CPU components in the fetch-decode-execute phase

Multiple Instruction Single Data (MISD)

In this model, n processors, each with their own control unit, share a single memory unit. In each clock cycle, the data received from the memory is processed by all processors simultaneously, each in accordance with the instructions received from its control unit.

In this case, the parallelism (instruction-level parallelism) is obtained by performing several operations on the same piece of data. The types of problems that can be solved efficiently in these architectures are rather special, such as data encryption. For this reason, the MISD computer has not found space in the commercial sector. MISD computers are more of an intellectual exercise than a practical configuration.

Single Instruction Multiple Data (SIMD)

A SIMD computer consists of n identical processors, each with their own local memory, where it is possible to store data. All processors work under the control of a single instruction stream. In addition to this, there are n data streams, one for each processor. The processors work simultaneously on each step and execute the same instructions, but on different data elements. This is an example of data-level parallelism.

The SIMD architectures are much more versatile than MISD architectures. Numerous problems covering a wide range of applications can be solved by parallel algorithms on SIMD computers. Another interesting feature is that the algorithms for these computers are relatively easy to design, analyze, and implement. The limitation is that only the problems that can be divided into a number of subproblems (which are all identical, each of which will then be solved simultaneously through the same set of instructions) can be addressed with the SIMD computer.

With the supercomputer developed according to this paradigm, we must mention the *Connection Machine* (Thinking Machine, 1985) and *MPP* (NASA, 1983).

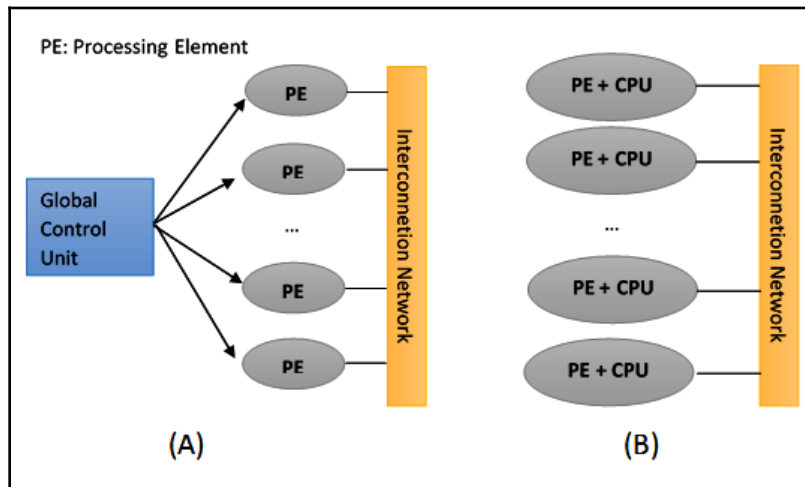
As we will see in Chapter 6, *Distributed Python*, and Chapter 7, *Cloud Computing*, the advent of modern graphics cards (GPUs), built with many SIMD-embedded units, has led to the more widespread use of this computational paradigm.

Multiple Instruction Multiple Data (MIMD)

This class of parallel computers is the most general and most powerful class, according to Flynn's classification. This contains n processors, n instruction streams, and n data streams. Each processor has its own control unit and local memory, which makes MIMD architectures more computationally powerful than SIMD architectures.

Each processor operates under the control of a flow of instructions issued by its own control unit. Therefore, the processors can potentially run different programs with different data, which allows them to solve subproblems that are different and can be a part of a single larger problem. In MIMD, the architecture is achieved with the help of the parallelism level with threads and/or processes. This also means that the processors usually operate asynchronously.

Nowadays, this architecture is applied to many PCs, supercomputers, and computer networks. However, there is a counter that you need to consider: asynchronous algorithms are difficult to design, analyze, and implement:



The SIMD architecture (A) and the MIMD architecture (B)

Flynn's taxonomy can be extended by considering that SIMD machines can be divided into two subgroups:

- Numerical supercomputers
- Vectorial machines

On the other hand, MIMD can be divided into machines that have a shared memory and those that have a distributed memory.

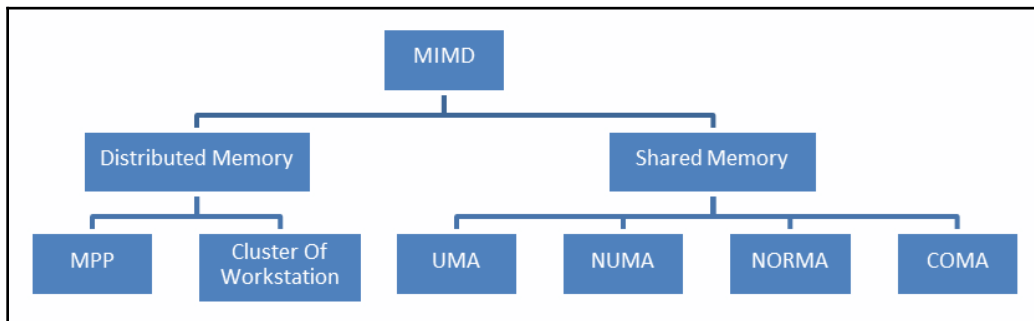
Indeed the next section focuses on this last aspect of the organization of the memory of MIMD machines.

Memory organization

Another aspect that we need to consider in order to evaluate parallel architectures is memory organization, or rather, the way in which data is accessed. No matter how fast the processing unit is, if memory cannot maintain and provide instructions and data at a sufficient speed, then there will be no improvement in performance.

The main problem that we need to overcome to make the response time of memory compatible with the speed of the processor is the memory cycle time, which is defined as the time that has elapsed between two successive operations. The cycle time of the processor is typically much shorter than the cycle time of memory.

When a processor initiates a transfer to or from memory, the processor's resources will remain occupied for the entire duration of the memory cycle; furthermore, during this period, no other device (for example, I/O controller, processor, or even the processor that made the request) will be able to use the memory due to the transfer in progress:



Memory organization in the MIMD architecture

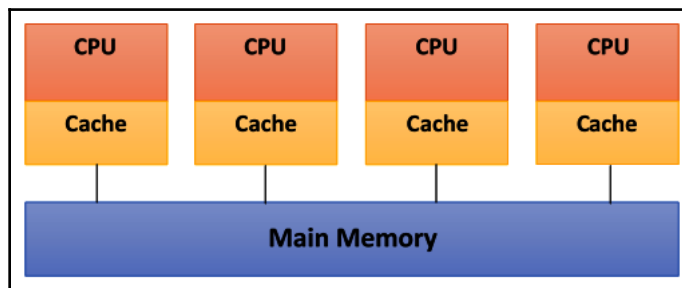
Solutions to the problem of memory access have resulted in a dichotomy of MIMD architectures. The first type of system, known as the *shared memory* system, has high virtual memory and all processors have equal access to data and instructions in this memory. The other type of system is the *distributed memory* model, wherein each processor has local memory that is not accessible to other processors.

What distinguishes memory shared by distributed memory is the management of memory access, which is performed by the processing unit; this distinction is very important for programmers because it determines how different parts of a parallel program must communicate.

In particular, a distributed memory machine must make copies of shared data in each local memory. These copies are created by sending a message containing the data to be shared from one processor to another. A drawback of this memory organization is that, sometimes, these messages can be very large and take a relatively long time to transfer, while in a shared memory system, there is no exchange of messages, and the main problem lies in synchronizing access to shared resources.

Shared memory

The schema of a shared memory multiprocessor system is shown in the following diagram. The physical connections here are quite simple:



Shared memory architecture schema

Here, the bus structure allows an arbitrary number of devices (**CPU + Cache** in the preceding diagram) that share the same channel (**Main Memory**, as shown in the preceding diagram). The bus protocols were originally designed to allow a single processor and one or more disks or tape controllers to communicate through the shared memory here.



Each processor has been associated with cache memory, as it is assumed that the probability that a processor needs to have data or instructions present in the local memory is very high.

The problem occurs when a processor modifies data stored in the memory system that is simultaneously used by other processors. The new value will pass from the processor cache that has been changed to the shared memory. Later, however, it must also be passed to all the other processors, so that they do not work with the obsolete value. This problem is known as the problem of *cache coherency*—a special case of the problem of memory consistency, which requires hardware implementations that can handle concurrency issues and synchronization, similar to that of thread programming.

The main features of shared memory systems are as follows:

- The memory is the same for all processors. For example, all the processors associated with the same data structure will work with the same logical memory addresses, thus accessing the same memory locations.
- The synchronization is obtained by reading the tasks of various processors and allowing the shared memory. In fact, the processors can only access one memory at a time.
- A shared memory location must not be changed from a task while another task accesses it.
- Sharing data between tasks is fast. The time required to communicate is the time that one of them takes to read a single location (depending on the speed of memory access).

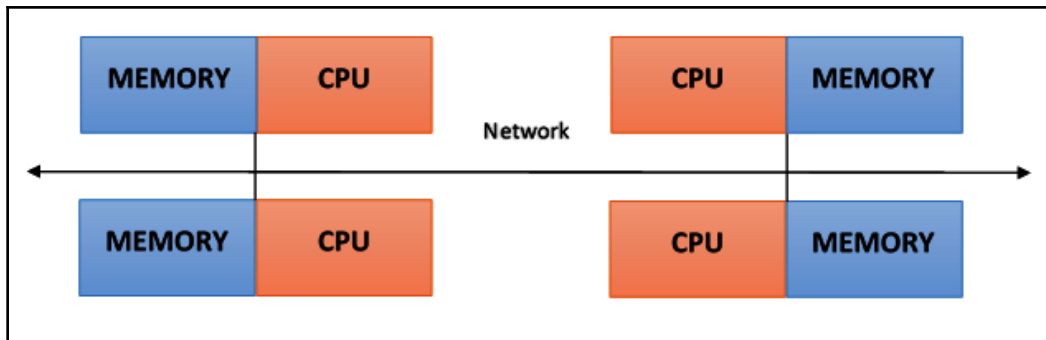
The memory access in shared memory systems is as follows:

- **Uniform Memory Access (UMA):** The fundamental characteristic of this system is the access time to the memory that is constant for each processor and for any area of memory. For this reason, these systems are also called **Symmetric Multiprocessors (SMPs)**. They are relatively simple to implement, but not very scalable. The coder is responsible for the management of the synchronization by inserting appropriate controls, semaphores, locks, and more in the program that manages resources.
- **Non-Uniform Memory Access (NUMA):** These architectures divide the memory into high-speed access area that is assigned to each processor, and also, a common area for the data exchange, with slower access. These systems are also called **Distributed Shared Memory (DSM)** systems. They are very scalable, but complex to develop.
- **No Remote Memory Access (NoRMA):** The memory is physically distributed among the processors (local memory). All local memories are private and can only access the local processor. The communication between the processors is through a communication protocol used for exchanging messages, which is known as the *message-passing protocol*.

- **Cache-Only Memory Architecture (COMA):** These systems are equipped with only cached memories. While analyzing NUMA architectures, it was noticed that this architecture kept the local copies of the data in the cache and that this data was stored as duplicates in the main memory. This architecture removes duplicates and keeps only the cached memories; the memory is physically distributed among the processors (local memory). All local memories are private and can only access the local processor. The communication between the processors is also through the message-passing protocol.

Distributed memory

In a system with distributed memory, the memory is associated with each processor and a processor is only able to address its own memory. Some authors refer to this type of system as a multicomputer, reflecting the fact that the elements of the system are, themselves, small and complete systems of a processor and memory, as you can see in the following diagram:

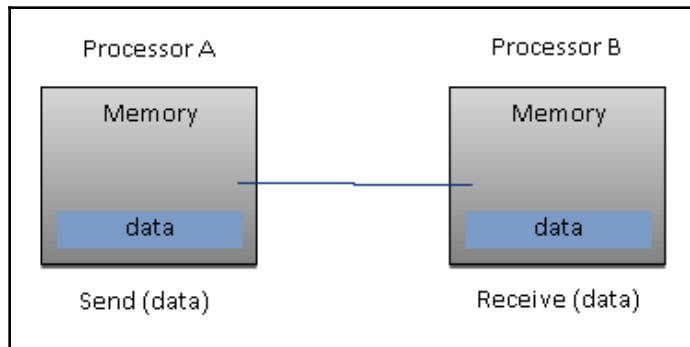


The distributed memory architecture schema

This kind of organization has several advantages:

- There are no conflicts at the level of the communication bus or switch. Each processor can use the full bandwidth of their own local memory without any interference from other processors.
- The lack of a common bus means that there is no intrinsic limit to the number of processors. The size of the system is only limited by the network used to connect the processors.
- There are no problems with cache coherency. Each processor is responsible for its own data and does not have to worry about upgrading any copies.

The main disadvantage is that communication between processors is more difficult to implement. If a processor requires data in the memory of another processor, then the two processors should not necessarily exchange messages via the message-passing protocol. This introduces two sources of slowdown: to build and send a message from one processor to another takes time, and also, any processor should be stopped in order to manage the messages received from other processors. A program designed to work on a distributed memory machine must be organized as a set of independent tasks that communicate via messages:



Basic message passing

The main features of distributed memory systems are as follows:

- Memory is physically distributed between processors; each local memory is directly accessible only by its processor.
- Synchronization is achieved by moving data (even if it's just the message itself) between processors (communication).
- The subdivision of data in the local memories affects the performance of the machine—it is essential to make subdivisions accurate, so as to minimize the communication between the CPUs. In addition to this, the processor that coordinates these operations of decomposition and composition must effectively communicate with the processors that operate on the individual parts of data structures.
- The message-passing protocol is used so that the CPUs can communicate with each other through the exchange of data packets. The messages are discrete units of information, in the sense that they have a well-defined identity, so it is always possible to distinguish them from each other.

Massively Parallel Processing (MPP)

MPP machines are composed of hundreds of processors (which can be as large as hundreds of thousands of processors in some machines) that are connected by a communication network. The fastest computers in the world are based on these architectures; some examples of these architecture systems are Earth Simulator, Blue Gene, ASCI White, ASCI Red, and ASCI Purple and Red Storm.

Clusters of workstations

These processing systems are based on classical computers that are connected by communication networks. Computational clusters fall into this classification.

In a cluster architecture, we define a node as a single computing unit that takes part in the cluster. For the user, the cluster is fully transparent—all the hardware and software complexity is masked and data and applications are made accessible as if they were all from a single node.

Here, we've identified three types of clusters:

- **Fail-over cluster:** In this, the node's activity is continuously monitored, and when one stops working, another machine takes over the charge of those activities. The aim is to ensure a continuous service due to the redundancy of the architecture.
- **Load balancing cluster:** In this system, a job request is sent to the node that has less activity. This ensures that less time is taken to process the job.
- **High-performance computing cluster:** In this, each node is configured to provide extremely high performance. The process is also divided into multiple jobs on multiple nodes. The jobs are parallelized and will be distributed to different machines.

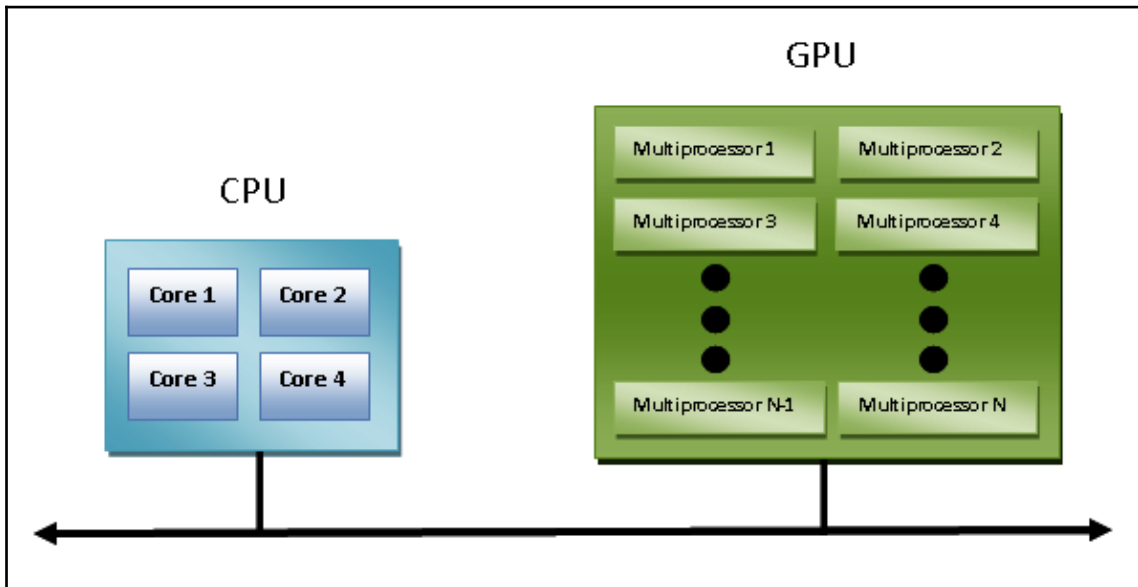
Heterogeneous architectures

The introduction of GPU accelerators in the homogeneous world of supercomputing has changed the nature of how supercomputers are both used and programmed now. Despite the high performance offered by GPUs, they cannot be considered as an autonomous processing unit as they should always be accompanied by a combination of CPUs. The programming paradigm, therefore, is very simple: the CPU takes control and computes in a serial manner, assigning tasks to the graphics accelerator that are, computationally, very expensive and have a high degree of parallelism.

The communication between a CPU and a GPU can take place, not only through the use of a high-speed bus but also through the sharing of a single area of memory for both physical or virtual memory. In fact, in the case where both the devices are not equipped with their own memory areas, it is possible to refer to a common memory area using the software libraries provided by the various programming models, such as *CUDA* and *OpenCL*.

These architectures are called *heterogeneous architectures*, wherein applications can create data structures in a single address space and send a job to the device hardware, which is appropriate for the resolution of the task. Several processing tasks can operate safely in the same regions to avoid data consistency problems, thanks to the atomic operations.

So, despite the fact that the CPU and GPU do not seem to work efficiently together, with the use of this new architecture, we can optimize their interaction with, and the performance of, parallel applications:



The heterogeneous architecture schema

In the following section, we introduce the main parallel programming models.

Parallel programming models

Parallel programming models exist as an abstraction of hardware and memory architectures. In fact, these models are not specific and do not refer to any particular types of machines or memory architectures. They can be implemented (at least theoretically) on any kind of machines. Compared to the previous subdivisions, these programming models are made at a higher level and represent the way in which the software must be implemented to perform parallel computation. Each model has its own way of sharing information with other processors in order to access memory and divide the work.

In absolute terms, no one model is better than the other. Therefore, the best solution to be applied will depend very much on the problem that a programmer should address and resolve. The most widely used models for parallel programming are as follows:

- Shared memory model
- Multithread model
- Distributed memory/message passing model
- Data-parallel model

In this recipe, we will give you an overview of these models.

Shared memory model

In this model, tasks share a single memory area in which we can read and write asynchronously. There are mechanisms that allow the coder to control the access to the shared memory; for example, locks or semaphores. This model offers the advantage that the coder does not have to clarify the communication between tasks. An important disadvantage, in terms of performance, is that it becomes more difficult to understand and manage data locality. This refers to keeping data local to the processor that works on conserving memory access, cache refreshes, and bus traffic that occurs when multiple processors use the same data.

Multithread model

In this model, a process can have multiple flows of execution. For example, a sequential part is created and, subsequently, a series of tasks are created that can be executed in parallel. Usually, this type of model is used on shared memory architectures. So, it will be very important for us to manage the synchronization between threads, as they operate on shared memory, and the programmer must prevent multiple threads from updating the same locations at the same time.

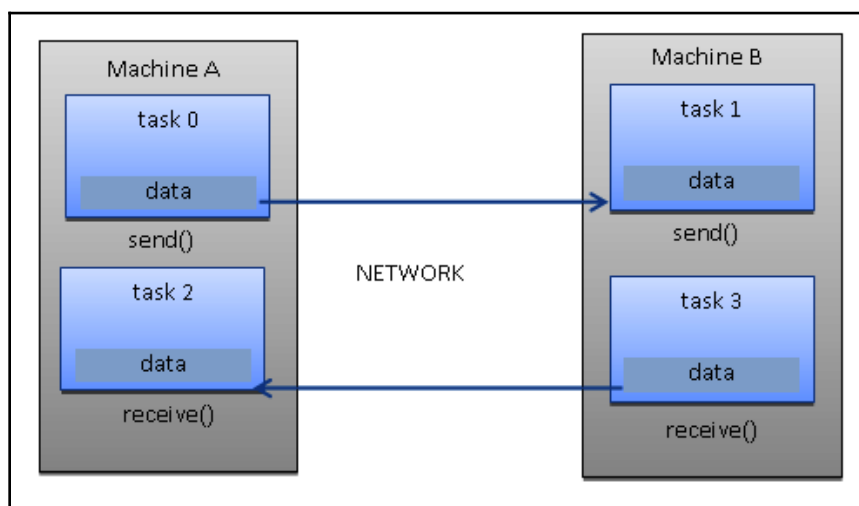
The current-generation CPUs are multithreaded in software and hardware. **POSIX** (short for **P**ortable **O**perating **S**ystem **I**nterface) threads are classic examples of the implementation of multithreading on software. Intel's Hyper-Threading technology implements multithreading on hardware by switching between two threads when one is stalled or waiting on I/O. Parallelism can be achieved from this model, even if the data alignment is nonlinear.

Message passing model

The message passing model is usually applied in cases where each processor has its own memory (distributed memory system). More tasks can reside on the same physical machine or on an arbitrary number of machines. The coder is responsible for determining the parallelism and data exchange that occurs through the messages, and it is necessary to request and call a library of functions within the code.

Some of the examples have been around since the 1980s, but only in the mid-1990s was a standardized model created, leading to a de facto standard called a **Message Passing Interface (MPI)**.

The MPI model is clearly designed with distributed memory, but being models of parallel programming, a multiplatform model can also be used with a shared memory machine:

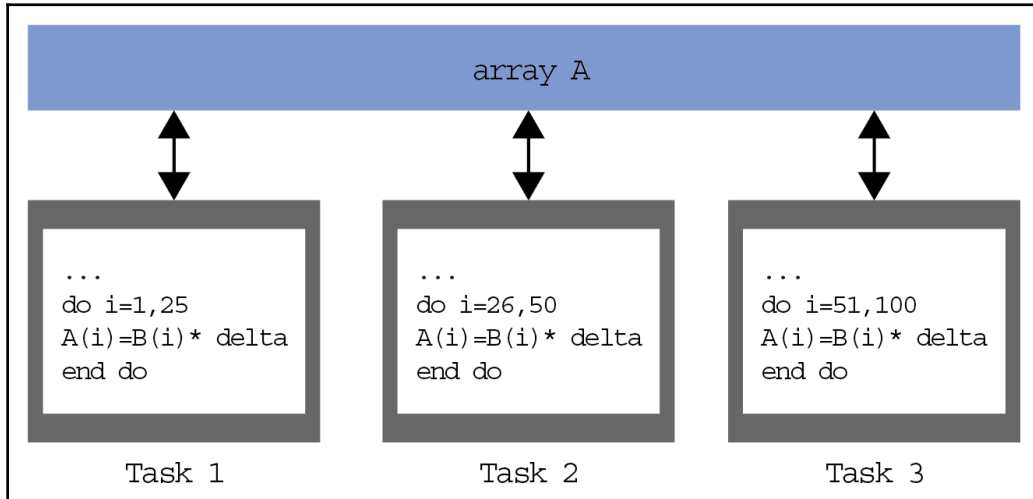


Message passing paradigm model

Data-parallel model

In this model, we have more tasks that operate on the same data structure, but each task operates on a different portion of data. In the shared memory architecture, all tasks have access to data through shared memory and distributed memory architectures, where the data structure is divided and resides in the local memory of each task.

To implement this model, a coder must develop a program that specifies the distribution and alignment of data; for example, the current-generation GPUs are highly operational only if data (**Task 1**, **Task 2**, **Task 3**) is aligned, as shown in the following diagram:



The data-parallel paradigm model

Designing a parallel program

The design of algorithms that exploit parallelism is based on a series of operations, which must be carried out for the program to perform the job correctly without producing partial or erroneous results. The macro operations that must be carried out for a correct parallelization of an algorithm are as follows:

- Task decomposition
- Task assignment
- Agglomeration
- Mapping

Task decomposition

In this first phase, the software program is split into tasks or a set of instructions that can then be executed on different processors to implement parallelism. To perform this subdivision, two methods are used:

- **Domain decomposition:** Here, the data of the problems is decomposed. The application is common to all the processors that work on different portions of data. This methodology is used when we have a large amount of data that must be processed.
- **Functional decomposition:** In this case, the problem is split into tasks, where each task will perform a particular operation on all the available data.

Task assignment

In this step, the mechanism by which the tasks will be distributed among the various processes is specified. This phase is very important because it establishes the distribution of workload among the various processors. Load balancing is crucial here; in fact, all processors must work with continuity, avoiding being in an idle state for a long time.

To perform this, the coder takes into account the possible heterogeneity of the system that tries to assign more tasks to better-performing processors. Finally, for greater efficiency of parallelization, it is necessary to limit communication as much as possible between processors, as they are often the source of slowdowns and consumption of resources.

Agglomeration

Agglomeration is the process of combining smaller tasks with larger ones in order to improve performance. If the previous two stages of the design process partitioned the problem into a number of tasks that greatly exceed the number of processors available, and if the computer is not specifically designed to handle a huge number of small tasks (some architectures, such as GPUs, handle this fine and indeed benefit from running millions, or even billions, of tasks), then the design can turn out to be highly inefficient.

Commonly, this is because tasks have to be communicated to the processor or thread so that they compute the said task. Most communications have costs that are disproportionate to the amount of data transferred, but also incur a fixed cost for every communication operation (such as the latency, which is inherent in setting up a TCP connection). If the tasks are too small, then this fixed cost can easily make the design inefficient.

Mapping

In the mapping stage of the parallel algorithm design process, we specify where each task is to be executed. The goal is to minimize the total execution time. Here, you must often make trade-offs, as the two main strategies often conflict with each other:

- The tasks that communicate frequently should be placed in the same processor to increase locality.
- The tasks that can be executed concurrently should be placed in different processors to enhance concurrency.

This is known as the *mapping problem*, and it is known to be **NP-complete**. As such, no polynomial-time solutions to the problem in the general case exist. For tasks of equal size and tasks with easily identified communication patterns, the mapping is straightforward (we can also perform agglomeration here to combine tasks that map to the same processor). However, if the tasks have communication patterns that are hard to predict or the amount of work varies per task, then it is hard to design an efficient mapping and agglomeration scheme.

For these types of problems, load balancing algorithms can be used to identify agglomeration and mapping strategies during runtime. The hardest problems are those in which the amount of communication or the number of tasks changes during the execution of the program. For these kinds of problems, dynamic load balancing algorithms can be used, which run periodically during the execution.

Dynamic mapping

Numerous load balancing algorithms exist for a variety of problems:

- **Global algorithms:** These require global knowledge of the computation being performed, which often adds a lot of overhead.
- **Local algorithms:** These rely only on information that is local to the task in question, which reduces overhead compared to global algorithms, but they are usually worse at finding optimal agglomeration and mapping.

However, the reduced overhead may reduce the execution time, even though the mapping is worse by itself. If the tasks rarely communicate other than at the start and end of the execution, then a task-scheduling algorithm is often used, which simply maps tasks to processors as they become idle. In a task-scheduling algorithm, a task pool is maintained. Tasks are placed in this pool and are taken from it by workers.

There are three common approaches in this model:

- **Manager/worker:** This is the basic dynamic mapping scheme in which all the workers connect to a centralized manager. The manager repeatedly sends tasks to the workers and collects the results. This strategy is probably the best for a relatively small number of processors. The basic strategy can be improved by fetching tasks in advance so that communication and computation overlap each other.
- **Hierarchical manager/worker:** This is the variant of a manager/worker that has a semi-distributed layout. Workers are split into groups, each with their own manager. These group managers communicate with the central manager (and possibly among themselves as well), while workers request tasks from the group managers. This spreads the load among several managers and can, as such, handle a larger number of processors if all workers request tasks from the same manager.
- **Decentralize:** In this scheme, everything is decentralized. Each processor maintains its own task pool and communicates with the other processors in order to request tasks. How the processors choose other processors to request tasks varies and is determined on the basis of the problem.

Evaluating the performance of a parallel program

The development of parallel programming created the need for performance metrics in order to decide whether its use is convenient or not. Indeed, the focus of parallel computing is to solve large problems in a relatively short period of time. The factors contributing to this objective are, for example, the type of hardware used, the degree of parallelism of the problem, and the parallel programming model adopted. To facilitate this, the analysis of basic concepts was introduced, which compares the parallel algorithm obtained from the original sequence.

The performance is achieved by analyzing and quantifying the number of threads and/or the number of processes used. To analyze this, let's introduce a few performance indexes:

- **Speedup**
- **Efficiency**
- **Scaling**

The limitations of parallel computation are introduced by **Amdahl's** law. To evaluate the *degree of efficiency* of the parallelization of a sequential algorithm, we have **Gustafson's** law.

Speedup

The **speedup** is the measure that displays the benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element (T_s) to the time required to solve the same problem on p identical processing elements (T_p).

We denote speedup as follows:

$$S = \frac{T_s}{T_p}$$

We have a linear speedup, where if $S=p$, then it means that the speed of execution increases with the number of processors. Of course, this is an ideal case. While the speedup is absolute when T_s is the execution time of the best sequential algorithm, the speedup is relative when T_s is the execution time of the parallel algorithm for a single processor.

Let's recap these conditions:

- $S = p$ is a linear or ideal speedup.
- $S < p$ is a real speedup.
- $S > p$ is a superlinear speedup.

Efficiency

In an ideal world, a parallel system with p processing elements can give us a speedup that is equal to p . However, this is very rarely achieved. Usually, some time is wasted in either idling or communicating. Efficiency is a measure of how much of the execution time a processing element puts toward doing useful work, given as a fraction of the time spent.

We denote it by E and can define it as follows:

$$E = \frac{S}{p} = \frac{T_s}{pT_p}$$

The algorithms with linear speedup have a value of $E = 1$. In other cases, they have the value of E is less than 1. The three cases are identified as follows:

- When $E = 1$, it is a linear case.
- When $E < 1$, it is a real case.
- When $E \ll 1$, it is a problem that is parallelizable with low efficiency.

Scaling

Scaling is defined as the ability to be efficient on a parallel machine. It identifies the computing power (speed of execution) in proportion to the number of processors. By increasing the size of the problem and, at the same time, the number of processors, there will be no loss in terms of performance.

The scalable system, depending on the increments of the different factors, may maintain the same efficiency or improve it.

Amdahl's law

Amdahl's law is a widely used law that is used to design processors and parallel algorithms. It states that the maximum speedup that can be achieved is limited by the serial component of the program:

$$S = \frac{1}{1 - P}$$



$1 - P$ denotes the serial component (not parallelized) of a program.

This means that, for example, if a program in which 90% of the code can be made parallel, but 10% must remain serial, then the maximum achievable speedup is 9, even for an infinite number of processors.

Gustafson's law

Gustafson's law states the following:

$$S(P) = P - \alpha(P - 1)$$

Here, as we indicated in the equation the following applies:

- P is the *number of processors*.
- S is the *speedup* factor.
- α is the *non-parallelizable fraction* of any parallel process.

Gustafson's law is in *contrast* to Amdahl's law, which, as we described, assumes that the overall workload of a program does not change with respect to the number of processors.

In fact, Gustafson's law suggests that programmers first set the *time* allowed for solving a problem in parallel and then based on that (that is time) *to size* the problem. Therefore, the *faster* the parallel system is, the *greater* the problems that can be solved over the same period of time.

The effect of Gustafson's law was to direct the objectives of computer research towards the selection or reformulation of problems in such a way that the solution of a larger problem would still be possible in the same amount of time. Furthermore, this law redefines the concept of *efficiency* as a need *to reduce at least the sequential part* of a program, despite the *increase in workload*.

Introducing Python

Python is a powerful, dynamic, and interpreted programming language that is used in a wide variety of applications. Some of its features are as follows:

- A clear and readable syntax.
- A very extensive standard library, where, through additional software modules, we can add data types, functions, and objects.
- Easy-to-learn rapid development and debugging. Developing Python code in Python can be up to 10 times faster than in C/C++ code. The code can also work as a prototype and then translated into C/C++.
- Exception-based error handling.
- A strong introspection functionality.
- The richness of documentation and a software community.

Python can be seen as a glue language. Using Python, better applications can be developed because different kinds of coders can work together on a project. For example, when building a scientific application, C/C++ programmers can implement efficient numerical algorithms, while scientists on the same project can write Python programs that test and use those algorithms. Scientists don't have to learn a low-level programming language and C/C++ programmers don't need to understand the science involved.



You can read more about this from <https://www.python.org/doc/essays/omg-darpa-mcc-position>.

Let's take a look at some examples of very basic code to get an idea of the features of Python.



The following section can be a refresher for most of you. We will use these techniques practically in Chapter 2, *Thread-Based Parallelism*, and Chapter 3, *Process-Based Parallelism*.

Help functions

The Python interpreter already provides a valid help system. If you want to know how to use an object, then just type `help(object)`.

Let's see, for example, how to use the `help` function on integer 0:

```
>>> help(0)
Help on int object:

class int(object)
| int(x=0) -> integer
| int(x, base=10) -> integer
|
| Convert a number or string to an integer, or return 0 if no
| arguments are given. If x is a number, return x.__int__(). For
| floating point numbers, this truncates towards zero.
|
| If x is not a number or if base is given, then x must be a string,
| bytes, or bytearray instance representing an integer literal in the
| given base. The literal can be preceded by '+' or '-' and be
| surrounded by whitespace. The base defaults to 10. Valid bases are 0
| and 2-36.
| Base 0 means to interpret the base from the string as an integer
```

```
| literal.
>>> int('0b100', base=0)
```

The description of the `int` object is followed by a list of methods that are applicable to it. The first five methods are as follows:

```
| Methods defined here:
|
| __abs__(self, /)
| abs(self)
|
| __add__(self, value, /)
| Return self+value.
|
| __and__(self, value, /)
| Return self&value.
|
| __bool__(self, /)
| self != 0
|
| __ceil__(...)
| Ceiling of an Integral returns itself.
```

Also useful is `dir(object)`, which lists the methods available for an object:

```
>>> dir(float)
['_abs_', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'_delattr_', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'_floor_', '__floordiv__', '__format__', '__ge__', '__getattr__',
'_getnewargs_', '__gt__', '__hash__', '__index__', '__init__', '__int__',
'_invert_', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'_ne_', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'_rand_', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'_rfloordiv_', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'_round_', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'_rtruediv_', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'_sub_', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

Finally, the relevant documentation for an object is provided by the `.__doc__` function, as shown in the following example:

```
>>> abs.__doc__
'Return the absolute value of the argument.'
```

Syntax

Python doesn't adopt statement terminators, and code blocks are specified through indentation. Statements that expect an indentation level must end in a colon (:). This leads to the following:

- The Python code is clearer and more readable.
- The program structure always coincides with that of the indentation.
- The style of indentation is uniform in any listing.

Bad indentation can lead to errors.

The following example shows how to use the `if` construct:

```
print("first print")
if condition:
    print("second print")
print("third print")
```

In this example, we can see the following:

- The following statements: `print("first print")`, `if condition:`, `print("third print")` have the same indentation level and are always executed.
- After the `if` statement, there is a block of code with a higher indentation level, which includes the `print ("second print")` statement.
- If the condition of `if` is true, then the `print ("second print")` statement is executed.
- If the condition of `if` is false, then the `print ("second print")` statement is not executed.

It is, therefore, very important to pay attention to indentation because it is always evaluated in the program parsing process.

Comments

Comments start with the hash sign (#) and are on a single line:

```
# single line comment
```

Multi-line strings are used for multi-line comments:

```
""" first line of a multi-line comment
second line of a multi-line comment."""
```

Assignments

Assignments are made with the equals symbol (=). For equality tests, the same amount (==) is used. You can increase and decrease a value using the += and -= operators, followed by an addendum. This works with many types of data, including strings. You can assign and use multiple variables on the same line.

Some examples are as follows:

```
>>> variable = 3
>>> variable += 2
>>> variable
5
>>> variable -= 1
>>> variable
4

>>> _string_ = "Hello"
>>> _string_ += " Parallel Programming CookBook Second Edition!"
>>> print (_string_)
Hello Parallel Programming CookBook Second Edition!
```

Data types

The most significant structures in Python are *lists*, *tuples*, and *dictionaries*. Sets have been integrated into Python since version 2.5 (the previous versions are available in the `sets` library):

- **Lists:** These are similar to one-dimensional arrays, but you can create lists that contain other lists.
- **Dictionaries:** These are arrays that contain key pairs and values (hash tables).
- **Tuples:** These are immutable mono-dimensional objects.

Arrays can be of any type, so you can mix variables such as integers and strings into your lists, dictionaries and tuples.

The index of the first object in any type of array is always zero. Negative indexes are allowed and count from the end of the array; -1 indicates the last element of the array:

```
#let's play with lists
list_1 = [1, ["item_1", "item_1"], ("a", "tuple")]
list_2 = ["item_1", -10000, 5.01]

>>> list_1
[1, ['item_1', 'item_1'], ('a', 'tuple')]

>>> list_2
['item_1', -10000, 5.01]

>>> list_1[2]
('a', 'tuple')

>>>list_1[1][0]
['item_1', 'item_1']

>>> list_2[0]
item_1

>>> list_2[-1]
5.01

#build a dictionary
dictionary = {"Key 1": "item A", "Key 2": "item B", 3: 1000}
>>> dictionary
{'Key 1': 'item A', 'Key 2': 'item B', 3: 1000}

>>> dictionary["Key 1"]
item A

>>> dictionary["Key 2"]
-1

>>> dictionary[3]
1000
```

You can get an array range using the colon (:):

```
list_3 = ["Hello", "Ruvika", "how" , "are" , "you?"]
>>> list_3[0:6]
['Hello', 'Ruvika', 'how', 'are', 'you?']

>>> list_3[0:1]
['Hello']

>>> list_3[2:6]
['how', 'are', 'you?']
```

Strings

Python strings are indicated using either the single (') or double (") quotation mark and they are allowed to use one notation within a string delimited by the other:

```
>>> example = "she loves ' giancarlo"
>>> example
"she loves ' giancarlo"
```

On multiple lines, they are enclosed in triple (or three single) quotation marks (''' multi-line string ''')

```
>>> _string_ = '''I am a
multi-line
string'''
>>> _string_
'I am a \nmulti-line\nstring'
```

Python also supports Unicode; just use the u "This is a unicode string" syntax:

```
>>> ustring = u"I am unicode string"
>>> ustring
'I am unicode string'
```

To enter values in a string, type the % operator and a tuple. Then, each % operator is replaced by a tuple element, from left to right:

```
>>> print ("My name is %s !" % ('Mr. Wolf'))
My name is Mr. Wolf!
```

Flow control

Flow control instructions are `if`, `for`, and `while`.

In the next example, we check whether the number is positive, negative, or zero and display the result:

```
num = 1

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

The following code block finds the sum of all the numbers stored in a list, using a `for` loop:

```
numbers = [6, 6, 3, 8, -3, 2, 5, 44, 12]
sum = 0
for val in numbers:
    sum = sum+val
print("The sum is", sum)
```

We will execute the `while` loop to iterate the code until the condition result is true. We will use this loop over the `for` loop since we are unaware of the number of iterations that will result in the code. In this example, we use `while` to add natural numbers up to $sum = 1+2+3+\dots+n$:

```
n = 10
# initialize sum and counter
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1 # update counter

# print the sum
print("The sum is", sum)
```

The outputs for the preceding three examples are as follows:

```
Positive number
The sum is 83
The sum is 55
>>>
```

Functions

Python functions are declared with the `def` keyword:

```
def my_function():  
    print("this is a function")
```

To run a function, use the function name, followed by parentheses, as follows:

```
>>> my_function()  
this is a function
```

Parameters must be specified after the function name, inside the parentheses:

```
def my_function(x):  
    print(x * 1234)  
  
>>> my_function(7)  
8638
```

Multiple parameters must be separated with a comma:

```
def my_function(x,y):  
    print(x*5+ 2*y)  
  
>>> my_function(7,9)  
53
```

Use the equals sign to define a default parameter. If you call the function without the parameter, then the default value will be used:

```
def my_function(x,y=10):  
    print(x*5+ 2*y)  
  
>>> my_function(1)  
25  
  
>>> my_function(1,100)  
205
```

The parameters of a function can be of any type of data (such as string, number, list, and dictionary). Here, the following list, `lcities`, is used as a parameter for `my_function`:

```
def my_function(cities):  
    for x in cities:  
        print(x)  
  
>>> lcities=["Napoli", "Mumbai", "Amsterdam"]
```



```
>>> my_function(lcities)
Napoli
Mumbai
Amsterdam
```

Use the `return` statement to return a value from a function:

```
def my_function(x,y):
    return x*y
```

```
>>> my_function(6,29)
174
```

Python supports an interesting syntax that allows you to define small, single-line functions on the fly. Derived from the Lisp programming language, these lambda functions can be used wherever a function is required.

An example of a lambda function, `functionvar`, is shown as follows:

```
# lambda definition equivalent to def f(x): return x + 1

functionvar = lambda x: x * 5
>>> print(functionvar(10))
50
```

Classes

Python supports multiple inheritances of classes. Conventionally (not a language rule), private variables and methods are declared by being preceded with two underscores (`__`). We can assign arbitrary attributes (properties) to the instances of a class, as shown in the following example:

```
class FirstClass:
    common_value = 10
    def __init__(self):
        self.my_value = 100
    def my_func(self, arg1, arg2):
        return self.my_value*arg1*arg2

# Build a first instance
>>> first_instance = FirstClass()
>>> first_instance.my_func(1, 2)
200

# Build a second instance of FirstClass
>>> second_instance = FirstClass()
```

```
#check the common values for both the instances
>>> first_instance.common_value
10

>>> second_instance.common_value
10

#Change common_value for the first_instance
>>> first_instance.common_value = 1500
>>> first_instance.common_value
1500

#As you can note the common_value for second_instance is not changed
>>> second_instance.common_value
10


# SecondClass inherits from FirstClass.
# multiple inheritance is declared as follows:
# class SecondClass (FirstClass1, FirstClass2, FirstClassN)

class SecondClass (FirstClass):
    # The "self" argument is passed automatically
    # and refers to the class's instance
    def __init__ (self, arg1):
        self.my_value = 764
        print (arg1)

>>> first_instance = SecondClass ("hello PACKT!!!!")
hello PACKT!!!!

>>> first_instance.my_func (1, 2)
1528
```

Exceptions

Exceptions in Python are managed with try-except blocks (exception_name):

```
def one_function():
    try:
        # Division by zero causes one exception
        10/0
    except ZeroDivisionError:
        print("Oops, error.")
    else:
        # There was no exception, we can continue.
```

```
        pass
    finally:
        # This code is executed when the block
        # try..except is already executed and all exceptions
        # have been managed, even if a new one occurs
        # exception directly in the block.
        print("We finished.")

>>> one_function()
Oops, error.
We finished
```

Importing libraries

External libraries are imported with `import [library name]`. Alternatively, you can use the `from [library name] import [function name]` syntax to import a specific function. Here is an example:

```
import random
randomint = random.randint(1, 101)

>>> print(randomint)
65

from random import randint
randomint = random.randint(1, 102)

>>> print(randomint)
46
```

Managing files

To allow us to interact with the filesystem, Python provides us with the built-in `open` function. This function can be invoked to open a file and return an object `file`. The latter allows us to perform various operations on the file, such as reading and writing. When we have finished interacting with the file, we must finally remember to close it by using the `file.close` method:

```
>>> f = open ('test.txt', 'w') # open the file for writing
>>> f.write ('first line of file \n') # write a line in file
>>> f.write ('second line of file \n') # write another line in file
>>> f.close () # we close the file
>>> f = open ('test.txt') # reopen the file for reading
>>> content = f.read () # read all the contents of the file
```

```
>>> print (content)
first line of the file
second line of the file
>>> f.close () # close the file
```

List comprehensions

List comprehensions are a powerful tool for creating and manipulating lists. They consist of an expression that is followed by a `for` clause and then followed by zero, or more, `if` clauses. The syntax for list comprehensions is simply the following:

```
[expression for item in list]
```

Then, perform the following:

```
#list comprehensions using strings
>>> list_comprehension_1 = [ x for x in 'python parallel programming
cookbook!' ]
>>> print( list_comprehension_1)

['p', 'y', 't', 'h', 'o', 'n', ' ', 'p', 'a', 'r', 'a', 'l', 'l', 'e', 'l',
' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', ' ', 'c', 'o',
'o', 'k', 'b', 'o', 'o', 'k', '!']

#list comprehensions using numbers
>>> l1 = [1,2,3,4,5,6,7,8,9,10]
>>> list_comprehension_2 = [ x*10 for x in l1 ]
>>> print( list_comprehension_2)

[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Running Python scripts

To execute a Python script, simply invoke the Python interpreter followed by the script name, in this case, `my_pythonscript.py`. Or, if we are in a different working directory, then use its full address:

```
> python my_pythonscript.py
```



From now on, for every invocation of a Python script, we will use the preceding notation; that is, `python`, followed by `script_name.py`, assuming that the directory from which the Python interpreter is launched is the one where the script to be executed resides.

Installing Python packages using pip

`pip` is a tool that allows us to search, download, and install Python packages found on the Python Package Index, which is a repository that contains tens of thousands of packages written in Python. This also allows us to manage the packages we have already downloaded, allowing us to update or remove them.

Installing pip

`pip` is already included in Python versions ≥ 3.4 and $\geq 2.7.9$. To check whether this tool is already installed, we can run the following command:

```
C:\>pip
```

If `pip` is already installed, then this command will show us the installed version.

Updating pip

It is also recommended to check that the `pip` version you are using is always up to date. To update it, we can use the following command:

```
C:\>pip install -U pip
```

Using pip

`pip` supports a series of commands that allow us, among other things, to *search*, *download*, *install*, *update*, and *remove* packages.

To install `PACKAGE`, just run the following command:

```
C:\>pip install PACKAGE
```

Introducing Python parallel programming

Python provides many libraries and frameworks that facilitate high-performance computations. However, doing parallel programming with Python can be quite insidious due to the **Global Interpreter Lock (GIL)**.

In fact, the most widespread and widely used Python interpreter, **CPython**, is developed in the C programming language. The CPython interpreter needs GIL for thread-safe operations. The use of GIL implies that you will encounter a global lock when you attempt to access any Python objects contained within threads. And only one thread at a time can acquire the lock for a Python object or C API.

Fortunately, things are not so serious, because, outside the realm of GIL, we can freely use parallelism. This category includes all the topics that we will discuss in the next chapters, including multiprocessing, distributed computing, and GPU computing.

So, Python is not really multithreaded. But what is a thread? What is a process? In the following sections, we will introduce these two fundamental concepts and how they are addressed by the Python programming language.

Processes and threads

Threads can be compared to light processes, in the sense that they offer advantages similar to those of processes, without, however, requiring the typical communication techniques of processes. Threads allow you to divide the main control flow of a program into multiple concurrently running control streams. Processes, by contrast, have their *own addressing space* and their own resources. It follows that communication between parts of code running on different processes can only take place through appropriate management mechanisms, including pipes, code FIFO, mailboxes, shared memory areas, and message passing. Threads, on the other hand, allow the creation of concurrent parts of the program, in which each part can access the same address space, variables, and constants.

The following table summarizes the main differences between threads and processes:

Threads	Processes
Share memory.	Do not share memory.
Start/change are computationally less expensive.	Start/change are computationally expensive.
Require fewer resources (light processes).	Require more computational resources.
Need synchronization mechanisms to handle data correctly.	No memory synchronization is required.

After this brief introduction, we can finally show how processes and threads operate.

In particular, we want to compare the serial, multithread, and multiprocessing execution times of the following function, `do_something`, which performs some basic calculations, including building a list of integers selected randomly (a `do_something.py` file):

```
import random

def do_something(count, out_list):
    for i in range(count):
        out_list.append(random.random())
```

Next, there is the serial (`serial_test.py`) implementation. Let's start with the relevant imports:

```
from do_something import *
import time
```

Note the importing of the module `time`, which will be used to evaluate the execution time, in this instance, and the serial implementation of the `do_something` function. size of the list to build is equal to 10000000, while the `do_something` function will be executed 10 times:

```
if __name__ == "__main__":
    start_time = time.time()
    size = 10000000
    n_exec = 10
    for i in range(0, n_exec):
        out_list = list()
        do_something(size, out_list)
    print ("List processing complete.")
    end_time = time.time()
    print("serial time=", end_time - start_time)
```

Next, we have the multithreaded implementation (`multithreading_test.py`).

Import the relevant libraries:

```
from do_something import *
import time
import threading
```

Note the import of the `threading` module in order to operate with the multithreading capabilities of Python.

Here, there is the multithreading execution of the `do_something` function. We will not comment in-depth on the instructions in the following code, as they will be discussed in more detail in Chapter 2, *Thread-Based Parallelism*.

However, it should be noted in this case, too, that the length of the list is obviously the same as in the serial case, `size = 100000000`, while the number of threads defined is 10, `threads = 10`, which is also the number of times the `do_something` function must be executed:

```
if __name__ == "__main__":
    start_time = time.time()
    size = 100000000
    threads = 10
    jobs = []
    for i in range(0, threads):
```

Note also the construction of the single thread, through the `threading.Thread` method:

```
    out_list = list()
    thread = threading.Thread(target=list_append(size, out_list))
    jobs.append(thread)
```

The sequence of cycles in which we start executing threads and then stop them immediately afterwards is as follows:

```
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()

    print ("List processing complete.")
    end_time = time.time()
    print("multithreading time=", end_time - start_time)
```

Finally, there is the multiprocessing implementation (`multiprocessing_test.py`).

We start by importing the necessary modules and, in particular, the multiprocessing library, whose features will be explained in-depth in Chapter 3, *Process-Based Parallelism*:

```
from do_something import *
import time
import multiprocessing
```

As in the previous cases, the length of the list to build, the size, and the execution number of the `do_something` function remain the same (`procs = 10`):

```
if __name__ == "__main__":
    start_time = time.time()
    size = 10000000
    procs = 10
    jobs = []
    for i in range(0, procs):
        out_list = list()
```

Here, the implementation of a single process through the `multiprocessing.Process` method call is affected as follows:

```
process = multiprocessing.Process\
    (target=do_something, args=(size, out_list))
jobs.append(process)
```

Next, the sequence of cycles in which we start executing processes and then stop them immediately afterwards is executed as follows:

```
for j in jobs:
    j.start()

for j in jobs:
    j.join()

print ("List processing complete.")
end_time = time.time()
print("multiprocesses time=", end_time - start_time)
```

Then, we open the command shell and run the three functions described previously.

Go to the folder where the functions have been copied and then type the following:

```
> python serial_test.py
```

The result, obtained on a machine with the following features—CPU Intel i7/8 GB of RAM, is as follows:

```
List processing complete.  
serial time= 25.428767204284668
```

In the case of the `multithreading` implementation, we have the following:

```
> python multithreading_test.py
```

The output is as follows:

```
List processing complete.  
multithreading time= 26.168917179107666
```

Finally, there is the `multiprocessing` implementation:

```
> python multiprocessing_test.py
```

Its result is as follows:

```
List processing complete.  
multiprocesses time= 18.929869890213013
```

As can be seen, the results of the serial implementation (that is, using `serial_test.py`) are similar to those obtained with the implementation of multithreading (using `multithreading_test.py`) where the threads are essentially launched one after the other, giving precedence to the one over the other until the end, while we have benefits in terms of execution times using the Python multiprocessing capability (using `multiprocessing_test.py`).

2

Thread-Based Parallelism

Currently, the most widely used programming paradigm for the management of concurrency in software applications is based on multithreading. Generally, an application is made by a single process that is divided into multiple independent threads, which represent activities of different types that run in parallel and compete with each other.

Nowadays, modern applications that use multithreading have been adopted on a massive scale. In fact, all current processors are multicore, just so they can perform parallel operations and exploit the computer's computational resources.

Hence, *multithreaded programming* is definitely a good way to achieve concurrent applications. However, multithreaded programming often hides some non-trivial difficulties, which must be managed appropriately to avoid errors such as deadlocks or synchronization issues.

We will first define the concepts of thread-based and multithreaded programming and then introduce the `multithreading` library. We will learn about the main directives for thread definition, management, and communication.

Through the `multithreading` library, we will see how to solve problems through different techniques, such as *lock*, *RLock*, *semaphores*, *condition*, *event*, *barrier*, and *queue*.

In this chapter, we will cover the following recipes:

- What is a thread?
- How to define a thread
- How to determine the current thread
- How to use a thread in a subclass
- Thread synchronization with a lock
- Thread synchronization with an `RLock`

- Thread synchronization with semaphores
- Thread synchronization with a condition
- Thread synchronization with an event
- Thread synchronization with a barrier
- Thread communication using a queue

We will also explore the main options offered by Python to program with threads. To do this, we will focus on using the `threading` module.

What is a thread?

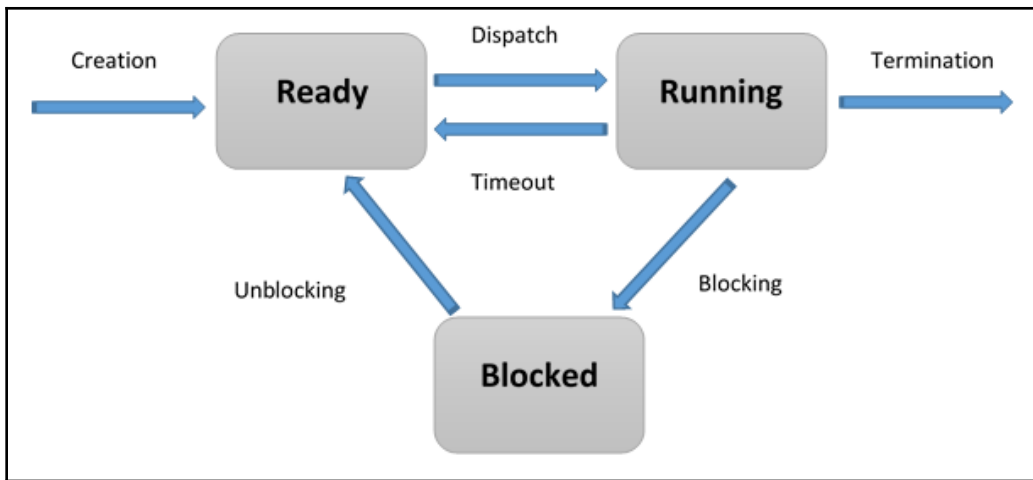
A *thread* is an independent execution flow that can be executed in parallel and concurrently with other threads in the system.

Multiple threads can share data and resources, taking advantage of the so-called space of shared information. The specific implementation of threads and processes depends on the OS on which you plan to run the application, but, in general, it can be stated that a thread is contained inside a process and that different threads in the same process conditions share some resources. In contrast to this, different processes do not share their own resources with other processes.

A thread is composed of three elements: program counters, registers, and stack. Shared resources with other threads of the same process essentially include *data* and *OS resources*. Moreover, threads have their own state of execution, namely, *thread state*, and can be *synchronized* with other threads.

A thread state can be ready, running, or blocked:

- When a thread is created, it enters the **Ready** state.
- A thread is scheduled for execution by the OS (or by the runtime support system) and, when its turn arrives, it begins execution by going into the **Running** state.
- The thread can wait for a condition to occur, passing from the **Running** state to the **Blocked** state. Once the locked condition is terminated, the **Blocked** thread returns to the **Ready** state:



Thread life cycle

The main advantage of multithreading programming lies in performances, as the context switch between processes turns out to be much heavier than the switch context between threads that belong to the same process.

In the next recipes, until the end of the chapter, we will examine the Python `threading` module, introducing its main functions through programming examples.

Python threading module

Python manages threads with the `threading` module provided by the Python standard library. This module provides some very interesting features that make the threading-based approach a whole lot easier; in fact, the `threading` module provides several synchronization mechanisms that are very simple to implement.

The major components of the threading module are as follows:

- The `thread` object
- The `lock` object
- The `RLock` object
- The `semaphore` object
- The `condition` object
- The `event` object

In the following recipes, we examine the features offered by the `threading` library with different application examples. For the examples that follow, we will refer to the Python 3.5.0 distribution (<https://www.python.org/downloads/release/python-350/>).

Defining a thread

The simplest way to use a thread is to instantiate it with a target function and then call the `start` method to let it begin the job.

Getting ready

The Python `threading` module provides a `Thread` class that is used to run processes and functions in a different thread:

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={})
```

Here are the parameters of the `Thread` class:

- `group`: This is the `group` value, which should be `None`; this is reserved for future implementations.
- `target`: This is the function that is to be executed when you start a thread activity.
- `name`: This is the name of the thread; by default, a unique name of the form of `Thread-N` is assigned to it.
- `args`: This is the tuple of arguments that are to be passed to a target.
- `kwargs`: This is the dictionary of keyword arguments that are to be used for the target function.

In the next section, let's learn about how to define a thread.

How to do it...

We'll define a thread by passing it a number, which represents the thread number, and finally, the result will be printed out:

1. Import the `threading` module by using the following Python command:

```
import threading
```

2. In the main program, a `Thread` object is instantiated with a target function called `my_func`. Then, an argument to the function that will be included in the output message is passed:

```
t = threading.Thread(target=function , args=(i,))
```

3. The thread does not start running until the start method is called, and the `join` method makes the calling thread and waits until the thread has finished the execution, as follows:

```
import threading

def my_func(thread_number):
    return print('my_func called by thread N°\
                {}'.format(thread_number))

def main():
    threads = []
    for i in range(10):
        t = threading.Thread(target=my_func, args=(i,))
        threads.append(t)
        t.start()
        t.join()

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

How it works...

In the `main` program, we initialize the thread's list, to which we add the instance of each thread that is created. The total number of threads created is 10, while the `i`-index for the i^{th} thread is passed as an argument to the i^{th} thread:

```
my_func called by thread N°0  
my_func called by thread N°1  
my_func called by thread N°2  
my_func called by thread N°3  
my_func called by thread N°4  
my_func called by thread N°5  
my_func called by thread N°6  
my_func called by thread N°7  
my_func called by thread N°8  
my_func called by thread N°9
```

There's more...

All current processors are multicore, thus offering the possibility of performing multiple parallel operations and making the most of the computer's computational resources. Although this is true, multithread programming hides a number of non-trivial difficulties, which must be managed appropriately to avoid errors such as deadlocks or synchronization problems.

Determining the current thread

Using arguments to identify or name the thread is cumbersome and unnecessary. Each `Thread` instance has a *name* with a default value that can be changed as the thread is created.

Naming threads is useful in server processes with multiple service threads that handle different operations.

Getting ready

This `threading` module provides the `currentThread().getName()` method, which returns the name of the current thread.

The following section shows us how to use this function to determine which thread is running.

How to do it...

Let's have a look at the following steps:

1. To determine which thread is running, we create three target functions and import the `time` module to introduce a suspended execution of two seconds:

```
import threading
import time

def function_A():
    print (threading.currentThread().getName()+str('-->\n\n')
           starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '-->\n\n')
           exiting \n'))
def function_B():
    print (threading.currentThread().getName()+str('-->\n\n')
           starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '-->\n\n')
           exiting \n'))
def function_C():
    print (threading.currentThread().getName()+str('-->\n\n')
           starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '-->\n\n')
           exiting \n'))
```

2. Three threads are instantiated with a target function. Then, we pass the name that is to be printed and, if it is not defined, then the default name will be used. Then, the `start()` and `join()` methods are called for each thread:

```
if __name__ == "__main__":

    t1 = threading.Thread(name='function_A', target=function_A)
    t2 = threading.Thread(name='function_B', target=function_B)
    t3 = threading.Thread(name='function_C',target=function_C)

    t1.start()
    t2.start()
    t3.start()
```

```
t1.join()  
t2.join()  
t3.join()
```

How it works...

We are going to set up three threads, each of which is assigned a `target` function. When the `target` function is executed and terminated, the function name is appropriately printed out.

For this example, the output should look like this (even if the order shown cannot be the same):

```
function_A--> starting  
function_B--> starting  
function_C--> starting  
  
function_A--> exiting  
function_B--> exiting  
function_C--> exiting
```

Defining a thread subclass

Creating a thread can require the definition of a subclass, which inherits from the `Thread` class. The latter, as explained in *Defining a thread* section, is included in the `threading` module, which must then be imported.

Getting ready

The class that we will define in the next section, which represents our thread, respects a precise structure: we will first have to define the `__init__` method, but, above all, we will have to override the `run` method.

How to do it...

The steps involved are as follows:

1. We defined the `MyThreadClass` class, which we can use to create all the threads we want. Each thread of this type will be characterized by the operations defined in the `run` method, which, in this simple example, limits itself to printing a string at the beginning and at the end of its execution:

```
import time
import os
from random import randint
from threading import Thread

class MyThreadClass (Thread):
```

2. Furthermore, in the `__init__` method, we have specified two initialization parameters, respectively, `name` and `duration`, that will be used in the `run` method:

```
    def __init__(self, name, duration):
        Thread.__init__(self)
        self.name = name
        self.duration = duration

    def run(self):
        print ("---> " + self.name + \
              " running, belonging to process ID " + \
              str(os.getpid()) + "\n")
        time.sleep(self.duration)
        print ("---> " + self.name + " over\n")
```

3. These parameters will then be set during the creation of the thread. In particular, the `duration` parameter is computed using the `randint` function that outputs a random integer between 1 and 10. Starting from the definition of `MyThreadClass`, let's see how to instantiate more threads, as follows:

```
def main():

    start_time = time.time()

    # Thread Creation
    thread1 = MyThreadClass("Thread#1 ", randint(1,10))
    thread2 = MyThreadClass("Thread#2 ", randint(1,10))
    thread3 = MyThreadClass("Thread#3 ", randint(1,10))
    thread4 = MyThreadClass("Thread#4 ", randint(1,10))
```

```
thread5 = MyThreadClass("Thread#5 ", randint(1,10))
thread6 = MyThreadClass("Thread#6 ", randint(1,10))
thread7 = MyThreadClass("Thread#7 ", randint(1,10))
thread8 = MyThreadClass("Thread#8 ", randint(1,10))
thread9 = MyThreadClass("Thread#9 ", randint(1,10))

# Thread Running
thread1.start()
thread2.start()
thread3.start()
thread4.start()
thread5.start()
thread6.start()
thread7.start()
thread8.start()
thread9.start()

# Thread joining
thread1.join()
thread2.join()
thread3.join()
thread4.join()
thread5.join()
thread6.join()
thread7.join()
thread8.join()
thread9.join()

# End
print("End")

#Execution Time
print("--- %s seconds ---" % (time.time() - start_time))

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

How it works...

In this example, we created nine threads, each with their own name and duration property, according to the definition of the `__init__` method.

We then run them using the `start` method, which is limited to executing the contents of the previously defined `run` method. Note that the process ID for each thread is the same, meaning that we are in a multithreaded process.

Also, note that the start method *is not blocking*: when it is executed, the control immediately goes to the next line, while the thread is started in the background. In fact, as you can see, the creation of threads *does not take place* in the order specified by the code. Likewise, thread termination is constrained to the value of the `duration` parameter, evaluated using the `randint` function, and passed by the parameter for each thread creation instance. To wait for a thread to finish, a `join` operation must be performed.

The output looks like this:

```
----> Thread#1 running, belonging to process ID 13084
----> Thread#5 running, belonging to process ID 13084
----> Thread#2 running, belonging to process ID 13084
----> Thread#6 running, belonging to process ID 13084
----> Thread#7 running, belonging to process ID 13084
----> Thread#3 running, belonging to process ID 13084
----> Thread#4 running, belonging to process ID 13084
----> Thread#8 running, belonging to process ID 13084
----> Thread#9 running, belonging to process ID 13084

----> Thread#6 over
----> Thread#9 over
----> Thread#5 over
----> Thread#2 over
----> Thread#7 over
----> Thread#4 over
----> Thread#3 over
----> Thread#8 over
----> Thread#1 over

End

--- 9.117518663406372 seconds ---
```

There's more...

The feature that is most frequently associated with OOP is *inheritance*, which is the ability to define a new class as a modified version of an already existing class. The main advantage of inheritance is that you can add new methods to a class without having to change the original definition.

The original class is often referred to as the parent class and the derived class, subclass. Inheritance is a powerful feature, and some programs can be written much more easily and concisely, providing the possibility to customize the behavior of a class without modifying the original class. The very fact that the inheritance structure can reflect that of the problem can, in some cases, make the program easier to understand.

However (to put the user on guard!), inheritance can make it more difficult to read the program. This is because, when invoking a method, it is not always clear where this has been defined within the code that must be traced within multiple modules, instead of being in a single well-defined place.

Many of the things that can be done with inheritance can usually be managed elegantly even without it, so it is appropriate to only use inheritance if the structure of the problem requires it. If used at the wrong time, then the harm inheritance can cause can outweigh the benefits of using it.

Thread synchronization with a lock

The `threading` module also includes a simple lock mechanism, which allows us to implement synchronization between threads.

Getting ready

A *lock* is nothing more than an object that is typically accessible by multiple threads, which a thread must possess before it can proceed to the execution of a protected section of a program. These locks are created by executing the `Lock()` method, which is defined in the `threading` module.

Once the lock has been created, we can use two methods that allow us to synchronize the execution of two (or more) threads: the `acquire()` method to acquire the lock control, and the `release()` method to release it.

The `acquire()` method accepts an optional parameter that, if not specified or set to `True`, forces the thread to suspend its execution until the lock is released and can then be acquired. If, on the other hand, the `acquire()` method is executed with an argument equal to `False`, then it immediately returns a Boolean result, which is `True` if the lock has been acquired, or `False` otherwise.

In the following example, we show the lock mechanism by modifying the code introduced in the previous recipe, *Defining a thread subclass*.

How to do it...

The steps involved are as follows:

1. As shown in the following code block, the `MyThreadClass` class has been modified, introducing the `acquire()` and `release()` methods within the `run` method, while the `Lock()` definition is outside the definition of the class itself:

```
import threading
import time
import os
from threading import Thread
from random import randint

# Lock Definition
threadLock = threading.Lock()

class MyThreadClass (Thread):
    def __init__(self, name, duration):
        Thread.__init__(self)
        self.name = name
        self.duration = duration
    def run(self):
        #Acquire the Lock
        threadLock.acquire()
        print ("---> " + self.name + \
              " running, belonging to process ID "\
              + str(os.getpid()) + "\n")
        time.sleep(self.duration)
        print ("---> " + self.name + " over\n")
        #Release the Lock
        threadLock.release()
```

2. The `main()` function has not changed with respect to the previous code sample:

```
def main():
    start_time = time.time()
    # Thread Creation
    thread1 = MyThreadClass("Thread#1 ", randint(1,10))
    thread2 = MyThreadClass("Thread#2 ", randint(1,10))
    thread3 = MyThreadClass("Thread#3 ", randint(1,10))
    thread4 = MyThreadClass("Thread#4 ", randint(1,10))
    thread5 = MyThreadClass("Thread#5 ", randint(1,10))
    thread6 = MyThreadClass("Thread#6 ", randint(1,10))
    thread7 = MyThreadClass("Thread#7 ", randint(1,10))
    thread8 = MyThreadClass("Thread#8 ", randint(1,10))
    thread9 = MyThreadClass("Thread#9 ", randint(1,10))
```

```
# Thread Running
thread1.start()
thread2.start()
thread3.start()
thread4.start()
thread5.start()
thread6.start()
thread7.start()
thread8.start()
thread9.start()

# Thread joining
thread1.join()
thread2.join()
thread3.join()
thread4.join()
thread5.join()
thread6.join()
thread7.join()
thread8.join()
thread9.join()

# End
print("End")
#Execution Time
print("--- %s seconds ---" % (time.time() - start_time))

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

How it works...

We have modified the code of the previous section by using a lock so that the threads will be executed in sequence.

The first thread acquires the lock and performs its task while the other eight remain *on hold*. At the end of the execution of the first thread, that is, when the `release()` method is executed, the second one will get the lock and the threads from three to eight will still be waiting until the end of the execution (that is, once again, only after running the `release()` method).

The *lock-acquire* and *lock-release* execution are repeated until the ninth thread, with the final result that as a result of the lock mechanism, this execution takes place in a sequential mode, as can be seen in the following output:

```
----> Thread#1 running, belonging to process ID 10632
----> Thread#1 over
----> Thread#2 running, belonging to process ID 10632
----> Thread#2 over
----> Thread#3 running, belonging to process ID 10632
----> Thread#3 over
----> Thread#4 running, belonging to process ID 10632
----> Thread#4 over
----> Thread#5 running, belonging to process ID 10632
----> Thread#5 over
----> Thread#6 running, belonging to process ID 10632
----> Thread#6 over
----> Thread#7 running, belonging to process ID 10632
----> Thread#7 over
----> Thread#8 running, belonging to process ID 10632
----> Thread#8 over
----> Thread#9 running, belonging to process ID 10632
----> Thread#9 over
```

End

```
--- 47.3672661781311 seconds ---
```

There's more...

The insertion points of the `acquire()` and `release()` methods determine the entire execution of the code. For this reason, it is very important that you take the time to analyze what threads you want to use and how you want to synchronize them.

For example, we can change the insertion point of the `release()` method in the `MyThreadClass` class like so:

```
import threading
import time
import os
from threading import Thread
from random import randint

# Lock Definition
threadLock = threading.Lock()
```

```
class MyThreadClass (Thread):
    def __init__(self, name, duration):
        Thread.__init__(self)
        self.name = name
        self.duration = duration
    def run(self):
        #Acquire the Lock
        threadLock.acquire()
        print ("---> " + self.name + \
              " running, belonging to process ID "\
              + str(os.getpid()) + "\n")
        #Release the Lock in this new point
        threadLock.release()
        time.sleep(self.duration)
        print ("---> " + self.name + " over\n")
```

In this case, the output changes quite significantly:

```
---> Thread#1 running, belonging to process ID 11228
---> Thread#2 running, belonging to process ID 11228
---> Thread#3 running, belonging to process ID 11228
---> Thread#4 running, belonging to process ID 11228
---> Thread#5 running, belonging to process ID 11228
---> Thread#6 running, belonging to process ID 11228
---> Thread#7 running, belonging to process ID 11228
---> Thread#8 running, belonging to process ID 11228
---> Thread#9 running, belonging to process ID 11228

---> Thread#2 over
---> Thread#4 over
---> Thread#6 over
---> Thread#5 over
---> Thread#1 over
---> Thread#3 over
---> Thread#9 over
---> Thread#7 over
---> Thread#8 over

End
--- 6.11468243598938 seconds ---
```

As you can see, only the thread creation happens in sequential mode. Once thread creation is complete, the new thread acquires the lock, while the previous one continues the computation in the background.

Thread synchronization with RLock

A reentrant lock, or simply an RLock, is a synchronization primitive that can be acquired multiple times by the same thread.

It uses the concept of the proprietary thread. This means that in the *locked state*, some threads own the lock, while in the *unlocked state*, the lock is not owned by any thread.

The next example demonstrates how to manage threads through the `RLock()` mechanism.

Getting ready

An RLock is implemented through the `threading.RLock()` class. It provides the `acquire()` and `release()` methods that have the same syntax as the `threading.Lock()` class.

An RLock block can be acquired multiple times by the same thread. Other threads will not be able to acquire the RLock block until the thread that owns it has made a `release()` call for every previous `acquire()` call. Indeed, the RLock block must be released, but only by the thread that acquired it.

How to do it...

The steps involved are as follows:

1. We introduced the `Box` class, which provides the `add()` and `remove()` methods that access the `execute()` method in order to perform the action to add or delete an item, respectively. Access to the `execute()` method is regulated by `RLock()`:

```
import threading
import time
import random

class Box:
    def __init__(self):
        self.lock = threading.RLock()
        self.total_items = 0

    def execute(self, value):
        with self.lock:
            self.total_items += value
```

```
def add(self):
    with self.lock:
        self.execute(1)

def remove(self):
    with self.lock:
        self.execute(-1)
```

2. The following functions are called by the two threads. They have the `box` class and the total number of items to add or to remove as parameters:

```
def adder(box, items):
    print("N° {} items to ADD \n".format(items))
    while items:
        box.add()
        time.sleep(1)
        items -= 1
    print("ADDED one item -->{} item to ADD \n".format(items))

def remover(box, items):
    print("N° {} items to REMOVE\n".format(items))
    while items:
        box.remove()
        time.sleep(1)
        items -= 1
    print("REMOVED one item -->{} item to REMOVE\
\n".format(items))
```

3. Here, the total number of items to add or to remove from the box is set. As you can see, these two numbers will be different. The execution ends when both the `adder` and `remover` methods accomplish their tasks:

```
def main():
    items = 10
    box = Box()

    t1 = threading.Thread(target=adder, \
                           args=(box, random.randint(10,20)))
    t2 = threading.Thread(target=remover, \
                           args=(box, random.randint(1,10)))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

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

How it works...

In the main program, the two threads of `t1` and `t2` have been associated with the `add()` and `remove()` functions. The functions are active if the number of items is greater than zero.

The call to `RLock()` is carried out inside the `__init__` method of the `Box` class:

```
class Box:
    def __init__(self):
        self.lock = threading.RLock()
        self.total_items = 0
```

The two `add()` and `remove()` functions interact with the items of the `Box` class, respectively, and call the `Box` class methods of `add()` and `remove()`.

In each method call, a resource is captured and then released using the `lock` parameter that is set in the `__init__` method.

Here is the output:

```
N° 16 items to ADD
N° 1 items to REMOVE
```

```
ADDED one item -->15 item to ADD
REMOVED one item -->0 item to REMOVE
```

```
ADDED one item -->14 item to ADD
ADDED one item -->13 item to ADD
ADDED one item -->12 item to ADD
ADDED one item -->11 item to ADD
ADDED one item -->10 item to ADD
ADDED one item -->9 item to ADD
ADDED one item -->8 item to ADD
ADDED one item -->7 item to ADD
ADDED one item -->6 item to ADD
ADDED one item -->5 item to ADD
ADDED one item -->4 item to ADD
ADDED one item -->3 item to ADD
ADDED one item -->2 item to ADD
ADDED one item -->1 item to ADD
ADDED one item -->0 item to ADD
>>>
```

There's more...

The differences between *lock* and *RLock* are as follows:

- A *lock* can only be acquired once before it must be released. However, *RLock* can be acquired multiple times from the same thread; it must be released the same number of times in order to be released.
- Another difference is that an acquired lock can be released by any thread, whereas an acquired *RLock* can only be released by the thread that acquired it.

Thread synchronization with semaphores

A **semaphore** is an abstract data type managed by the OS to synchronize access by multiple threads to shared resources and data. It consists of an internal variable that identifies the amount of concurrent access to a resource with which it is associated.

Getting ready

The operation of a semaphore is based on two functions: `acquire()` and `release()`, as explained here:

- Whenever a thread wants to access a given or a resource that is associated with a semaphore, it must invoke the `acquire()` operation, which *decreases the internal variable of the semaphore* and allows access to the resource if the value of this variable appears to be non-negative. If the value is negative, then the thread will be suspended and the release of the resource by another thread will be placed on hold.
- Having finished using shared resources, the thread frees resources through the `release()` instruction. In this way, the internal variable of the semaphore is increased, allowing, for a *waiting* thread (if any), the opportunity to access the newly freed resource.

The semaphore is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra.

The following example shows how to synchronize threads through a semaphore.

How to do it...

The following code describes a problem where we have two threads, `producer()` and `consumer()`, that share a common resource, which is the item. The task of `producer()` is to generate the item while the `consumer()` thread's task is to use the item that has been produced.

If the item has not yet produced the `consumer()` thread, then it has to wait. As soon as the item is produced, the `producer()` thread notifies the consumer that the resource should be used:

1. By initializing a semaphore to 0, we obtain a so-called semaphore event whose sole purpose is to synchronize the computation of two or more threads. Here, a thread must make use of data or common resources simultaneously:

```
semaphore = threading.Semaphore(0)
```

2. This operation is very similar to that described in the lock mechanism of the lock. The `producer()` thread creates the item and, after that, it frees the resource by calling the `release()` method:

```
semaphore.release()
```

3. Similarly, the `consumer()` thread acquires the data by the `acquire()` method. If the semaphore's counter is equal to 0, then it blocks the condition's `acquire()` method until it gets notified by a different thread. If the semaphore's counter is greater than 0, then it decrements the value. When the producer creates an item, it releases the semaphore, and then the consumer acquires it and consumes the shared resource:

```
semaphore.acquire()
```

4. The synchronization process that is done via the semaphores is shown in the following code block:

```
import logging
import threading
import time
import random

LOG_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %\
              (message)s'
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)

semaphore = threading.Semaphore(0)
```

```
item = 0

def consumer():
    logging.info('Consumer is waiting')
    semaphore.acquire()
    logging.info('Consumer notify: item number {}'.format(item))

def producer():
    global item
    time.sleep(3)
    item = random.randint(0, 1000)
    logging.info('Producer notify: item number {}'.format(item))
    semaphore.release()

#Main program
def main():
    for i in range(10):
        t1 = threading.Thread(target=consumer)
        t2 = threading.Thread(target=producer)

        t1.start()
        t2.start()

        t1.join()
        t2.join()

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

How it works...

The data acquired is then printed on the standard output:

```
print ("Consumer notify : consumed item number %s " %item)
```

This is the result that we get after 10 runs:

```
2019-01-27 19:21:19,354 Thread-1 INFO Consumer is waiting
2019-01-27 19:21:22,360 Thread-2 INFO Producer notify: item number 388
2019-01-27 19:21:22,385 Thread-1 INFO Consumer notify: item number 388
2019-01-27 19:21:22,395 Thread-3 INFO Consumer is waiting
2019-01-27 19:21:25,398 Thread-4 INFO Producer notify: item number 939
2019-01-27 19:21:25,450 Thread-3 INFO Consumer notify: item number 939
2019-01-27 19:21:25,453 Thread-5 INFO Consumer is waiting
2019-01-27 19:21:28,459 Thread-6 INFO Producer notify: item number 388
2019-01-27 19:21:28,468 Thread-5 INFO Consumer notify: item number 388
2019-01-27 19:21:28,476 Thread-7 INFO Consumer is waiting
```



```
2019-01-27 19:21:31,478 Thread-8 INFO Producer notify: item number 700
2019-01-27 19:21:31,529 Thread-7 INFO Consumer notify: item number 700
2019-01-27 19:21:31,538 Thread-9 INFO Consumer is waiting
2019-01-27 19:21:34,539 Thread-10 INFO Producer notify: item number 685
2019-01-27 19:21:34,593 Thread-9 INFO Consumer notify: item number 685
2019-01-27 19:21:34,603 Thread-11 INFO Consumer is waiting
2019-01-27 19:21:37,604 Thread-12 INFO Producer notify: item number 503
2019-01-27 19:21:37,658 Thread-11 INFO Consumer notify: item number 503
2019-01-27 19:21:37,668 Thread-13 INFO Consumer is waiting
2019-01-27 19:21:40,670 Thread-14 INFO Producer notify: item number 690
2019-01-27 19:21:40,719 Thread-13 INFO Consumer notify: item number 690
2019-01-27 19:21:40,729 Thread-15 INFO Consumer is waiting
2019-01-27 19:21:43,731 Thread-16 INFO Producer notify: item number 873
2019-01-27 19:21:43,788 Thread-15 INFO Consumer notify: item number 873
2019-01-27 19:21:43,802 Thread-17 INFO Consumer is waiting
2019-01-27 19:21:46,807 Thread-18 INFO Producer notify: item number 691
2019-01-27 19:21:46,861 Thread-17 INFO Consumer notify: item number 691
2019-01-27 19:21:46,874 Thread-19 INFO Consumer is waiting
2019-01-27 19:21:49,876 Thread-20 INFO Producer notify: item number 138
2019-01-27 19:21:49,924 Thread-19 INFO Consumer notify: item number 138
>>>
```

There's more...

A particular use of semaphores is the *mutex*. A mutex is nothing but a semaphore with an internal variable initialized to the value of 1, which allows the realization of mutual exclusion in access to data and resources.

Semaphores are still commonly used in programming languages that are multithreaded; however, they have two major problems, which we have discussed, as follows:

- They do not prevent the possibility of a thread performing more wait operations on the same semaphore. It is very easy to forget to do all the necessary signals in relation to the number of waits performed.
- You can run into situations of deadlock. For example, a deadlock situation is created when the t_1 thread executes a wait on the s_1 semaphore, while the t_2 thread executes a wait on the thread t_1 , executes a wait on s_2 and t_2 , and then executes a wait on s_1 .

Thread synchronization with a condition

A *condition* identifies a change of state in the application. It is a synchronization mechanism where a thread waits for a specific condition and another thread notifies that this *condition* has taken place.

Once the condition takes place, the thread *acquires* the lock in order to get *exclusive access* to the shared resource.

Getting ready

A good way to illustrate this mechanism is by looking again at a producer/consumer problem. The class producer writes to a buffer if it is not full, and the class consumer takes the data from the buffer (eliminating them from the latter) if the buffer is full. The class producer will notify the consumer that the buffer is not empty, while the consumer will report to the producer that the buffer is not full.

How to do it...

The steps involved are as follows:

1. The class consumer acquires the shared resource that is modelled through the `items[]` list:

```
condition.acquire()
```

2. If the length of the list is equal to 0, then the consumer is placed in a waiting state:

```
if len(items) == 0:  
    condition.wait()
```

3. Then it makes a `pop` operation from the `items` list:

```
items.pop()
```

4. So, the consumer's state is notified to the producer and the shared resource is released:

```
condition.notify()
```

5. The class producer acquires the shared resource and then it verifies that the list is completely full (in our example, we place the maximum number of items, 10, that can be contained in the items list). If the list is full, then the producer is placed in the wait state until the list is consumed:

```
condition.acquire()
if len(items) == 10:
    condition.wait()
```

6. If the list is not full, then a single item is added. The state is notified and the resource is released:

```
condition.notify()
condition.release()
```

7. To show you the condition mechanism, we will use the *consumer/producer* model again:

```
import logging
import threading
import time

LOG_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %\
              (message)s'
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)

items = []
condition = threading.Condition()

class Consumer(threading.Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def consume(self):

        with condition:

            if len(items) == 0:
                logging.info('no items to consume')
                condition.wait()

            items.pop()
            logging.info('consumed 1 item')

            condition.notify()
```

```
def run(self):
    for i in range(20):
        time.sleep(2)
        self.consume()

class Producer(threading.Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def produce(self):
        with condition:
            if len(items) == 10:
                logging.info('items produced {}. \
                    Stopped'.format(len(items)))
                condition.wait()

            items.append(1)
            logging.info('total items {}'.format(len(items)))

            condition.notify()

    def run(self):
        for i in range(20):
            time.sleep(0.5)
            self.produce()
```

How it works...

`producer` generates the item and stores it in the buffer continuously. At the same time, `consumer` uses the data produced, removing it from the buffer from time to time.

As soon as `consumer` has picked up an object from the buffer, it will wake up `producer`, who will start to fill the buffer again.

Similarly, `consumer` will suspend if the buffer is empty. As soon as `producer` has downloaded the data into the buffer, `consumer` will wake up.

As you can see, even in this case, the use of the `condition` directive allows the threads to be properly synchronized.

The result that we get after a single run is as follows:

```
2019-08-05 14:33:44,285 Producer INFO total items 1
2019-08-05 14:33:44,786 Producer INFO total items 2
2019-08-05 14:33:45,286 Producer INFO total items 3
2019-08-05 14:33:45,786 Consumer INFO consumed 1 item
2019-08-05 14:33:45,787 Producer INFO total items 3
2019-08-05 14:33:46,287 Producer INFO total items 4
2019-08-05 14:33:46,788 Producer INFO total items 5
2019-08-05 14:33:47,289 Producer INFO total items 6
2019-08-05 14:33:47,787 Consumer INFO consumed 1 item
2019-08-05 14:33:47,790 Producer INFO total items 6
2019-08-05 14:33:48,291 Producer INFO total items 7
2019-08-05 14:33:48,792 Producer INFO total items 8
2019-08-05 14:33:49,293 Producer INFO total items 9
2019-08-05 14:33:49,788 Consumer INFO consumed 1 item
2019-08-05 14:33:49,794 Producer INFO total items 9
2019-08-05 14:33:50,294 Producer INFO total items 10
2019-08-05 14:33:50,795 Producer INFO items produced 10. Stopped
2019-08-05 14:33:51,789 Consumer INFO consumed 1 item
2019-08-05 14:33:51,790 Producer INFO total items 10
2019-08-05 14:33:52,290 Producer INFO items produced 10. Stopped
2019-08-05 14:33:53,790 Consumer INFO consumed 1 item
2019-08-05 14:33:53,790 Producer INFO total items 10
2019-08-05 14:33:54,291 Producer INFO items produced 10. Stopped
2019-08-05 14:33:55,790 Consumer INFO consumed 1 item
2019-08-05 14:33:55,791 Producer INFO total items 10
2019-08-05 14:33:56,291 Producer INFO items produced 10. Stopped
2019-08-05 14:33:57,791 Consumer INFO consumed 1 item
2019-08-05 14:33:57,791 Producer INFO total items 10
2019-08-05 14:33:58,292 Producer INFO items produced 10. Stopped
2019-08-05 14:33:59,791 Consumer INFO consumed 1 item
2019-08-05 14:33:59,791 Producer INFO total items 10
2019-08-05 14:34:00,292 Producer INFO items produced 10. Stopped
2019-08-05 14:34:01,791 Consumer INFO consumed 1 item
2019-08-05 14:34:01,791 Producer INFO total items 10
2019-08-05 14:34:02,291 Producer INFO items produced 10. Stopped
2019-08-05 14:34:03,791 Consumer INFO consumed 1 item
2019-08-05 14:34:03,792 Producer INFO total items 10
2019-08-05 14:34:05,792 Consumer INFO consumed 1 item
2019-08-05 14:34:07,793 Consumer INFO consumed 1 item
2019-08-05 14:34:09,794 Consumer INFO consumed 1 item
2019-08-05 14:34:11,795 Consumer INFO consumed 1 item
2019-08-05 14:34:13,795 Consumer INFO consumed 1 item
2019-08-05 14:34:15,833 Consumer INFO consumed 1 item
```

```
2019-08-05 14:34:17,833 Consumer INFO consumed 1 item
2019-08-05 14:34:19,833 Consumer INFO consumed 1 item
2019-08-05 14:34:21,834 Consumer INFO consumed 1 item
2019-08-05 14:34:23,835 Consumer INFO consumed 1 item
```

There's more...

It's interesting to see the Python internals for the condition synchronization mechanism. The internal class `_Condition` creates an `RLock()` object if no existing lock has been passed to the class's constructor. Also, the lock will be managed when `acquire()` and `released()` are called:

```
class _Condition(_Verbose):
    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

Thread synchronization with an event

An event is an object that is used for communication between threads. A thread waits for a signal while another thread outputs it. Basically, an event object manages an internal flag that can be set to `false` with `clear()`, set to `true` with `set()`, and tested with `is_set()`.

A thread can hold a signal by means of the `wait()` method, which sends the call with the `set()` method.

Getting ready

To understand thread synchronization through the event object, let's take a look at the producer/consumer problem.

How to do it...

Again, to explain how to synchronize threads through events, we will refer to the *producer/consumer* problem. The problem describes two processes, a producer and a consumer, who share a common buffer of a fixed size. The producer's task is to generate items and deposit them in the continuous buffer. At the same time, the consumer will use the items produced, removing them from the buffer from time to time.

The problem is to ensure that the producer does not process new data if the buffer is full and that the consumer does not look for data if the buffer is empty.

Now, let's see how to implement the consumer/producer problem by using thread synchronization with an event statement:

1. Here, the relevant libraries are imported as follows:

```
import logging
import threading
import time
import random
```

2. Then, we define the log output format. It is useful to clearly visualize what's happening:

```
LOG_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %\n\
              (message)s'
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
```

3. Set the `items` list. This parameter will be used by the `Consumer` and `Producer` classes:

```
items = []
```

4. The `event` parameter is defined as follows. This parameter will be used to synchronize the communication between threads:

```
event = threading.Event()
```

5. The `Consumer` class is initialized with the list of items and the `Event()` function. In the `run` method, the consumer waits for a new item to consume. When the item arrives, it is popped from the `item` list:

```
class Consumer(threading.Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

```

def run(self):
    while True:
        time.sleep(2)
        event.wait()
        item = items.pop()
        logging.info('Consumer notify: {} popped by {}'.format(item, self.name))

```

6. The `Producer` class is initialized with the list of items and the `Event()` function. Unlike the example with condition objects, the item list is not global, but it is passed as a parameter:

```

class Producer(threading.Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

```

7. In the `run` method for each item that is created, the `Producer` class appends it to the list of items and then notifies the event:

```

def run(self):
    for i in range(5):
        time.sleep(2)
        item = random.randint(0, 100)
        items.append(item)
        logging.info('Producer notify: item {} appended by {}'.format(item, self.name))

```

8. There are two steps that you need to take for this and the first step, which are as follows:

```

event.set()
event.clear()

```

9. The `t1` thread appends a value to the list and then sets the event to notify the consumer. The consumer's call to `wait()` stops blocking and the integer is retrieved from the list:

```

if __name__ == "__main__":
    t1 = Producer()
    t2 = Consumer()

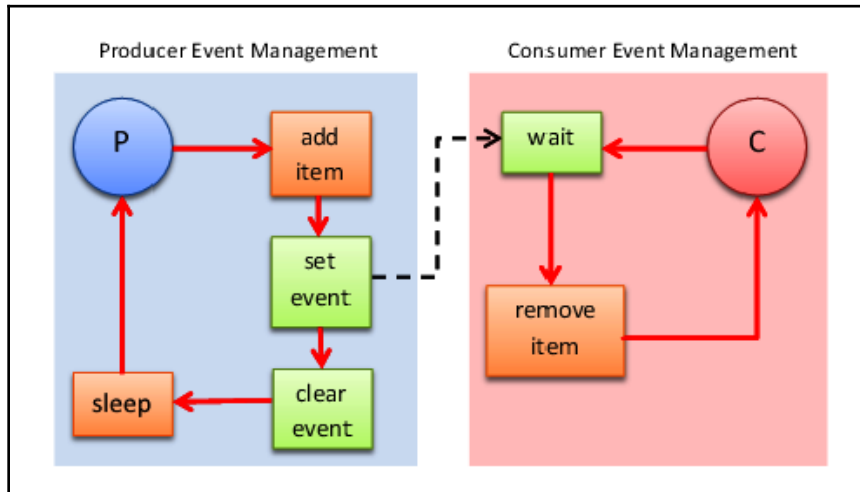
    t1.start()
    t2.start()

    t1.join()
    t2.join()

```


How it works...

All the operations between the `Producer` and the `Consumer` classes can be easily resumed with the help of the following schema:



Thread synchronization with event objects

In particular, the `Producer` and the `Consumer` classes have the following behavior:

- `Producer` acquires a lock, adds an item to the queue, and notifies this event to `Consumer` (`set event`). It then sleeps until it receives a new item to add.
- `Consumer` acquires a block and then begins to listen to the elements in a continuous cycle. The moment the event arrives, the consumer abandons the block, thus allowing other producers/consumers to enter and acquire the block. If `Consumer` is reactivated, then it reacquires the lock by safely processing new items from the queue:

```
2019-02-02 18:23:35,125 Thread-1 INFO Producer notify: item 68
appended by Thread-1
2019-02-02 18:23:35,133 Thread-2 INFO Consumer notify: 68 popped by
Thread-2
2019-02-02 18:23:37,138 Thread-1 INFO Producer notify: item 45
appended by Thread-1
2019-02-02 18:23:37,143 Thread-2 INFO Consumer notify: 45 popped by
Thread-2
2019-02-02 18:23:39,148 Thread-1 INFO Producer notify: item 78
appended by Thread-1
```

```
2019-02-02 18:23:39,153 Thread-2 INFO Consumer notify: 78 popped by
Thread-2
2019-02-02 18:23:41,158 Thread-1 INFO Producer notify: item 22
appended by Thread-1
2019-02-02 18:23:43,173 Thread-1 INFO Producer notify: item 48
appended by Thread-1
2019-02-02 18:23:43,178 Thread-2 INFO Consumer notify: 48 popped by
Thread-2
```

Thread synchronization with a barrier

Sometimes, an application can be divided into phases with the rule that no process can continue if first, all threads of the process have completed their own task. A **barrier** implements this concept: a thread that has finished its phase calls a primitive barrier and stops. When all the threads involved have finished their stage of execution and have also invoked the primitive barrier, the system unlocks them all, allowing threads to move to a later stage.

Getting ready

Python's `threading` module implements barriers through the `Barrier` class. In the next section, let's learn about how to use this synchronization mechanism in a very simple example.

How to do it...

In this example, we simulate a run with three participants, Huey, Dewey, and Louie, in which a barrier is assimilated to that of a finish line.

Moreover, the race can end on its own when all three participants cross the finish line.

The barrier is implemented through the `Barrier` class, in which the number of threads to be completed must be specified as an argument to move to the next stage:

```
from random import randrange
from threading import Barrier, Thread
from time import ctime, sleep

num_runners = 3
finish_line = Barrier(num_runners)
runners = ['Huey', 'Dewey', 'Louie']
```

```
def runner():
    name = runners.pop()
    sleep(randrange(2, 5))
    print('%s reached the barrier at: %s \n' % (name, ctime()))
    finish_line.wait()

def main():
    threads = []
    print('START RACE!!!!')
    for i in range(num_runners):
        threads.append(Thread(target=runner))
        threads[-1].start()
    for thread in threads:
        thread.join()
    print('Race over!')

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

How it works...

First, we set the number of runners to `num_runners = 3` in order to set the final goal on the next line through the `Barrier` directive. The runners are set in the runners' list; each of them will have an arrival time that is determined in the `runner` function, using the `randrange` directive.

When a runner arrives at the finish line, call the `wait` method, which will block all the runners (the threads) that have made that call. The output for this is as follows:

```
START RACE!!!!
Dewey reached the barrier at: Sat Feb 2 21:44:48 2019

Huey reached the barrier at: Sat Feb 2 21:44:49 2019

Louie reached the barrier at: Sat Feb 2 21:44:50 2019

Race over!
```

In this case, Dewey won the race.

Thread communication using a queue

Multithreading can be complicated when threads need to share data or resources. Luckily, the `threading` module provides many synchronization primitives, including semaphores, condition variables, events, and locks.

However, it is considered a best practice to use the `queue` module. In fact, a queue is much easier to deal with and makes threaded programming considerably safer, as it effectively funnels all access to a resource of a single thread and allows for a cleaner and more readable design pattern.

Getting ready

We will simply consider these queue methods:

- `put()`: Puts an item in the queue
- `get()`: Removes and returns an item from the queue
- `task_done()`: Needs to be called each time an item has been processed
- `join()`: Blocks until all items have been processed

How to do it...

In this example, we will see how to use the `threading` module with the `queue` module.

Also, we have two entities here that try to share a common resource, a queue. The code is as follows:

```
from threading import Thread
from queue import Queue
import time
import random

class Producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
    def run(self):
        for i in range(5):
            item = random.randint(0, 256)
            self.queue.put(item)
            print('Producer notify : item N%d appended to queue by\
                %s\n'\
```

```
        % (item, self.name))
        time.sleep(1)

class Consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print('Consumer notify : %d popped from queue by %s'\
                  % (item, self.name))
            self.queue.task_done()

if __name__ == '__main__':
    queue = Queue()
    t1 = Producer(queue)
    t2 = Consumer(queue)
    t3 = Consumer(queue)
    t4 = Consumer(queue)

    t1.start()
    t2.start()
    t3.start()
    t4.start()

    t1.join()
    t2.join()
    t3.join()
    t4.join()
```

How it works...

First, with the `producer` class, we don't need to pass the integers list because we use the queue to store the integers that are generated.

The thread in the `producer` class generates integers and puts them in the queue in a `for` loop. The `producer` class uses `Queue.put(item[, block[, timeout]])` to insert data in the queue. It has the logic to acquire the lock before inserting data in a queue.

There are two possibilities:

- If the optional arguments `block` is `true` and `timeout` is `None` (this is the default case that we used in the example), then it is necessary for us to block until a free slot is available. If the timeout is a positive number, then it blocks at most timeout seconds and raises the full exception if no free slot is available within that time.
- If the block is `false`, then put an item in the queue if a free slot is immediately available, otherwise, raise the full exception (timeout is ignored in this case). Here, `put` checks whether the queue is full and then calls `wait` internally, after which, the producer starts waiting.

Next is the `consumer` class. The thread gets the integer from the queue and indicates that it is done working on it by using `task_done`. The `consumer` class uses `Queue.get([block[, timeout]])` and acquires the lock before removing data from the queue. The consumer is placed in a waiting state, in case the queue is empty. Finally, in the `main` function, we create four threads, one for the `producer` class and three for the `consumer` class, respectively.

The output should be like this:

```
Producer notify : item N°186 appended to queue by Thread-1
Consumer notify : 186 popped from queue by Thread-2

Producer notify : item N°16 appended to queue by Thread-1
Consumer notify : 16 popped from queue by Thread-3

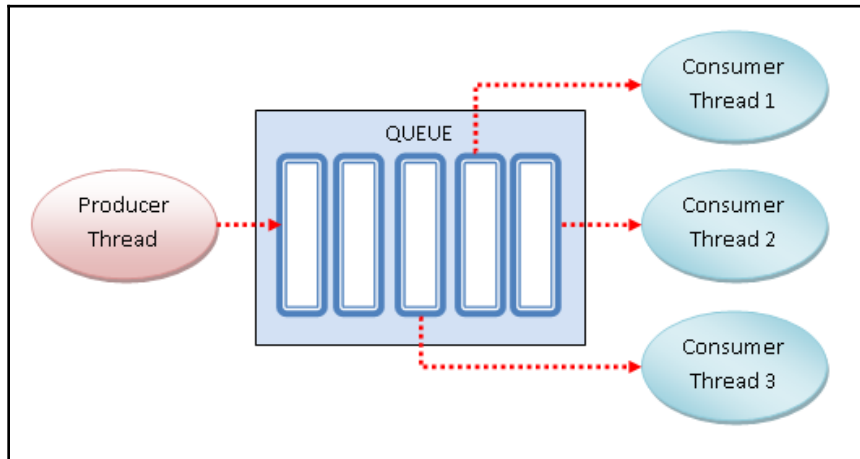
Producer notify : item N°72 appended to queue by Thread-1
Consumer notify : 72 popped from queue by Thread-4

Producer notify : item N°178 appended to queue by Thread-1
Consumer notify : 178 popped from queue by Thread-2

Producer notify : item N°214 appended to queue by Thread-1
Consumer notify : 214 popped from queue by Thread-3
```

There's more...

All the operations between the `producer` class and the `consumer` class can easily be resumed with the following schema:



Thread synchronization with the queue module

- The `Producer` thread acquires the lock and then inserts data in the `QUEUE` data structure.
- The `Consumer` threads get the integers from the `QUEUE`. These threads acquire the lock before removing data from the `QUEUE`.

If the `QUEUE` is empty, then the `consumer` threads get in a **waiting** state.

With this recipe, the chapter dedicated to thread-based parallelism comes to an end.

3

Process-Based Parallelism

In the previous chapter, we learned how to use threads to implement concurrent applications. This chapter will examine the process-based approach that we introduced in Chapter 1, *Getting Started with Parallel Computing and Python*. In particular, the focus of the chapter is on the Python `multiprocessing` module.

The Python `multiprocessing` module, which is a part of the standard library of the language, implements the shared memory programming paradigm, that is, the programming of a system that consists of *one or more processors* that have access to a shared memory.

In this chapter, we will cover the following recipes:

- Understanding Python's `multiprocessing` module
- Spawning a process
- Naming a process
- Running processes in the background
- Killing a process
- Defining a process in a subclass
- Using a queue to exchange objects
- Using pipes to exchange objects
- Synchronizing processes
- Managing a state between processes
- Using a process pool

Understanding Python's multiprocessing module

The introduction of the Python multiprocessing documentation (<https://docs.python.org/2.7/library/multiprocessing.html#introduction>) clearly mentions that all the functionality within this package requires the main module to be importable to the children (<https://docs.python.org/3.3/library/multiprocessing.html>).

The `__main__` module is not importable to the children in IDLE, even if you run the script as a file with IDLE. To get the correct result, we will run all the examples from Command Prompt:

```
> python multiprocessing_example.py
```

Here, `multiprocessing_example.py` is the script's name.

Spawning a process

Spawning a process is the creation of a *child process* from a *parent process*. The latter continues its execution asynchronously or waits until the child process ends.

Getting ready

The multiprocessing library allows spawning processes by following these steps:

1. *Define* the process object.
2. *Call* the `start()` method of the process to run it.
3. *Call* the `join()` method of the process. It waits until the process has completed the job and then exits.

How to do it...

Let's have a look at the following steps:

1. To create a process, we need to import the `multiprocessing` module with the following command:

```
import multiprocessing
```

2. Each process is associated with the `myFunc(i)` function. This function outputs the numbers from 0 to `i`, where `i` is the ID associated with the process number:

```
def myFunc(i):  
    print ('calling myFunc from process n°: %s' %i)  
    for j in range (0,i):  
        print('output from myFunc is :%s' %j)
```

3. Then, we define the process object with `myFunc` as the target function:

```
if __name__ == '__main__':  
    for i in range(6):  
        process = multiprocessing.Process(target=myFunc, args=(i,))
```

4. Finally, we call the `start` and `join` methods on the process created:

```
process.start()  
process.join()
```

Without the `join` method, child processes do not end and must be killed manually.

How it works...

In this section, we have therefore seen how it is possible to create processes by starting from a parent process. This feature is called *spawning a process*.

Python's `multiprocessing` library allows easy process management by following three simple steps. The first step is the process definition through the `multiprocessing` class method, `Process`:

```
process = multiprocessing.Process(target=myFunc, args=(i,))
```

The `Process` method has as an argument of the function to spawn, `myFunc`, and any arguments of the function itself.

The following two steps are necessary to execute and exit the process:

```
process.start()  
process.join()
```

To run the process and display the results, let's open Command Prompt, preferably in the same folder containing the example file (`spawning_processes.py`), and then type the following command:

```
> python spawning_processes.py
```

For each process created (there are six in all), the output of the target function is shown. Remember that this is a simple counter from 0 up to the index of the process ID:

```
calling myFunc from process n°: 0  
calling myFunc from process n°: 1  
output from myFunc is :0  
calling myFunc from process n°: 2  
output from myFunc is :0  
output from myFunc is :1  
calling myFunc from process n°: 3  
output from myFunc is :0  
output from myFunc is :1  
output from myFunc is :2  
calling myFunc from process n°: 4  
output from myFunc is :0  
output from myFunc is :1  
output from myFunc is :2  
output from myFunc is :3  
calling myFunc from process n°: 5  
output from myFunc is :0  
output from myFunc is :1  
output from myFunc is :2  
output from myFunc is :3  
output from myFunc is :4
```

There's more...

This reminds us once again of the importance of instantiating the `Process` object within the main section: this is because the child process created imports the script file where the target function is contained. Then, by instantiating the `process` object within this block, we prevent an infinite recursive call of such instantiations.

A valid workaround is used to define the `target` function in a different script, namely `myFunc.py`:

```
def myFunc(i):
    print ('calling myFunc from process n°: %s' %i)
    for j in range (0,i):
        print('output from myFunc is :%s' %j)
    return
```

The main program containing the process instance is defined in a second file (`spawning_processes_namespace.py`):

```
import multiprocessing
from myFunc import myFunc

if __name__ == '__main__':
    for i in range(6):
        process = multiprocessing.Process(target=myFunc, args=(i,))
        process.start()
        process.join()
```

To run this example, type the following command:

```
> python spawning_processes_names.py
```

The output is the same as the previous example.

See also

The official guide for the `multiprocessing` library can be found at <https://docs.python.org/3/>.

Naming a process

In the previous example, we identified the processes and how to pass a variable to the target function. However, it is very useful to associate a name to the processes as debugging an application requires the processes to be well marked and identifiable.

Getting ready

At some point in your code, it may be crucial to know what process is currently being executed. For this purpose, the `multiprocessing` library provides the `current_process()` method, which uses the `name` attribute to identify which process is currently running. In the following section, we'll learn about this topic.

How to do it...

Let's perform the following steps:

1. The target function for both the processes is the `myFunc` function. It outputs the process name by evaluating the `multiprocessing.current_process().name` method:

```
import multiprocessing
import time

def myFunc():
    name = multiprocessing.current_process().name
    print ("Starting process name = %s \n" %name)
    time.sleep(3)
    print ("Exiting process name = %s \n" %name)
```

2. Then, we create `process_with_name` simply by instantiating the `name` parameter and `process_with_default_name`:

```
if __name__ == '__main__':
    process_with_name = multiprocessing.Process\
        (name='myFunc process',\
         target=myFunc)

    process_with_default_name = multiprocessing.Process\
        (target=myFunc)
```

3. Finally, the processes are started and then joined:

```
process_with_name.start()
process_with_default_name.start()
process_with_name.join()
process_with_default_name.join()
```

How it works...

In the `main` program, the processes are created using the same target function, `myFunc`. This function simply prints the process name.

To run the example, open Command Prompt and type the following command:

```
> python naming_processes.py
```

The output looks like this:

```
Starting process name = myFunc process
Starting process name = Process-2

Exiting process name = Process-2
Exiting process name = myFunc process
```

There's more...

The main Python process is `multiprocessing.process._MainProcess`, while child processes are `multiprocessing.process.Process`. It can be tested by simply typing the following:

```
>>> import multiprocessing
>>> multiprocessing.current_process().name
'MainProcess'
```

See also

More on this topic can be found at <https://doughellmann.com/blog/2012/04/30/determining-the-name-of-a-process-from-python/>.

Running processes in the background

Running in the background is a mode of execution that is typical of some programs that do not require the presence or intervention of the user, and which may be concurrent to the execution of other programs (and therefore, it is only possible in multitasking systems), resulting in the user being unaware about it. Background programs typically perform long or time-consuming tasks such as peer-to-peer filesharing programs or defragmentation of filesystems. Many OS processes also run in the background.

In Windows, programs in this mode (scanning antiviruses or OS updates) often place an icon in the system tray (the area of the desktop next to the system clock) in order to signal their activity and adopt behaviors that reduce the use of resources so as to not interfere with the user's interactive activities, such as slowing down or causing interruptions. In Unix and Unix-like systems, processes that run in the background are called **daemons**. Using a task manager can highlight all running programs including those in the background.

Getting ready

The `multiprocessing` module allows—through the `daemonic` option—to run background processes. In the following example, two processes are defined:

- `background_process` with their `daemon` parameter set to `True`
- `NO_background_process` with their `daemon` parameter set to `False`

How to do it...

In the following example, we implement a target function, namely `foo`, which displays the digits from 0 to 4 **if** the child process is in the **background**; otherwise, it prints the digits from 5 to 9:

1. Let's import the relevant libraries:

```
import multiprocessing
import time
```

2. Then, we define the `foo()` function. As previously specified, the printed digits depend on the value of the `name` parameter:

```
def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    if name == 'background_process':
        for i in range(0,5):
            print ('--> %d \n' %i)
            time.sleep(1)
    else:
        for i in range(5,10):
            print ('--> %d \n' %i)
            time.sleep(1)
    print ("Exiting %s \n" %name)
```

3. Finally, we define the following processes: `background_process` and `NO_background_process`. Notice that the `daemon` parameter is set for the two processes:

```
if __name__ == '__main__':
    background_process = multiprocessing.Process\
        (name='background_process',\
         target=foo)
    background_process.daemon = True

    NO_background_process = multiprocessing.Process\
        (name='NO_background_process',\
         target=foo)
    NO_background_process.daemon = False
    background_process.start()
    NO_background_process.start()
```

How it works...

Note that only the `daemon` parameter of the process defines whether the process should run in the background or not. To run this example, type the following command:

```
> python run_background_processes.py
```

The output clearly reports only the `NO_background_process` output:

```
Starting NO_background_process
----> 5

----> 6

----> 7

----> 8

----> 9
Exiting NO_background_process
```

The output changes the setting of the `daemon` parameter for `background_process` to `False`:

```
background_process.daemon = False
```


To run this example, type the following:

```
C:\>python run_background_processes_no_daemons.py
```

The output reports the execution of both the `background_process` and `NO_background_process` processes:

```
Starting NO_background_process
Starting background_process
----> 5

----> 0
----> 6

----> 1
----> 7

----> 2
----> 8

----> 3
----> 9

----> 4

Exiting NO_background_process
Exiting background_process
```

See also

A code snippet on how to run a Python script in the background in Linux can be found at <https://janakiev.com/til/python-background/>.

Killing a process

There is no perfect software and even in the best applications, you can nest a bug that leads to blocking the application, which is why modern OSes have developed several methods to terminate the processes of applications in order to free the system resources and allow the user to use them for other operations as soon as possible. This section will show you how to kill a process in your multiprocessing application.

Getting ready

It's possible to kill a process immediately by using the `terminate` method. Also, we use the `is_alive` method to keep track of whether the process is alive or not.

How to do it...

The following steps allow us to perform the recipe:

1. Let's import the relevant libraries:

```
import multiprocessing
import time
```

2. Then, a simple `target` function is implemented. In this example, the `target` function, `foo()`, prints the first 10 digits:

```
def foo():
    print ('Starting function')
    for i in range(0,10):
        print('-->%d\n' %i)
        time.sleep(1)
    print ('Finished function')
```

3. In the main program, we create a process monitoring its lifetime by the `is_alive` method; then, we finish it with a call to `terminate`:

```
if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())
    p.start()
    print ('Process running:', p, p.is_alive())
    p.terminate()
    print ('Process terminated:', p, p.is_alive())
    p.join()
    print ('Process joined:', p, p.is_alive())
```

4. Then, we verify the status code when the process is finished and read the attribute of the `ExitCode` process:

```
print ('Process exit code:', p.exitcode)
```

5. The possible values of `ExitCode` are as follows:

- `== 0`: No error was produced.
- `> 0`: The process had an error and exited that code.
- `< 0`: The process was killed with a signal of `-1 * ExitCode`.

How it works...

The sample code consists of a target function, `foo()`, whose task is to print out the first 10 integer numbers on the screen. In the `main` program, the process is executed and then killed by the `terminate` instruction. The process is then joined and `ExitCode` is determined.

To run the code, type the following command:

```
> python killing_processes.py
```

Then, we get the following output:

```
Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Process terminated: <Process(Process-1, started)> True
Process joined: <Process(Process-1, stopped[SIGTERM])> False
Process exit code: -15
```

Notice that the output value of the `ExitCode` code is equal to `-15`. The negative value of `-15` indicates that the child was terminated by an interrupt signal, which is identified by the number 15.

See also

On a Linux machine, a Python process can be identified and then killed simply by following the tutorial at <http://www.cagrimmett.com/til/2016/05/06/killing-rogue-python-processes.html>.

Defining processes in a subclass

The `multiprocessing` module provides access to process management functionalities. In this section, we'll learn about how to define a process in a subclass of the `multiprocessing.Process` class.

Getting ready

To implement a multiprocessing custom subclass, we need to do the following things:

- *Define* a subclass of the `multiprocessing.Process` class, redefining the `run()` method.
- *Override* the `__init__(self [,args])` method to add additional arguments, if needed.
- *Override* the `run(self [,args])` method to implement what `Process` should do when it is started.

Once you have created the new `Process` subclass, you can create an instance of it and then start by invoking the `start` method, which will, in turn, call the `run` method.

How to do it...

Just consider a very simple example, as follows:

1. Import the relevant library first:

```
import multiprocessing
```

2. Then, define a subclass, `MyProcess`, overriding only the `run` method, which returns the process' name:

```
class MyProcess(multiprocessing.Process):  
  
    def run(self):  
        print ('called run method by %s' %self.name)  
        return
```

3. In the main program, we define a subclass of 10 processes:

```
if __name__ == '__main__':
    for i in range(10):
        process = MyProcess()
        process.start()
        process.join()
```

How it works...

Each process subclass is represented by a class that extends the `Process` class and overrides the `run()` method. This method is the starting point of `Process`:

```
class MyProcess (multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return
```

In the main program, we create several objects of the `MyProcess()` type. The execution of the thread begins when the `start()` method is called:

```
p = MyProcess()
p.start()
```

The `join()` command just handles the termination of processes. To run the script from Command Prompt, type the following command:

```
> python process_in_subclass.py
```

The output looks like this:

```
called run method by MyProcess-1
called run method by MyProcess-2
called run method by MyProcess-3
called run method by MyProcess-4
called run method by MyProcess-5
called run method by MyProcess-6
called run method by MyProcess-7
called run method by MyProcess-8
called run method by MyProcess-9
called run method by MyProcess-10
```

There's more...

In object-oriented programming, a subclass is a class that inherits all properties from a superclass, whether they are objects or methods. An alternative name to subclass is *derived class*. *Inheritance* is the specific term that indicates this process by which the daughter or derived classes inherit the properties of parent classes or superclasses.

You can think of a subclass as a particular genre of its superclass; in fact, it can use methods and/or attributes, as well as redefine them through *overriding*.

See also

More information on class definition techniques can be found at http://buildingskills.itmaybeahack.com/book/python-2.6/html/p03/p03c02_adv_class.html.

Using a queue to exchange data

A *queue* is a data structure of the **First-In, First-Out (FIFO)** type (the first input is the first to exit). A practical example is the queues to get a service, how to pay at the supermarket, or get your hair cut at the hairdresser. Ideally, you are served in the same order as you were presented to. This is exactly how a FIFO queue works.

Getting ready

In this section, we show you how to use a queue for a *producer-consumer* problem, that is a classic example of *process synchronization*.

The **producer-consumer** problem describes two *processes*: one is the *producer* and the other is a *consumer*, sharing a **common buffer** of a **fixed size**.

The task of the *producer* is to generate data and to deposit it in the buffer continuously. At the same time, the *consumer* will use the data produced, removing it from the buffer from time to time. The problem is to ensure that the producer does not process new data if the buffer is full and that the consumer does not look for data if the buffer is empty. The solution for the producer is to suspend its execution if the buffer is full.

As soon as the consumer has taken an item from the buffer, the producer wakes up and starts to fill the buffer again. Similarly, the consumer will suspend if the buffer is empty. As soon as the producer has downloaded the data into the buffer, the consumer wakes up.

How to do it...

This solution can be implemented by means of communication strategies between processes, shared memory, or message passing. An incorrect solution could result in a deadlock, in which both processes wait to be awakened:

```
import multiprocessing
import random
import time
```

Let's perform the steps as follows:

1. The producer class is responsible for entering 10 items in the queue by using the `put` method:

```
class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended \
                  to queue %s" \
                  % (item, self.name))
            time.sleep(1)
            print ("The size of queue is %s" \
                  % self.queue.qsize())
```

2. The consumer class has the task of removing the items from the queue (using the `get` method) and verifying that the queue is not empty. If this happens, then the flow inside the while loop ends with a `break` statement:

```
class consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
```

```
while True:
    if (self.queue.empty()):
        print("the queue is empty")
        break
    else :
        time.sleep(2)
        item = self.queue.get()
        print ('Process Consumer : item %d popped \
              from by %s \n'\
              % (item, self.name))
        time.sleep(1)
```

3. The multiprocessing class has its queue object instantiated in the main program:

```
if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process_producer = producer(queue)
    process_consumer = consumer(queue)
    process_producer.start()
    process_consumer.start()
    process_producer.join()
    process_consumer.join()
```

How it works...

Within the main program, we define the queue using the `multiprocessing.Queue` object. Then, it is passed as an argument to the `producer` and `consumer` processes:

```
queue = multiprocessing.Queue()
process_producer = producer(queue)
process_consumer = consumer(queue)
```

In the `producer` class, the `queue.put` method is used to append new items to the queue:

```
self.queue.put(item)
```

While in the `consumer` class, the `queue.get` method is used to pop out the items:

```
self.queue.get()
```

Execute the code by typing the following command:

```
> python communicating_with_queue.py
```


The following output reports the interaction between the producer and the consumer:

```
Process Producer : item 79 appended to queue producer-1
The size of queue is 1
Process Producer : item 50 appended to queue producer-1
The size of queue is 2
Process Consumer : item 79 popped from by consumer-2
Process Producer : item 33 appended to queue producer-1
The size of queue is 2
Process Producer : item 57 appended to queue producer-1
The size of queue is 3
Process Producer : item 227 appended to queue producer-1
Process Consumer : item 50 popped from by consumer-2
The size of queue is 3
Process Producer : item 98 appended to queue producer-1
The size of queue is 4
Process Producer : item 64 appended to queue producer-1
The size of queue is 5
Process Producer : item 182 appended to queue producer-1
Process Consumer : item 33 popped from by consumer-2
The size of queue is 5
Process Producer : item 206 appended to queue producer-1
The size of queue is 6
Process Producer : item 214 appended to queue producer-1
The size of queue is 7
Process Consumer : item 57 popped from by consumer-2
Process Consumer : item 227 popped from by consumer-2
Process Consumer : item 98 popped from by consumer-2
Process Consumer : item 64 popped from by consumer-2
Process Consumer : item 182 popped from by consumer-2
Process Consumer : item 206 popped from by consumer-2
Process Consumer : item 214 popped from by consumer-2
the queue is empty
```

There's more...

A queue has the `JoinableQueue` subclass. This provides the following methods:

- `task_done()`: This method indicates that a task is complete, for example, after using the `get()` method to fetch items from the queue. So `task_done()` must be used only by queue consumers.
- `join()`: This method blocks the processes until all the items in the queue have been completed and processed.

See also

A good tutorial on how to use a queue is available at <https://www.pythoncentral.io/use-queue-beginners-guide/>.

Using pipes to exchange objects

A *pipe* does the following:

- It returns a pair of connection objects connected by a pipe.
- Every connection object has to send/receive methods to communicate between processes.

Getting ready

The multiprocessing library allows you to implement a pipe data structure using the `multiprocessing.Pipe (duplex)` function. This returns a pair of objects, (`conn1`, `conn2`), which represent the end of the pipe.

The `duplex` parameter determines whether the pipe for the last case is bidirectional (that is, `duplex = True`), or unidirectional (that is, `duplex = False`). `conn1` can only be used for receiving messages, and `conn2` can only be used for sending messages.

Now, let's see how to exchange objects using pipes.

How to do it...

Here is a simple example of pipes. We have one process pipe that outputs numbers from 0 to 9, and a second process pipe that takes the numbers and squares them:

1. Let's import the `multiprocessing` library:

```
import multiprocessing
```

2. The `pipe` function returns a pair of connection objects connected by a *two-way* pipe. In the example, `out_pipe` contains the numbers from 0 to 9, which were generated by the target function of `create_items`:

```
def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()
```

3. The `multiply_items` function is based on two pipes, `pipe_1` and `pipe_2`:

```
def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
```

4. This function returns the product of the elements of each pipe:

```
        output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()
```

5. In the main program, `pipe_1`, and `pipe_2` are defined:

```
if __name__ == '__main__':
```

6. First, process `pipe_1` with numbers from 0 to 9:

```
    pipe_1 = multiprocessing.Pipe(True)
    process_pipe_1 = \
        multiprocessing.Process\
            (target=create_items, args=(pipe_1,))
    process_pipe_1.start()
```

7. Then, process `pipe_2`, which picks up the numbers from `pipe_1` and squares them:

```
    pipe_2 = multiprocessing.Pipe(True)
    process_pipe_2 = \
        multiprocessing.Process\
            (target=multiply_items, args=(pipe_1, pipe_2,))
    process_pipe_2.start()
```

8. Close the processes:

```
pipe_1[0].close()
pipe_2[0].close()
```

9. Print out the results:

```
try:
    while True:
        print (pipe_2[1].recv())
except EOFError:
    print("End")
```

How it works...

Essentially, the two pipes, `pipe_1` and `pipe_2`, are created by the `multiprocessing.Pipe(True)` statement:

```
pipe_1 = multiprocessing.Pipe(True)
pipe_2 = multiprocessing.Pipe(True)
```

The first pipe, `pipe_1`, simply created a list of integers from 0 to 9, while the second pipe, `pipe_2`, processed each element of the list created by `pipe_1`, calculating the squared value of each element:

```
process_pipe_2 = \
    multiprocessing.Process\
        (target=multiply_items, args=(pipe_1, pipe_2,))
```

Therefore, both processes are closed:

```
pipe_1[0].close()
pipe_2[0].close()
```

And the final result is printed:

```
print (pipe_2[1].recv())
```

Execute the code by typing the following command:

```
> python communicating_with_pipe.py
```

The following result shows the square of the first 9 digits:

```
0
1
4
```

9
16
25
36
49
64
81

There's more...

If you need more than two points to communicate, then use a `Queue()` method. However, if you need absolute performance, then a `Pipe()` method is much faster because `Queue()` is built on top of `Pipe()`.

See also

More information on Python and pipes can be found at <https://www.python-course.eu/pipes.php>.

Synchronizing processes

Multiple processes can work together to perform a given task. Usually, they share data. It is important that access to shared data by various processes does not produce inconsistent data. Processes that cooperate by sharing data must, therefore, act in an orderly manner in order for that data to be accessible. Synchronization primitives are quite like those encountered for the library and threading.

Synchronization primitives are as follows:

- **Lock:** This object can be in either the locked or unlocked state. A locked object has two methods, `acquire()` and `release()`, to manage access to a shared resource.
- **Event:** This object realizes simple communication between processes; one process signals an event and the other processes wait for it. An event object has two methods, `set()` and `clear()`, to manage its own internal flag.
- **Condition:** This object is used to synchronize parts of a workflow, in sequential or parallel processes. It has two basic methods: `wait()` is used to wait for a condition and `notify_all()` is used to communicate the condition that was applied.

- **Semaphore:** This is used to share a common resource, for example, to support a fixed number of simultaneous connections.
- **RLock:** This defines the *recursive lock* object. The methods and functionality of RLock are the same as the `threading` module.
- **Barrier:** This divides a program into phases as it requires all processes to reach the barrier before any of the proceeds. Code that is executed after a barrier cannot be concurrent with the code that was executed before the barrier.

Getting ready

Barrier objects in Python are used to wait for the execution of a fixed number of threads to complete before a given thread can proceed with the execution of the program.

The following example shows how to synchronize simultaneous tasks with a `barrier()` object.

How to do it...

Let's consider four processes, wherein process `p1` and process `p2` are managed by a barrier statement, while process `p3` and process `p4` have *no synchronization* directives.

To do this, perform the following steps:

1. Import the relevant libraries:

```
import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime
```

2. The `test_with_barrier` function executes the barrier's `wait()` method:

```
def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()
```

3. When the two processes have called the `wait()` method, they are released simultaneously:

```
with serializer:
    print("process %s ----> %s" \
```

```

        %(name, datetime.fromtimestamp(now)))

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("process %s ----> %s" \
          %(name, datetime.fromtimestamp(now)))

```

4. In the main program, we created four processes. However, we also need a barrier and lock primitive. The 2 parameter in the Barrier statement stands for the total number of processes to manage:

```

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
             ,target=test_with_barrier,\
             args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
             ,target=test_with_barrier,\
             args=(synchronizer,serializer)).start()
    Process(name='p3 - test_without_barrier'\
             ,target=test_without_barrier).start()
    Process(name='p4 - test_without_barrier'\
             ,target=test_without_barrier).start()

```

How it works...

The Barrier object provides one of the Python synchronization techniques with which single or multiple threads wait until a point in a set of activities and make progress together.

In the main program, the Barrier object (that is, synchronizer) is defined through the following statement:

```
synchronizer = Barrier(2)
```

Note that the number 2 within the parentheses represents the number of processes that the barrier should wait upon.

Then, we implement a set of four processes, but only for the p1 and p2 processes. Note that synchronizer is passed as an argument:

```

Process(name='p1 - test_with_barrier'\
          ,target=test_with_barrier,\

```

```
args=(synchronizer,serializer)).start()
Process(name='p2 - test_with_barrier'\
, target=test_with_barrier,\
args=(synchronizer,serializer)).start()
```

Indeed, in the body of the `test_with_barrier` function, the barrier's `wait()` method is used in order to synchronize the processes:

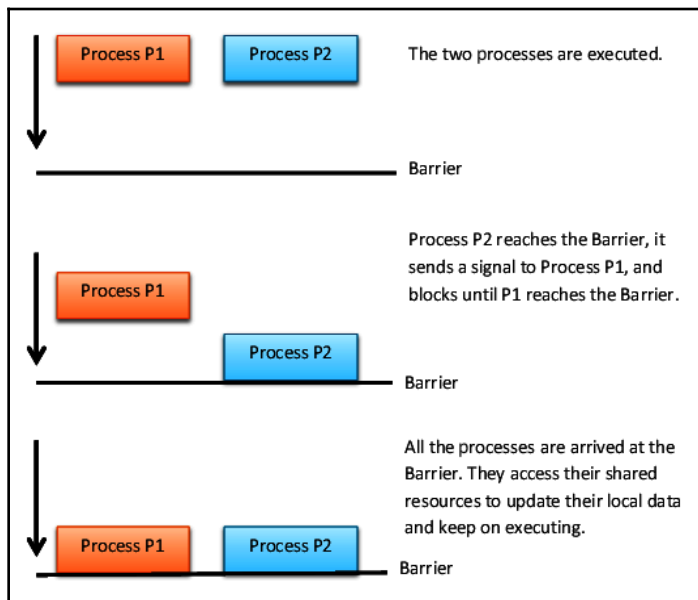
```
synchronizer.wait()
```

By running the script, we can see that the `p1` and `p2` processes print out the same timestamps as expected:

```
> python processes_barrier.py
process p4 - test_without_barrier ----> 2019-03-03 08:58:06.159882
process p3 - test_without_barrier ----> 2019-03-03 08:58:06.144257
process p1 - test_with_barrier ----> 2019-03-03 08:58:06.175505
process p2 - test_with_barrier ----> 2019-03-03 08:58:06.175505
```

There's more...

The following diagram shows you how a barrier works with the two processes:



Process management with a barrier

See also

Please read <https://pymotw.com/2/multiprocessing/communication.html> for more examples of process synchronization.

Using a process pool

The process pool mechanism allows the execution of a function across multiple input values to be parallelized, distributing *the* input data between processes. The process pool, therefore, allows implementing the so-called **data parallelism** that is based on the distribution of data through the different processes that operate on data in parallel.

Getting ready

The multiprocessing library provides the `Pool` class for simple parallel processing tasks.

The `Pool` class has the following methods:

- `apply()`: This blocks until the result is ready.
- `apply_async()`: This is a variant of the `apply()` (<https://docs.python.org/2/library/functions.html#apply>) method, which returns a result object. It is an asynchronous operation that will not lock the main thread until all the child classes are executed.
- `map()`: This is the parallel equivalent of the built-in `map()` (<https://docs.python.org/2/library/functions.html#map>) function. This blocks until the result is ready, and it chops the iterable data in a number of chunks that are submitted to the process pool as separate tasks.
- `map_async()`: This is a variant of the `map()` (<https://docs.python.org/2/library/multiprocessing.html?highlight=pool%20class#multiprocessing.pool.multiprocessing.Pool.map>) method, which returns a result object. If a callback is specified, then it should be callable, which accepts a single argument. When the result becomes ready, a callback is applied to it (unless the call fails). A callback should be completed immediately; otherwise, the thread that handles the results will get blocked.

How to do it...

This example shows you how to implement a process pool to perform a parallel application. We create a pool of four processes and then we use the pool's `map` method to perform a simple function:

1. Import the `multiprocessing` library:

```
import multiprocessing
```

2. The `Pool` method applies `function_square` to the input element to perform a simple calculation:

```
def function_square(data):  
    result = data*data  
    return result
```

```
if __name__ == '__main__':
```

3. The parameter inputs are a list of integers from 0 to 100:

```
inputs = list(range(0,100))
```

4. The total number of parallel processes is 4:

```
pool = multiprocessing.Pool(processes=4)
```

5. The `pool.map` method submits to the process pool as separate tasks:

```
pool_outputs = pool.map(function_square, inputs)  
pool.close()  
pool.join()
```

6. The result of the calculation is stored in `pool_outputs`:

```
print ('Pool      : ', pool_outputs)
```

It is important to note that the result of the `pool.map()` method is equivalent to Python's built-in `map()` function, except that the processes run in parallel.

How it works...

Here, we have created a pool of four processes using the following statement:

```
pool = multiprocessing.Pool(processes=4)
```

Each process has a list of integers as input. Here, `pool.map` works in the same way as the `map`, but uses multiple processes, whose number, four, was previously defined during pool creation:

```
pool_outputs = pool.map(function_square, inputs)
```

To terminate the computation of the pool, the usual `close` and `join` functions are used:

```
pool.close()
pool.join()
```

To execute this, type the following command:

```
> python process_pool.py
```

This is the result that we get after completing the calculation:

```
Pool : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900,
961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764,
1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916,
3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356,
4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084,
6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921, 8100,
8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```

There's more...

In the previous example, we saw that `Pool` also provides the `map` method, which allows us to apply a function to a different set of data. In particular, the scenario in which the same operation is performed in parallel on the elements of the input is referred to as *data parallelism*.

In the following example, in which we use `Pool` and `map`, we create `pool` with 5 workers and, through the `map` method, a function of `f` is applied to a list of 10 elements:

```
from multiprocessing import Pool

def f(x):
    return x+10

if __name__ == '__main__':
    p=Pool(processes=5)
    print(p.map(f, [1, 2, 3,5,6,7,8,9,10]))
```

The output is as follows:

```
11 12 13 14 15 16 17 18 19 20
```

See also

To learn more information about process pools, use the following link: https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_pool_of_processes.htm.

4

Message Passing

This chapter will briefly cover the **Message Passing Interface (MPI)**, which is a specification for message exchange. The primary goal of the MPI is to establish an efficient, flexible, and portable standard for message exchange communication.

Mainly, we will show the functions of the library that include synchronous and asynchronous communication primitives, such as (send/receive) and (broadcast/all-to-all), the operations of combining the partial results of the calculation (gather/reduce), and finally, the synchronization primitives between processes (barriers).

Furthermore, the control functions of the communication network will be presented by defining the topologies.

In this chapter, we will cover the following recipes:

- Using the `mpi4py` Python module
- Implementing point-to-point communication
- Avoiding deadlock problems
- Collective communication using a broadcast
- Collective communication using the `scatter` function
- Collective communication using the `gather` function
- Collective communication using `Alltoall`
- The reduction operation
- Optimizing communication

Technical requirements

You will need the `mpich` and `mpi4py` libraries for this chapter.

The `mpich` library is a portable implementation of MPI. It is free software and is available for various versions of Unix (including Linux and macOS) and Microsoft Windows.

To install `mpich`, use the installer downloaded from the downloads page (<http://www.mpich.org/static/downloads/1.4.1p1/>). Moreover, make sure to choose between the 32-bit or 64-bit versions to get the right one for your machine.

The `mpi4py` Python module provides Python bindings for the MPI (<https://www.mpi-forum.org>) standard. It is implemented on top of the MPI-1/2/3 specification and exposes an API that is based on the standard MPI-2 C++ bindings.

The installation procedure of `mpi4py` on a Windows machine is as follows:

```
C:>pip install mpi4py
```

Anaconda users must type the following:

```
C:>conda install mpi4py
```

Note that for all the examples in this chapter, we used `mpi4py` installed by using the `pip` installer

This implies that the notation used to run the `mpi4py` examples is as follows:

```
C:>mpiexec -n x python mpi4py_script_name.py
```

The `mpiexec` command is the typical way to start parallel jobs: `x` is the total number of processes to use, while `mpi4py_script_name.py` is the name of the script to be executed.

Understanding the MPI structure

The MPI standard defines the primitives for the management of virtual topologies, synchronization, and communication between processes. There are several MPI implementations that differ in the version and features of the standard supported.

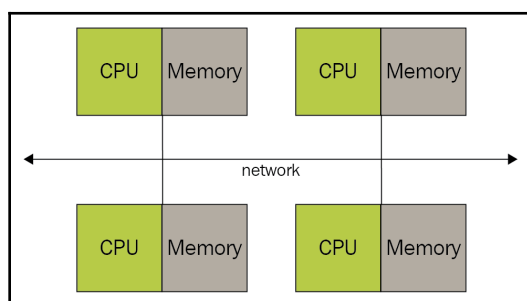
We will introduce the MPI standard through the Python `mpi4py` library.

Before the 1990s, writing parallel applications for different architectures was a more difficult job than what it is today. Many libraries facilitated the process, but there was not a standard way to do it. At that time, most parallel applications were destined for scientific research environments.

The model that was most commonly adopted by the various libraries was the message-passing model, in which the communication between the processes takes place through the exchange of messages and without the use of shared resources. For example, the master process can assign a job to the slaves simply by sending a message that describes the work to be done. A second, very simple, example here is a parallel application that performs a merge sort. The data is sorted locally to the processes and the results are passed to other processes that will deal with the merge.

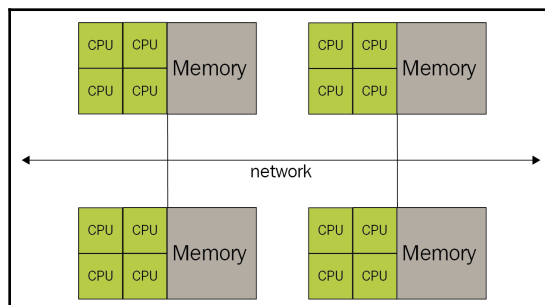
Since the libraries largely used the same model, albeit with minor differences from each other, the authors of the various libraries met in 1992 to define a standard interface for the exchange of messages, and, from here, MPI was born. This interface had to allow programmers to write portable parallel applications on most parallel architectures, using the same features and models they were already used to.

Originally, MPI was designed for distributed memory architectures, which began to grow in popularity 20 years ago:



The distributed memory architecture schema

Over time, distributed memory systems began to be combined with each other, creating hybrid systems with distributed/shared memory:



The hybrid system architecture schema

Today, MPI runs on distributed memory, shared memory, and hybrid systems. However, the programming model remains that of distributed memory, although the true architecture on which the calculation is performed may be different.

The strengths of MPI can be summarized as follows:

- **Standardization:** It is supported by all **High-Performance Computing (HPC)** platforms.
- **Portability:** The changes applied to the source code are minimal, which is useful if you decide to use the application on a different platform that also supports the same standard.
- **Performance:** Manufacturers can create implementations optimized for a specific type of hardware and get better performance.
- **Functionality:** Over 440 routines are defined in MPI-3, but many parallel programs can be written using fewer than even 10 routines.

In the following sections, we will examine the main Python library for message passing: the `mpi4py` library.

Using the `mpi4py` Python module

The Python programming language provides several MPI modules to write parallel programs. The most interesting of these is the `mpi4py` library. It is constructed on top of the MPI-1/2 specifications and provides an object-oriented interface, which closely follows the MPI-2 C++ bindings. A C MPI user could use this module without learning a new interface. Therefore, it is widely used as an almost-full package of an MPI library in Python.

The main applications of the module, which will be described in this chapter, are as follows:

- Point-to-point communication
- Collective communication
- Topologies

How to do it...

Let's start our journey to the MPI library by examining the classic code of a program that prints the phrase `Hello, world!` on each process that is instantiated:

1. Import the `mpi4py` library:

```
from mpi4py import MPI
```



In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers called **ranks**.

2. If we have a number (p of processes) that runs a program, then the processes will have a `rank` that goes from 0 to $p-1$. In particular, in order to assess the rank of each process, we must use the `COMM_WORLD` MPI function in particular. This function is called a **communicator**, as it defines its own set of all processes that can communicate together:

```
comm = MPI.COMM_WORLD
```

3. Finally, the following `Get_rank()` function returns `rank` of the process calling it:

```
rank = comm.Get_rank()
```

4. Once evaluated, `rank` is printed:

```
print ("hello world from process ", rank)
```

How it works...

According to the MPI execution model, our application consists of N (5 in this example) autonomous processes, each with their own local memory able to communicate data through the exchange of messages.

The communicator defines a group of processes that can communicate with each other. The `MPI_COMM_WORLD` work used here is the default communicator and includes all processes.

The identification of a process is based on ranks. Each process is assigned a rank for each communicator to which it belongs. The rank is an integer that is assigned, which starts from zero and identifies each individual process in the context of a specific communicator. The common practice is to define the process with a global rank of 0 as the master process. Through the rank, the developer can specify what the sending process is and what the recipient processes are instead.

It should be noted that, for illustration purposes only, the `stdout` output will not always be ordered, as multiple processes can apply at the same time by writing on the screen and the OS arbitrarily chooses the order. So, we are ready for a fundamental observation: every process involved in the execution of MPI runs the same compiled binary, so each process receives the same instructions to be executed.

To execute the code, type the following command line:

```
C:>mpiexec -n 5 python helloworld_MPI.py
```

This is the result that we will get after executing this code (notice how the order of execution of the processes *is not sequential*):

```
hello world from process 1
hello world from process 0
hello world from process 2
hello world from process 3
hello world from process 4
```



It should be noted that the number of processes to be used is strictly dependent on the characteristics of the machine on which the program must run.

There's more...

MPI belongs to the **Single Program Multiple Data (SPMD)** programming technique.

SPMD is a programming technique in which all processes execute the same program, each on different data. The distinction in executions between different processes occurs by differentiating the flow of the program, based on the local rank of the process.

SPMD is a programming technique in which a single program is executed by several processes at the same time, but each process can operate on different data. At the same time, the processes can execute both the same instruction and different instructions. Obviously, the program will contain appropriate instructions that allow the execution of only parts of the code and/or to operate on a subset of the data. This can be implemented using different programming models, and all executables start at the same time.

See also

The complete reference to the `mpi4py` library can be found at <https://mpi4py.readthedocs.io/en/stable/>.

Implementing point-to-point communication

Point-to-point operations consist of the exchange of messages between two processes. In a perfect world, every sending operation would be perfectly synchronized with the respective reception operation. Obviously, this is not the case, and the MPI implementation must be able to preserve the data sent when the sender and recipient processes are not synchronized. Typically, this occurs using a buffer, which is transparent to the developer and entirely managed by the `mpi4py` library.

The `mpi4py` Python module enables point-to-point communication via two functions:

- `Comm.Send(data, process_destination)`: This function sends data to the destination process identified by its rank in the communicator group.
- `Comm.Recv(process_source)`: This function receives data from the sourcing process, which is also identified by its rank in the communicator group.

The `Comm` parameter, which is short for *communicator*, defines the group of processes that may communicate through message passing using `comm = MPI.COMM_WORLD`.

How to do it...

In the following example, we will utilize the `comm.send` and `comm.recv` directives to exchange messages between different processes:

1. Import the relevant `mpi4py` library:

```
from mpi4py import MPI
```

2. Then, we define the communicator parameter, namely `comm`, through the `MPI.COMM_WORLD` statement:

```
comm=MPI.COMM_WORLD
```

3. The `rank` parameter is used to identify the process itself:

```
rank = comm.rank
```

4. It is useful to print out the `rank` of a process:

```
print("my rank is : " , rank)
```

5. Then, we start considering the `rank` of the process. In this case, for the process of `rank` equal to 0, we set `destination_process` and `data` (in this case `data = 10000000`) to be sent:

```
if rank==0:
    data= 10000000
    destination_process = 4
```

6. Then, by using the `comm.send` statement, the data that was previously set is sent to the destination process:

```
comm.send(data,dest=destination_process)
print ("sending data %s " %data + \
      "to process %d" %destination_process)
```

7. For the process of `rank` equal to 1, the `destination_process` value is 8, while the data to be sent is the "hello" string:

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
    print ("sending data %s :" %data + \
          "to process %d" %destination_process)
```

8. The process of `rank` equal to 4 is a receiver process. Indeed, the source process (that is, the process of `rank` equal to 0) is set as a parameter in the `comm.recv` statement:

```
if rank==4:
    data=comm.recv(source=0)
```

9. Now, using the following code, the data received from the process of 0 must be displayed:

```
print ("data received is = %s" %data)
```

10. The last process to be set is number 9. Here, we define the source process of rank equal to 1 as a parameter in the `comm.recv` statement:

```
if rank==8:  
    data1=comm.recv(source=1)
```

11. The `data1` value is then printed:

```
print ("data1 received is = %s" %data1)
```

How it works...

We ran the example with a total number of processes equal to 9. So, in the `comm` communicator group, we have nine tasks that can communicate with each other:

```
comm=MPI.COMM_WORLD
```

Also, to identify a task or processes inside the group, we use their `rank` value:

```
rank = comm.rank
```

We have two sender processes and two receiver processes. The process of rank equal to 0 sends numerical data to the receiver process of rank equal to 4:

```
if rank==0:  
    data= 10000000  
    destination_process = 4  
    comm.send(data,dest=destination_process)
```

Similarly, we must specify the receiver process of rank equal to 4. We also note that the `comm.recv` statement must contain, as an argument, the rank of the sender process:

```
if rank==4:  
    data=comm.recv(source=0)
```

For the other sender and receiver processes (the process of rank equal to 1 and the process of rank equal to 8, respectively), the situation is the same, the only difference being the type of data.

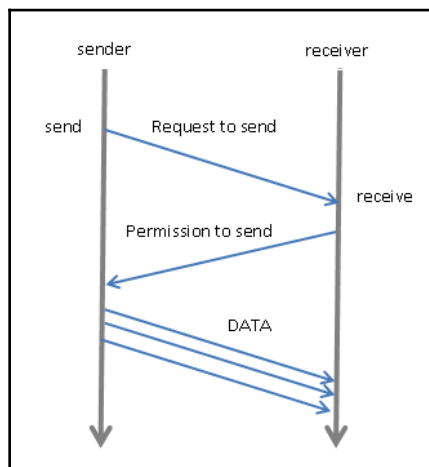
In this case, for the sender process, we have a string that is to be sent:

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
```

For the receiver process of rank equal to 8, the rank of the sender process is pointed out:

```
if rank==8:
    data1=comm.recv(source=1)
```

The following diagram summarizes the point-to-point communication protocol in `mpi4py`:



The send/receive transmission protocol

As you can see, it describes a two-step process, consisting of sending some **DATA** from one task (*sender*) and another task (*receiver*) receiving this data. The sending task must specify the data to be sent and its destination (the *receiver* process), while the receiving task has to specify the source of the message to be received.

To run the script, we shall use 9 processes:

```
C:>mpirun -n 9 python pointToPointCommunication.py
```

This is the output that you'll get after you run the script:

```
my rank is : 7
my rank is : 5
my rank is : 2
my rank is : 6
my rank is : 3
my rank is : 1
sending data hello :to process 8
my rank is : 0
sending data 10000000 to process 4
my rank is : 4
data received is = 10000000
my rank is : 8
data1 received is = hello
```

There's more...

The `comm.send()` and `comm.recv()` functions are blocking functions, which means that they block the caller until the buffered data involved can be used safely. Also, in MPI, there are two management methods of sending and receiving messages:

- **Buffered mode:** The flow control returns to the program as soon as the data to be sent has been copied to a buffer. This does not mean that the message is sent or received.
- **Synchronous mode:** The function only gets terminated when the corresponding `receive` function begins receiving the message.

See also

An interesting tutorial on this topic can be found at https://github.com/antolonappan/MPI_tutorial.

Avoiding deadlock problems

A common problem we face is deadlock. This is a situation where two (or more) processes block each other and wait for the other to perform a certain action that serves another and vice versa. The `mpi4py` module doesn't provide any specific functionality to resolve the deadlock problem, but there are some measures that the developer must follow in order to avoid the problem of deadlock.

How to do it...

Let's first analyze the following Python code, which will introduce a typical deadlock problem. We have two processes—rank equal to 1 and rank equal to 5—that communicate with each other and both have the data sender and data receiver functionalities:

1. Import the `mpi4py` library:

```
from mpi4py import MPI
```

2. Define the communicator as `comm` and the `rank` parameter:

```
comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is %i" % (rank))
```

3. The process of rank equal to 1 sends and receives data from the process of rank equal to 5:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)
    print ("sending data %s " %data_send + \
          "to process %d" %destination_process)
    print ("data received is = %s" %data_received)
```

4. In the same way, here, we define the process of rank equal to 5:

```
if rank==5:
    data_send= "b"
```

5. The destination and sender processes are equal to 1:

```
destination_process = 1
source_process = 1
comm.send(data_send,dest=destination_process)
data_received=comm.recv(source=source_process)
print ("sending data %s :" %data_send + \
      "to process %d" %destination_process)
print ("data received is = %s" %data_received)
```


How it works...

If we try to run this program (it makes sense to execute it with only two processes), then we note that none of the two processes can proceed:

```
C:\>mpiexec -n 9 python deadLockProblems.py
```

```
my rank is : 8
my rank is : 6
my rank is : 7
my rank is : 2
my rank is : 4
my rank is : 3
my rank is : 0
my rank is : 1
sending data a to process 5
data received is = b
my rank is : 5
sending data b :to process 1
data received is = a
```

Both the processes prepare to receive a message from the other and get stuck there. This happens because of the `comm.recv()` MPI function and the `comm.send()` MPI blocking them. This means that the calling process awaits their completion. As for the `comm.send()` MPI, the completion occurs when the data has been sent and may be overwritten without modifying the message.

The completion of the `comm.recv()` MPI instead occurs when the data has been received and can be used. To solve this problem, the first idea is to invert the `comm.recv()` MPI with the `comm.send()` MPI, as follows:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

    print ("sending data %s " %data_send + \
          "to process %d" %destination_process)
    print ("data received is = %s" %data_received)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
```

```
comm.send(data_send, dest=destination_process)

print ("sending data %s :" %data_send + \
      "to process %d" %destination_process)
print ("data received is = %s" %data_received)
```

This solution, even if correct, does not guarantee that we will avoid deadlock. In fact, communication is performed through a buffer with the instruction of `comm.send()`.

MPI copies the data to be sent. This mode works without problems, but only if the buffer is able to keep them all. If this does not happen, then there is a deadlock: the sender cannot finish sending the data because the buffer is busy, and the receiver cannot receive data because it is blocked by the `comm.send()` MPI call, which has not yet completed.

At this point, the solution that allows us to avoid deadlocks is used to swap the sending and receiving functions so as to make them asymmetrical:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send, dest=destination_process)
    data_received=comm.recv(source=source_process)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send, dest=destination_process)
    data_received=comm.recv(source=source_process)
```

Finally, we get the correct output:

```
C:\>mpiexec -n 9 python deadLockProblems.py
```

```
my rank is : 4
my rank is : 0
my rank is : 3
my rank is : 8
my rank is : 6
my rank is : 7
my rank is : 2
my rank is : 1
sending data a to process 5
data received is = b
my rank is : 5
sending data b :to process 1
data received is = a
```

There's more...

The solution proposed to the deadlock is not the only solution.

There is, for example, a function that unifies the single call that sends a message to a given process and receives another message that comes from another process. This function is called `Sendrecv`:

```
Sendrecv(self, sendbuf, int dest=0, int sendtag=0, recvbuf=None, int
source=0, int recvtag=0, Status status=None)
```

As you can see, the required parameters are the same as the `comm.send()` and `comm.recv()` MPI (in this case, also the function blocks). However, `Sendrecv` offers the advantage of leaving the communication subsystem responsible for checking the dependencies between sending and receiving, thus avoiding the deadlock.

In this way, the code of the previous example becomes the following:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.sendrecv(data_send,dest=\
                                destination_process,\
                                source =source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.sendrecv(data_send,dest=\
                                destination_process,\
                                source=source_process)
```

See also

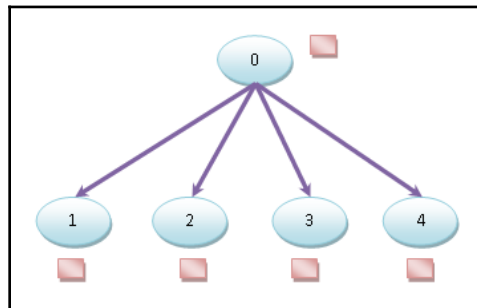
An interesting analysis of how parallel programming is difficult due to deadlock management can be found at <https://codewithoutrules.com/2017/08/16/concurrency-python/>.

Collective communication using a broadcast

During the development of parallel code, we often find ourselves in a situation where we must share, between multiple processes, the value of a certain variable at runtime or certain operations on variables that each process provides (presumably with different values).

To resolve these types of situations, communication trees are used (for example, process 0 sends data to the processes 1 and 2, which will, respectively, take care of sending them to processes 3, 4, 5, 6, and so on).

Instead, MPI libraries provide functions that are ideal for the exchange of information or the use of multiple processes that are clearly optimized for the machine in which they are performed:



Broadcasting data from process 0 to processes 1, 2, 3, and 4

A communication method that involves all the processes that belong to a communicator is called a collective communication. Consequently, collective communication generally involves more than two processes. However, instead of this, we will call the collective communication broadcast, wherein a single process sends the same data to any other process.

Getting ready

The `mpi4py` broadcast functionalities are offered by the following method:

```
buf = comm.bcast(data_to_share, rank_of_root_process)
```

This function sends the information contained in the message process root to every other process that belongs to the `comm` communicator.

How to do it...

Let's now see an example in which we've used the `broadcast` function. We have a root process of rank equal to 0 that shares its own data, `variable_to_share`, with the other processes defined in the communicator group:

1. Let's import the `mpi4py` library:

```
from mpi4py import MPI
```

2. Now, let's define the communicator and the rank parameter:

```
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()
```

3. As far as the process of rank equal to 0 is concerned, we define the variable to be shared among the other processes:

```
if rank == 0:  
    variable_to_share = 100  
else:  
    variable_to_share = None
```

4. Finally, we define a broadcast, having the rank process equal to zero as its root:

```
variable_to_share = comm.bcast(variable_to_share, root=0)  
print("process = %d" %rank + " variable shared = %d " \  
      %variable_to_share)
```

How it works...

The root process of rank equal to 0 instantiates a variable, `variable_to_share`, which is equal to 100. This variable will be shared with the other processes of the communication group:

```
if rank == 0:  
    variable_to_share = 100
```

To perform this, we also introduce the broadcast communication statement:

```
variable_to_share = comm.bcast(variable_to_share, root=0)
```

Here, the parameters in the function are as follows:

- The data to be shared (`variable_to_share`).
- The root process, that is, the process of rank equal to 0 (`root=0`).

Running the code, we have a communication group of 10 processes, and `variable_to_share` is shared between the other processes in the group. Finally, the `print` statement visualizes the rank of the running process and the value of its variable:

```
print("process = %d" %rank + " variable shared = %d " \
      %variable_to_share)
```

After setting 10 processes, the output obtained is as follows:

```
C:\>mpiexec -n 10 python broadcast.py
process = 0
variable shared = 100
process = 8
variable shared = 100
process = 2 variable
shared = 100
process = 3
variable shared = 100
process = 4
variable shared = 100
process = 5
variable shared = 100
process = 9
variable shared = 100
process = 6
variable shared = 100
process = 1
variable shared = 100
process = 7
variable shared = 100
```

There's more...

Collective communication allows simultaneous data transmission between multiple processes in a group. The `mpi4py` library provides collective communications, but only in the blocking version (that is, it blocks the caller method until the buffered data involved can safely be used).

The most commonly used collective communication operations are as follows:

- Barrier synchronization across the group's processes
- Communication functions:
 - Broadcasting data from one process to all processes in the group
 - Gathering data from all processes to one process
 - Scattering data from one process to all processes
- Reduction operations

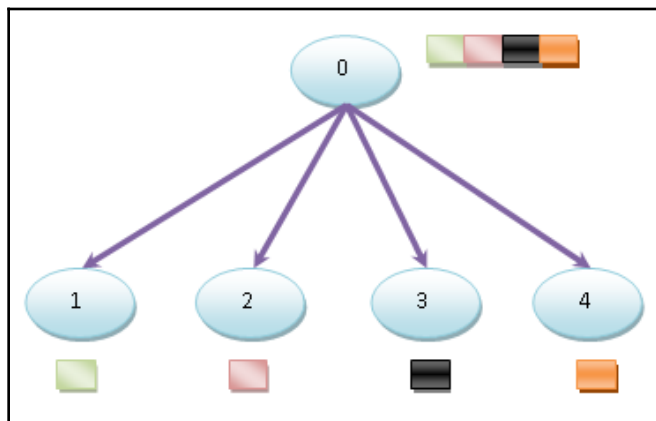
See also

Refer to this link (<https://nyu-cds.github.io/python-mpi/>) to find a complete introduction to Python and MPI.

Collective communication using the scatter function

The scatter functionality is very similar to a scatter broadcast, but with one major difference: while `comm.bcast` sends the same data to all listening processes, `comm.scatter` can send chunks of data in an array to different processes.

The following diagram illustrates the scatter functionality:



Scattering data from process 0 to processes 1, 2, 3, and 4

The `comm.scatter` function takes the elements of the array and distributes them to the processes according to their rank, for which the first element will be sent to process 0, the second element to process 1, and so on. The function implemented in `mpi4py` is as follows:

```
recvbuf = comm.scatter(sendbuf, rank_of_root_process)
```

How to do it...

In the following example, we'll see how to distribute data to different processes using the `scatter` functionality:

1. Import the `mpi4py` library:

```
from mpi4py import MPI
```

2. Next, we define the `comm` and `rank` parameters in the usual way:

```
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()
```

3. For the process of rank equal to 0, the following array will be scattered:

```
if rank == 0:  
    array_to_share = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
else:  
    array_to_share = None
```

4. Then, `recvbuf` is set. The root process is the process of rank equal to 0:

```
recvbuf = comm.scatter(array_to_share, root=0)  
print("process = %d" %rank + " recvbuf = %d " %recvbuf)
```

How it works...

The process of rank equal to 0 distributes the `array_to_share` data structure to other processes:

```
array_to_share = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The `recvbuf` parameter indicates the value of the i^{th} variable that will be sent to the process through the `comm.scatter` statement:

```
recvbuf = comm.scatter(array_to_share, root=0)
```


The output is as follows:

```
C:\>mpiexec -n 10 python scatter.py
process = 0 variable shared = 1
process = 4 variable shared = 5
process = 6 variable shared = 7
process = 2 variable shared = 3
process = 5 variable shared = 6
process = 3 variable shared = 4
process = 7 variable shared = 8
process = 1 variable shared = 2
process = 8 variable shared = 9
process = 9 variable shared = 10
```

We also remark that one of the restrictions to `comm.scatter` is that you can scatter as many elements as the processors you specify in the execution statement. In fact, if you attempt to scatter more elements than the processors specified (three, in this example), then you will get an error similar to the following:

```
C:\> mpiexec -n 3 python scatter.py
Traceback (most recent call last):
  File "scatter.py", line 13, in <module>
    recvbuf = comm.scatter(array_to_share, root=0)
  File "Comm.pyx", line 874, in mpi4py.MPI.Comm.scatter
    (c:\users\utente\appdata\local\temp\pip-build-h14iaj\mpi4py\
src\mpi4py.MPI.c:73400)
  File "pickled.pxi", line 658, in mpi4py.MPI.PyMPI_scatter
    (c:\users\utente\appdata\local\temp\pip-build-h14iaj\mpi4py\src\
mpi4py.MPI.c:34035)
  File "pickled.pxi", line 129, in mpi4py.MPI._p_Pickle.dumpv
    (c:\users\utente\appdata\local\temp\pip-build-h14iaj\mpi4py
\src\mpi4py.MPI.c:28325)
ValueError: expecting 3 items, got 10
mpiexec aborting job...

job aborted:
rank: node: exit code[: error message]
0: Utente-PC: 123: mpiexec aborting job
1: Utente-PC: 123
2: Utente-PC: 123
```

There's more...

The `mpi4py` library provides two other functions that are used to scatter data:

- `comm.scatter(sendbuf, recvbuf, root=0)`: This function sends data from one process to all other processes in a communicator.
- `comm.scatterv(sendbuf, recvbuf, root=0)`: This function scatters data from one process to all other processes in a given group that provide a different amount of data and displacements at the sending side.

The `sendbuf` and `recvbuf` arguments must be given in terms of a list (as in the `comm.send` point-to-point function):

```
buf = [data, data_size, data_type]
```

Here, `data` must be a buffer-like object of the `data_size` size and of the `data_type` type.

See also

An interesting tutorial on MPI broadcasting is presented at <https://pythonprogramming.net/mpi-broadcast-tutorial-mpi4py/>.

Collective communication using the gather function

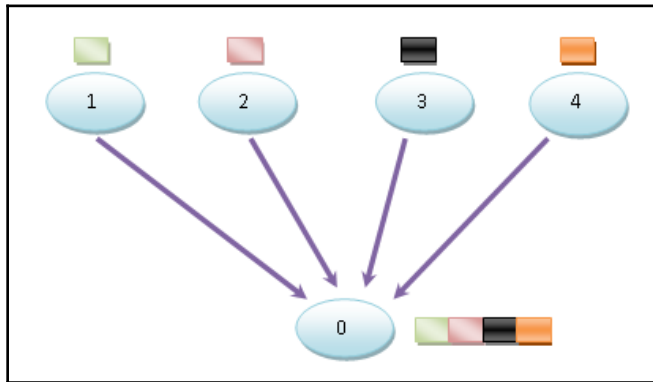
The `gather` function performs the inverse of the `scatter` function. In this case, all processes send data to a root process that collects the data received.

Getting ready

The `gather` function, which is implemented in `mpi4py`, is as follows:

```
recvbuf = comm.gather(sendbuf, rank_of_root_process)
```

Here, `sendbuf` is the data that is sent, and `rank_of_root_process` represents the processing of the receiver of all the data:



Gathering data from processes 1, 2, 3, and 4

How to do it...

In the following example, we'll represent the condition shown in the preceding diagram, in which each process builds its own data, which is to be sent to the root processes that are identified with the `rank` zero:

1. Type the necessary import:

```
from mpi4py import MPI
```

2. Next, we define the following three parameters. The `comm` parameter is the communicator, `rank` provides the rank of the process, and `size` is the total number of processes:

```
comm = MPI.COMM_WORLD  
size = comm.Get_size()  
rank = comm.Get_rank()
```

3. Here, we define the data to be gathered from the process of `rank` zero:

```
data = (rank+1)**2
```

4. Finally, the gathering is provided through the `comm.gather` function. Also, note that the root process (the process that will gather the data from the other ones) is the zero rank process:

```
data = comm.gather(data, root=0)
```

5. For the rank equal to the 0 process, the data gathered and the sending process are printed out:

```
if rank == 0:
    print ("rank = %s " %rank +\
          "...receiving data to other process")
    for i in range(1,size):
        value = data[i]
        print(" process %s receiving %s from process %s"\
              %(rank , value , i))
```

How it works...

The root process of 0 receives data from the other four processes, as represented in the previous diagram.

We set $n (= 5)$ processes sending their data:

```
data = (rank+1)**2
```

If the rank of the process is 0, then the data is collected in an array:

```
if rank == 0:
    for i in range(1,size):
        value = data[i]
```

The gathering of data is given, instead, by the following function:

```
data = comm.gather(data, root=0)
```

Finally, we run the code setting the group of processes equal to 5:

```
C:\>mpiexec -n 5 python gather.py
rank = 0 ...receiving data to other process
process 0 receiving 4 from process 1
process 0 receiving 9 from process 2
process 0 receiving 16 from process 3
process 0 receiving 25 from process 4
```

There's more...

To collect data, `mpi4py` provides the following functions:

- Gathering to one task: `comm.Gather`, `comm.Gatherv`, and `comm.gather`
- Gathering to all tasks: `comm.Allgather`, `comm.Allgatherv`, and `comm.allgather`

See also

More information on `mpi4py` can be found at <http://www.cecil-hpc.be/assets/training/mpi4py.pdf>.

Collective communication using `Alltoall`

The `Alltoall` collective communication combines the `scatter` and `gather` functionalities.

How to do it...

In the following example, we'll see an `mpi4py` implementation of `comm.Alltoall`. We'll consider a communicator a group of processes, where each process sends and receives an array of numerical data from the other processes defined in the group:

1. For this example, the relevant `mpi4py` and `numpy` libraries must be imported:

```
from mpi4py import MPI
import numpy
```

2. As in the previous example, we need to set the same parameters, `comm`, `size`, and `rank`:

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
```

3. Hence, we must define the data that each process will send (`senddata`) and, at the same time, receive (`recvdata`) from the other processes:

```
senddata = (rank+1)*numpy.arange(size, dtype=int)
recvdata = numpy.empty(size, dtype=int)
```

4. Finally, the `Alltoall` function is executed:

```
comm.Alltoall(senddata, recvdata)
```

5. The data that is sent and received for each process is displayed:

```
print(" process %s sending %s receiving %s" \
      %(rank , senddata , recvdata))
```

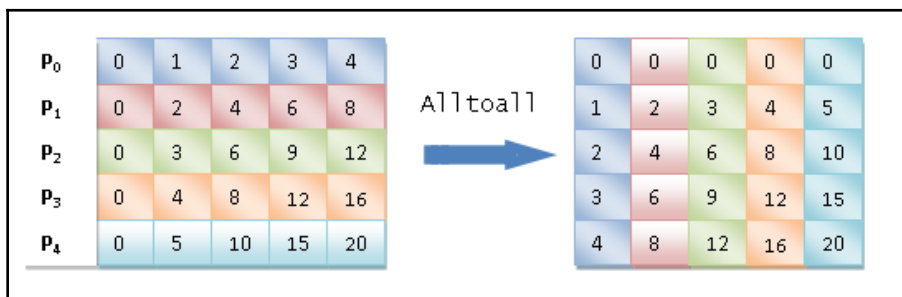
How it works...

The `comm.alltoall` method takes the i^{th} object from the `sendbuf` argument of task j and copies it into the j^{th} object of the `recvbuf` argument of task i .

If we run the code with a communicator group of 5 processes, then our output is as follows:

```
C:\>mpirun -n 5 python alltoall.py
process 0 sending [0 1 2 3 4] receiving [0 0 0 0 0]
process 1 sending [0 2 4 6 8] receiving [1 2 3 4 5]
process 2 sending [ 0 3 6 9 12] receiving [ 2 4 6 8 10]
process 3 sending [ 0 4 8 12 16] receiving [ 3 6 9 12 15]
process 4 sending [ 0 5 10 15 20] receiving [ 4 8 12 16 20]
```

We could also figure out what happened by using the following schema:



The Alltoall collective communication

Our observations regarding the schema are as follows:

- The $P0$ process contains the [0 1 2 3 4] data array, where it assigns 0 to itself, 1 to the $P1$ process, 2 to the $P2$ process, 3 to the $P3$ process, and 4 to the $P4$ process;
- The $P1$ process contains the [0 2 4 6 8] data array, where it assigns 0 to the $P0$ process, 2 to itself, 4 to the $P2$ process, 6 to the $P3$ process, and 8 to the $P4$ process;
- The $P2$ process contains the [0 3 6 9 12] data array, where it assigns 0 to the $P0$ process, 3 to the $P1$ process, 6 to itself, 9 to the $P3$ process, and 12 to the $P4$ process;
- The $P3$ process contains the [0 4 8 12 16] data array, where it assigns 0 to the $P0$ process, 4 to the $P1$ process, 8 to the $P2$ process, 12 to itself, and 16 to the $P4$ process;
- The $P4$ process contains the [0 5 10 15 20] data array, where it assigns 0 to the $P0$ process, 5 to the $P1$ process, 10 to the $P2$ process, 15 to the $P3$ process, and 20 to itself.

There's more...

Alltoall personalized communication is also known as a total exchange. This operation is used in a variety of parallel algorithms, such as the fast Fourier transform, matrix transpose, sample sort, and some parallel database join operations.

In `mpi4py`, there are *three types* of Alltoall collective communication:

- `comm.Alltoall(sendbuf, recvbuf)`: The Alltoall scatter/gather sends data from all-to-all processes in a group.
- `comm.Alltoallv(sendbuf, recvbuf)`: The Alltoall scatter/gather vector sends data from all-to-all processes in a group, providing a different amount of data and displacements.
- `comm.Alltoallw(sendbuf, recvbuf)`: Generalized Alltoall communication allows different counts, displacements, and datatypes for each partner.

See also

An interesting analysis of MPI Python modules can be downloaded from <https://www.duo.uio.no/bitstream/handle/10852/10848/WenjingLinThesis.pdf>.

The reduction operation

Similar to `comm.gather`, `comm.reduce` takes an array of input elements in each process and returns an array of output elements to the root process. The output elements contain the reduced result.

Getting ready

In `mpi4py`, we define the reduction operation through the following statement:

```
comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op =
type_of_reduction_operation)
```

We must note that the difference with the `comm.gather` statement resides in the `op` parameter, which is the operation that you wish to apply to your data, and the `mpi4py` module contains a set of reduction operations that can be used.

How to do it...

Now, we'll see how to implement the sum of an array of elements with the `MPI.SUM` reduction operation by using the reduction functionality. Each process will manipulate an array of size 10.

For array manipulation, we use the functions provided by the `numpy` Python module:

1. Here, the relevant libraries, `mpi4py` and `numpy`, are imported:

```
import numpy
from mpi4py import MPI
```

2. Define the `comm`, `size`, and `rank` parameters:

```
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank
```

3. Then, the size of the array (`array_size`) is set:

```
array_size = 10
```


4. The data to be sent and received is defined:

```
recvdata = numpy.zeros(array_size,dtype=numpy.int)
senddata = (rank+1)*numpy.arange(array_size,dtype=numpy.int)
```

5. The process sender and the sent data are printed out:

```
print(" process %s sending %s " %(rank , senddata))
```

6. Finally, the Reduce operation is executed. Note that the root process is set to 0 and the op parameter is set to `MPI.SUM`:

```
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
```

7. The output of the reduction operation is then shown, as follows:

```
print ('on task',rank,'after Reduce:    data = ',recvdata)
```

How it works...

To perform the reduction sum, we use the `comm.Reduce` statement. Also, we identify with rank zero, which is the root process that will contain `recvbuf`, which represents the final result of the computation:

```
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
```

It makes sense to run the code with a communicator group of 10 processes, as this is the size of the manipulated array.

The output appears as follows:

```
C:\>mpiexec -n 10 python reduction.py
process 1 sending [ 0 2 4 6 8 10 12 14 16 18]
on task 1 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 5 sending [ 0 6 12 18 24 30 36 42 48 54]
on task 5 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 7 sending [ 0 8 16 24 32 40 48 56 64 72]
on task 7 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 3 sending [ 0 4 8 12 16 20 24 28 32 36]
on task 3 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 9 sending [ 0 10 20 30 40 50 60 70 80 90]
on task 9 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 6 sending [ 0 7 14 21 28 35 42 49 56 63]
on task 6 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 2 sending [ 0 3 6 9 12 15 18 21 24 27]
on task 2 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
```

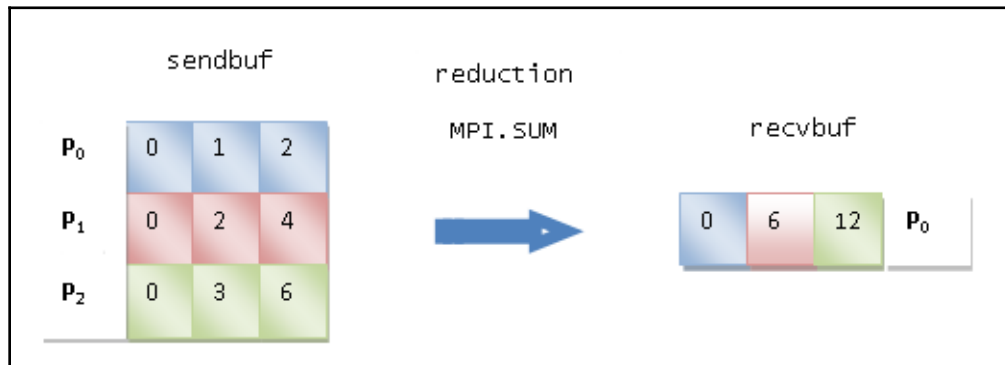
```

process 8 sending [ 0 9 18 27 36 45 54 63 72 81]
on task 8 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 4 sending [ 0 5 10 15 20 25 30 35 40 45]
on task 4 after Reduce: data = [0 0 0 0 0 0 0 0 0 0]
process 0 sending [0 1 2 3 4 5 6 7 8 9]
on task 0 after Reduce: data = [ 0 55 110 165 220 275 330 385 440 495]

```

There's more...

Note that with the `op=MPI.SUM` option, we apply the sum operation to all the elements of the column array. To better understand how the reduction operates, let's look at the following diagram:



Reduction in collective communication

The sending operation is as follows:

- The P_0 process sends the [0 1 2] data array.
- The P_1 process sends the [0 2 4] data array.
- The P_2 process sends the [0 3 6] data array.

The reduction operation sums the i^{th} elements of each task and then puts the result in the i^{th} element of the array in the P_0 root process. For the receiving operation, the P_0 process receives the [0 6 12] data array.

Some of the reduction operations defined by MPI are as follows:

- `MPI.MAX`: This returns the maximum element.
- `MPI.MIN`: This returns the minimum element.
- `MPI.SUM`: This sums up the elements.

- `MPI.PROD`: This multiplies all elements.
- `MPI.LAND`: This performs the AND logical operation across the elements.
- `MPI.MAXLOC`: This returns the maximum value and the rank of the process that owns it.
- `MPI.MINLOC`: This returns the minimum value and the rank of the process that owns it.

See also

At <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>, you can find a good tutorial on this topic and much more.

Optimizing communication

An interesting feature that is provided by MPI regards virtual topologies. As already noted, all the communication functions (point-to-point or collective) refer to a group of processes. We have always used the `MPI_COMM_WORLD` group that includes all processes. It assigns a rank of 0 to $n-1$ for each process that belongs to a communicator of the size n .

However, MPI allows us to assign a virtual topology to a communicator. It defines an assignment of labels to the different processes: by building a virtual topology, each node will communicate only with its virtual neighbor, improving performance because it reduces execution times.

For example, if the rank was randomly assigned, then a message could be forced to pass to many other nodes before it reaches the destination. Beyond the question of performance, a virtual topology makes sure that the code is clearer and more readable.

MPI provides two building topologies. The first construct creates Cartesian topologies, while the latter creates any kind of topologies. Specifically, in the second case, we must supply the adjacency matrix of the graph that you want to build. We will only deal with Cartesian topologies, through which it is possible to build several structures that are widely used, such as mesh, ring, and toroid.

The `mpi4py` function used to create a Cartesian topology is as follows:

```
comm.Create_cart((number_of_rows,number_of_columns))
```

Here, `number_of_rows` and `number_of_columns` specify the rows and columns of the grid that is to be made.

How to do it...

In the following example, we see how to implement a Cartesian topology of the size $M \times N$. Also, we define a set of coordinates to understand how all the processes are disposed of:

1. Import all the relevant libraries:

```
from mpi4py import MPI
import numpy as np
```

2. Define the following parameter in order to move along the topology:

```
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3
```

3. For each process, the following array defines the neighbor processes:

```
neighbour_processes = [0,0,0,0]
```

4. In the main program, the `comm.rank` and `size` parameters are then defined:

```
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.rank
    size = comm.size
```

5. Now, let's build the topology:

```
grid_rows = int(np.floor(np.sqrt(comm.size)))
grid_column = comm.size // grid_rows
```

6. The following conditions ensure that the processes are always within the topology:

```
if grid_rows*grid_column > size:
    grid_column -= 1
if grid_rows*grid_column > size:
    grid_rows -= 1
```

7. The rank equal to 0 process starts the topology construction:

```
if (rank == 0) :
    print("Building a %d x %d grid topology:"\
          % (grid_rows, grid_column) )
    cartesian_communicator = \
        comm.Create_cart( \
```

```

        (grid_rows, grid_column), \
        periods=(False, False), \
        reorder=True)

my_mpi_row, my_mpi_col = \
    cartesian_communicator.Get_coords\
    ( cartesian_communicator.rank )

neighbour_processes[UP], neighbour_processes[DOWN]\
    = cartesian_communicator.Shift(0, 1)
neighbour_processes[LEFT], \
    neighbour_processes[RIGHT] = \
    cartesian_communicator.Shift(1, 1)

print ("Process = %s
\row = %s\n \
column = %s ----> neighbour_processes[UP] = %s \
neighbour_processes[DOWN] = %s \
neighbour_processes[LEFT] =%s neighbour_processes[RIGHT]=%s" \
    %(rank, my_mpi_row, \
    my_mpi_col,neighbour_processes[UP], \
    neighbour_processes[DOWN], \
    neighbour_processes[LEFT] , \
    neighbour_processes[RIGHT]))

```

How it works...

For each process, the output should read as follows: if `neighbour_processes = -1`, then it has no topological proximity, otherwise, `neighbour_processes` shows the rank of the process closely.

The resulting topology is a mesh of 2x2 (refer to the previous diagram for a mesh representation), the size of which is equal to the number of processes in the input; that is, four:

```

grid_row = int(np.floor(np.sqrt(comm.size)))
grid_column = comm.size // grid_row
if grid_row*grid_column > size:
    grid_column -= 1
if grid_row*grid_column > size:
    grid_rows -= 1

```

Then, the Cartesian topology is built using the `comm.Create_cart` function (note also the parameter, `periods = (False, False)`):

```
cartesian_communicator = comm.Create_cart( \
    (grid_row, grid_column), periods=(False, False), reorder=True)
```

To know the position of the process, we use the `Get_coords()` method in the following form:

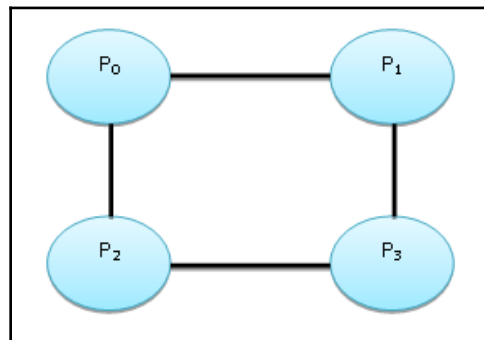
```
my_mpi_row, my_mpi_col = \
    cartesian_communicator.Get_coords(cartesian_communicator.rank )
```

For the processes, in addition to getting their coordinates, we must calculate and find out which processes are topologically closer. For this purpose, we use the `comm.Shift(rank_source, rank_dest)` function:

```
neighbour_processes[UP], neighbour_processes[DOWN] = \
    cartesian_communicator.Shift(0, 1)

neighbour_processes[LEFT], neighbour_processes[RIGHT] = \
    cartesian_communicator.Shift(1, 1)
```

The topology obtained is as follows:



The virtual mesh 2x2 topology

As the diagram shows, the **P₀** process is chained to the **P₁** (RIGHT) and **P₂** (DOWN) processes. The **P₁** process is chained to the **P₃** (DOWN) and **P₀** (LEFT) processes, the **P₃** process is chained to the **P₁** (UP) and **P₂** (LEFT) processes, and the **P₂** process is chained to the **P₃** (RIGHT) and **P₀** (UP) processes.

Finally, by running the script, we obtain the following result:

```
C:\>mpiexec -n 4 python virtualTopology.py
Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0
---->
neighbour_processes[UP] = -1
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] = -1
neighbour_processes[RIGHT]=1

Process = 2 row = 1 column = 0
---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] = -1
neighbour_processes[RIGHT]=3

Process = 1 row = 0 column = 1
---->
neighbour_processes[UP] = -1
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] = 0
neighbour_processes[RIGHT]=-1

Process = 3 row = 1 column = 1
---->
neighbour_processes[UP] = 1
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] = 2
neighbour_processes[RIGHT]=-1
```

There's more...

To obtain a toroidal topology of the size $M \times N$, let's use `comm.Create_cart` again, but, this time, let's set the `periods` parameter to `periods=(True, True)`:

```
cartesian_communicator = comm.Create_cart( (grid_row, grid_column), \
                                           periods=(True, True), reorder=True)
```

The following output is obtained:

```
C:\>mpiexec -n 4 python virtualTopology.py
Process = 3 row = 1 column = 1
---->
neighbour_processes[UP] = 1
```

```

neighbour_processes[DOWN] = 1
neighbour_processes[LEFT] = 2
neighbour_processes[RIGHT] = 2

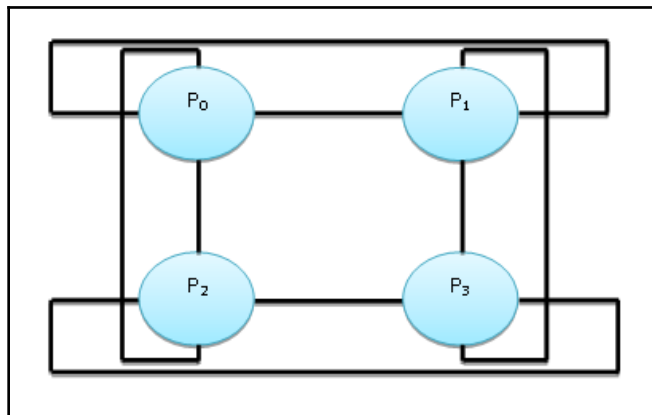
Process = 1 row = 0 column = 1
---->
neighbour_processes[UP] = 3
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] = 0
neighbour_processes[RIGHT] = 0

Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0
---->
neighbour_processes[UP] = 2
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] = 1
neighbour_processes[RIGHT] = 1

Process = 2 row = 1 column = 0
---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = 0
neighbour_processes[LEFT] = 3
neighbour_processes[RIGHT] = 3

```

The output covers the topology represented here:



The virtual toroidal 2x2 topology

The topology represented in the previous diagram indicates that the **P0** process is chained to the **P1** (RIGHT and LEFT) and **P2** (UP and DOWN) processes, the **P1** process is chained to the **P3** (UP and DOWN) and **P0** (RIGHT and LEFT) processes, the **P3** process is chained to the **P1** (UP and DOWN) and **P2** (RIGHT and LEFT) processes, and the **P2** process is chained to the **P3** (LEFT and RIGHT) and **P0** (UP and DOWN) processes.

See also

More information on MPI can be found at <http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-topo.html>.

5 Asynchronous Programming

Beside the sequential and parallel execution models, there is a third model that is of fundamental importance together with the concept of event programming: the *asynchronous model*.

The execution model of asynchronous tasks can be implemented through a single main control flow, both in single-processor systems and in multiprocessor systems. In the concurrent asynchronous execution model, the executions of various tasks intersect along the timeline, and everything happens under the action of a single flow of control (single-threaded). Once started, the execution of tasks can be suspended and then resumed over time, alternating with the execution of other current tasks that are present.

The development of code for the asynchronous model is completely different from that for multithreaded programming. A substantial difference between the concurrent multithreaded parallel model and the single-threaded concurrent asynchronous model lies in the fact that, in the first case, the OS decides on the timeline if we suspend the activity of one thread and start another.

This remains outside the control of the coder, unlike the asynchronous model. The execution or termination of a task continues as long as it is explicitly required.

The most important feature of this type of programming is that the code is not performed on multiple threads, as in the classic concurrent programming, but on a single thread. Thus, it is not at all true that two tasks are executed at the same time, but, according to this approach, they are performed at almost the same time.

In particular, we will describe the `asyncio` Python module, which was introduced in Python 3.4. This allows us to use coroutines and futures to make writing asynchronous code easier and to make it more readable.

In this chapter, we will cover the following recipes:

- Using the `concurrent.futures` Python module
- Managing the event loop with `asyncio`
- Handling coroutines with `asyncio`
- Manipulating tasks with `asyncio`
- Dealing with `asyncio` and futures

Using the `concurrent.futures` Python module

The `concurrent.futures` module, which is part of the standard Python library, provides a level of abstraction on threads by modelling them as asynchronous functions.

This module is built by two main classes:

- `concurrent.futures.Executor`: This is an abstract class that provides methods to execute calls asynchronously.
- `concurrent.futures.Future`: This encapsulates the asynchronous execution of a callable. Future objects are instantiated by submitting tasks (functions with optional parameters) to Executors.

Here are some of the main methods of the module:

- `submit(function, argument)`: This schedules the execution of the callable function on the arguments.
- `map(function, argument)`: This executes the functions of arguments in asynchronous mode.
- `shutdown(Wait=True)`: This signals the executor to free any resource.

The executors are accessed through their subclasses: `ThreadPoolExecutor` or `ProcessPoolExecutor`. Because the instantiation of threads and processes is a resource-demanding task, it is better to pool these resources and use them as repeatable launchers or executors (hence the `Executors` concept) for parallel or concurrent tasks.

The approach we are taking here involves using a pool executor. We will submit the assets to the pool (thread and process) and get the futures, which are the results that will be available to us in the future. Of course, we can wait for all futures to become real results.

A thread or process pool (also called *pooling*) indicates a management software that is being used to optimize and simplify the use of threads and/or processes within a program. Through pooling, you can submit the task (or tasks) in order to execute them to the pooler.

The pool is equipped with an internal queue of tasks pending and several threads *or* processes that execute them. A recurring concept in pooling is reusing: a thread (or process) is used several times for different tasks during its life cycle. This decreases the overhead of creating new threads or processes and increases the performance of the program.

Reuse *is not a rule*, but it is one of the main reasons that lead a coder to use pooling in their application.

Getting ready

The `concurrent.futures` module provides two subclasses of the `Executor` class, which asynchronously manipulate a pool of threads and a pool of processes. The two subclasses are as follows:

- `concurrent.futures.ThreadPoolExecutor(max_workers)`
- `concurrent.futures.ProcessPoolExecutor(max_workers)`

The `max_workers` parameter identifies the maximum number of workers that execute the call asynchronously.

How to do it...

Here is an example of thread and process pool usage, where we will compare the execution time with the time it takes for sequential execution.

The task to be performed is as follows: we have a list of 10 elements. Each element of the list is made to count up to 100,000,000 (just to waste time), and then the last number is multiplied by the *i-th* element of the list. In particular, we are evaluating the following cases:

- **Sequential execution**
- **Thread pool with five workers**
- **Process pool with five workers**

Now, let's look at how to do it:

1. Import the relevant libraries:

```
import concurrent.futures
import time
```

2. Define the list of numbers from 1 to 10:

```
number_list = list(range(1, 11))
```

3. The `count(number)` function counts the numbers from 1 to 1000000000, and then returns the product of `number × 100,000,000`:

```
def count(number):
    for i in range(0, 1000000000):
        i += 1
    return i*number
```

4. The `evaluate(item)` function evaluates the `count` function on the `item` parameter. It prints out the `item` value and the result of `count(item)`:

```
def evaluate(item):
    result_item = count(item)
    print('Item %s, result %s' % (item, result_item))
```

5. In `__main__`, the sequential execution, thread pool, and process pool are executed:

```
if __name__ == '__main__':
```

6. For the sequential execution, the `evaluate` function is executed for each item of `number_list`. Then, the execution time is printed out:

```
    start_time = time.clock()
    for item in number_list:
        evaluate(item)
    print('Sequential Execution in %s seconds' % (time.clock() -\
        start_time))
```

7. Regarding thread and process pool execution, the same number of workers (max_workers=5) is used. Of course, for both pools, execution times are displayed:

```
start_time = time.clock()
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as\
executor:
    for item in number_list:
        executor.submit(evaluate, item)
print('Thread Pool Execution in %s seconds' % (time.clock() -\
start_time))
start_time = time.clock()
with concurrent.futures.ProcessPoolExecutor(max_workers=5) as\
executor:
    for item in number_list:
        executor.submit(evaluate, item)
print('Process Pool Execution in %s seconds' % (time.clock() -\
start_time))
```

How it works...

We build a list of numbers stored in number_list:

```
number_list = list(range(1, 11))
```

For each element in the list, we operate the counting procedure until we reach 100000000 iterations, and then multiply the resulting value for 100000000:

```
def count(number) :
    for i in range(0, 100000000):
        i=i+1
    return i*number

def evaluate_item(x):
    result_item = count(x)
```

In the main program, we execute the same task in sequential mode:

```
if __name__ == "__main__":
    for item in number_list:
        evaluate_item(item)
```

Then, in parallel mode, use the `concurrent.futures` pooling capabilities for a thread pool:

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate, item)
```

And do the same for a process pool:

```
with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate, item)
```



Note that both the thread *and* process pools are set with `max_workers=5`; moreover, if `max_workers` is equal to `None`, it will default to the number of processors on the machine.

To run this example, open Command Prompt and, in the same folder where the example is contained, type the following:

```
> python concurrent_futures_pooling.py
```

By executing the preceding example, we can see the execution of the three execution models with relative times:

```
Item 1, result 10000000
Item 2, result 20000000
Item 3, result 30000000
Item 4, result 40000000
Item 5, result 50000000
Item 6, result 60000000
Item 7, result 70000000
Item 8, result 80000000
Item 9, result 90000000
Item 10, result 100000000
Sequential Execution in 6.8109448 seconds
Item 2, result 20000000
Item 1, result 10000000
Item 4, result 40000000
Item 5, result 50000000
Item 3, result 30000000
Item 8, result 80000000
Item 7, result 70000000
Item 6, result 60000000
Item 10, result 100000000
Item 9, result 90000000
```

```
Thread Pool Execution in 6.805766899999999 seconds
Item 1, result 10000000
Item 4, result 40000000
Item 2, result 20000000
Item 3, result 30000000
Item 5, result 50000000
Item 6, result 60000000
Item 7, result 70000000
Item 9, result 90000000
Item 8, result 80000000
Item 10, result 100000000
Process Pool Execution in 4.166398899999999 seconds
```

It should be noted that although the example is not expensive in computational terms, sequential and thread pool execution are comparable in terms of time. Using a process pool gives us the fastest execution time.

The pool then distributes the processes (in this case, five processes) between the available cores (for this example, a machine with four cores was used) in **FIFO** (short for **first in, first out**) mode.

So, for each core, the assigned process runs in series. Only after the I/O operation is performed does the pool schedule the execution of another process. Of course, the execution mechanism is the same if you use a thread pool.

The computational times, which are lower in the case of the process pool, must be traced back to the fact that I/O operations are not significant. This allows the pool of processes to be faster because, unlike threads, they do not require any synchronization mechanisms (as explained in [Chapter 1, *Getting Started with Parallel Computing and Python*](#), in the *Introducing parallel programming* recipe).

There's more...

The pooling technique is widely used in server applications, as it is necessary to manage multiple simultaneous requests from any number of clients.

Many other applications, however, require that every activity be performed immediately or that you have more control over the thread that runs it: in this case, pooling is not the best choice.

See also

An interesting tutorial on `concurrent.futures` can be found here: <http://masnun.com/2016/03/29/python-a-quick-introduction-to-the-concurrent-futures-module.html>.

Managing the event loop with `asyncio`

The `asyncio` Python module provides facilities for managing events, coroutines, tasks, as well as threads, and synchronization primitives for writing concurrent code.

The main components of this module are as follows:

- **Event loop:** The `asyncio` module allows one event loop per process. This is the entity that deals with managing and distributing the execution of different tasks. In particular, this registers the tasks and manages them by switching the control flow from one task to another.
- **Coroutines:** This is a generalization of the concept of the subroutine. Also, a coroutine can be suspended during execution to wait for external processing (some routine in I/O) and return from the point it had stopped at when the external processing is done.
- **Futures:** This defines the `Future` object exactly like the `concurrent.futures` module. It represents a computation that *has still not been accomplished*.
- **Tasks:** This is a subclass of `asyncio` that is used to encapsulate and manage coroutines in a parallel mode.

In this recipe, the focus is on the concept of events and event management (namely, event loops) within a software program.

Understanding event loops

In computer science, an *event* is an action intercepted by the program that can be managed by the program itself. As an example, an event could be the virtual pressure of a key by the user during interaction with the graphical interface, the pressure of a key on the physical keyboard, an external interrupt signal, or, more abstractly, the reception of data through the network. But more generally, any other form of event that has happened that can be detected and managed in some way.

Within a system, the entity that can generate events is called an *event source*, while the entity that deals with handling an event that occurs are an event handler.

The *event loop* programming construct realizes the functionality of managing events within a program. More precisely, the event loop acts cyclically during the whole execution of the program, keeping track of events that have occurred within a data structure to queue and then process them one at a time by invoking the event handler if the main thread is free.

The pseudocode of the event loop manager is shown here:

```
while (1) {
    events = getEvents()
    for (e in events)
        processEvent(e)
}
```

All the events that are fed into the `while` loop are caught and then processed by the event handler. The handler that processes an event is the only activity taking place in the system. When the handler has ended, control passes to the next event scheduled.

`asyncio` provides the following methods to manage an event loop:

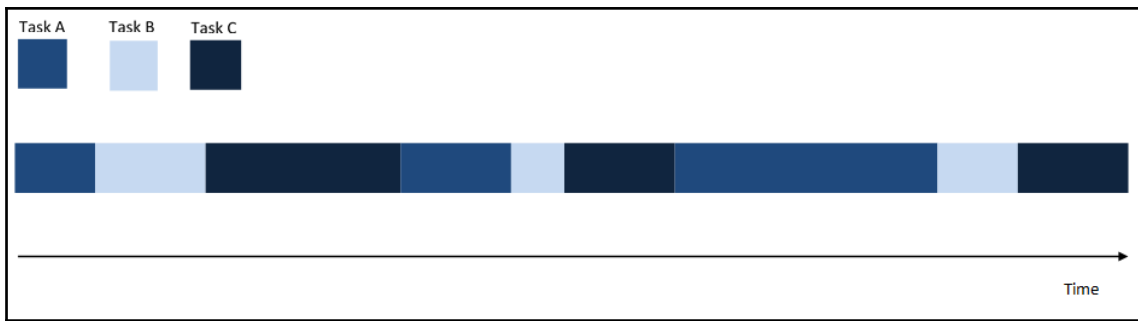
- `loop = get_event_loop()`: This gets the event loop for the current context.
- `loop.call_later(time_delay, callback, argument)`: This arranges for the callback to be called after the given `time_delay`, in seconds.
- `loop.call_soon(callback, argument)`: This arranges for a callback to be called as soon as possible. The callback is called after `call_soon()` (<https://docs.python.org/3/library/asyncio-eventloop.html>) returns when control returns to the event loop.
- `loop.time()`: This returns the current time as a float value (<https://docs.python.org/3/library/functions.html>), according to the event loop's internal clock.
- `asyncio.set_event_loop()`: This sets the event loop for the current context to loop.
- `asyncio.new_event_loop()`: This creates and returns a new event loop object according to this policy's rules.
- `loop.run_forever()`: This runs until `stop()` (<https://docs.python.org/3/library/asyncio-eventloop.html>) is called.

How to do it...

In this example, we look at how to use the event loop statements provided by the `asyncio` library, in order to build an application that works in asynchronous mode.

In this example, we defined three tasks. Each task has an execution time determined by a time random parameter. Once the execution is finished, **Task A** calls **Task B**, **Task B** calls **Task C**, and **Task C** calls **Task A**.

The event loop will continue until a termination condition is met. As we can imagine, this example follows this asynchronous schema:



Asynchronous programming model

Let's have a look at the following steps:

1. Let's start by importing the libraries needed for our implementation:

```
import asyncio
import time
import random
```

2. Then, we define `task_A`, whose execution time is determined randomly and can vary from 1 to 5 seconds. At the end of the execution, if the termination condition is not satisfied, then the computation goes to `task_B`:

```
def task_A(end_time, loop):
    print ("task_A called")
    time.sleep(random.randint(0, 5))
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, task_B, end_time, loop)
    else:
        loop.stop()
```

3. Here, `task_B` is defined. Its execution time is determined randomly and can vary from 4 to 7 seconds. At the end of the execution, if the termination condition is not satisfied, then the computation goes to `task_B`:

```
def task_B(end_time, loop):
    print ("task_B called ")
    time.sleep(random.randint(3, 7))
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, task_C, end_time, loop)
    else:
        loop.stop()
```

4. Then, `task_C` is implemented. Its execution time is determined randomly and can vary from 6 to 10 seconds. At the end of the execution, if the termination condition is not satisfied, then the computation goes back to `task_A`:

```
def task_C(end_time, loop):
    print ("task_C called")
    time.sleep(random.randint(5, 10))
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, task_A, end_time, loop)
    else:
        loop.stop()
```

5. The next statement defines the `loop` parameter, which simply gets the current event loop:

```
loop = asyncio.get_event_loop()
```

6. The `end_loop` value defines the termination condition. The execution of this example code must last 60 seconds:

```
end_loop = loop.time() + 60
```

7. Then, let's request the execution of `task_A`:

```
loop.call_soon(task_A, end_loop, loop)
```

8. Now, we set a long duration cycle that continues to respond to events until it is stopped:

```
loop.run_forever()
```

9. Now, close the event loop:

```
loop.close()
```

How it works...

In order to manage the execution of the three tasks, `task_A`, `task_B`, and `task_C`, we need to capture the event loop:

```
loop = asyncio.get_event_loop()
```

Then, we schedule the first call to `task_A` by using the `call_soon` construct:

```
end_loop = loop.time() + 60
loop.call_soon(function_1, end_loop, loop)
```

Let's note the definition of `task_A`:

```
def task_A(end_time, loop):
    print ("task_A called")
    time.sleep(random.randint(0, 5))
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, task_B, end_time, loop)
    else:
        loop.stop()
```

The asynchronous behavior of the application is determined by the following parameters:

- `time.sleep(random.randint(0, 5))`: This defines the duration time of the task execution.
- `end_time`: This defines the upper time limit within `task_A` and makes the call to `task_B` through the `call_later` method.
- `loop`: This is the event loop captured previously with the `get_event_loop()` method.

After executing the task, `loop.time` is compared to `end_time`. If the execution time is within the maximum time (60 seconds), then the computation continues by calling `task_B`, otherwise, the computation ends, closing the event loop:

```
if (loop.time() + 1.0) < end_time:
    loop.call_later(1, task_B, end_time, loop)
else:
    loop.stop()
```

For the other two tasks, the operations are practically the same, but only the execution time and the call to the next task vary.

Now, let me summarize the situation:

1. `task_A` calls `task_B` with a random execution time between 1 and 5 seconds.
2. `task_B` calls `task_C` with a random execution time between 4 and 7 seconds.
3. `task_C` calls `task_A` with a random execution time between 6 and 10 seconds.

When the running time expires, the event loop must end:

```
loop.run_forever()  
loop.close()
```

A possible output of this example would be the following:

```
task_A called  
task_B called  
task_C called  
task_A called  
task_B called  
task_C called  
task_A called  
task_B called  
task_C called  
task_A called  
task_B called  
task_C called  
task_A called  
task_B called  
task_C called
```

There's more...

Asynchronous event programming replaces a type of concurrent programming in which several parts of the program are executed simultaneously by different threads that have access to the same data in memory, thus giving rise to the problem of critical runs. At the same time, it has become essential to be able to exploit the different cores of modern CPUs because, in certain areas, performance similar to that made available by the latter can no longer be achieved with a single-core processor.

See also

Here is a good introduction to `asyncio`: <https://hackernoon.com/a-simple-introduction-to-pythons-asyncio-595d9c9ecf8c>.

Handling coroutines with `asyncio`

Throughout the various examples presented, we have seen that when a program becomes very long and complex, it is convenient to divide it into subroutines, each of which implements a specific task. However, subroutines cannot be executed independently, but only at the request of the main program, which is responsible for coordinating the use of subroutines.

In this section, we introduce a generalization of the concept of subroutines, known as coroutines: just like subroutines, coroutines compute a single computational step, but unlike subroutines, there is no `main` program to coordinate the results. The coroutines link themselves together to form a pipeline without any supervising function responsible for calling them in a particular order.

In a coroutine, the execution point can be suspended and resumed later, since the coroutine keeps track of the state of execution. Having a pool of coroutines, it is possible to interleave the computations: the first one runs until it *yields control back*, then the second runs and goes on down the line.

The interleaving is managed by the event loop, which was described in the *Managing the event loop with `asyncio`* recipe. It keeps track of all the coroutines and schedules when they will be executed.

Other important aspects of coroutines are as follows:

- Coroutines allow for multiple entry points that can yield multiple times.
- Coroutines can transfer execution to any other coroutine.

The term *yield* is used here to describe a coroutine pausing and passing the control flow to another coroutine.

Getting ready

We will use the following notation to work with coroutines:

```
import asyncio

@asyncio.coroutine
def coroutine_function(function_arguments):
    .....
    DO_SOMETHING
    .....
```

Coroutines use the `yield from` syntax introduced in PEP 380 (read more at <https://www.python.org/dev/peps/pep-0380/>) to stop the execution of the current computation and suspends the coroutine's internal state.

In particular, in the case of `yield from future`, the coroutine is suspended until `future` is done, then the result of `future` will be propagated (or raise an exception); in the case of `yield from coroutine`, the coroutine waits for another coroutine to produce a result that will be propagated (or raise an exception).

As we shall see in the next example, in which the coroutines will be used to simulate a finite state machine, we will use the `yield from` coroutine notation.



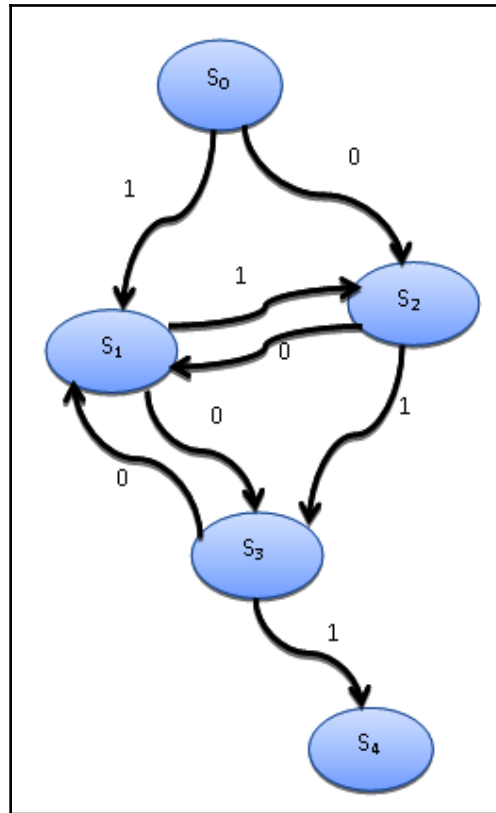
More on coroutines with `asyncio` are available at <https://docs.python.org/3.5/library/asyncio-task.html>.

How to do it...

In this example, we see how to use coroutines to simulate a finite state machine with five states.

A **finite state machine** or **finite state automaton** is a mathematical model that is widely used in engineering disciplines, but also in sciences such as mathematics and computer science.

The automaton that we want to simulate the behavior of using coroutines is as follows:



Finite-state machine

The states of the system are **S0**, **S1**, **S2**, **S3**, and **S4**, with **0** and **1**: the values for which the automaton can pass from one state to the next state (this operation is called a *transition*). So, for example, state **S0** can pass to state **S1**, but only for the value **1**, and **S0** can pass to state **S2**, but only for the value **0**.

The following Python code simulates a transition of the automaton from state **S0** (the start state), up to state **S4** (the end state):

1. The first step is obviously to import the relevant libraries:

```
import asyncio
import time
from random import randint
```

2. Then, we define the coroutine relative to `start_state`. The `input_value` parameter is evaluated randomly; it can be 0 or 1. If it is 0, then the control goes to coroutine `state2`; otherwise, it changes to coroutine `state1`:

```
@asyncio.coroutine
def start_state():
    print('Start State called\n')
    input_value = randint(0, 1)
    time.sleep(1)
    if input_value == 0:
        result = yield from state2(input_value)
    else:
        result = yield from state1(input_value)
    print('Resume of the Transition:\nStart State calling'+ result)
```

3. Here is the coroutine for `state1`. The `input_value` parameter is evaluated randomly; it can be 0 or 1. If it is 0, then the control goes to `state2`; otherwise, it changes to `state1`:

```
@asyncio.coroutine
def state1(transition_value):
    output_value = 'State 1 with transition value = %s\n' % \
        transition_value

    input_value = randint(0, 1)
    time.sleep(1)
    print('...evaluating...')
    if input_value == 0:
        result = yield from state3(input_value)
    else:
        result = yield from state2(input_value)
    return output_value + 'State 1 calling %s' % result
```

4. The coroutine for `state1` has the `transition_value` argument that allowed the passage of the state. Also, in this case, `input_value` is randomly evaluated. If it is 0, then the state transitions to `state3`; otherwise, the control changes to `state2`:

```
@asyncio.coroutine
def state2(transition_value):
    output_value = 'State 2 with transition value = %s\n' % \
        transition_value

    input_value = randint(0, 1)
    time.sleep(1)
    print('...evaluating...')
    if input_value == 0:
        result = yield from state1(input_value)
```

```

else:
    result = yield from state3(input_value)
    return output_value + 'State 2 calling %s' % result

```

5. The coroutine for `state3` has the `transition_value` argument, which allowed the passage of the state. `input_value` is randomly evaluated. If it is 0, then the state transitions to `state1`; otherwise, the control changes to `end_state`:

```

@asyncio.coroutine
def state3(transition_value):
    output_value = 'State 3 with transition value = %s\n' % \
                                                            transition_value

    input_value = randint(0, 1)
    time.sleep(1)
    print('...evaluating...')
    if input_value == 0:
        result = yield from state1(input_value)
    else:
        result = yield from end_state(input_value)
    return output_value + 'State 3 calling %s' % result

```

6. `end_state` prints out the `transition_value` argument, which allowed the passage of the state, and then stops the computation:

```

@asyncio.coroutine
def end_state(transition_value):
    output_value = 'End State with transition value = %s\n' % \
                                                            transition_value

    print('...stop computation...')
    return output_value

```

7. In the `__main__` function, the event loop is acquired, and then we start the simulation of the finite state machine, calling the automaton's `start_state`:

```

if __name__ == '__main__':
    print('Finite State Machine simulation with Asyncio Coroutine')
    loop = asyncio.get_event_loop()
    loop.run_until_complete(start_state())

```

How it works...

Each state of the automaton has been defined by using the decorator:

```
@asyncio.coroutine
```

For example, state **S0** is defined here:

```
@asyncio.coroutine
def StartState():
    print ("Start State called \n")
    input_value = randint(0,1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else :
        result = yield from State1(input_value)
```

The transition to the next state is determined by `input_value`, which is defined by the `randint (0,1)` function of Python's `random` module. This function randomly provides a value of 0 or 1.

In this manner, `randint` randomly determines the state to which the finite state machine will pass:

```
input_value = randint(0,1)
```

After determining the values to pass, the coroutine calls the next coroutine using the `yield from` command:

```
if (input_value == 0):
    result = yield from State2(input_value)
else :
    result = yield from State1(input_value)
```

The `result` variable is the value that each coroutine returns. It is a string, and, at the end of the computation, we can reconstruct the transition from the initial state of the automaton, `start_state`, up to `end_state`.

The main program starts the evaluation inside the event loop:

```
if __name__ == "__main__":
    print("Finite State Machine simulation with Asyncio Coroutine")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(StartState())
```

Running the code, we have an output like this:

```
Finite State Machine simulation with Asyncio Coroutine
Start State called
...evaluating...
...evaluating...
...evaluating...
...evaluating...
...stop computation...
Resume of the Transition :
Start State calling State 1 with transition value = 1
State 1 calling State 2 with transition value = 1
State 2 calling State 1 with transition value = 0
State 1 calling State 3 with transition value = 0
State 3 calling End State with transition value = 1
```

There's more...

Before Python 3.5 was released, the `asyncio` module used generators to mimic asynchronous calls and, therefore, had a different syntax than the current version of Python 3.5.

Python 3.5 introduced the `async` and `await` keywords. Notice the lack of parentheses around the `await func()` call.

The following is an example of "Hello, world!", using `asyncio` with the new syntax introduced by Python 3.5+:

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

See also

Coroutines in Python are well described here: <https://www.geeksforgeeks.org/coroutine-in-python/>.

Manipulating tasks with `asyncio`

The `asyncio` module is designed to handle asynchronous processes and concurrent task execution over an event loop. It also provides the `asyncio.Task()` class for the purpose of wrapping coroutines in a task (<https://docs.python.org/3/library/asyncio-task.html>). Its use is to allow independently running tasks to run concurrently with other tasks over the same event loop.

When a coroutine is wrapped in a task, it connects `Task` to the event loop and then runs automatically when the loop is started, thus providing a mechanism for automatically driving the coroutine.

The `asyncio` module provides the `asyncio.Task(coroutine)` method to handle computations with tasks; moreover, `asyncio.Task(coroutine)` schedules the execution of a coroutine (<https://docs.python.org/3/library/asyncio-task.html>).

A task is responsible for executing a coroutine object in an *event loop*.

If the wrapped coroutine uses the `yields from future` notation, as already described in the *Handling coroutines with `asyncio`* section, then the task suspends the execution of the wrapped coroutine and awaits completion of the future.

When the future is done, the execution of the wrapped coroutine restarts with the result or the exception of the future. Also, we must note that an event loop only runs one task at a time. Other tasks may run in parallel if other event loops are running in different threads.

While a task waits for the completion of a future, the event loop executes a new task.

How to do it...

In this example, we show how three mathematical functions can be executed concurrently by the `asyncio.Task()` statement:

1. Of course, let's start by importing the `asyncio` library:

```
import asyncio
```

2. In the first coroutine, the factorial function is defined:

```
@asyncio.coroutine
def factorial(number):
    f = 1
    for i in range(2, number + 1):
        print("Asyncio.Task: Compute factorial(%s)" % (i))
        yield from asyncio.sleep(1)
        f *= i
    print("Asyncio.Task - factorial(%s) = %s" % (number, f))
```

3. After which, the second function is defined—the fibonacci function:

```
@asyncio.coroutine
def fibonacci(number):
    a, b = 0, 1
    for i in range(number):
        print("Asyncio.Task: Compute fibonacci (%s)" % (i))
        yield from asyncio.sleep(1)
        a, b = b, a + b
    print("Asyncio.Task - fibonacci(%s) = %s" % (number, a))
```

4. The last function to be executed concurrently is the binomial coefficient:

```
@asyncio.coroutine
def binomial_coefficient(n, k):
    result = 1
    for i in range(1, k + 1):
        result = result * (n - i + 1) / i
        print("Asyncio.Task: Compute binomial_coefficient (%s)" %
              (i))
        yield from asyncio.sleep(1)
    print("Asyncio.Task - binomial_coefficient(%s , %s) = %s" %
          (n, k, result))
```

5. In the `__main__` function, `task_list` contains the functions that must be performed in parallel using the `asyncio.Task` function:

```
if __name__ == '__main__':
    task_list = [asyncio.Task(factorial(10)),
                 asyncio.Task(fibonacci(10)),
                 asyncio.Task(binomial_coefficient(20, 10))]
```

6. Finally, we acquire the event loop and start the computation:

```
loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(task_list))
loop.close()
```

How it works...

Each coroutine is defined by the `@asyncio.coroutine` annotation (called the *decorator*):

```
@asyncio.coroutine
def function (args):
    do something
```

To run in parallel, each function is an argument of the `asyncio.Task` module, and therefore, they are included in `task_list`:

```
if __name__ == '__main__':
    task_list = [asyncio.Task(factorial(10)),
                 asyncio.Task(fibonacci(10)),
                 asyncio.Task(binomial_coefficient(20, 10))]
```

Then, we get the event loop:

```
loop = asyncio.get_event_loop()
```

Finally, we add the execution of `task_list` to the event loop:

```
loop.run_until_complete(asyncio.wait(task_list))
loop.close()
```



Note that the `asyncio.wait(task_list)` statement waits for the given coroutines to complete.

The output for the preceding code looks like this:

```
Asyncio.Task: Compute factorial(2)
Asyncio.Task: Compute fibonacci(0)
Asyncio.Task: Compute binomial_coefficient(1)
Asyncio.Task: Compute factorial(3)
Asyncio.Task: Compute fibonacci(1)
Asyncio.Task: Compute binomial_coefficient(2)
Asyncio.Task: Compute factorial(4)
Asyncio.Task: Compute fibonacci(2)
Asyncio.Task: Compute binomial_coefficient(3)
Asyncio.Task: Compute factorial(5)
Asyncio.Task: Compute fibonacci(3)
Asyncio.Task: Compute binomial_coefficient(4)
Asyncio.Task: Compute factorial(6)
Asyncio.Task: Compute fibonacci(4)
Asyncio.Task: Compute binomial_coefficient(5)
Asyncio.Task: Compute factorial(7)
Asyncio.Task: Compute fibonacci(5)
Asyncio.Task: Compute binomial_coefficient(6)
Asyncio.Task: Compute factorial(8)
Asyncio.Task: Compute fibonacci(6)
Asyncio.Task: Compute binomial_coefficient(7)
Asyncio.Task: Compute factorial(9)
Asyncio.Task: Compute fibonacci(7)
Asyncio.Task: Compute binomial_coefficient(8)
Asyncio.Task: Compute factorial(10)
Asyncio.Task: Compute fibonacci(8)
Asyncio.Task: Compute binomial_coefficient(9)
Asyncio.Task - factorial(10) = 3628800
Asyncio.Task: Compute fibonacci(9)
Asyncio.Task: Compute binomial_coefficient(10)
Asyncio.Task - fibonacci(10) = 55
Asyncio.Task - binomial_coefficient(20, 10) = 184756.0
```

There's more...

asyncio provides other ways to schedule tasks using the `ensure_future()` or `AbstractEventLoop.create_task()` methods, which both accept a coroutine object.

See also

More on `asyncio` and tasks can be found here: <https://tutorialedge.net/python/concurrency/asyncio-tasks-tutorial/>.

Dealing with `asyncio` and futures

Another key component of the `asyncio` module is the `asyncio.Future` class. It is very similar to `concurrent.Futures`, but, of course, it is adapted to the main mechanism of `asyncio`: the event loop.

The `asyncio.Future` class represents a result (but can also be an exception) that is not yet available.

Hence, it represents an abstraction of something that is yet to be achieved. The callbacks that have to process any results are, in fact, added to instances of this class.

Getting ready

To define a `future` object, the following syntax must be used:

```
future = asyncio.Future
```

The main methods to manage this object are the following:

- `cancel()`: This cancels the `future` object and schedules callbacks.
- `result()`: This returns the result that this `future` represents.
- `exception()`: This returns the exception that was set on this `future`.
- `add_done_callback(fn)`: This adds a callback to be run when `future` is done.
- `remove_done_callback(fn)`: This removes all instances of a callback from the call when done.
- `set_result(result)`: This marks `future` as done and sets its result.
- `set_exception(exception)`: This marks `future` as done and sets an exception.

How to do it...

The following example shows how to use the `asyncio.Future` class for the management of two coroutines: `first_coroutine` and `second_coroutine`, which perform the following tasks. `first_coroutine` performs the sum of the first N integers, and `second_coroutine` performs the factorial of N :

1. Now, let's import the relevant libraries:

```
import asyncio
import sys
```

2. `first_coroutine` implements the sum function of the first N integers:

```
@asyncio.coroutine
def first_coroutine(future, num):
    count = 0
    for i in range(1, num + 1):
        count += i
    yield from asyncio.sleep(1)
    future.set_result('First coroutine (sum of N integers)\
        result = %s' % count)
```

3. In `second_coroutine`, we still implement the factorial function:

```
@asyncio.coroutine
def second_coroutine(future, num):
    count = 1
    for i in range(2, num + 1):
        count *= i
    yield from asyncio.sleep(2)
    future.set_result('Second coroutine (factorial) result = %s' %\
        count)
```

4. Using the `got_result` function, we print the output of the computation:

```
def got_result(future):
    print(future.result())
```

5. In the main function, the `num1` and `num2` parameters must be set by the user. They will be used as parameters for the functions implemented by the first and second coroutines:

```
if __name__ == "__main__":
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
```

6. Now, let's take the event loop:

```
loop = asyncio.get_event_loop()
```

7. Here, the futures are defined by the `asyncio.Future` function:

```
future1 = asyncio.Future()
future2 = asyncio.Future()
```

8. The two coroutines—`first_couroutine` and `second_couroutine`—included in the `tasks` list have the `future1` and `future2` futures, the user-defined arguments, and the `num1` and `num2` parameters:

```
tasks = [first_coroutine(future1, num1),
         second_coroutine(future2, num2)]
```

9. The futures have added a callback:

```
future1.add_done_callback(got_result)
future2.add_done_callback(got_result)
```

10. Then, the `tasks` list is added to the event loop, so that the computation can begin:

```
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

How it works...

In the main program, we define the future objects, `future1` and `future2` respectively, using via the `asyncio.Future()` directive:

```
if __name__ == "__main__":
    future1 = asyncio.Future()
    future2 = asyncio.Future()
```

In defining the tasks, we pass the future objects as an argument of the two coroutines `first_couroutine` and `second_couroutine`:

```
tasks = [first_coroutine(future1, num1),
         second_coroutine(future2, num2)]
```

Finally, we add a callback to be run when `future` is done:

```
future1.add_done_callback(got_result)
future2.add_done_callback(got_result)
```

Here, `got_result` is a function that prints the result of `future`:

```
def got_result(future):
    print(future.result())
```

In the coroutine, we pass the `future` object as an argument. After the computation, we set sleep times of 3 seconds for the first coroutine and 4 seconds for the second one:

```
yield from asyncio.sleep(sleep_time)
```

The following output is obtained by executing the command with different values:

```
> python asyncio_and_futures.py 1 1
First coroutine (sum of N integers) result = 1
Second coroutine (factorial) result = 1

> python asyncio_and_futures.py 2 2
First coroutine (sum of N integers) result = 2
Second coroutine (factorial) result = 2

> python asyncio_and_futures.py 3 3
First coroutine (sum of N integers) result = 6
Second coroutine (factorial) result = 6

> python asyncio_and_futures.py 5 5
First coroutine (sum of N integers) result = 15
Second coroutine (factorial) result = 120

> python asyncio_and_futures.py 50 50
First coroutine (sum of N integers) result = 1275
Second coroutine (factorial) result =
304140932017133780436126081660647688443776415689605120000000000000
First coroutine (sum of N integers) result = 1275
```

There's more...

We can invert the output results, that is, have the output of `second_coroutine` first, by simply swapping the sleep time between the coroutines: `yield from asyncio.sleep(2)` in the `first_coroutine` definition, and `yield from asyncio.sleep(1)` in the `second_coroutine` definition. This can be shown by the following example:

```
> python asyncio_and_future.py 1 10
second coroutine (factorial) result = 3628800
first coroutine (sum of N integers) result = 1
```

See also

More examples of `asyncio` and futures can be found at <https://www.programcreek.com/python/example/102763/asyncio.futures>.

6 Distributed Python

This chapter will introduce some important Python modules for distributed computing. In particular, we will describe the `socket` module, which allows you to implement simple applications distributed through the client-server model.

Then, we will introduce the Celery module, which is a powerful Python framework that is used to manage distributed tasks. Finally, we will describe the `Pyro4` module, which allows you to call methods that are used in different processes, potentially on a different machine.

In this chapter, we will cover the following recipes:

- Introducing distributed computing
- Using the Python `socket` module
- Distributed task management with Celery
- Remote Method Invocation (RMI) with `Pyro4`

Introducing distributed computing

Parallel and *distributed computing* are similar technologies designed to increase the amount of processing power available for a specific task. Generally, these methods are used to solve problems that require large computational capabilities.

When the problem is divided into many small pieces, individual sections of the problem can be calculated by many processors simultaneously. This allows more processing power to be exercised on the problem than can be provided by a single processor.

The main difference between parallel and distributed processing is that parallel configurations include many processors within a single system, while distributed configurations exploit the processing power of many computers simultaneously.

Let's look at the other differences:

Parallel processing	Distributed processing
Parallel processing has the advantage of providing reliable processing power with a very low degree of latency.	Distributed processing is not extremely efficient on a processor-by-processor basis, as the data must travel over the network rather than through the internal connections of a single system.
By concentrating all the processing power in one system, speed loss due to data transfer is minimized.	Each processor will contribute much less processing power than any processor in a parallel system since data transfer creates a bottleneck that limits processing power.
The only real limit is the number of processors incorporated in the system.	The system is almost infinitely scalable since there is no actual upper limit to the number of processors in a distributed system.

However, in the context of computer applications, it is customary to distinguish between local and distributed architectures:

Local architectures	Distributed architectures
All the components are on the same machine.	Applications and components can reside on different nodes that are connected by a network.

The advantages of using distributed computing consist mainly of the possibility of concurrent use of the programs, the centralization of the data, and the distribution of the processing load, which all come at the price of greater complexity, especially with communication between the various components.

Types of distributed applications

Distributed applications can be classified according to the degree of distribution:

- **Client-server applications**
- **Multi-level applications**

Client-server applications

There are only two levels and the operations are carried out entirely on the server. As an example, we can mention the classic static or dynamic websites. The tools for the realization of these types of applications are the network sockets, whose programming is possible in various languages, including C, C ++, Java, and, of course, Python.

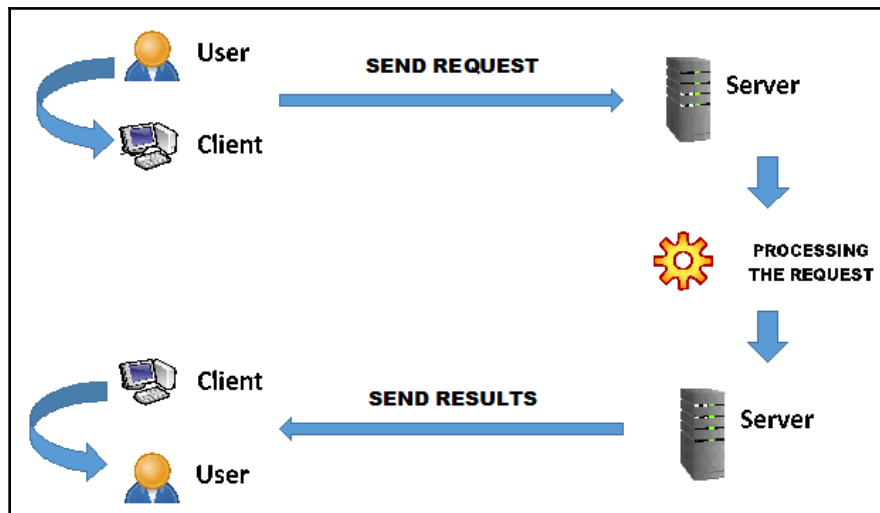
The term *client-server system* refers to a network architecture in which a client computer or client terminal is generally connected to a server for the use of a certain service; for example, the sharing of a certain hardware/software resource with other clients, or relying on the underlying protocol architecture.

Client-server architecture

The client-server architecture is a system that realizes a distribution of both processing and data. The central element of the architecture is the server. The server can be considered both from a logical point of view and from a physical point of view. From the physical point of view—hardware—a server is a machine dedicated to running a software server.

From a logical point of view, a server is software. The server, as a logical process, provides services to other processes that take on the role of requester or client. Generally, the server does not send the results to the requester until the results are requested by the client.

A feature that distinguishes the client from its server is that the client can initiate a transaction with the server, while the server can never initiate a transaction with the client on its own initiative:



Client-server architecture

In fact, the specific tasks of the client are to start transactions, request specific services, notify the completion of the service, and receive results from the server, as shown in the preceding diagram.

Client-server communications

Communication between clients and servers can take place using a variety of mechanisms—from geographic to local networks, up to communication services—between applications at the OS level. Furthermore, a client-server architecture must be independent of the physical connection method that exists between the client and the server.

It should also be noted that it is not necessary for a client-server process to reside on physically separate systems. In fact, the server process and the client process can reside on the same computing platform.

The main objective of the client-server architecture, in the context of data management, is to allow client applications to access data managed by the servers. The server (understood in a logical sense as software) is often running on a remote system (for example, in another city or on a local network).

Therefore, client-server applications are often associated with distributed processing.

TCP/IP client-server architecture

The TCP/IP connection establishes a point-to-point connection between two applications. The extremes of this connection are marked by an IP address, which identifies the workstation by a port number, which makes it possible to have several connections that are connected to independent applications on the same workstation.

Once the connection is established and the protocol can exchange data over it, the underlying TCP/IP protocol takes care of sending this data, divided into packets, from one end of the connection to the other. In particular, the TCP protocol deals with assembling and disassembling the packets, as well as managing the handshaking that guarantees the reliability of the connection, while the IP protocol takes care of transporting the individual packets and the choice of the best routing of the packets along with the network.

This mechanism underlies the robustness of the TCP/IP protocol, which, in turn, represents one of the reasons for the development of the protocol itself in the military sphere (ARPANET).

The various existing standard applications (such as web browsing, file transfer, and email) use standardized application protocols, such as HTTP, FTP, POP3, IMAP, and SMTP.

Each specific client-server application must instead define and apply its own proprietary application protocol. This can involve the exchange of data in blocks of a fixed size (which is the simplest solution).

Multi-level applications

There are a greater number of levels that enable the processing load of the servers to be alleviated. Those that are, in fact, subdivided are the functionalities of the server-side, leaving the characteristics of the client part that has the task of hosting the application interface largely unchanged. An example of this type of architecture is that of the three-tier model, having a structure divided into three layers or levels:

- Frontend or presentation tier or interface
- Middle tier or application logic
- Backend or data tier or persistent data management

This nomenclature is typical of web applications. More generally, it is possible to refer to a subdivision in three levels that are applicable to any software application, which is the following:

- **Presentation Layer (PL):** This is the visualization part of the data (such as modules and controls of input) necessary for the user interface.
- **Business Logic Layer (BLL):** This is the main part of the application, which defines the various entities and their relationships independently of the methods of presentation available to the user and saved in the archives.
- **Data Access Layer (DAL):** This contains everything necessary for the management of persistent data (essentially, database management systems).

This chapter will present some of the solutions proposed by Python for the implementation of distributed architectures. We will begin by describing the `socket` module with which we will implement some examples of the fundamental client-server model.

Using the Python socket module

A socket is a software object that allows data to be sent and received between remote hosts (via a network) or between local processes, such as **Inter-Process Communication (IPC)**.

Sockets were invented at Berkeley as part of the **BSD Unix** project. They are based precisely on the management model of input and output of Unix files. In fact, the operations of opening, reading, writing, and closing a socket occur in the same way as the management of Unix files, but with the difference that should be considered are the useful parameters for communication, such as addresses, port numbers, and protocols.

The success and spread of socket technology have gone hand in hand with the development of the internet. In fact, the combination of sockets with the internet has made communication between machines that are of any type, and/or are scattered throughout the world, incredibly easy (at least when compared with other systems).

Getting ready

The socket Python module exposes low-level C APIs for communication over a network using the **BSD** (short for **Berkeley Software Distribution**) socket interface.

This module includes the `Socket` class, which includes the main methods for managing the following tasks:

- `socket ([family [, type [, protocol]])`: Builds a socket using the following as arguments:
 - The family address, which can be `AF_INET` (default), `AF_INET6`, or `AF_UNIX`
 - The type socket, which can be `SOCK_STREAM` (default), `SOCK_DGRAM`, or perhaps one of the other "SOCK_" constants
 - The protocol number (that is usually zero)
- `gethostname ()`: Returns the current IP address of the machine.
- `accept ()`: Returns the following pair of values (`conn` and `address`), where `conn` is a socket type object (to send/receive data on the connection), while `address` is the address connected to the socket on the other end of the connection.
- `bind (address)`: Associates the socket with the address server.



This method historically accepted a couple of parameters for the `AF_INET` addresses instead of a single tuple.

- `close ()`: Provides the option to clean up the connection once communication with the client is finished. The sockets are closed and collected by the garbage collector.
- `connect (address)`: Connects a remote socket to an address. The address format depends on the family address.

How to do it...

In the following example, the server is listening on a default port, and by following a TCP/IP connection, the client sends to the server the date and time that the connection was established.

Here is the server implementation for `server.py`:

1. Import the relevant Python modules:

```
import socket
import time
```

2. Create a new socket using the given address, socket type, and protocol number:

```
serversocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

3. Get the local machine name (host):

```
host=socket.gethostname()
```

4. Set the port number:

```
port=9999
```

5. Connect (bind) the socket to host and to port:

```
serversocket.bind((host,port))
```

6. Listen for connections made to the socket. The argument of 5 specifies the maximum number of connections in the queue. The maximum value depends on the system (usually, it is 5) and the minimum value is always 0:

```
serversocket.listen(5)
```

7. Establish a connection:

```
while True:
```

8. Then, the connection is accepted. The return value is a pair (conn, address), where conn is a new socket object that is used to send and receive data, and address is the address linked to the socket. Once accepted, a new socket is created and it will have its own identifier. This new socket is only used with this particular client:

```
clientsocket, addr=serversocket.accept()
```

9. The address and the port that is connected are printed out:

```
print ("Connected with[addr], [port]%"%str(addr))
```

10. currentTime is evaluated:

```
currentTime=time.ctime(time.time())+"\r\n"
```

11. The following statement sends data to the socket, returning the number of bytes sent:

```
clientsocket.send(currentTime.encode('ascii'))
```

12. The following statement indicates the socket closure (that is, the communication channel); all subsequent operations on the socket will fail. The sockets are automatically closed when they are rejected, but it is always recommended to close them with the close() operation:

```
clientsocket.close()
```

The code for the client (`client.py`) is as follows:

1. Import the `socket` library:

```
import socket
```

2. The `socket` object is then created:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

3. Get the local machine name (`host`):

```
host=socket.gethostname()
```

4. Set the port number:

```
port=9999
```

5. Set up the connection to `host` and `port`:

```
s.connect((host,port))
```



The maximum number of bytes that can be received is no more than 1,024 bytes: `(tm=s.recv(1024))`.

6. Now, close the connection and finally print out the connection time to the server:

```
s.close()  
print ("Time connection server:%s"%tm.decode('ascii'))
```

How it works...

Clients and servers create their respective sockets, and the server listens to them on a port. The client makes a connection request to the server. It should be noted that we can have two different port numbers because one could only be dedicated to outgoing traffic, and the other could only be dedicated to entry. This depends on the host configuration.

Essentially, the client's local port does not necessarily coincide with the server's remote port. The server receives the request and, if accepted, a new connection is created. Now, the client and server communicate through a virtual channel, between the socket and the server, which is created specifically for the data flow of the data socket connection.

Consistent with what was mentioned in the first phase, the server creates the data socket because the first one is used exclusively for managing requests. Therefore, it is possible that there are many clients communicating with the server using the data socket created by the server for them. The TCP protocol is connection-oriented, which means that when there is no longer a need to communicate, the client communicates this to the server and the connection is closed.

To run the example, execute the server:

```
C:\>python server.py
```

Then, execute the client (in a different Windows terminal):

```
C:\>python client.py
```

The result on the client side should report the address (`addr`) and report port as connected:

```
Connected with[addr], [port] ('192.168.178.11', 58753)
```

However, on the server side, the result should be as follows:

```
Time connection server:Sun Mar 31 20:59:38 2019
```

There's more...

With a small change to the previous code, we can create a simple client-server application for file transfer. The server instantiates the socket and waits for the connection instance from the client. Once connected to the server, the client starts the data transfer.

The data to be transferred, which is in the `mytext.txt` file, is copied byte by byte and sent to the server through the call to the `conn.send` function. The server then receives the data and writes it to a second file, `received.txt`.

The source code for `client2.py` is as follows:

```
import socket
s = socket.socket()
host = socket.gethostname()
port = 60000
s.connect((host, port))
s.send('HelloServer!'.encode())
with open('received.txt', 'wb') as f:
    print('file opened')
    while True :
```



```
        print ('receiving data...')
        data=s.recv(1024)
        if not data:
            break
        print ('Data=>',data.decode())
        f.write(data)
    f.close()
    print ('Successfully get the file')
    s.close()
    print ('connection closed')
```

Here is the source code for `client.py`:

```
import socket
port=60000
s =socket.socket()
host=socket.gethostname()
s.bind((host,port))
s.listen(15)
print('Server listening....')
while True :
    conn,addr=s.accept()
    print ('Got connection from',addr)
    data=conn.recv(1024)
    print ('Server received',repr(data.decode()))
    filename='mytext.txt'
    f =open(filename,'rb')
    l =f.read(1024)
    while True:
        conn.send(l)
        print ('Sent',repr(l.decode()))
        l =f.read(1024)
        f.close()
        print ('Done sending')
        conn.send('->Thank you for connecting'.encode())
        conn.close()
```

Types of sockets

We can distinguish between the following three types of sockets, which are characterized by connection modes:

- **Stream sockets:** These are connection-oriented sockets, and they are based on reliable protocols such as TCP or SCTP.
- **Datagram sockets:** These are not connection-oriented (connectionless) sockets, and are based on the fast but unreliable UDP protocol.
- **Raw socket (raw IP):** The transport level is bypassed, and the header is accessible at the application level.

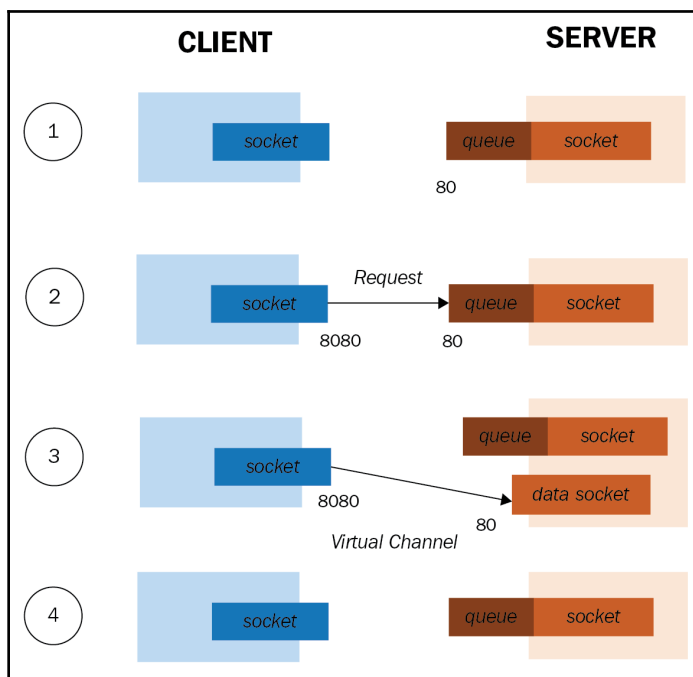
Stream sockets

We will see more in particular of this type of socket only. Being based on transport layer protocols such as TCP, they guarantee a reliable, full-duplex, and connection-oriented communication, with a variable-length byte stream.

Communication through this socket consists of these phases:

1. **Creation of sockets:** Clients and servers create their respective sockets, and the server listens to them on a port. Since the server can create multiple connections with different clients (but also with the same one), it needs a queue to handle the various requests.
2. **Connection request:** The client requests a connection to the server. Note that we can have different port numbers because one could only be assigned to the outgoing traffic, and the other only to entry. This depends on the host configuration. Essentially, the client's local port does not necessarily coincide with the server's remote port. The server receives the request and, if accepted, a new connection is created. In the diagram, the port of the client socket is 8080, while for the socket server, the port is 80.
3. **Communication:** Now, the client and server communicate through a virtual channel, between the client's socket, and a new socket (server side), created specifically for the data flow of this connection: a data socket. As it was mentioned in the first phase, the server creates the data socket because the first data socket is used exclusively for managing requests. Therefore, it is possible that there are many clients communicating with the server, each with the data socket specifically created by the server for them.
4. **Closure of the connection:** Since the TCP is a connection-oriented protocol when there is no longer a need to communicate, the client communicates it to the server, which deallocates the data socket.

The phases of communication through stream sockets are shown in the following diagram:



Stream socket phases

See also

More information on Python sockets can be found at <https://docs.python.org/3/howto/sockets.html>.

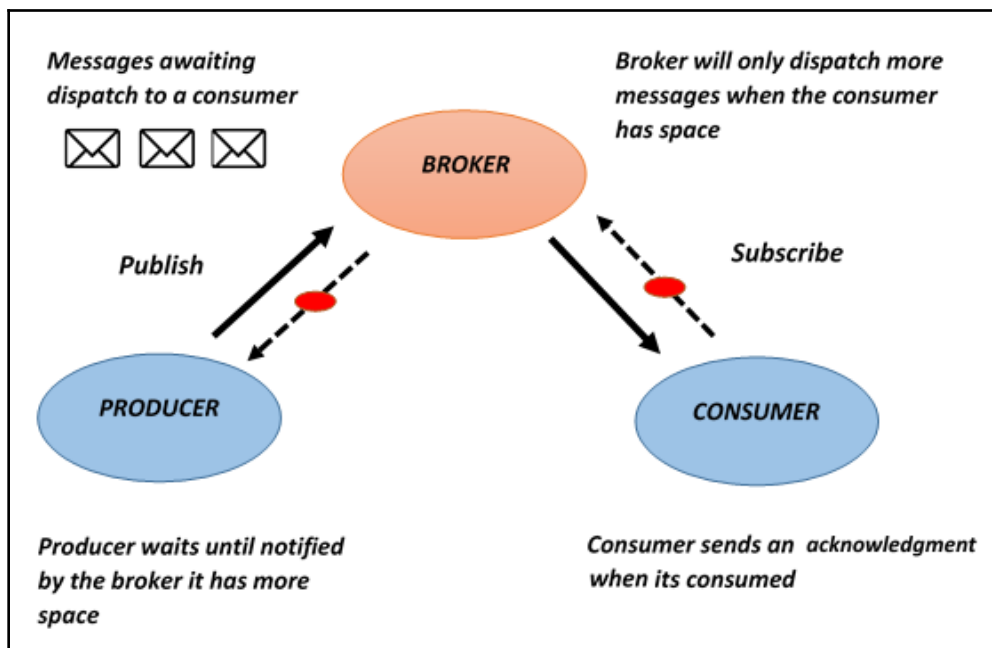
Distributed task management with Celery

Celery is a Python framework that manages distributed tasks by following the object-oriented middleware approach. Its main feature is handling many small tasks and distributing them on many computational nodes. Finally, the result of each task will then be reworked in order to compose the overall solution.

To use Celery, a message broker is required. This is an independent (from Celery) software component that has the function of middleware, which is used to send and receive messages to distributed task workers.

In fact, a message broker—also known as message middleware—deals with the exchange of messages in a communication network.: the addressing scheme of this type of middleware is no longer of the point-to-point type, but is message-oriented addressing.

The reference architecture, with which the message broker manages the exchange of messages, is based on the so-called publish/subscribe paradigm, which is depicted as follows:



Message broker architecture

Celery supports many types of brokers. However, the more complete ones are RabbitMQ and Redis.

Getting ready

To install Celery, use the `pip` installer as follows:

```
C:\>pip install celery
```

Then, a message broker must be installed. There are several choices available, but for our example, it is recommended to install RabbitMQ from the following link: <http://www.rabbitmq.com/download.html>.



RabbitMQ is a message-oriented middleware that implements the **Advanced Message Queuing Protocol (AMQP)**. The RabbitMQ server is written in the Erlang programming language, so in order to install it, you need to install Erlang after downloading it from <http://www.erlang.org/download.html>.

The steps involved are as follows:

1. To check the `celery` installation, first start the message broker (for example, RabbitMQ). Then, type the following:

```
C:\>celery --version
```

2. The following output, which indicates the `celery` version, is as follows:

```
4.2.2 (Windowlicker)
```

Next, let's learn about how to create and call a task using the `celery` module.

`celery` provides the following two methods to perform a call to a task:

- `apply_async(args[, kwargs[, ...]])`: This sends a task message.
- `delay(*args, **kwargs)`: This is a shortcut to send a task message, but it does not support execution options.



The `delay` method is easier to use because it is called as a **regular function**: `task.delay(arg1, arg2, kwarg1='x', kwarg2='y')`. However, for `apply_async`, the syntax is `task.apply_async(args=[arg1, arg2] kwargs={'kwarg1': 'x', 'kwarg2': 'y'})`.

Windows setup

In order to use Celery in a Windows environment, you must perform the following procedure:

1. Go to **System Properties | Environment Variables | User or System variables | New**.
2. Set the following values:
 - Variable name: `FORKED_BY_MULTIPROCESSING`
 - Variable value: `1`

The reason for this setup is because of Celery's dependence on the `billiard` package (<https://github.com/celery/billiard>), which uses the `FORKED_BY_MULTIPROCESSING` variable.

For more information on Celery's Windows setup, read <https://www.distributedpython.com/2018/08/21/celery-4-windows/>.

How to do it...

The task here is a sum of two numbers. To perform this easy task, we have to compose the `addTask.py` and `addTask_main.py` script files:

1. For `addTask.py`, start to import the Celery framework as follows:

```
from celery import Celery
```

2. Then, define the task. In our example, the task is a sum of two numbers:

```
app = Celery('tasks', broker='amqp://guest@localhost//')
@app.task
def add(x, y):
    return x + y
```

3. Now, import the `addTask.py` file that was defined previously into `addtask_main.py`:

```
import addTask
```

4. Then, call `addTask.py` to execute the sum of two numbers:

```
if __name__ == '__main__':
    result = addTask.add.delay(5,5)
```

How it works...

In order to use Celery, the first thing to do is to run the RabbitMQ service, and then execute the Celery worker server (that is, the `addTask.py` file script) by typing the following:

```
C:\>celery -A addTask worker --loglevel=info
```

The output is as follows:

```
Microsoft Windows [Versione 10.0.17134.648]
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Giancarlo>cd C:\Users\Giancarlo\Desktop\Python Parallel
Programming CookBook 2nd edition\Python Parallel Programming NEW
BOOK\chapter_6 - Distributed Python\esempi

C:\Users\Giancarlo\Desktop\Python Parallel Programming CookBook 2nd
edition\Python Parallel Programming NEW BOOK\chapter_6 - Distributed
Python\esempi>celery -A addTask worker --loglevel=info

----- celery@pc-giancarlo v4.2.2 (windowlicker)
---- **** -----
--- * *** * -- Windows-10.0.17134 2019-04-01 21:32:37
-- * - **** ---
- ** ----- [config]
- ** ----- .> app: tasks:0x1deb8f46940
- ** ----- .> transport: amqp://guest:**@localhost:5672//
- ** ----- .> results: disabled://
- *** --- * --- .> concurrency: 4 (prefork)
-- ***** ----- .> task events: OFF (enable -E to monitor tasks in this
worker)
--- ***** -----
----- [queues]
               .> celery exchange=celery(direct) key=celery

[tasks]
    . addTask.add

[2019-04-01 21:32:37,650: INFO/MainProcess] Connected to
amqp://guest:**@127.0.0.1:5672//
[2019-04-01 21:32:37,745: INFO/MainProcess] mingle: searching for neighbors
[2019-04-01 21:32:39,353: INFO/MainProcess] mingle: all alone
[2019-04-01 21:32:39,479: INFO/SpawnPoolWorker-2] child process 10712
calling self.run()
[2019-04-01 21:32:39,512: INFO/SpawnPoolWorker-3] child process 10696
calling self.run()
[2019-04-01 21:32:39,536: INFO/MainProcess] celery@pc-giancarlo ready.
[2019-04-01 21:32:39,551: INFO/SpawnPoolWorker-1] child process 6084
calling self.run()
```

```
[2019-04-01 21:32:39,615: INFO/SpawnPoolWorker-4] child process 2080
calling self.run()
```

Then, the second script is launched using Python:

```
C:\>python addTask_main.py
```

Finally, the result should be as follows in the first Command Prompt:

```
[2019-04-01 21:33:00,451: INFO/MainProcess] Received task:
addTask.add[6fc350a9-e925-486c-bc41-c239ebd96041]
[2019-04-01 21:33:00,452: INFO/SpawnPoolWorker-2] Task
addTask.add[6fc350a9-e925-486c-bc41-c239ebd96041] succeeded in 0.0s: 10
```

As you can see, the result is 10. Let's focus on the first script, `addTask.py`: in the first two lines of code, we create a `Celery` application instance that uses the `RabbitMQ` service broker:

```
from celery import Celery
app = Celery('addTask', broker='amqp://guest@localhost//')
```

The first argument in the `Celery` function is the name of the current module (`addTask.py`), and the second is the broker keyword argument; this indicates the URL that is used to connect the broker (`RabbitMQ`).

Now, let's introduce the task to be accomplished.

Each task must be added with the `@app.task` annotation (namely, decorator); the decorator helps `Celery` to identify which functions can be scheduled in the task queue.

After the decorator, we create the task that the workers can execute: this will be a simple function that performs the sum of two numbers:

```
@app.task
def add(x, y):
    return x + y
```

In the second script, `addTask_main.py`, we call our task by using the `delay()` method:

```
if __name__ == '__main__':
    result = addTask.add.delay(5,5)
```

Let's remember that this method is a shortcut to the `apply_async()` method, which gives us greater control over the task execution.

There's more...

Celery usage is very simple. It can be executed by using the following commands:

```
Usage: celery <command> [options]
```

Here, the options are as follows:

```
positional arguments:
  args
```

```
optional arguments:
  -h, --help            show this help message and exit
  --version              show program's version number and exit
```

```
Global Options:
  -A APP, --app APP
  -b BROKER, --broker BROKER
  --result-backend RESULT_BACKEND
  --loader LOADER
  --config CONFIG
  --workdir WORKDIR
  --no-color, -C
  --quiet, -q
```

The main commands are as follows:

```
+ Main:
| celery worker
| celery events
| celery beat
| celery shell
| celery multi
| celery amqp

+ Remote Control:
| celery status

| celery inspect --help
| celery inspect active
| celery inspect active_queues
| celery inspect clock
| celery inspect conf [include_defaults=False]
| celery inspect memdump [n_samples=10]
| celery inspect memsample
| celery inspect objgraph [object_type=Request] [num=200 [max_depth=10]]
| celery inspect ping
| celery inspect query_task [id1 [id2 [... [idN]]]]
```

```

| celery inspect registered [attr1 [attr2 [... [attrN]]]]
| celery inspect report
| celery inspect reserved
| celery inspect revoked
| celery inspect scheduled
| celery inspect stats

| celery control --help
| celery control add_consumer <queue> [exchange [type [routing_key]]]
| celery control autoscale [max [min]]
| celery control cancel_consumer <queue>
| celery control disable_events
| celery control election
| celery control enable_events
| celery control heartbeat
| celery control pool_grow [N=1]
| celery control pool_restart
| celery control pool_shrink [N=1]
| celery control rate_limit <task_name> <rate_limit (e.g., 5/s | 5/m |
5/h)>
| celery control revoke [id1 [id2 [... [idN]]]]
| celery control shutdown
| celery control terminate <signal> [id1 [id2 [... [idN]]]]
| celery control time_limit <task_name> <soft_secs> [hard_secs]

+ Utils:
| celery purge
| celery list
| celery call
| celery result
| celery migrate
| celery graph
| celery upgrade

+ Debugging:
| celery report
| celery logtool

+ Extensions:
| celery flower
-----

```

Celery protocol can be implemented in any language by using Webhooks (<https://developer.github.com/webhooks/>).

See also

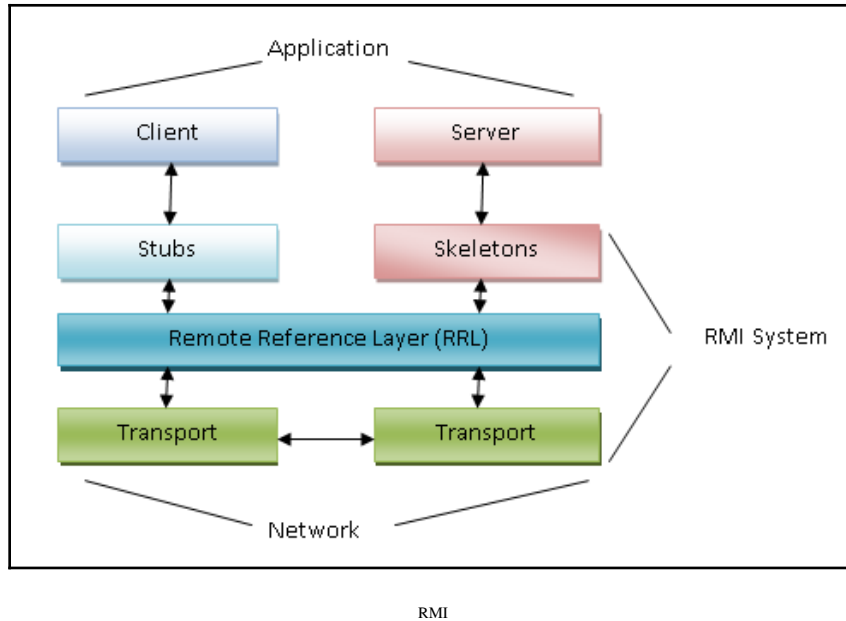
- More information on Celery can be found at <http://www.celeryproject.org/>.
- The recommended message brokers (https://en.wikipedia.org/wiki/Message_broker) are RabbitMQ (<https://en.wikipedia.org/wiki/RabbitMQ>) or Redis (<https://en.wikipedia.org/wiki/Redis>). Additionally, there is MongoDB (<https://en.wikipedia.org/wiki/MongoDB>), Beanstalk, Amazon SQS (https://en.wikipedia.org/wiki/Amazon_Simple_Queue_Service), CouchDB (https://en.wikipedia.org/wiki/Apache_CouchDB), and IronMQ (<https://www.iron.io/mq>).

RMI with Pyro4

Pyro is short for **Python Remote Objects**. It works exactly like the Java **RMI** (short for **Remote Method Invocation**) allowing to invoke a method of a remote object (belonging to a different process) exactly as if the object were local (belonging to the same process in which the invocation runs).

The use of an RMI mechanism, in an object-oriented system, involves significant advantages of uniformity and symmetry in the project, as this mechanism enables the modelling of interactions between distributed processes using the same conceptual tool.

As you can see from the following diagram, `Pyro4` enables objects to be distributed in a client/server style; this means that the main parts of a `Pyro4` system may switch from a client caller to a remote object, which is called to serve a function:



It is important to note that during the remote calling, there are always two distinct parts: a client and a server that accept and execute the client call.

Getting ready

The entire method of managing this mechanism in a distributed way is provided by `Pyro4`. To install the latest release of `Pyro4`, use the `pip` installer (Windows installation is used here) and add the following command:

```
C:\>pip install Pyro4
```

We are using the `pyro_server.py` and `pyro_client.py` codes for this recipe.

How to do it...

In this example, we'll see how to build and use a simple client-server communication using the Pyro4 middleware. The code for the client is `pyro_server.py`:

1. Import the Pyro4 library:

```
import Pyro4
```

2. Define the `Server` class that contains the `welcomeMessage()` method that will be exposed:

```
class Server(object):
    @Pyro4.expose
    def welcomeMessage(self, name):
        return ("Hi welcome " + str (name))
```



Note that the decorator, `@Pyro4.expose`, means that the preceding method will be remotely accessible.

3. The `startServer` function contains all the instructions that are used to start the server:

```
def startServer():
```

4. Next, build the `server` instance of the `Server` class:

```
server = Server()
```

5. Then, define the Pyro4 daemon:

```
daemon = Pyro4.Daemon()
```

6. To execute this script, we must run a Pyro4 statement to locate a nameserver:

```
ns = Pyro4.locateNS()
```

7. Register the object `server` as *Pyro object*; it will only be known inside the Pyro daemon:

```
uri = daemon.register(server)
```

8. Now, we can register the object server with a name in the nameserver:

```
ns.register("server", uri)
```

9. The function ends with a call to the daemon's `requestLoop` method. This starts the event loop of the server and waits for calls:

```
print("Ready. Object uri =", uri)
daemon.requestLoop()
```

10. Finally, call `startServer` via the main program:

```
if __name__ == "__main__":
    startServer()
```

Here is the code for the client (`pyro_client.py`):

1. Import the `Pyro4` library:

```
import Pyro4
```

2. The `Pyro4` API enables the developer to distribute objects in a transparent way. In this example, the client script sends requests to the server program in order to execute the `welcomeMessage()` method:

```
uri = input("What is the Pyro uri of the greeting object? ")
uri.strip()
name = input("What is your name? ").strip()
```

3. Then, the remote call is created:

```
server = Pyro4.Proxy("PYRONAME:server")
```

4. Finally, the client calls the server, printing a message:

```
print(server.welcomeMessage(name))
```

How it works...

The preceding example is composed of two main functions: `pyro_server.py` and `pyro_client.py`.

In `pyro_server.py`, the `Server` class object provides the `welcomeMessage()` method, returning a string equal to the name inserted in the client session:

```
class Server(object):
    @Pyro4.expose
    def welcomeMessage(self, name):
        return ("Hi welcome " + str (name))
```

Pyro4 uses daemon objects to dispatch incoming calls to appropriate objects. A server must create just one daemon that manages everything from its instance. Each server has a daemon that knows about all the Pyro objects that the server provides:

```
daemon = Pyro4.Daemon()
```

As for the `pyro_client.py` function, the remote call is first performed and creates a `Proxy` object. In particular, the Pyro4 client uses proxy objects to forward method calls to the remote objects, and then passes the results back to the calling code:

```
server = Pyro4.Proxy("PYRONAME:server")
```

In order to execute a client-server connection, we need to have a Pyro4 nameserver running. In Command Prompt, type the following:

```
C:\>python -m Pyro4.naming
```

After this, you'll see the following message:

```
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

The preceding message means that the nameserver is running in your network. Finally, we can start the server and the client scripts in two separate Windows consoles:

1. To run `pyro_server.py`, just type the following:

```
C:\>python pyro_server.py
```

2. Following that, you'll see something like this:

```
Ready. Object uri =
PYRO:obj_76046e1c9d734ad5b1b4f6a61ee77425@localhost:63269
```

3. Then, run the client by typing the following:

```
C:\>python pyro_client.py
```

4. The following message will be printed out:

```
What is your name?
```

5. Insert a name (for example, Ruvika):

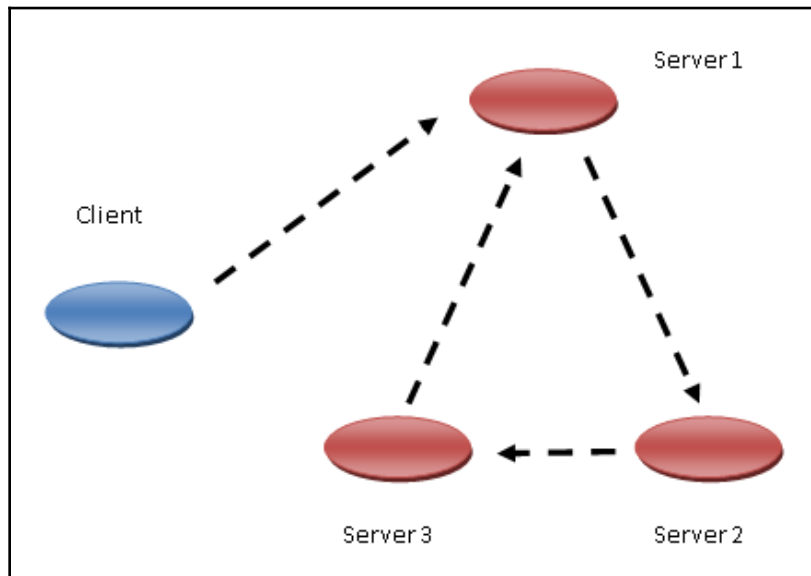
```
What is your name? Ruvika
```

6. The following welcome message will be displayed:

```
Hi welcome Ruvika
```

There's more...

Among the features of `Pyro4`, there is the creation of object topologies. For example, let's suppose we want to build a distributed architecture that follows a chain topology, as follows:



Chaining objects with Pyro4

The Client makes a request to **Server 1**, and then the request is forwarded to **Server 2**, which then calls **Server 3**. The chain call ends when **Server 3** calls **Server 1**.

Implementing chain topology

To implement a chain topology using Pyro4, we need to implement a `chain` object and the `client` and `server` objects. The `Chain` class allows the call to be redirected to the next server by processing the input message and reconstructing the server address to which the request should be addressed.

Also note, in this case, the `@Pyro4.expose` decorator, which allows all the methods of the class (`chainTopology.py`) to be exposed:

```
import Pyro4

@Pyro4.expose
class Chain(object):
    def __init__(self, name, next_server):
        self.name = name
        self.next_serverName = next_server
        self.next_server = None
    def process(self, message):
        if self.next_server is None:
            self.next_server = Pyro4.core.Proxy("PYRONAME:example.\
chainTopology." + self.next_serverName)
```

If the chain is closed (the last call is done from `server_chain_3.py` to `server_chain_1.py`), then a closing message is printed out:

```
if self.name in message:
    print("Back at %s;the chain is closed!" % self.name)
    return ["complete at " + self.name]
```

A forwarding message is printed out if there is a next element in the chain:

```
else:
    print("%s forwarding the message to the object %s" %\
          (self.name, self.next_serverName))
    message.append(self.name)
    result = self.next_server.process(message)
    result.insert(0, "passed on from " + self.name)
    return result
```

Next, we have the source code for the client (`client_chain.py`):

```
import Pyro4

obj = Pyro4.core.Proxy("PYRONAME:example.chainTopology.1")
print("Result=%s" % obj.process(["hello"]))
```

Following this is the source code for the first server (namely, `server_1`) in the chain that is called from the client (`server_chain_1.py`). Here, the relevant libraries are imported. Note, the import to the `chainTopology.py` file that was described previously:

```
import Pyro4
import chainTopology
```

Note also that the source code for the servers only differs as regards the definitions of the current and the next servers of the chain:

```
current_server= "1"
next_server = "2"
```

The remaining lines of code define the communication with the next element in the chain:

```
servername = "example.chainTopology." + current_server
daemon = Pyro4.core.Daemon()
obj = chainTopology.Chain(current_server, next_server)
uri = daemon.register(obj)
ns = Pyro4.locateNS()
ns.register(servername, uri)
print("server_%s started " % current_server)
daemon.requestLoop()
```

To execute this example, first run the `Pyro4` nameserver:

```
C:\>python -m Pyro4.naming
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

Run the three servers in three different terminals, typing each of them respectively (Windows terminals are used here):

The first server (`server_chain_1.py`) in the first terminal:

```
C:\>python server_chain_1.py
```

Followed by the second server (`server_chain_2.py`) in the second terminal:

```
C:\>python server_chain_2.py
```

And finally, the third server (`server_chain_3.py`) in the third terminal:

```
C:\>python server_chain_3.py
```

Then, run the `client_chain.py` script from another terminal:

```
C:\>python client_chain.py
```

This is the output, as shown in the Command Prompt:

```
Result=['passed on from 1','passed on from 2','passed on from 3','complete  
at 1']
```

The preceding message is displayed as a result of the forwarding request passed across the three servers after it returned the fact that the task is completed to `server_chain_1`.

Also, we can focus on the behavior of the object servers while the request is forwarded to the next object in the chain (refer to the message underneath the start message):

1. `server_1` is started and the following message is forwarded to `server_2`:

```
server_1 started  
1 forwarding the message to the object 2
```

2. `server_2` forwards the following message to `server_3`:

```
server_2 started  
2 forwarding the message to the object 3
```

3. `server_3` forwards the following message to `server_1`:

```
server_3 started  
3 forwarding the message to the object 1
```

4. Finally, the message returns to the starting point (in other words, `server_1`), closing the chain :

```
server_1 started  
1 forwarding the message to the object 2  
Back at 1; the chain is closed!
```

See also

The Pyro4 documentation is available

at <https://buildmedia.readthedocs.org/media/pdf/pyro4/stable/pyro4.pdf>.

This contains a description and some application examples of the 4.75 release.

7 Cloud Computing

Cloud computing is the distribution of computing services, such as servers, storage resources, databases, networks, software, analysis, and intelligence, via the internet (*the cloud*). The purpose of this chapter is to provide an overview of the main cloud computing technologies in relation to the Python programming language.

First, we will describe the PythonAnywhere platform, with which we will deploy Python applications on the cloud. In the context of cloud computing, two emerging technologies will be identified: containers and serverless technologies.

Containers represent the new approach to the virtualization of resources, and the *serverless* technologies represent a step forward in the field of cloud services because they can speed up the release of applications.

In fact, you do not have to worry about the provisioning, the servers, or the infrastructure configurations. You only have to create functions (namely, Lambda functions) that can operate independently from the applications.

In this chapter, we will cover the following recipes:

- What is cloud computing?
- Understanding the cloud computing architecture
- Developing web applications with PythonAnywhere
- Dockerizing a Python application
- Introducing serverless computing

We will also see how to take advantage of the *AWS Lambda* framework for the development of Python applications.

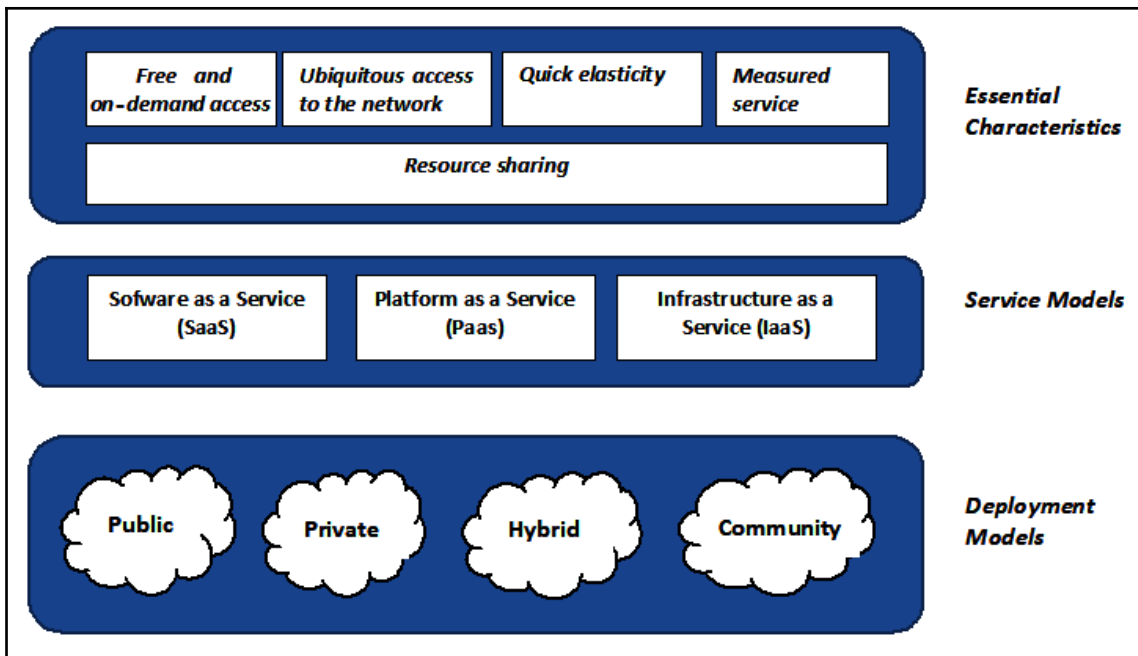
What is cloud computing?

Cloud computing is a computational model for the distribution of services based on a set of resources, such as virtual processing, mass memory, and networking, which can be dynamically aggregated and activated as platforms to run applications, satisfying appropriate levels of service and optimizing the efficiency of resource use.

This can be acquired and released quickly with minimum management effort or interaction with the service provider. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

In particular, the five essential characteristics are as follows:

- **Free and on-demand access:** This allows users to access—through *user friendly* interfaces—the services offered by the provider without human interaction.
- **Ubiquitous access to the network:** Resources are available throughout the network and can be accessed—via standard devices—such as *smartphones, tablets, and personal computers*.
- **Quick elasticity:** This is the ability of the cloud to increase or reduce the resources assigned in a rapid and automatic way, such as making it seem that they are infinite to the user. This provides great scalability to the system.
- **Measured service:** Cloud systems constantly monitor the resources offered and optimize them automatically based on the estimated use. In this way, the customer only pays for the resources that are actually used in that particular session.
- **Resource sharing:** The provider provides its resources through a multi-tenant model so that they can be assigned and reassigned dynamically, based on the customer's request, and used by multiple consumers:



Cloud computing main features

However, there are many definitions of cloud computing, each of which has different interpretations and meanings. The *National Institute of Standards and Technology* (NIST) has tried to provide a detailed and official explanation (<https://csrc.nist.gov/publications/detail/sp/800-145/final>).

Another feature (not listed in the definition of NIST, but which is the basis of cloud computing) is the concept of virtualization. This is the possibility of executing multiple OSes on the same physical resources, guaranteeing numerous advantages, such as scalability, cost reduction, and greater speed in providing new resources to customers.

The most common approaches to virtualization are as follows:

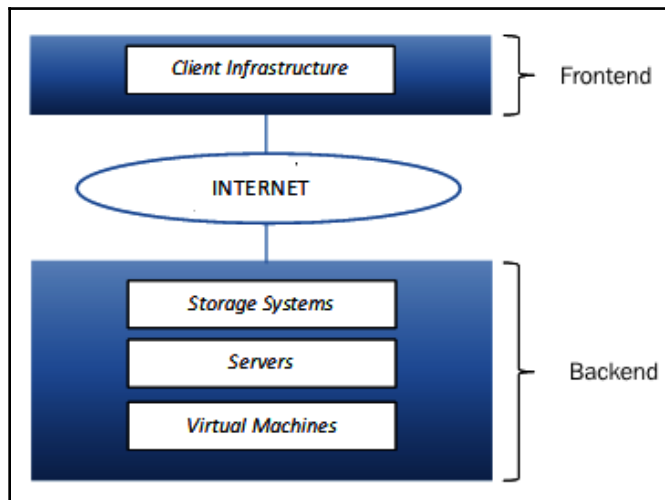
- Containers
- Virtual machines

Both solutions have almost the same advantages as far as the isolation of applications is concerned, but they work at different levels of virtualization because the containers virtualize the OS and the virtual machines virtualize the hardware. This means that the containers are more portable and efficient.

The most common application for virtualizing through containers is Docker. We will go through a brief introduction to this framework and we will see how to containerize (or dockerize) a Python application.

Understanding the cloud computing architecture

The architecture of cloud computing refers to a series of components and sub-components that make up the structure of the system. Typically, it can be grouped into the two main sections of *Frontend* and *Backend*:



Cloud computing architecture

Each section has a very specific meaning and scope and is linked to the other via a virtual network or an internet network.

The *Frontend* refers to the part of the cloud computing system that is visible to the user, which is realized through a series of interfaces and applications that allow the consumer to access the cloud system. Different cloud computing systems have different UIs.

The *Backend* is the part that is not visible to the customer. This section contains all the resources that allow the provider to provide cloud computing services such as servers, storage systems, and virtual machines. The idea behind the creation of the backend is to entrust the management of the entire system to a single central server, which will, therefore, have to constantly monitor traffic and user requests, perform access control, and implement communication protocols.

Among the various components of this architecture, the most important is the Hypervisor, also called the *Virtual Machine Manager*. This is a firmware that allocates resources dynamically and also allows you to share a single instance among multiple users. In short, this is the program that realizes virtualization, which is one of the main attributes of cloud computing.

After providing a definition of cloud computing and explaining the essential features, we'll introduce the *service models* in which cloud computing services can be provided.

Service models

The cloud computing services offered by the providers fall into three broad categories:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

This classification led to the definition of a scheme that takes the name of the **SPI** model (see the **bold** initials in the previous list). Sometimes it is called the cloud computing stack, as these categories are based on each other.

A detailed description of each of these levels will now be given, following a top-down approach.

SaaS

A SaaS provider provides users with software applications on-demand, which are accessible through any internet device, such as a web browser. Furthermore, the provider hosts the software application and the underlying infrastructure, relieving the customer from the burden of management and maintenance activities such as software updates and the application of security patches.

There are many advantages of using this model for both the user and the provider. For the user, there is a considerable reduction in management costs, and for the provider, they have more control over the traffic, thus allowing them to avoid any overloads. An example of SaaS is any web-based email service, such as **Gmail**, **Outlook**, **Salesforce** and **Yahoo!**.

PaaS

Unlike SaaS, this service refers to the entire development environment of an application, not just its use. So, the PaaS solution provides a cloud platform that is accessible through a web browser for the development, testing, distribution, and management of software applications. Furthermore, the provider provides web-based interfaces, a multi-tenant architecture, and communication tools in order to allow developers to create applications in a simpler way. This supports the entire life cycle of the software and also favoring the cooperation.

Examples of PaaS are **Microsoft Azure Services**, **Google App Engine**, and **Amazon Web Services**.

IaaS

IaaS is a model that offers the computing infrastructure as an on-demand service. You can, therefore, purchase virtual machines, on which you can run your own software, storage resources (with the possibility of rapidly increasing or reducing the storage capacity based on your needs), networks, and OS by paying based on what you actually use. A dynamic infrastructure of this type adds greater scalability, while also significantly reducing costs.

This model is used both by small emerging companies that do not have large capital to invest and by established companies seeking to streamline their hardware architecture. The range of IaaS sellers is very wide, including **Amazon Web Services**, **IBM**, and **Oracle**.

Distribution models

Cloud computing architectures are not all the same. In fact, there are four different distribution models:

- **The public cloud**
- **The private cloud**
- **The cloud community**
- **The hybrid cloud**

Public cloud

This distribution model is open to all, both individual users and companies. Typically, the public cloud runs in a data center owned by the service provider that handles hardware, software, and other support infrastructure. In this way, the user is exempt from any maintenance activities/expenses.

Private cloud

Also known as *internal clouds*, private clouds offer the same advantages as public clouds, but provide greater control over data and processes. This model is presented as a cloud infrastructure that works exclusively for a company and is therefore managed and hosted within the borders of the given company. Clearly, the organization that uses it can extend its architecture to any group that it is linked to by a business relationship.

By adopting this type of solution, possible problems concerning the violation of sensitive data and industrial espionage are avoided without neglecting the possibility of using a simplified, configurable, and high-performing working provisioning system. Precisely for this reason, in recent years, the number of companies that use the private cloud has increased significantly.

Cloud community

Conceptually, this model describes a shared infrastructure that is implemented and managed by several companies with common interests. This type of solution is rarely used because sharing the responsibilities and management activities among the various members of the community could become complicated.

Hybrid cloud

NIST defines this as the result of the composition of the three implementation models mentioned previously (the private, public, and community clouds), trying to take the advantages of each of the three in order to make up for where the others are weaker. The clouds used remain distinct entities, and this can cause a lack of operational consistency. Therefore, companies that adopt this model have the task of guaranteeing, through proprietary technologies, the interoperability of their servers, optimizing them for the specific roles they must play.

A feature that distinguishes the hybrid cloud from all others is the cloudburst or the possibility of being able to dynamically transfer excess traffic from the private cloud to the public cloud in the presence of large peak demand.

This implementation model is adopted by those companies that intend to share their software applications while retaining their sensitive data in internal clouds.

Cloud computing platforms

Cloud computing platforms are sets of software and technologies that enable the delivery of resources in the cloud (on-demand, scalable, and virtualized resources). Among the most popular platforms are those of Google and, of course, the milestone of cloud computing: **Amazon Web Services (AWS)**. Both support Python as a development language.

However, in the next recipe, we will focus on PythonAnywhere, which is a cloud platform developed specifically for the deployment of web applications in the Python programming language.

Developing web applications with PythonAnywhere

PythonAnywhere is an online hosting development and service environment based on the Python programming language. Once registered on the site, you will be directed to the dashboard, which includes an advanced shell and text editor that is made entirely with HTML code. With this, you can create, modify, and execute your own scripts.

Moreover, this development environment also allows you to choose which version of Python to work with. In this, a simple wizard helps us to preconfigure an application.

Getting ready

Let's first see how to get login credentials to the site.

The following screenshot shows the various types of subscriptions, and also, the possibility of obtaining a free account (please go to <https://www.pythonanywhere.com/registration/register/beginner/>):

Plans and pricing

Beginner: Free!

A limited account with one web app at `your-username.pythonanywhere.com`, restricted outbound Internet access from your apps, low CPU/bandwidth, no IPython/Jupyter notebook support.

It works and it's a great way to get started!

Create a Beginner account

Education accounts

Are you a teacher looking for a place your students can code Python? You're not alone. Click through to find out more about our [Education beta](#).

All of our paid plans come with a no-quibble 30-day money-back guarantee — you're billed monthly and you can cancel at any time. The minimum contract length is just one month. You get unrestricted Internet access from your applications, unlimited in-browser Python, Bash and database consoles, and full SSH access to your account. All accounts (including free ones) have screen-sharing with other PythonAnywhere accounts, and free SSL support (though you'll need to get a certificate for your own domains).

Hacker	\$5/month	Web dev	\$12/month	Startup	\$99/month	Custom	\$5 to \$500/month
Run your Python code in the cloud from one web app and the console		If you want to host small Python-based websites for you or for your clients		Start a business and don't worry about having to scale to handle traffic spikes		Want a combination that's not on the list? Create your own! All custom plans have:	
A Python IDE in your browser with unlimited Python/bash consoles		A Python IDE in your browser with unlimited Python/bash consoles		A Python IDE in your browser with unlimited Python/bash consoles		A Python IDE in your browser with unlimited Python/bash consoles	

PythonAnywhere: Registration page

Once access to the site has been obtained (it is recommended that you create a beginner account), we log in. Given that the Python shells that are integrated into the browsers are very useful, especially for beginners and for introductory programming courses, they are certainly not new from a technological point of view.

Instead, the added value of PythonAnywhere is perceived as soon as you log in by accessing the personal dashboard:

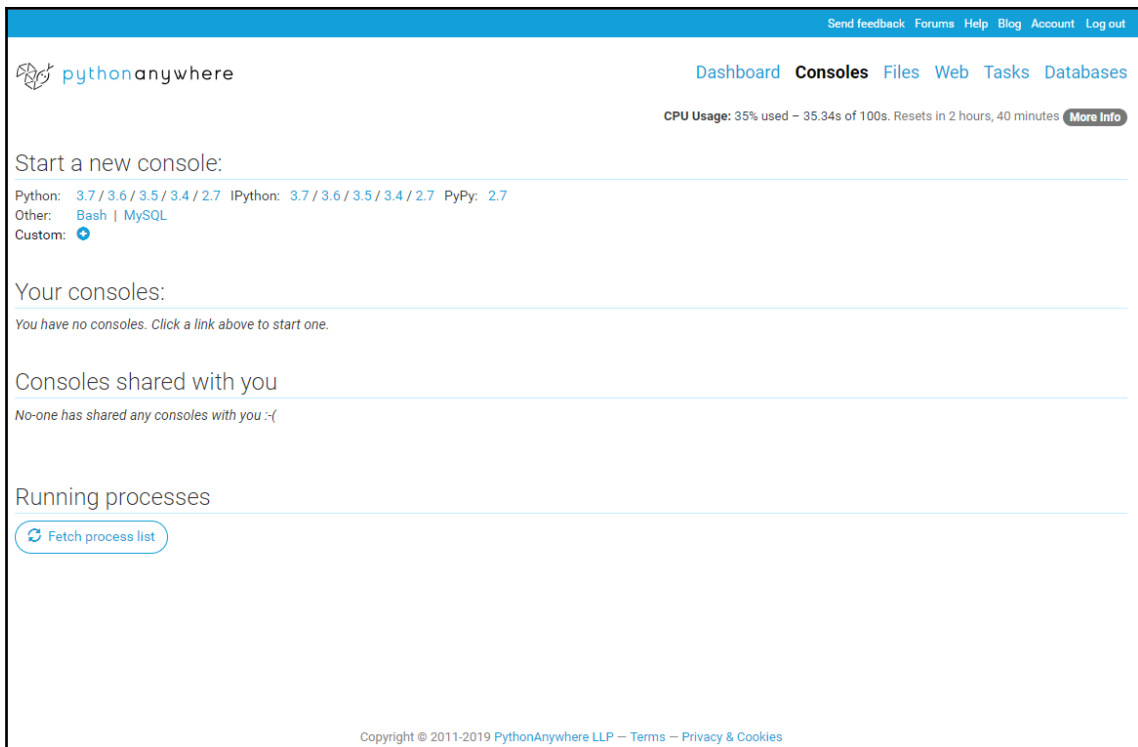
The screenshot displays the PythonAnywhere Dashboard for a user named 'giazax'. The top navigation bar includes links for 'Send feedback', 'Forums', 'Help', 'Blog', 'Account', and 'Log out'. The main header shows the 'pythonanywhere' logo and navigation tabs for 'Dashboard', 'Consoles', 'Files', 'Web', 'Tasks', and 'Databases'. The dashboard is divided into several sections:

- CPU Usage:** 35% used – 35.34s of 100s. Resets in 2 hours, 55 minutes. A 'More Info' button is available.
- File storage:** 14% full – 70.6 MB of your 512.0 MB quota.
- Recent Consoles:** A section with a '+ 7 -' button. It states 'You have no recent consoles.' and includes a 'View all' button.
- Recent Files:** A section with a '+ 5 -' button. It lists three files:
 - `/home/giazax/mysite/flask_app.py`
 - `/home/giazax/pythoninthecloud/pytho...wsgi.py`
 - `/home/giazax/pythoninthecloud/pytho...urls.py`Below the list are buttons for '+ Open another file' and 'Browse files'.
- Recent Notebooks:** A section with a '+ 5 -' button. It contains a message: 'Your account does not support Jupyter Notebooks. Upgrade your account to get access!'.
- All Web apps:** A section showing 'giazax.pythonanywhere.com' and an 'Open Web tab' button.
- New console:** A section with a message: 'You can have up to 2 consoles. To get more, upgrade your account!'.

The footer of the dashboard includes the copyright notice: 'Copyright © 2011-2019 PythonAnywhere LLP – Terms – Privacy & Cookies'.

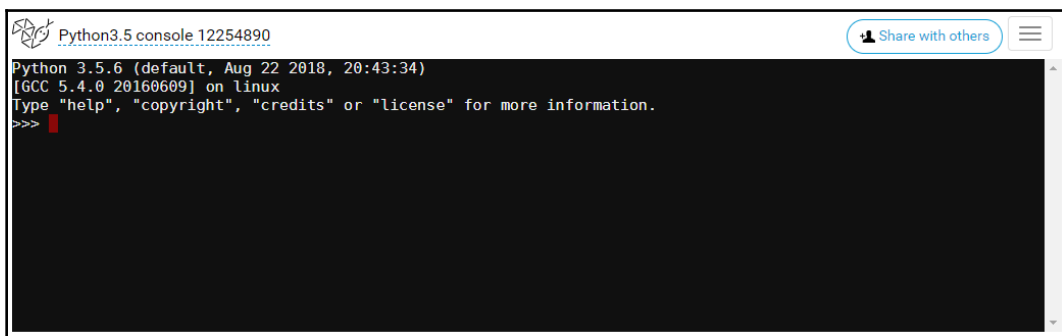
PythonAnywhere: Dashboard

Through the personal dashboard, we can choose which version of Python to run between 2.7 and 3.7, with or without the IPython interface:



PythonAnywhere: Console view

The number of consoles that can be used varies according to the type of subscription you have. In our case, having made a beginner account, we can use two Python consoles at most. Once selecting a Python shell, such as version 3.5, the following view should open on the web browser:



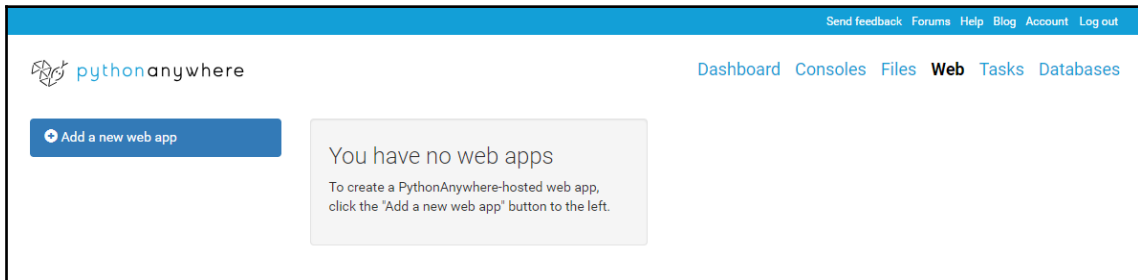
PythonAnywhere: Python shell

In the following section, we want to show you how to use PythonAnywhere to write a simple web application.

How to do it...

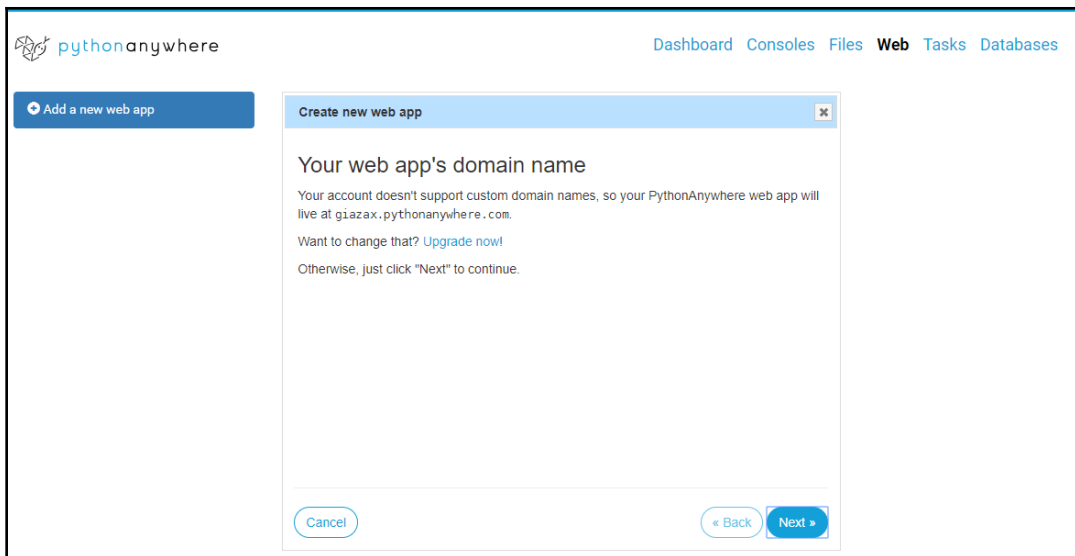
Let's have a look at the following steps:

1. On the **Dashboard**, open the **Web** tab:



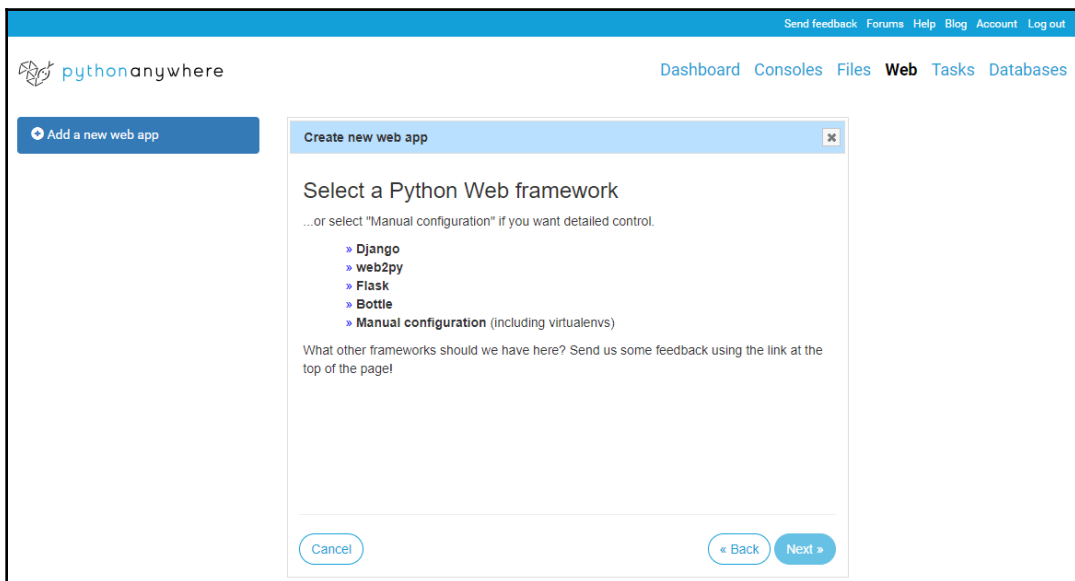
PythonAnywhere: Web app view

2. The interface tells us that we don't have a web application yet. By selecting **Add a new web app**, the following view opens. It tells us that our applications will have the following web address: `loginname.pythonanywhere.com` (for this example, the web address of the application will be `giazax.pythonanywhere.com`):



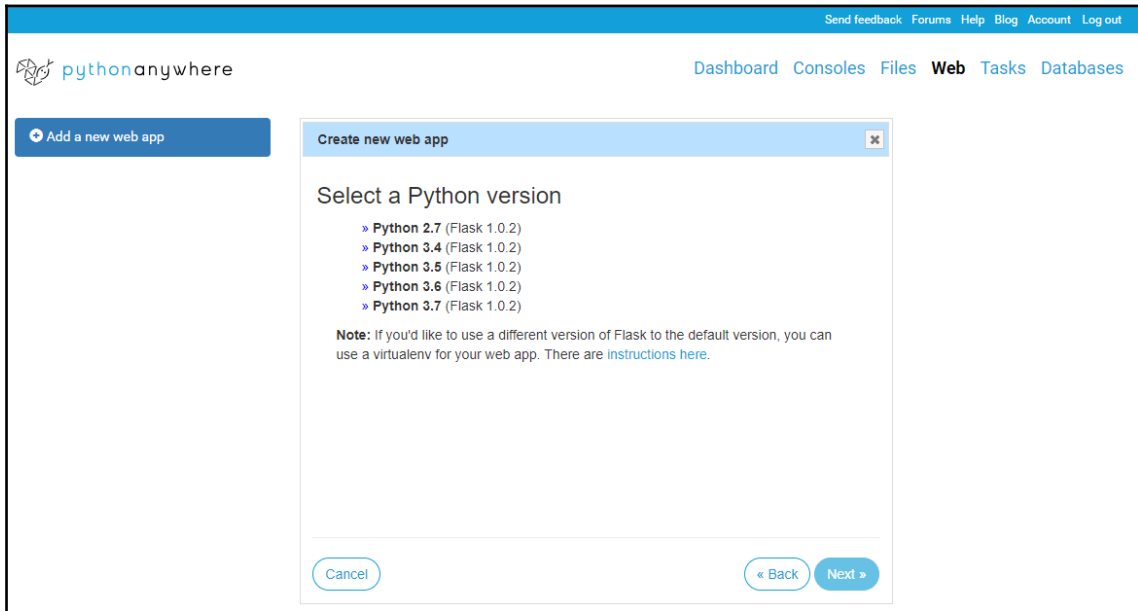
PythonAnywhere: Web app wizard

3. When we click on **Next**, we can select the Python web framework we want to use:



PythonAnywhere: Web framework wizard

4. We select **Flask** as a web framework, and then click on **Next** to choose which Python version we want to use, as shown here:



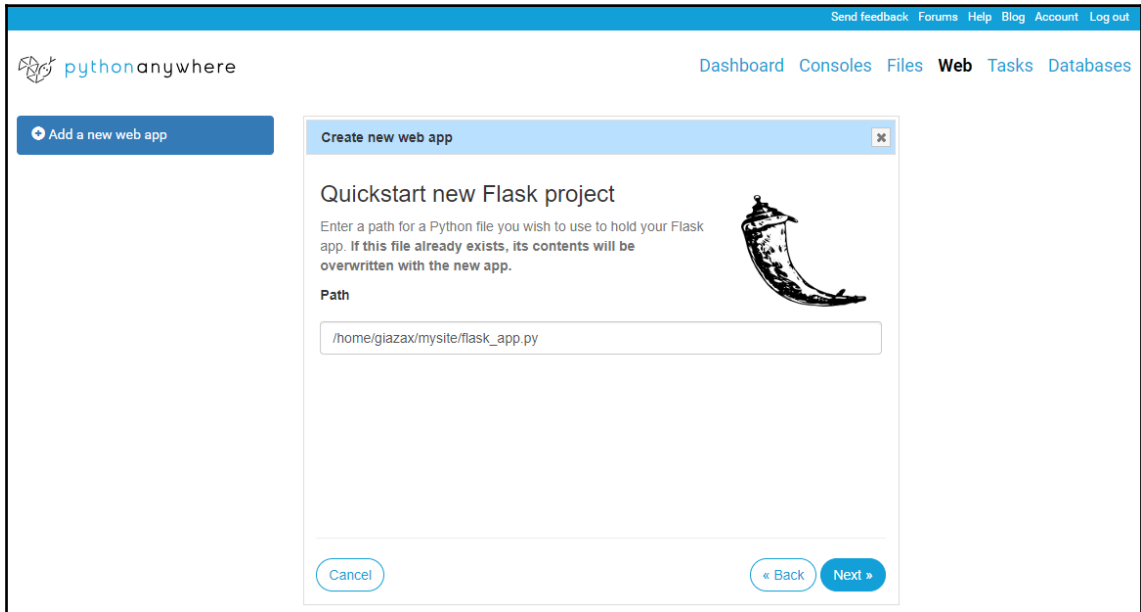
PythonAnywhere: Web framework wizard

Flask is a micro-framework for Python that is easy to install and use, and is used by companies such as Pinterest and LinkedIn.



If you don't know what a framework for creating web applications is, then you can imagine a set of programs with the aim of facilitating the creation of web services such as web servers and APIs. More information on Flask can be found at <http://flask.pocoo.org/docs/1.0/>.

5. In the preceding screenshot, we select **Python 3.5** for **Flask 1.0.2**, then let's click on **Next** to enter the path for a Python file to use in order to hold the Flask application. Here, the default file is selected:



The screenshot shows the PythonAnywhere web interface. At the top, there's a navigation bar with links: Send feedback, Forums, Help, Blog, Account, Log out. Below this, the main header includes the PythonAnywhere logo and a set of tabs: Dashboard, Consoles, Files, Web (selected), Tasks, Databases. On the left, there's a button 'Add a new web app'. The central focus is a 'Create new web app' dialog box. Inside, it says 'Quickstart new Flask project' and provides instructions: 'Enter a path for a Python file you wish to use to hold your Flask app. If this file already exists, its contents will be overwritten with the new app.' To the right of the text is a small illustration of a flask. Below the text, there's a 'Path' label and a text input field containing the default path '/home/giazax/mysite/flask_app.py'. At the bottom of the dialog, there are three buttons: 'Cancel', '« Back', and 'Next »'.

PythonAnywhere: Flask project definition

6. When we click on **Next** for the final time, the following screen is displayed, which summarizes the web application's configuration parameters:

The screenshot shows the PythonAnywhere configuration page for the application `giazax.pythonanywhere.com`. The page has a blue header with navigation links: Send feedback, Forums, Help, Blog, Account, Log out. Below the header is a green success message: "All done! Your web app is now set up. Details below." The main content area is titled "Configuration for `giazax.pythonanywhere.com`". On the left, there is a sidebar with a button "Add a new web app". The main content area includes a "Reload" section with a button "Reload `giazax.pythonanywhere.com`", a "Best before date" section with a warning message and a button "Run until 3 months from today", and a "Traffic" section with a table showing site usage.

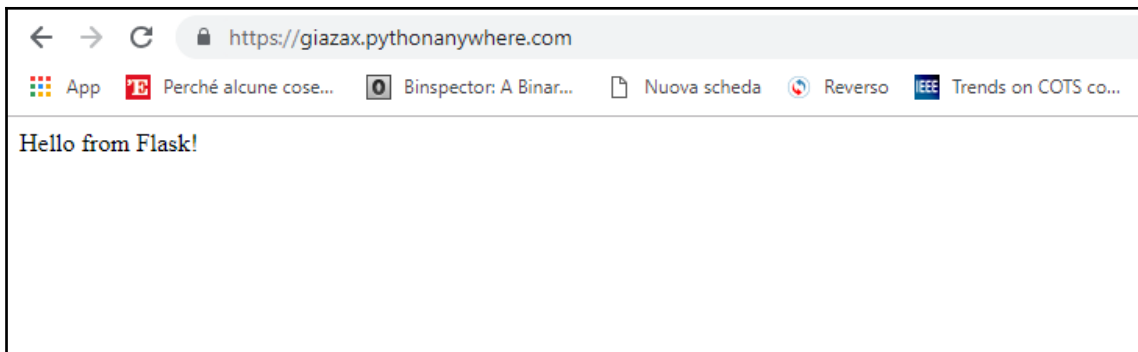
How busy is your site?		
This month (previous month)	38	(0)
Today (yesterday)	1	(37)

PythonAnywhere: Configuration page for `giazax.pythonanywhere.com`

Now, let's see what happens with this.

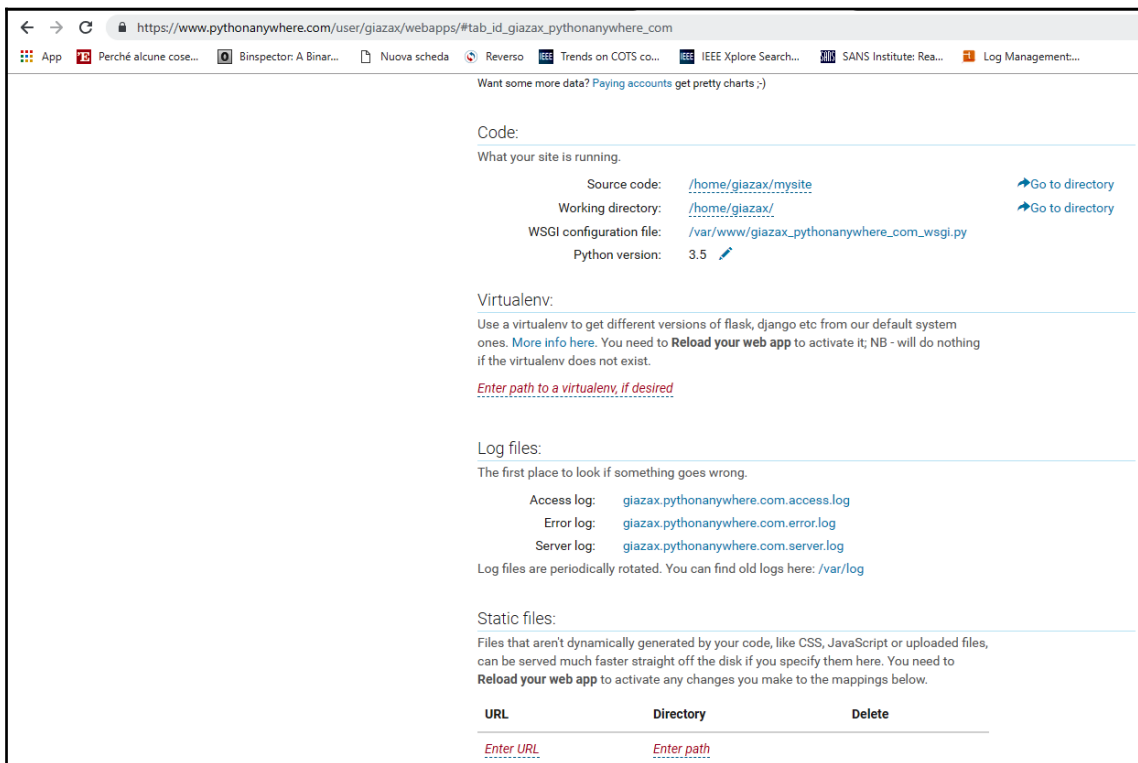
How it works...

In the address bar of the web browser, type the URL of our web application, in our case, `https://giazax.pythonanywhere.com/`. The site shows a simple welcome phrase:



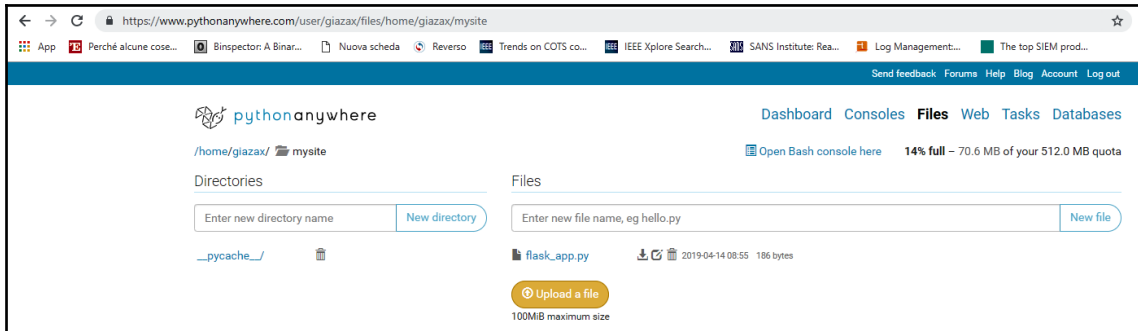
giazax.pythonanywhere.com site page

The source code for this application can be seen by selecting **Go to directory** in correspondence with the **Source code** label:



PythonAnywhere: Configuration page

Here, it is possible to analyze the files that make up the web application:



PythonAnywhere: Project site repository

It is also possible to upload new files and possibly modify the contents. Here, we select the `flask_app.py` file of our first web application. The content looks like a minimal Flask application:

```
# A very simple Flask Hello World app for you to get started with...

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello from Flask!'
```

The `route()` decorator is used by Flask to define the URL that should trigger the `hello_world` function. This simple function returns the message displayed in the web browser.

There's more...

The PythonAnywhere shell is made with HTML, making it virtually portable across multiple platforms and browsers, including Apple's mobile versions. It is possible to keep several shells open (in variable numbers according to the chosen account profile), share them with other users, or terminate them as needed.

PythonAnywhere has a rather advanced text editor with syntax coloring and automatic indentation, through which you can create, modify, and execute your own scripts. The files are stored in a storage area of varying size depending on the profile of the account, but if there is not enough space or if you wanted a more fluid integration with the filesystem of your PC, then PythonAnywhere allows you to use a Dropbox account, making your shared folder accessible on the popular storage service.

Each shell can contain a WSGI script that corresponds to a specific URL. It is also possible to start a bash shell from which to invoke Git and interact with the filesystem. Finally, as we have seen, there is a wizard available that allows us to preconfigure a **Django** and **web2py** or Flask application.

Furthermore, there is the possibility of exploiting a **MySQL** database, which is a series of cron jobs that allows us to execute certain scripts periodically. Therefore, we will get the true essence of PythonAnywhere: deployment of web applications at the speed of light.

PythonAnywhere relies completely on the **Amazon EC2** infrastructure, so there should be no reason not to trust the service. For this reason, it is strongly recommended for those who think of making a personal use. The beginner account offers more resources than the corresponding one on **Heroku** (<https://www.heroku.com/>), the deployment is simpler than on **OpenShift** (<https://www.openshift.com/>), and the whole system is generally much more flexible than **Google App Engine** (<https://cloud.google.com/appengine/>).

See also

- The main resources on PythonAnywhere can be found here: <https://www.pythonanywhere.com>.
- For web programming via Python, PythonAnywhere supports **Django** (<https://www.djangoproject.com/>) and **web2py** (<http://www.web2py.com/>), in addition to **Flask**.

As with **Flask**, it is recommended that you visit these sites for information on how to work with these libraries.

Dockerizing a Python application

Containers are virtualization environments. They include everything that the software needs, namely libraries, dependencies, filesystems, and network interfaces. Unlike classic virtual machines, all the aforementioned elements share the kernel with the machine they are running on. In this way, the impact on the use of the resources of the host node is greatly reduced.

This makes the container a very attractive technology in terms of scalability, performance, and isolation. Containers are not young technology; they had success with the launch of Docker in 2013. Since then, they have completely revolutionized the standards used for application development and management.

Docker is a container platform based on the implementation of **Linux Containers (LXC)**, which extends the functionality of this technology with the ability to manage containers as self-contained images, and adds additional tools for coordinating their life cycle and saving their state.

The idea of containerization is precisely to allow a given application to be executed on any type of system since all its dependencies are already included in the container itself.

In this way, the application becomes highly portable and can be easily tested and deployed on any type of environment, both on-premises and, above all, in the cloud.

Now, let's see how to dockerize a Python application using Docker.

Getting ready

The Docker team's intuition was to take the concept of a container and build an ecosystem around it that would simplify its use. This ecosystem includes a series of tools:

- Docker engine (<https://www.docker.com/products/docker-engine>)
- Docker toolbox (<https://docs.docker.com/toolbox/>)
- Swarm (<https://docs.docker.com/engine/swarm/>)
- Kitematic (<https://kitematic.com/>)

Installing Docker for Windows

The installation is quite simple: once you have downloaded the installer (<https://docs.docker.com/docker-for-windows/install/>), just run it and you're done. The installation process is generally very linear. The only thing that needs attention is the final phase of the installation, in which it might be required to enable Hyper-V features. If so, then we accept and restart the machine.

Once the computer is restarted, the Docker icon should appear in the system tray in the bottom right of the screen.

Open Command Prompt or the PowerShell console and check whether everything is okay by executing the `docker version` command:

```
C:\>docker version
Client: Docker Engine - Community
Version: 18.09.2
API version: 1.39
Go version: go1.10.8
Git commit: 6247962
Built: Sun Feb 10 04:12:31 2019
OS/Arch: windows/amd64
Experimental: false

Server: Docker Engine - Community
Engine:
Version: 18.09.2
API version: 1.39 (minimum version 1.12)
Go version: go1.10.6
Git commit: 6247962
Built: Sun Feb 10 04:13:06 2019
OS/Arch: linux/amd64
Experimental: false
```

The most interesting part of the output is the subdivision that is made between the client and the server. The client is our local Windows system, while the server is the Linux virtual machine that Docker instantiated behind the scenes. The parts communicate with each other thanks to the API layer, as mentioned in the introduction of this recipe.

Now, let's see how to containerize (or dockerize) a simple Python application.

How to do it...

Let's imagine we want to deploy the following Python application, which we call `dockerize.py`:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "Hello World!"
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int("5000"), debug=True)
```

The example application uses the `Flask` module. It implements a simple web application at the localhost address, 5000.

The first step is to create the following text file, with the extension of `.py`, which we will call `Dockerfile.py`:

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD python ./dockerize.py
```

The directives listed in the previous code perform the following tasks:

- `FROM python: alpine3.7` instructs Docker to use Python version 3.7.
- `COPY` copies the application into the container image.
- `WORKDIR` sets the working directory (`WORKDIR`).
- The `RUN` instruction calls the `pip` installer, pointing to the `requirements.txt` file. It contains the list of dependencies that the application must execute (in our case the only dependence is `flask`).
- The `EXPOSE` directive exposes to the port that is used by Flask.

So, in summary, we have written three files:

- The application to be containerized: `dockerize.py`
- `Dockerfile`
- The dependency list file

So, we need to create an image of the `dockerize.py` application:

```
docker build --tag dockerize.py
```

This will tag the `my-python-app` image and build it.

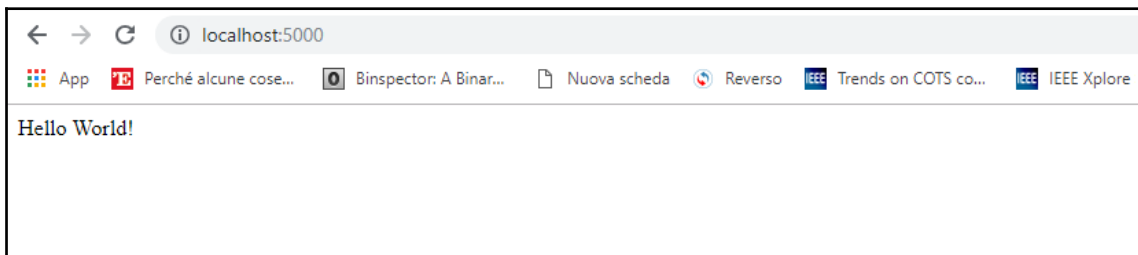
How it works...

After the `my-python-app` image is built, you can run it as a container:

```
docker run -p 5000:5000 dockerize.py
```

The application is then started as a container, after which the name parameter sends the name to the container and the `-p` parameter maps the 5000 host port to the container port of 5000.

Next, you need to open your web browser, then on the address bar, type `localhost:5000`. If everything worked the right way, then you should see the following web page:



Docker application

Docker runs the `dockerize.py` container by using the `run` command, and the result is a web application. The image contains the instructions necessary for the operation of the container.

The correlation between container and image can be understood by referring to the object-oriented programming paradigm by associating the image with a class and the container with the class instance.

It is useful to recap what happens when we create an instance of a container:

- The image of the container is (if not already present) unloaded locally.
- An environment in which to start the container is created.
- A message is printed on the screen.
- The previously created environment is then abandoned.

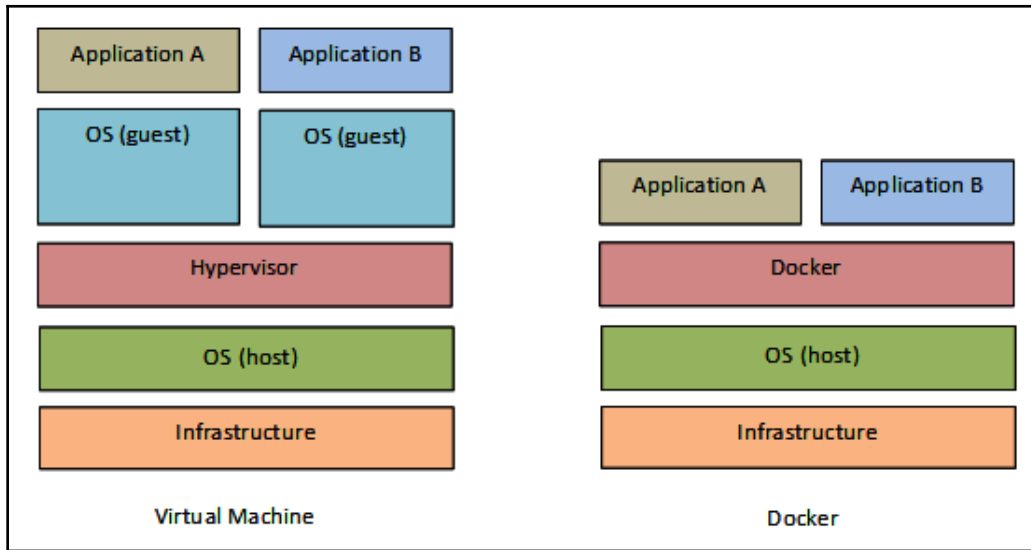
This all takes place in a few seconds and with a simple, intuitive, and readable command.

There's more...

Apparently, containers and virtual machines seem to be very similar concepts. But although these two solutions have common characteristics, they are profoundly different technologies, in the same way, that we must start thinking about how the architectures of our applications are different. We can create a container with our monolithic application inside, but in this way, we will not fully exploit the strength of the containers, and therefore, of Docker.

A possible software architecture suitable for a container infrastructure is the classic microservice architecture. The idea is to break down the application into many small components—each with their own specific task—that are able to exchange messages and cooperate with each other. The deployment of these components will then take place individually, in the form of many containers.

A scenario that can be handled with microservices is absolutely impractical with a virtual machine since every new virtual machine instantiated would require a good expenditure of energy for the host machine. Containers, on the other hand, are very light, since they carry out completely different virtualization from that practiced by virtual machines:



Microservice architecture in virtual machine and Docker implementation

In virtual machines, a tool called a **Hypervisor** takes care of reserving (statically or dynamically) a certain amount of resources from the host OS to be dedicated to one or more OSes, called **guests** or **hosts**. A guest OS will be completely isolated from the host OS. This mechanism is very expensive in terms of resources, so the idea of combining a microservice with a virtual machine is completely impossible.

Containers, on the other hand, make a completely different contribution to the issue. The isolation is much blander and all the running containers share the same kernel as the underlying OS. Hypervisor overhead completely disappears, and a single host can host hundreds of containers.

When we ask Docker to run a container from its image, it must be present on the local disk, otherwise Docker will warn us of the problem (with a message reading **Unable to find image 'hello-world: latest' locally**) and will download it autonomously.

To find out which images were downloaded from Docker on our computer, we use the `docker images` command:

```
C:\>docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
dockerize.py latest bc3d70b05ed4 23 hours ago 91.8MB
<none> <none> ca18efb44b3c 24 hours ago 91.8MB
python alpine3.7 00be2573e9f7 2 months ago 81.3MB
```

The repository is a container of related images. For example, the `dockerize` repository contains various versions of the `dockerize` image. In the Docker world, the term **tag** is more correctly used to express the concept of image versioning. In the preceding code example, the image has been tagged as the `latest` and is the only tag available for the `dockerize` repository.

The latest tag is the default tag: whenever we refer to a repository without specifying the tag name, Docker will implicitly refer to the latest tag, and if this does not exist, then an error will be shown. Therefore, as a best practice, the repository tag form would be preferable as it allows greater predictability regarding the content of the image, avoiding possible conflicts between containers and errors due to the lack of the latest tag.

See also

Container technology is a very broad concept that can be explored by consulting numerous articles and examples of applications on the web. However, before starting this long and difficult journey, it is advisable to start from the website (<https://www.docker.com/>), which is complete and fully informative.

In the next section, we will examine the main features of serverless computing, whose main goal is to make it easier for a software developer to compose code that is designed to run on a cloud platform.

Introducing serverless computing

In recent years, a new service model named **Function as a Service (FaaS)** has been developed, which is also known as **serverless computing**.

Serverless computing is a cloud computing paradigm that allows the execution of applications without worrying about problems related to the underlying infrastructure. The term **serverless** could be misleading; in fact, it could be thought that this model does not foresee the use of processing servers. In reality, it indicates that the provisioning, scalability, and management of the servers on which the applications are executed are administered automatically and in a completely transparent manner for the developer. Everything is possible thanks to a new architecture model called **serverless**.

The first FaaS model dates back to **Amazon**, when the **AWS Lambda** service was released in 2014. Over time, other alternatives were added to the Amazon solution, which were developed by other major vendors such as **Microsoft**, with its **Azure Functions**, and by **IBM** and **Google**, with their own **Cloud Functions**. There are also valid open source solutions: among the most commonly used, we have **Apache OpenWhisk**, which is used by **IBM** on **Bluemix** for its serverless offering, but also **OpenLambda** and **IronFunctions**, with the latter being based on Docker's container technology.

In this recipe, we see how to implement a serverless Python function via **AWS Lambda**.

Getting ready

AWS is a whole class of cloud services offered and administered through a common interface. The common interface through which the services are offered in the AWS web console is reachable at <https://console.aws.amazon.com/>.

This type of service is charged. However, for the first year, a *free tier* is available. This a set of services use the minimum amount of resources and can be used for free to both evaluate the services and for the development of applications.



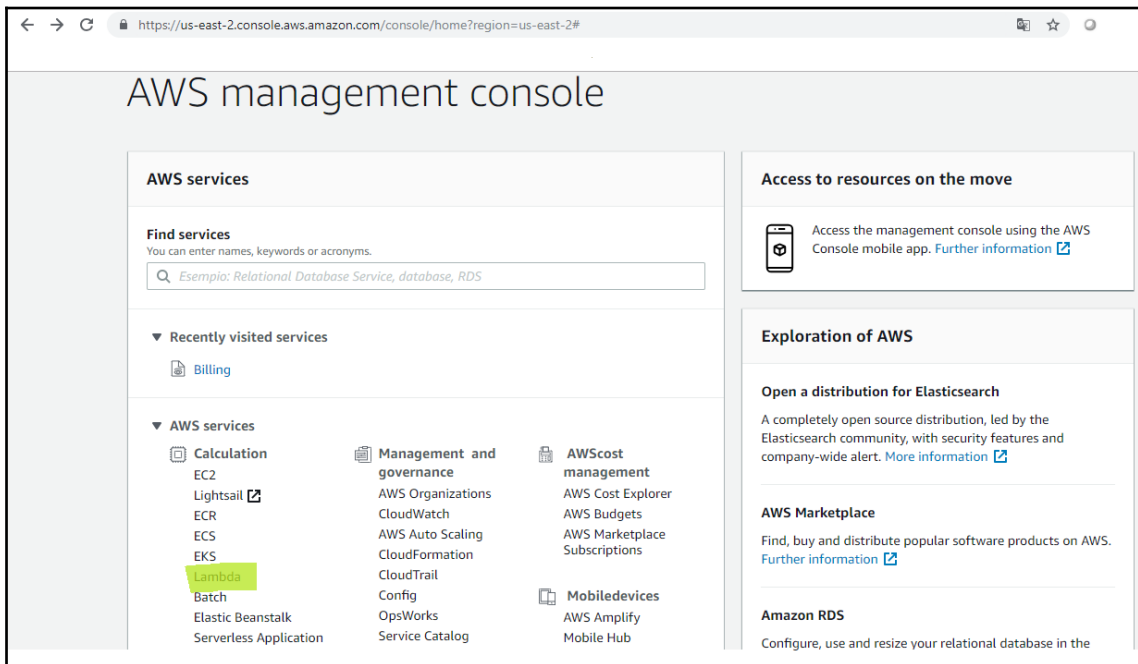
For details on how to create a free account with AWS, please refer to the official Amazon documentation at <https://aws.amazon.com>.

In these sections, we will outline the basics of running code in AWS Lambda without having to provision or manage any servers. We will show how to create a `Hello World` function in Lambda by using the AWS Lambda console. We will also explain how to manually call up the Lambda function by using sample event data and how to interpret the output parameters. All the operations shown in this tutorial can be performed as part of the free plan at <https://aws.amazon.com/free>.

How to do it...

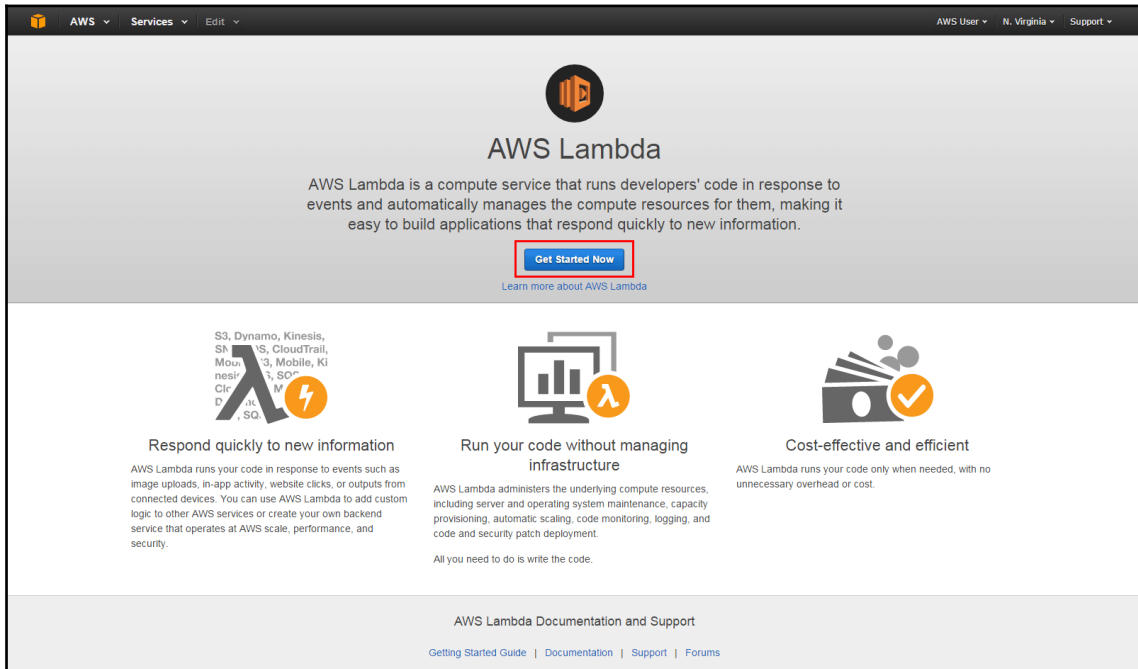
Let's have a look at the following steps:

1. The first thing to do is log in to the Lambda console (<https://console.aws.amazon.com/console/home>). Then, you need to locate and select **Lambda** under compute in order to open the AWS Lambda console (which is highlighted in green in the following screenshot):



AWS: Selecting a Lambda service

2. Then, in the AWS Lambda console, select **Get Started Now** and then create a Lambda function:



AWS: Lambda start page

3. In the filter box, type `hello-world-python` and select the **hello-world-python** blueprint.
4. Now we need to configure the Lambda function. The following list shows the configurations and provides example values:
 - **Configure function:**
 - **Name:** Enter the name of the function here. For this tutorial, enter `hello-world-python`.
 - **Description:** Here, you can enter a brief description of the function. This box is pre-filled with the phrase **A starter AWS Lambda Function**.
 - **Runtime:** At the moment, it is possible to write the code of the Lambda function in **Java**, **Node.js**, and **Python 2.7**, **3.6**, and **3.7**. For this tutorial, set up Python 2.7 as a runtime.
 - **Lambda function code:**
 - As you can see in the following screenshot, it is possible to review the Python sample code.
 - **Lambda function handler and role:**
 - **Handler:** You can specify a method in which AWS Lambda can start executing the code. AWS Lambda provides event data as input to the handler, which will process the events. In this example, Lambda identifies the event from the example code, so the field will be compiled with `lambda_function.lambda_handler`.

- **Role:** Click on the drop-down menu and select **Basic Execution Role**:

Lambda > New function using blueprint hello-world-python

Step 1: Select blueprint

Step 2: Configure function

Step 3: Review

Configure function

A Lambda function consists of the custom code you want to execute. [Learn more](#) about Lambda functions.

Name* hello-world-python

Description A starter AWS Lambda function.

Runtime* Python 2.7

Lambda function code

Provide the code for your function. Use the editor if your code does not require custom libraries (other than boto3). If you need custom libraries, you can upload your code and libraries as a .ZIP file.

Code entry type ☒ Edit code inline ☐ Upload a .ZIP file ☐ Upload a .ZIP from Amazon S3

```
1 from __future__ import print_function
2
3 import json
4
5 print('Loading function')
6
7
8 def lambda_handler(event, context):
9     #print("Received event: " + json.dumps(event, indent=2))
10    print("value1 = " + event['key1'])
11    print("value2 = " + event['key2'])
12    print("value3 = " + event['key3'])
13    return event['key1'] # Echo back the first key value
14    #raise Exception('Something went wrong')
```

Lambda function handler and role

Handler* lambda_function.lambda_handler ⓘ

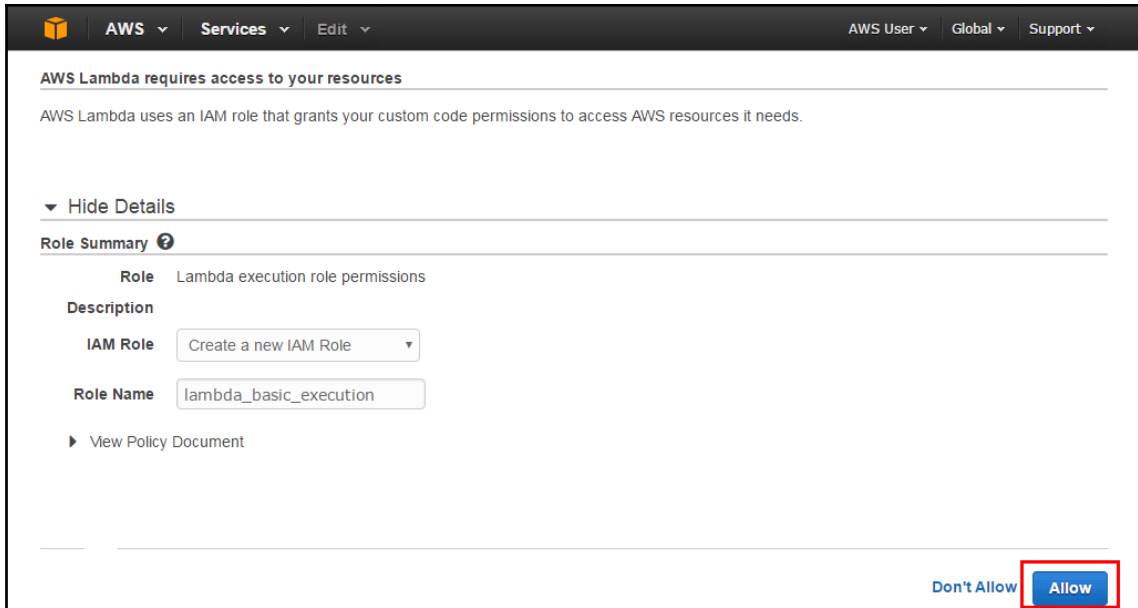
Role* lambda_basic_execution ⓘ

Ensure that popups are enabled to create a new role. [Learn more](#) about Lambda execution roles.

Advanced settings

AWS Configure function page

5. At this point, it is necessary to create a role for execution (named **IAM Role**) with the necessary authorizations to be interpreted by AWS Lambda as the executor of the Lambda function. By clicking **Allow**, the **Configure function** page will be returned, and the **lambda_basic_execution** function will be selected:



The screenshot shows the AWS IAM console's 'Role Summary' page for the role 'lambda_basic_execution'. The page has a dark header with navigation links: AWS, Services, Edit, AWS User, Global, and Support. Below the header, a message states 'AWS Lambda requires access to your resources' and explains that AWS Lambda uses an IAM role. A 'Hide Details' link is present. The 'Role Summary' section includes the role name 'lambda_basic_execution', a description 'Lambda execution role permissions', and a dropdown menu for 'IAM Role' set to 'Create a new IAM Role'. A link to 'View Policy Document' is also visible. At the bottom right, there are two buttons: 'Don't Allow' and 'Allow', with the 'Allow' button highlighted by a red rectangle.

AWS Lambda requires access to your resources

AWS Lambda uses an IAM role that grants your custom code permissions to access AWS resources it needs.

▼ Hide Details

Role Summary ?

Role Lambda execution role permissions

Description

IAM Role Create a new IAM Role ▼

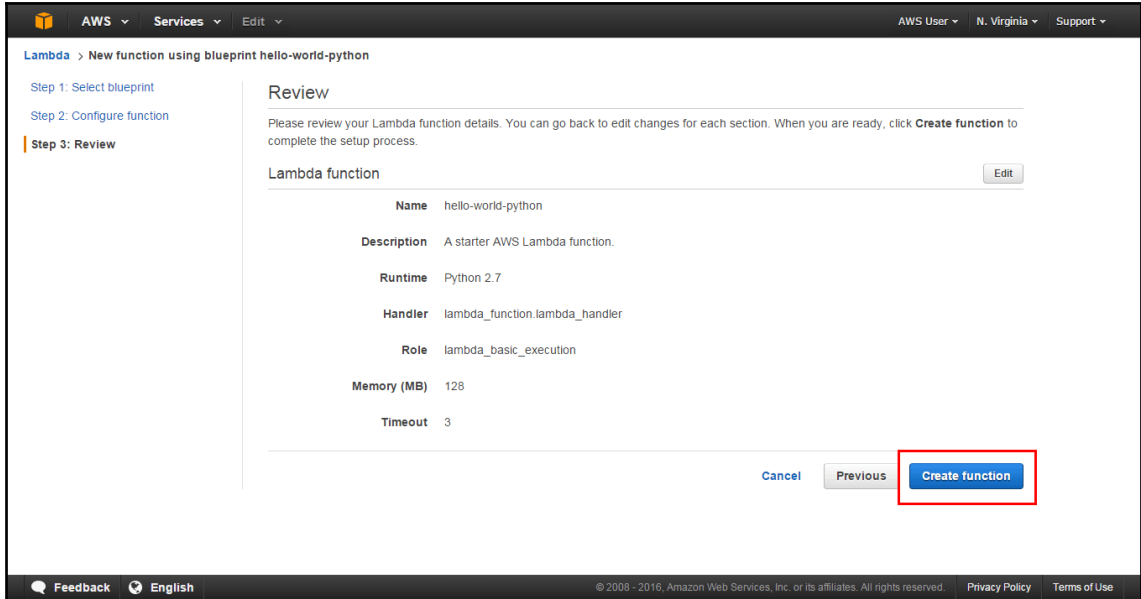
Role Name lambda_basic_execution

► View Policy Document

Don't Allow Allow

AWS: Role summary page

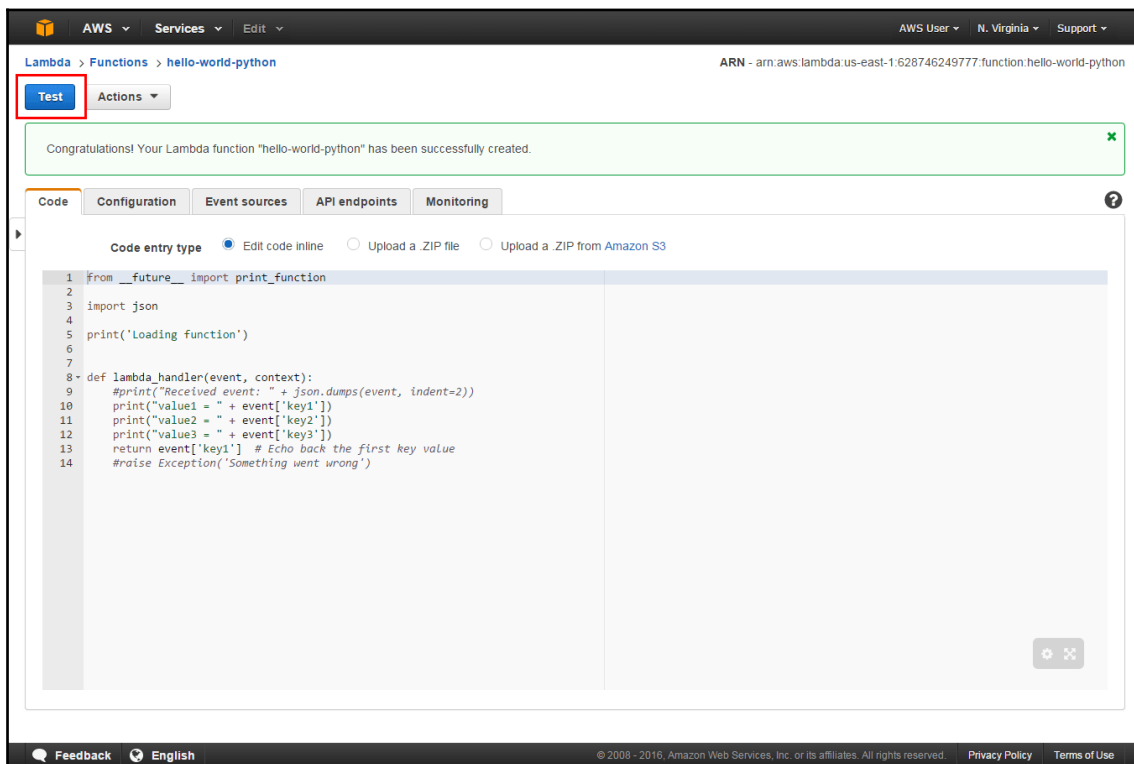
6. The console saves the code in a compressed file, which represents the distribution package. The console then loads the distribution package into AWS Lambda to create the Lambda function:



AWS: Lambda review page

It is now possible to test the functions, check the results, and display the logs:

1. To run our first Lambda function, click on **Test**:



AWS: Lambda testing page

2. Enter an event in the pop-up editor to test the function.
3. Select **Hello World** from the **Sample event template** list on the **Input test event** page:

Input test event

It looks like you have not configured a test event for this function yet. Use the editor below to enter an event to test your function with (please remember that this will actually execute the code!). You can always edit the event later by choosing **Configure test event** in the Actions list. Note that changes to the event will only be saved locally.

Sample event template Hello World ▼

```
1 {  
2   "key3": "value3",  
3   "key2": "value2",  
4   "key1": "value1"  
5 }
```

Cancel Save Save and test

AWS: Lambda template

Click **Save and test**. Then, AWS Lambda will perform the function on your behalf.

How it works...

When execution is complete, it is possible to see the results in the console:

- The **Execution result** section documents the correct execution of the function.
- The **Summary** section shows the most important information reported in the **Log output** section.
- The **Log output** section shows the logs generated by the Lambda function execution:

The screenshot displays the AWS Lambda console's execution results for a function. It is divided into three main sections: Execution result, Summary, and Log output.

Execution result: succeeded (logs)
The area below shows the result returned by your function execution.
"hello, world!"

Summary

Code SHA-256	j3x8AeuD7gdjpESGMi91Pa2d1cWSSzMMr cJZ2AkEovo=
Request ID	a2b6e0e5-e7bc-11e5-b30a- 7d151ecbe64d
Duration	0.27 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	9 MB

Log output
The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. [Click here](#) to view the CloudWatch log group.

```
START RequestId: a2b6e0e5-e7bc-11e5-b30a-7d151ecbe64d Version: $LATEST
value1 = hello, world!
value2 = value2
value3 = value3
END RequestId: a2b6e0e5-e7bc-11e5-b30a-7d151ecbe64d
REPORT RequestId: a2b6e0e5-e7bc-11e5-b30a-7d151ecbe64d  Duration: 0.27 ms    Billed Duration: 100 ms    Memory Size: 128 MB    Max Memory Used: 9 MB
```

AWS: Execution results

There's more...

AWS Lambda monitors the functions and generates parameter reports automatically via **Amazon CloudWatch** (see the following screenshot). To simplify the monitoring of the code during execution, AWS Lambda automatically tracks the number of requests, the latency per request, and the number of requests with errors, publishing the associated parameters:



What is a Lambda function?

A Lambda function contains code that a developer wants to execute in response to certain events. The developer takes care of configuring this code and specifying the requirements in terms of resources within the console of the reference provider. Everything else, including the sizing of resources, is managed automatically by the provider, based on the workload required.

Why serverless?

The benefits of serverless computing are as follows:

- **No infrastructure management:** Developers can focus on the product to be built rather than on the operation and management of runtime servers.
- **Automatic scalability:** The resources are automatically recalibrated to cope with any type of workload, without requiring a configuration for scaling, but reacting to real-time events.

- **Resource use optimization:** Since the processing and storage resources are dynamically allocated, it is no longer necessary to invest in excess capacity in advance.
- **Cost reduction:** In traditional cloud computing, payment of running resources is expected even when they are not actually used. In the serverless case, the applications are event-driven, meaning that when the application code is not running, no cost is charged, so you won't have to pay for unused resources.
- **High availability:** The services that manage the infrastructure and the application guarantee high availability and fault tolerance.
- **Time to Market improvement:** The elimination of infrastructure management charges allows developers to focus on product quality and bring the code to production faster.

Possible problems and limitations

There are some cons to take into consideration when evaluating the adoption of serverless computing:

- **Possible loss of performance:** If the code is not used very frequently, then latency problems may occur in its execution. These are prominent in comparison to cases in which it is in continuous execution on a server, a virtual machine, or a container. This happens because (contrary to what occurs when using autoscaling policies) with the serverless model, the cloud provider often deallocates resources completely if the code is not used. This implies that if the runtime takes some time to start, then additional latency is inevitably created in the initial start phase.
- **Stateless mode:** Serverless functions operate in stateless mode. This means that if you want to add logic to save some elements, such as parameters to pass as arguments to a different function, then you need to add a persistent storage component to the application flow and link the events to each other. For example, Amazon provides an additional tool called **AWS Step Functions**, which coordinates and manages the status of all microservices and distributed components of serverless applications.

- **Limit to resources:** Serverless computing is not suitable for some types of workloads or use cases, particularly with high-performance ones and for the limits on the use of resources that are imposed by the cloud provider (for example, AWS limits the number of concurrent runs of Lambda functions). These are both due to the difficulty in provisioning the number of desired servers in a limited and fixed period of time.
- **Debugging and monitoring:** If you rely on non-open source solutions, then the developers will depend on vendors for debugging and monitoring applications, and therefore, will not be able to diagnose any problems in detail by using additional profilers or debuggers. Thus, they will have to rely on the tools provided by their respective providers.

See also

As we have seen, the reference point for working with serverless architectures is the AWS framework (<https://aws.amazon.com/>). At the preceding URL, you can find a lot of information and tutorials, including the example described in this section.

8

Heterogeneous Computing

This chapter will help us to explore the **Graphics Processing Unit (GPU)** programming techniques through the Python language. The continuous evolution of GPUs is revealing how these architectures can bring great benefits to performing complex calculations.

GPUs certainly cannot replace CPUs. However, they are a well-structured and heterogeneous code that is able to exploit the strengths of both types of processors that can, in fact, bring considerable advantages.

We will examine the main development environments for heterogeneous programming, namely, the **PyCUDA** and **Numba** environments for **Compute Unified Device Architecture (CUDA)** and **PyOpenCL** environments, which are for **Open Computing Language (OpenCL)** frameworks in their Python version.

In this chapter, we will cover the following recipes:

- Understanding heterogeneous computing
- Understanding the GPU architecture
- Understanding GPU programming
- Dealing with PyCUDA
- Heterogeneous programming with PyCUDA
- Implementing memory management with PyCUDA
- Introducing PyOpenCL
- Building applications with PyOpenCL
- Element-wise expressions with PyOpenCL
- Evaluating PyOpenCL applications
- GPU programming with Numba

Let's start with understanding heterogeneous computing in detail.

Understanding heterogeneous computing

Over the years, the search for better performance for increasingly complex calculations has led to the adoption of new techniques in the use of computers. One of these techniques is called *heterogeneous computing*, which aims to cooperate with different (or heterogeneous) processors in such a way as to have advantages (in particular) in terms of temporal computational efficiency.

In this context, the processor on which the main program is run (generally the CPU) is called the *host*, while the coprocessors (for example, the GPUs) are called *devices*. The latter are generally physically separated from the host and manage their own memory space, which is also separated from the host's memory.

In particular, following significant market demand, the GPU has evolved into a highly parallel processor, transforming the GPU from devices for graphics rendering to devices for parallelizable and computationally intensive general-purpose calculations.

In fact, the use of GPU for tasks other than rendering graphics on the screen is called heterogeneous computing.

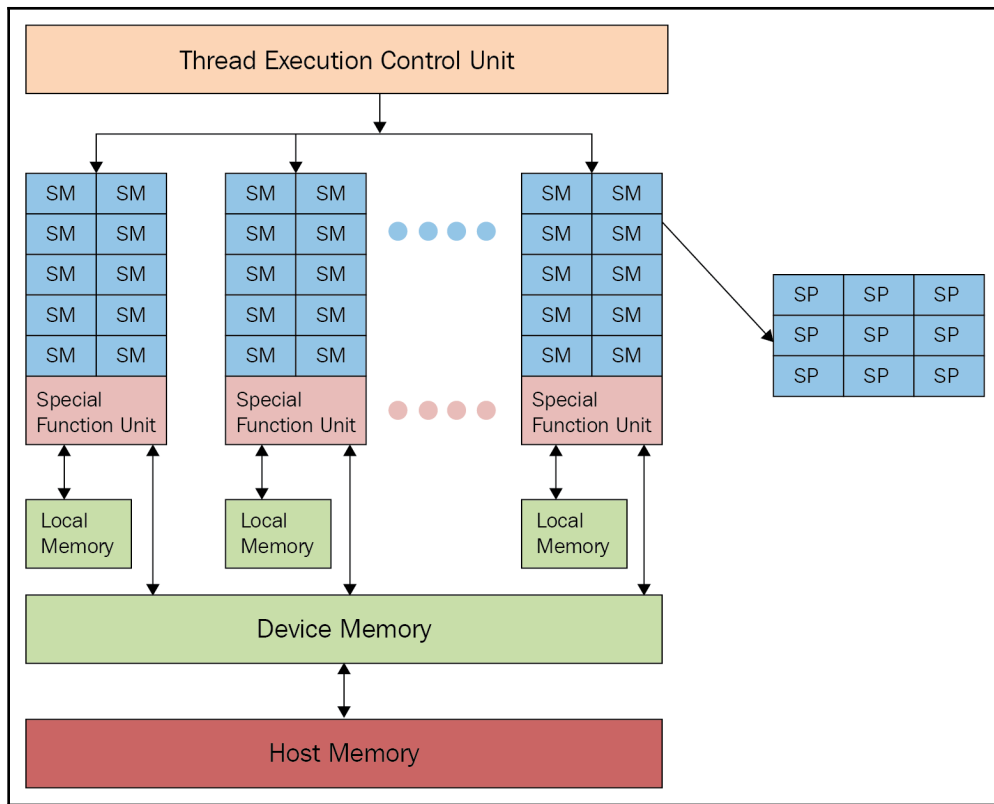
Finally, the task of good GPU programming is to make the most of the great level of parallelism and mathematical capabilities offered by the graphics card, minimizing all the disadvantages presented by it, such as the delay of the physical connection between the host and device.

Understanding the GPU architecture

A GPU is a specialized CPU/core for vector processing of graphical data to render images from polygonal primitives. The task of a good GPU program is to make the most of the great level of parallelism and mathematical capabilities offered by the graphics card and minimize all the disadvantages presented by it, such as the delay in the physical connection between the host and device.

GPUs are characterized by a highly parallel structure that allows you to manipulate large datasets in an efficient manner. This feature is combined with rapid improvements in hardware performance programs, bringing the attention of the scientific world to the possibility of using GPUs for purposes other than just rendering images.

A GPU (refer to the following diagram) is composed of several processing units called **Streaming Multiprocessors (SMs)**, which represent the first logic level of parallelism. In fact, each SM works simultaneously and independently from the others:



GPU architecture

Each SM is divided into a group of **Streaming Processors (SPs)**, which have a core that can run a thread sequentially. The SP represents the smallest unit of execution logic and the level of finer parallelism.

In order to best program this type of architecture, we need to introduce GPU programming, which is described in the next section.

Understanding GPU programming

GPUs have become increasingly programmable. In fact, their set of instructions has been extended to allow the execution of a greater number of tasks.

Today, on a GPU, it is possible to execute classic CPU programming instructions, such as cycles and conditions, memory access, and floating-point calculations. The two major discrete video card manufacturers—**NVIDIA** and **AMD**—have developed their GPU architectures, providing developers with related development environments that allow programming in different programming languages, including Python.

At present, developers have valuable tools for programming software that uses GPUs in contexts that aren't purely graphics-related. Among the main development environments for heterogeneous computing, we have CUDA and OpenCL.

Let's now have a look at them in detail.

CUDA

CUDA is a proprietary hardware architecture of NVIDIA, which also gives its name to the related development environment. Currently, CUDA has a pool of hundreds of thousands of active developers, which demonstrates the growing interest that is developing around this technology in the parallel programming environment.

CUDA offers extensions for the most commonly used programming languages, including Python. The most well known CUDA Python extensions are as follows:

- PyCUDA (<https://mathematician.de/software/PyCUDA/>)
- Numba (<http://numba.pydata.org>)

We'll use these extensions in the coming sections.

OpenCL

The second protagonist in parallel computing is OpenCL, which (unlike its NVIDIA counterpart) is open standard and can be used not only with GPUs of different manufacturers but also with microprocessors of different types.

However, OpenCL is a more complete and versatile solution as it does not boast the maturity and simplicity of use that CUDA has.

The OpenCL Python extension is PyOpenCL (<https://mathematician.de/software/pyopencl/>).

In the following sections, the CUDA and OpenCL programming models will be analyzed in their Python extension and will be accompanied by some interesting application examples.

Dealing with PyCUDA

PyCUDA is a binding library that provides access to CUDA's Python API by Andreas Klöckner. The main features include automatic cleanup, which is tied to an object's lifetime, thus preventing leaks, convenient abstraction over modules and buffers, full access to the driver, and built-in error handling. It is also very light.

The project is open source under the MIT license, the documentation is very clear, and many different sources found online can provide help and support. The main purpose of PyCUDA is to let a developer invoke CUDA with minimal abstraction from Python, and it also supports CUDA metaprogramming and templatzation.

Getting ready

Please follow the instructions on the Andreas Klöckner home page (<https://mathematikian.de/software/pycuda/>) to install PyCUDA.

The next programming example has a dual function:

- The first is to verify that PyCUDA is properly installed.
- The second is to read and to print the characteristics of the GPU cards.

How to do it...

Let's look at the steps, as follows:

1. With the first instruction, we import the Python driver (that is, `pycuda.driver`) to the CUDA library installed on our PC:

```
import pycuda.driver as drv
```

2. Initialize CUDA. Note also that the following instruction must be called before any other instruction in the `pycuda.driver` module:

```
drv.init()
```

3. Enumerate the number of GPU cards on the PC:

```
print ("%d device(s) found." % drv.Device.count())
```

4. For each of the GPU cards present, print the model name, the computing capability, and the total amount of memory on the device in kilobytes:

```
for ordinal i n range(drv.Device.count()):
    dev = drv.Device(ordinal)
    print ("Device #d: %s" % (ordinal, dev.name()))
    print ("Compute Capability: %d.%d"%
dev.compute_capability())
    print ("Total Memory: %s KB" % (dev.total_memory()//(1024)))
```

How it works...

The execution is pretty simple. In the first line of code, `pycuda.driver` is imported and then initialized:

```
import pycuda.driver as drv
drv.init()
```

The `pycuda.driver` module exposes the driver level to the programming interface of CUDA, which is more flexible than the CUDA C runtime-level programming interface, and it has a few features that are not present in the runtime.

Then, it cycles into the `drv.Device.count()` function and, for each GPU card, the name of the card and its main characteristics (computing capability and total memory) are printed:

```
print ("Device #d: %s" % (ordinal, dev.name()))
print ("Compute Capability: %d.%d" % dev.compute_capability())
print ("Total Memory: %s KB" % (dev.total_memory()//(1024)))
```

Execute the following code:

```
C:\>python dealingWithPycuda.py
```

When you've done so, the installed GPU will be shown on the screen, as in the following example:

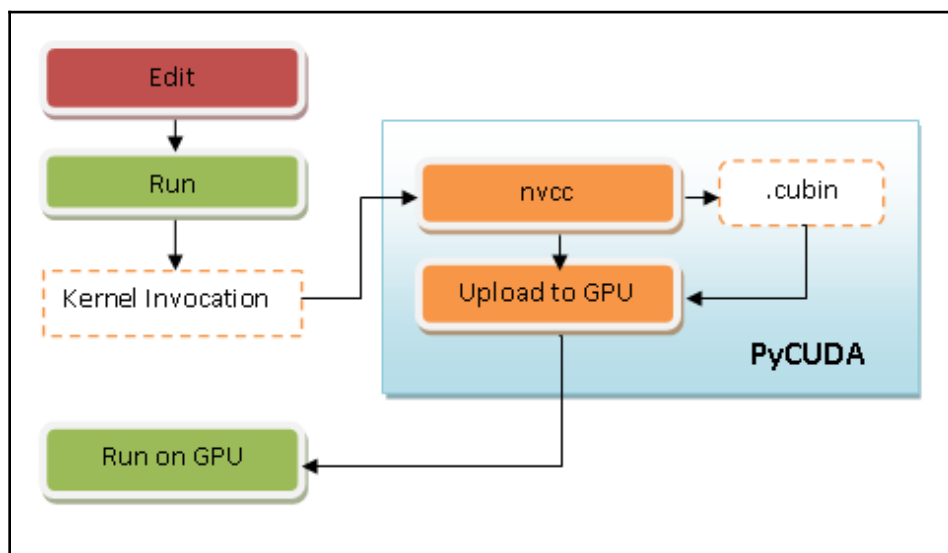
```
1 device(s) found.
Device #0: GeForce GT 240
Compute Capability: 1.2
Total Memory: 1048576 KB
```


There's more...

The CUDA programming model (and consequently PyCUDA, which is a Python wrapper) is implemented through specific extensions to the standard library of the C language. These extensions have been created just like function calls in the standard C library, allowing a simple approach to a heterogeneous programming model that includes the host and device code. The management of the two logical parts is done by the `nvcc` compiler.

Here is a brief description of how this works:

1. *Separate* device code from the host code.
2. *Invoke* a default compiler (for example, GCC) to compile the host code.
3. *Build* the device code in binary form (`.cubin` objects) or in assembly form (PTX objects):



PyCUDA execution model

All the preceding steps are performed by PyCUDA during execution, with an increase in the application loading time compared to a CUDA application.

See also

- The CUDA programming guide is available here: <https://docs.nvidia.com/CUDA/CUDA-c-programming-guide/>
- The PyCUDA documentation is available here: <https://documen.tician.de/PyCUDA/>

Heterogeneous programming with PyCUDA

The CUDA programming model (and, hence, that of PyCUDA) is designed for the joint execution of a software application on a CPU and GPU, in order to perform the sequential parts of the application on the CPU and those that can be parallelized on the GPU.

Unfortunately, the computer is not smart enough to understand how to distribute the code autonomously, so it is up to the developer to indicate which parts should be run by the CPU and by the GPU.

In fact, a CUDA application is composed of serial components, which are executed by the system CPU or host, or by parallel components called kernels, which are executed by the GPU or by the device instead.

A kernel is defined as a *grid* and can, in turn, be decomposed into blocks that are sequentially assigned to the various multiprocessors, thus implementing *coarse-grained parallelism*. Inside the blocks, there is the fundamental computational unit, the thread, with a very *fine parallel granularity*. A thread can belong to only one block and is identified by a unique index for the whole kernel. For convenience, there is the possibility of using two-dimensional indexes for blocks and three-dimensional indexes for threads. The kernels are executed sequentially between them. Blocks and threads, on the other hand, are executed in parallel. The number of threads running (in parallel) depends on their organization in blocks and on their requests in terms of resources, with respect to the resources available in the device.



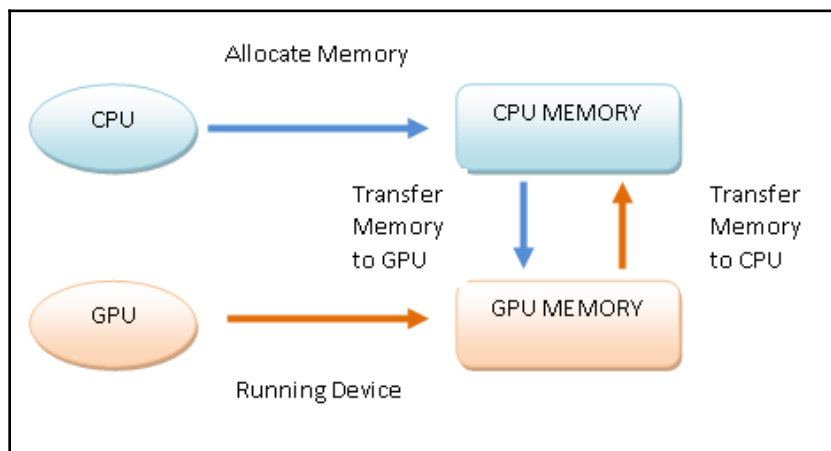
To visualize the concepts expressed previously, please refer to (Figure 5) at <https://sites.google.com/site/computationvisualization/programming/cuda/article1>.

The blocks are designed to guarantee scalability. In fact, if you have an architecture with two multiprocessors and another with four, then, a GPU application can be performed on both architectures, obviously with different times and levels of parallelism.

The execution of a heterogeneous program according to the PyCUDA programming model is thus structured as follows:

1. *Allocate* memory on the host.
2. *Transfer* data from the host memory to the device memory.
3. *Run* the device through the invocation of the kernel functions.
4. *Transfer* the results from the device memory to the host memory.
5. *Release* the memory allocated on the device.

The following diagram shows the execution flow of a program according to the PyCUDA programming model:



PyCUDA programming model

In the next example, we will go through a concrete example of the programming methodology to follow in order to build PyCUDA applications.

How to do it...

In order to show the PyCUDA programming model, we consider the task of having to double all the elements of a 5×5 matrix:

1. We import the libraries needed for the task we want to perform:

```
import PyCUDA.driver as CUDA
import PyCUDA.autoinit
from PyCUDA.compiler import SourceModule
import numpy
```

2. The `numpy` library, which we imported, allows us to construct the input to our problem, that is, a 5×5 matrix whose values are chosen randomly:

```
a = numpy.random.randn(5, 5)
a = a.astype(numpy.float32)
```

3. The matrix, thus built, must be copied from the memory of the host to the memory of the device. For this, we allocate a memory space (`a_gpu`) on the device that is necessary to contain matrix `a`. For this purpose, we use the `mem_alloc` function, which has the allocated memory space as its subject. In particular, the number of bytes of matrix `a`, as expressed by the `a.nbytes` parameter, is as follows:

```
a_gpu = cuda.mem_alloc(a.nbytes)
```

4. After that, we can transfer the matrix from the host to the memory area, created specifically on the device by using the `memcpy_htod` function:

```
cuda.memcpy_htod(a_gpu, a)
```

5. Inside the device, the `doubleMatrix` kernel function will operate. Its purpose will be to multiply each element of the input matrix by 2. As you can see, the syntax of the `doubleMatrix` function is C-like, while the `SourceModule` statement is a real directive for the NVIDIA compiler (the `nvcc` compiler), which creates a module that, in this case, consists of the `doubleMatrix` function only:

```
mod = SourceModule("""
__global__ void doubles_matrix(float *a){
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;}
""")
```

6. With the `func` parameter, we identify the `doubleMatrix` function, which is contained in the `mod` module:

```
func = mod.get_function("doubles_matrix")
```

7. Finally, we run the kernel function. In order to successfully execute a kernel function on the device, the CUDA user must specify the input for the kernel and the size of the execution thread block. In the following case, the input is the `a_gpu` matrix that was previously copied to the device, while the dimension of the thread block is `(5, 5, 1)`:

```
func(a_gpu, block=(5,5,1))
```

8. Therefore, we allocate an area of memory of size equal to that of the input matrix `a`:

```
a_doubled = numpy.empty_like(a)
```

9. Then, we copy the contents of the memory area allocated to the device—that is, the `a_gpu` matrix—to the previously defined memory area, `a_doubled`:

```
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

10. Finally, we print the contents of the input matrix `a` and the output matrix in order to verify the quality of the implementation:

```
print ("ORIGINAL MATRIX")
print (a)
print ("DOUBLED MATRIX AFTER PyCUDA EXECUTION")
print (a_doubled)
```

How it works...

Let's start with looking at which libraries are imported for this example:

```
import PyCUDA.driver as CUDA
import PyCUDA.autoint
from PyCUDA.compiler import SourceModule
```

In particular, the `autoint` import automatically identifies which GPU on our system is available for execution, while `SourceModule` is the directive for the compiler of NVIDIA (`nvcc`) that allows us to identify the objects that must be compiled and uploaded to the device.

Then, we build the 5×5 input matrix by using the `numpy` library:

```
import numpy
a = numpy.random.randn(5,5)
```

In this case, the elements in the matrix are converted to single-precision mode (since the graphics card on which this example is executed only supports single precision):

```
a = a.astype(numpy.float32)
```

Then, we copy the array from the host to the device, using the following two operations:

```
a_gpu = CUDA.mem_alloc(a.nbytes)
CUDA.memcpy_htod(a_gpu, a)
```

Note that the device and host memory may never communicate during the execution of a kernel function. For this reason, in order to parallel execute the kernel function on the device, all input data relating to the kernel function must also be present in the memory of the device.

It should also be noted that the `a_gpu` matrix is linearized, that is, it is one-dimensional, and therefore we must manage it as such.

Moreover, all these operations do not require kernel invocation. This means that they are made directly by the host.

The `SourceModule` entity allows the definition of the `doubleMatrix` kernel function. `__global__`, which is an `nvcc` directive, indicates that the `doubleMatrix` function will be processed by the device:

```
mod = SourceModule("""
__global__ void doubleMatrix(float *a)
```

Let's consider the kernel's body. The `idx` parameter is the matrix index, which is identified by the `threadIdx.x` and `threadIdx.y` thread coordinates:

```
int idx = threadIdx.x + threadIdx.y*4;
a[idx] *= 2;
```

Then, `mod.get_function("doubleMatrix")` returns an identifier to the `func` parameter:

```
func = mod.get_function("doubleMatrix ")
```

In order to execute the kernel, we need to configure the execution context. This means setting the three-dimensional structure of the threads that belong to the block grid by using the `block` parameter inside the `func` call:

```
func(a_gpu, block = (5, 5, 1))
```

`block = (5, 5, 1)` tells us that we are calling a kernel function with the `a_gpu` linearized input matrix and a single thread block of size 5 (that is, 5 threads) in the x -direction, 5 threads in the y -direction, and 1 thread in the z -direction, which makes 16 threads in total. Note that each thread executes the same kernel code (25 threads in total).

After the computation in the GPU device, we use an array to store the results:

```
a_doubled = numpy.empty_like(a)
CUDA.memcpy_dtoh(a_doubled, a_gpu)
```

To run the example, type the following on Command Prompt:

```
C:\>python heterogeneousPycuda.py
```

The output should be like this:

ORIGINAL MATRIX

```
[[-0.59975582  1.93627465  0.65337795  0.13205571 -0.46468592]
 [ 0.01441949  1.40946579  0.5343408  -0.46614054 -0.31727529]
 [-0.06868593  1.21149373 -0.6035406  -1.29117763  0.47762445]
 [ 0.36176383 -1.443097  1.21592784 -1.04906416 -1.18935871]
 [-0.06960868 -1.44647694 -1.22041082  1.17092752  0.3686313  ]]
```

DOUBLED MATRIX AFTER PyCUDA EXECUTION

```
[[-1.19951165  3.8725493  1.3067559  0.26411143 -0.92937183]
 [ 0.02883899  2.81893158  1.0686816  -0.93228108 -0.63455057]
 [-0.13737187  2.42298746 -1.2070812  -2.58235526  0.95524889]
 [ 0.72352767 -2.886194  2.43185568 -2.09812832 -2.37871742]
 [-0.13921736 -2.89295388 -2.44082164  2.34185504  0.73726263  ]]
```

There's more...

The key feature of CUDA that makes this programming model substantially different from other parallel models (normally used on CPUs) is that in order to be efficient, it requires thousands of threads to be active. This is made possible by the typical structure of GPUs, which use light threads and also allow the creation and modification of execution contexts in a very fast and efficient way.

Note that the scheduling of threads is directly linked to the GPU architecture and its intrinsic parallelism. In fact, a block of threads is assigned to a single SM. Here, the threads are further divided into groups, called warps. The threads that belong to the same warp are managed by the *warp scheduler*. To take full advantage of the inherent parallelism of the SM, the threads of the same warp must execute the same instruction. If this condition does not occur, then we speak of *threads divergence*.

See also

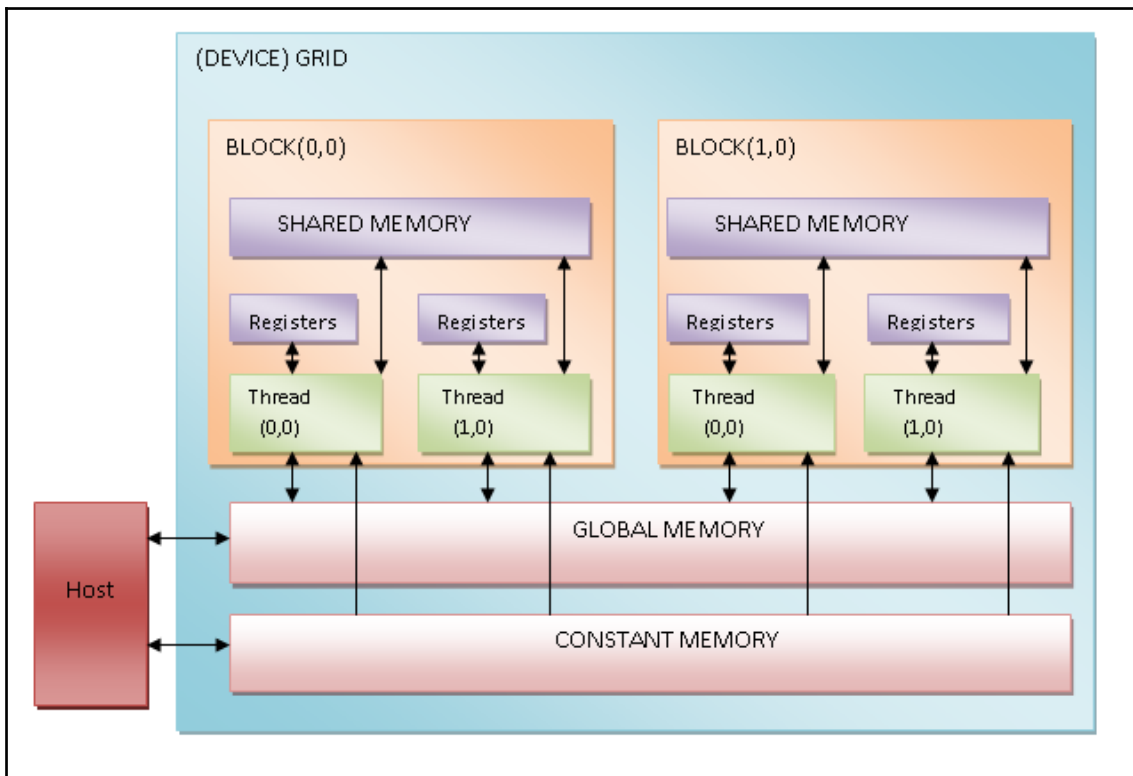
- The complete tutorial on using PyCUDA is available at the following site: <https://documen.tician.de/pycuda/tutorial.html>.
- To install PyCUDA on Windows 10, take a look at the following link: <https://github.com/kdcoadd/Win10-PyCUDA-Install>.

Implementing memory management with PyCUDA

PyCUDA programs should respect the rules dictated by the structure and the internal organization of SM that impose constraints on thread performances. In fact, the knowledge and the correct use of various types of memory that the GPU makes available are fundamental in order to achieve maximum efficiency. In those GPU cards, enabled for CUDA use, there are four types of memory, which are as follows:

- **Registers:** Each thread is assigned a memory register which only the assigned thread can access, even if the threads belong to the same block.
- **Shared memory:** Each block has its own shared memory between the threads that belong to it. Even this memory is extremely fast.
- **Constant memory:** All threads in a grid have constant access to the memory, but can only be accessed in reading. The data present in it persists for the entire duration of the application.

- **Global memory:** All the threads of the grid, and therefore all the kernels, have access to the global memory. Moreover, data persistence is exactly like a constant memory:



GPU memory model

Getting ready

For best performance, a PyCUDA program must, therefore, make the most of every type of memory. In particular, it must make the most of shared memory, minimizing access to memory on a global level.

To do this, the problem domain is typically subdivided so that a single block of threads is able to execute its processing in a closed subset of data. In this way, the threads operating on the single block will all work together on the same shared memory area, optimizing access.

The basic steps for each thread are as follows:

1. *Load* data from global memory to shared memory.
2. *Synchronize* all threads of the block so that everyone can read safety positions and shared memory filled by other threads.
3. *Process* the data of the shared memory. Making a new synchronization is necessary to ensure that the shared memory has been updated with the results.
4. *Write* the results in global memory.

To clarify this approach, in the following section, we will present an example based on the calculation of the product of two matrices.

How to do it...

The following code fragment shows the calculation of the product of two matrices, $M \times N$, in the standard method, which is based on a sequential approach. Each element of the output matrix, P , is obtained by taking a row element from matrix M , and a column element from matrix N :

```
void SequentialMatrixMultiplication(float*M,float *N,float *P, int width){
    for (int i=0; i< width; ++i)
        for(int j=0;j < width; ++j) {
            float sum = 0;
            for (int k = 0 ; k < width; ++k) {
                float a = M[I * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[I * width + j] = sum;
        }
    }
P[I * width + j] = sum;
```

In this case, if each thread had been given the task of calculating each element of the matrix, then access to the memory would have dominated the execution time of the algorithm.

What we can do is rely on a block of threads to calculate one output submatrix at a time. In this way, the threads that access the same memory block cooperate to optimize accesses, thereby minimizing the total calculation time:

1. The first step is to load all the necessary modules to implement the algorithm:

```
import numpy as np
from pycuda import driver, compiler, gpuarray, tools
```

2. Then, initialize the GPU device:

```
import pycuda.autoinit
```

3. We implement `kernel_code_template`, which implements the product of two matrices that are respectively indicated with `a` and `b`, while the resulting matrix is indicated with the parameter `c`. Note that the `MATRIX_SIZE` parameter will be defined in the next step:

```
kernel_code_template = """
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }
    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}"""
```

4. The following parameter will be used to set the dimensions of the matrices. In this case, the size is 5×5 :

```
MATRIX_SIZE = 5
```

5. We define the two input matrices, `a_cpu` and `b_cpu`, that will contain random floating-point values:

```
a_cpu = np.random.randn(MATRIX_SIZE,
MATRIX_SIZE).astype(np.float32)
b_cpu = np.random.randn(MATRIX_SIZE,
MATRIX_SIZE).astype(np.float32)
```

6. Then, we calculate the product of the two matrices, `a` and `b`, on the host device:

```
c_cpu = np.dot(a_cpu, b_cpu)
```

7. We allocate memory areas on the device (GPU), equal in size to the input matrices:

```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
```

8. We allocate a memory area on the GPU, equal in size to the output matrix resulting from the product of the two matrices. In this case, the resulting matrix, `c_gpu`, will have a size of 5×5 :

```
c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)
```

9. The following `kernel_code` redefines `kernel_code_template`, but with the `matrix_size` parameter set:

```
kernel_code = kernel_code_template % {
    'MATRIX_SIZE': MATRIX_SIZE}
```

10. The `SourceModule` directive tells `nvcc` (*NVIDIA CUDA Compiler*) that it will have to create a module—that is, a collection of functions—containing the previously defined `kernel_code`:

```
mod = compiler.SourceModule(kernel_code)
```

11. Finally, we take the `MatrixMulKernel` function from the module, `mod`, to which we give the name `matrixmul`:

```
matrixmul = mod.get_function("MatrixMulKernel")
```

12. We execute the product between two matrices, `a_gpu` and `b_gpu`, resulting in the `c_gpu` matrix. The size of the thread block is defined as `MATRIX_SIZE`, `MATRIX_SIZE`, 1:

```
matrixmul(  
    a_gpu, b_gpu,  
    c_gpu,  
    block = (MATRIX_SIZE, MATRIX_SIZE, 1))
```

13. Print the input matrices:

```
print ("-" * 80)  
print ("Matrix A (GPU):")  
print (a_gpu.get())  
print ("-" * 80)  
print ("Matrix B (GPU):")  
print (b_gpu.get())  
print ("-" * 80)  
print ("Matrix C (GPU):")  
print (c_gpu.get())
```

14. To check the validity of the calculation performed on the GPU, we compare the results of the two implementations, which are the one performed on the host device (CPU) and the one performed on the device (GPU). To do this, we use the `numpy.allclose` directive, which verifies that two element-wise arrays are equal within a tolerance equal to `1e-05`:

```
np.allclose(c_cpu, c_gpu.get())
```

How it works...

Let's consider the PyCUDA programming workflow. Let's prepare the input matrix, the output matrix, and where to store the results:

```
MATRIX_SIZE = 5  
a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)  
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)  
c_cpu = np.dot(a_cpu, b_cpu)
```

Then, we transfer these matrices to the GPU device by using the `gpuarray.to_gpu()` PyCUDA function:

```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)
```

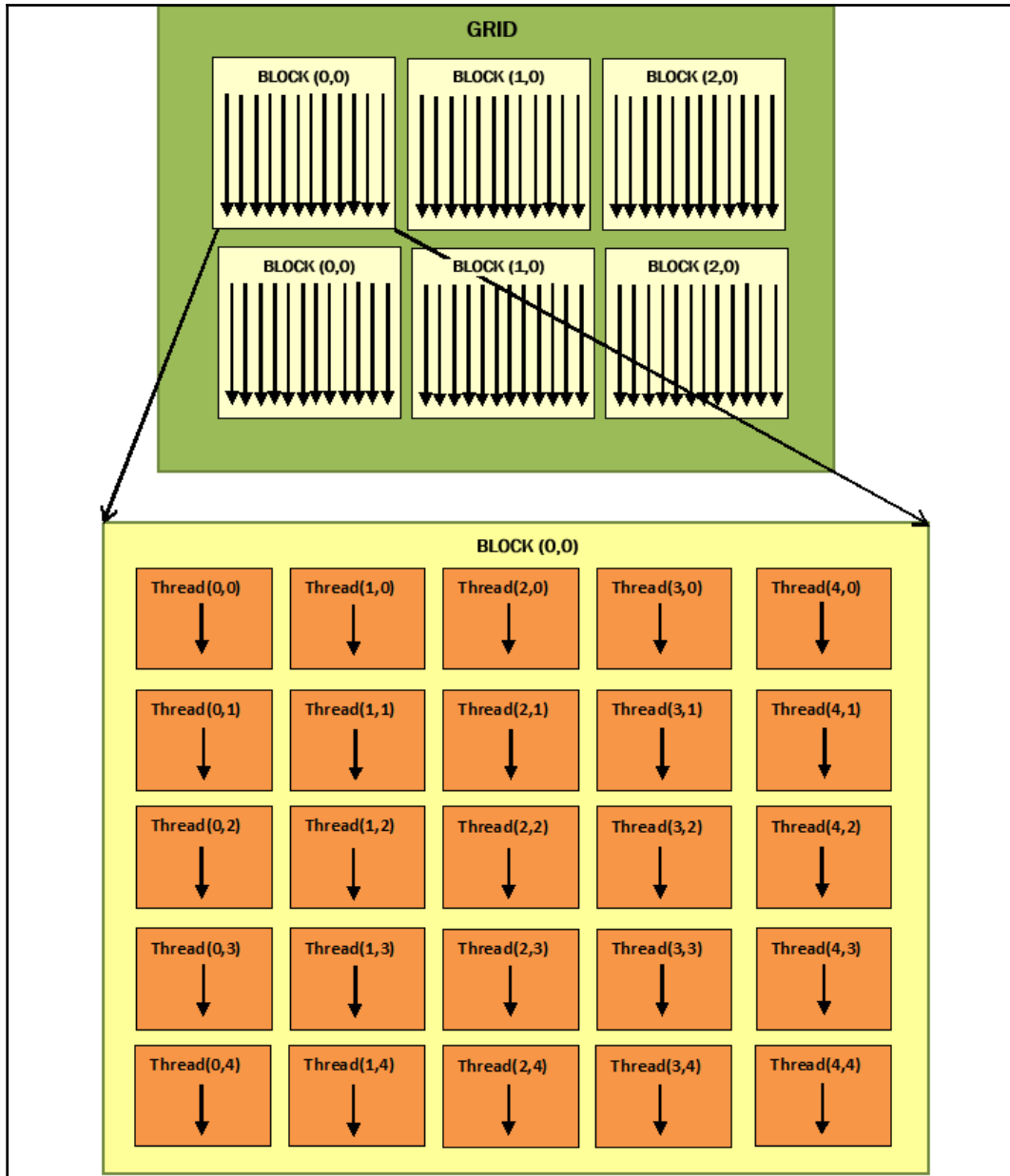
The core of the algorithm is the following kernel function. Let's remark that the `__global__` keyword specifies that this function is a kernel function, which means that it will be executed by the device (GPU) following a call from the host code (CPU):

```
__global__ void MatrixMulKernel(float *a, float *b, float *c){
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }
    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}
```

`threadIdx.x` and `threadIdx.y` are coordinates that allow the identification of the threads in the grid of two-dimensional blocks. Note that the threads within the grid block execute the same kernel code but on different pieces of data. If we compare the parallel version with the sequential one, then we immediately notice that the cycle indexes, i and j , have been replaced by the `threadIdx.x` and `threadIdx.y` indexes.

This means that in the parallel version, we will have only one iteration of the cycle. In fact, the `MatrixMulKernel` kernel will be executed on a grid of dimensions of 5×5 parallel threads.

This condition is expressed in the following diagram:



Grid and block of thread organization for the example

Then, we verify the product computation just by comparing the two resulting matrices:

```
np.allclose(c_cpu, c_gpu.get())
```

The output is as follows:

```
C:\>python memManagementPycuda.py
```

```
-----
Matrix A (GPU):
```

```
[[ 0.90780383 -0.4782407 0.23222363 -0.63184392 1.05509627]
 [-1.27266967 -1.02834761 -0.15528528 -0.09468858 1.037099 ]
 [-0.18135822 -0.69884419 0.29881889 -1.15969539 1.21021318]
 [ 0.20939326 -0.27155793 -0.57454145 0.1466181 1.84723163]
 [ 1.33780348 -0.42343542 -0.50257754 -0.73388749 -1.883829 ]]
```

```
-----
Matrix B (GPU):
```

```
[[ 0.04523897 0.99969769 -1.04473436 1.28909719 1.10332143]
 [-0.08900332 -1.3893919 0.06948703 -0.25977209 -0.49602833]
 [-0.6463753 -1.4424541 -0.81715286 0.67685211 -0.94934392]
 [ 0.4485206 -0.77086055 -0.16582981 0.08478995 1.26223004]
 [-0.79841441 -0.16199949 -0.35969591 -0.46809086 0.20455229]]
```

```
-----
Matrix C (GPU):
```

```
[[ -1.19226956 1.55315971 -1.44614291 0.90420711 0.43665022]
 [-0.73617989 0.28546685 1.02769876 -1.97204924 -0.65403283]
 [-1.62555301 1.05654192 -0.34626681 -0.51481217 -1.35338223]
 [-1.0040834 1.00310731 -0.4568972 -0.90064859 1.47408712]
 [ 1.59797418 3.52156591 -0.21708387 2.31396151 0.85150564]]
```

```
-----
TRUE
```

There's more...

The data allocated in shared memory has limited visibility in the single-threaded block. It is easy to see that the PyCUDA programming model adapts to specific classes of applications.

In particular, the features that these applications must present concern the presence of many mathematical operations, with a high degree of data parallelism (that is, the same sequence of operations being repeated on large amounts of data).

The application fields that possess these characteristics all belong to the following sciences: cryptography, computational chemistry, and image and signal analysis.

See also

- More examples of using PyCUDA can be found at <https://github.com/zamorays/miniCursoPycuda>.

Introducing PyOpenCL

PyOpenCL is a sister project to PyCUDA. It is a binding library that provides full access to OpenCL's API from Python and is also by Andreas Klöckner. It features many of the same concepts as PyCUDA, including cleanup for out-of-scope objects, partial abstraction over data structures, and error handling, all with minimal overhead. The project is available under the MIT license; its documentation is very good and plenty of guides and tutorials can be found online.

The main focus of PyOpenCL is to provide a lightweight connection between Python and OpenCL, but it also includes support for templates and metaprograms. The flow of a PyOpenCL program is almost exactly the same as a C or C++ program for OpenCL. The host program prepares the call of the device program, launches it, and then waits for the result.

Getting ready

The main reference for the PyOpenCL installation is the Andreas Klöckner home page: <https://mathematician.de/software/pyopencl/>.

If you are using Anaconda, then it is advisable to perform the following steps:

1. Install the latest Anaconda distribution with Python 3.7 from the following link: <https://www.anaconda.com/distribution/#download-section>. For this section, the Anaconda 2019.07 for Windows Installer has been installed.
2. Get the PyOpenCL prebuilt binary from Christoph Gohlke from this link: <https://www.lfd.uci.edu/~gohlke/pythonlibs/>. Select the right combination of OS and CPython versions. Here, we use `pyopencl-2019.1+cl12-cp37-cp37m-win_amd64.whl`.

3. Use `pip` to install the previous package. Simply type this in your Anaconda Prompt:

```
(base) C:\> pip install <directory>\pyopencl-2019.1+cl12-cp37-  
cp37m-win_amd64.whl
```

<directory> is the folder where the PyOpenCL package is located.

Moreover, the following notation indicates that we are operating on the Anaconda Prompt:

```
(base) C:\>
```

How to do it...

In the following example, we will use a function of PyOpenCL that allows us to enumerate the features of the GPU on which it will operate.

The code we implement is very simple and logical:

1. In the first step, we import the `pyopencl` library:

```
import pyopencl as cl
```

2. We build a function whose output will provide us with the characteristics of the GPU hardware in use:

```
def print_device_info() :  
    print('\n' + '=' * 60 + '\nOpenCL Platforms and Devices')  
    for platform in cl.get_platforms():  
        print('=' * 60)  
        print('Platform - Name: ' + platform.name)  
        print('Platform - Vendor: ' + platform.vendor)  
        print('Platform - Version: ' + platform.version)  
        print('Platform - Profile: ' + platform.profile)  
  
        for device in platform.get_devices():  
            print(' ' + '-' * 56)  
            print(' Device - Name: ' \  
                  + device.name)  
            print(' Device - Type: ' \  
                  + cl.device_type.to_string(device.type))  
            print(' Device - Max Clock Speed: {0} Mhz'\  
                  .format(device.max_clock_frequency))  
            print(' Device - Compute Units: {0}'\  
                  .format(device.max_compute_units))  
            print(' Device - Local Memory: {0:.0f} KB'
```

```

        .format(device.local_mem_size/1024.0))
    print(' Device - Constant Memory: {0:.0f} KB'\
        .format(device.max_constant_buffer_size/1024.0))
    print(' Device - Global Memory: {0:.0f} GB'\
        .format(device.global_mem_size/1073741824.0))
    print(' Device - Max Buffer/Image Size: {0:.0f} MB'\
        .format(device.max_mem_alloc_size/1048576.0))
    print(' Device - Max Work Group Size: {0:.0f}'\
        .format(device.max_work_group_size))

    print('\n')

```

3. So, we implement the main function, which calls the previously implemented `print_device_info` function:

```

if __name__ == "__main__":
    print_device_info()

```

How it works...

The following command is used to import the `pyopenc1` library:

```
import pyopenc1 as cl
```

This allows us to use the `get_platforms` method, which returns a list of platform instances, that is, a list of devices in the system:

```
for platform in cl.get_platforms():
```

Then, for each device found, the following main features are shown:

- Name and device type
- Max clock speed
- Compute units
- Local/constant/global memory

The output for this example is as follows:

```

(base) C:\>python deviceInfoPyopenc1.py

=====
OpenCL Platforms and Devices
=====
Platform - Name: NVIDIA CUDA
Platform - Vendor: NVIDIA Corporation
Platform - Version: OpenCL 1.2 CUDA 10.1.152

```

```
Platform - Profile: FULL_PROFILE
-----
Device - Name: GeForce 840M
Device - Type: GPU
Device - Max Clock Speed: 1124 Mhz
Device - Compute Units: 3
Device - Local Memory: 48 KB
Device - Constant Memory: 64 KB
Device - Global Memory: 2 GB
Device - Max Buffer/Image Size: 512 MB
Device - Max Work Group Size: 1024
=====
Platform - Name: Intel(R) OpenCL
Platform - Vendor: Intel(R) Corporation
Platform - Version: OpenCL 2.0
Platform - Profile: FULL_PROFILE
-----
Device - Name: Intel(R) HD Graphics 5500
Device - Type: GPU
Device - Max Clock Speed: 950 Mhz
Device - Compute Units: 24
Device - Local Memory: 64 KB
Device - Constant Memory: 64 KB
Device - Global Memory: 3 GB
Device - Max Buffer/Image Size: 808 MB
Device - Max Work Group Size: 256
-----
Device - Name: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Device - Type: CPU
Device - Max Clock Speed: 2400 Mhz
Device - Compute Units: 4
Device - Local Memory: 32 KB
Device - Constant Memory: 128 KB
Device - Global Memory: 8 GB
Device - Max Buffer/Image Size: 2026 MB
Device - Max Work Group Size: 8192
```

There's more...

OpenCL is currently managed by the Khronos Group, a non-profit consortium of companies that collaborate in defining the specifications of this (and many other) standards and compliance parameters for the creation of OpenCL-specific drivers for each type of platform.

These drivers also provide functions for compiling programs that are written in the kernel language: these are converted into programs in some form of intermediate language that is usually vendor-specific, and then executed on the reference architectures.

More info on OpenCL can be found at the following link: <https://www.khronos.org/registry/OpenCL/>.

See also

- PyOpenCL documentation is available here: <https://documen.tician.de/pyopencl/>.
- One of the best introductions to PyOpenCL, even if somewhat dated, can be found at the following link: <http://www.drdobbs.com/open-source/easy-opencl-with-python/240162614>.

Building applications with PyOpenCL

The first step in the construction of a program for PyOpenCL is the coding of the host application. This is performed on the CPU and has the task of managing the possible execution of the kernel on the GPU card (that is, the device).

A *kernel* is a basic unit of executable code, similar to a C function. It can be data-parallel or task-parallel. However, the cornerstone of PyOpenCL is the exploitation of parallelism.

A fundamental concept is a *program*, which is a collection of kernels and other functions, analogous to dynamic libraries. So, we can group instructions in a kernel and group different kernels into a program.

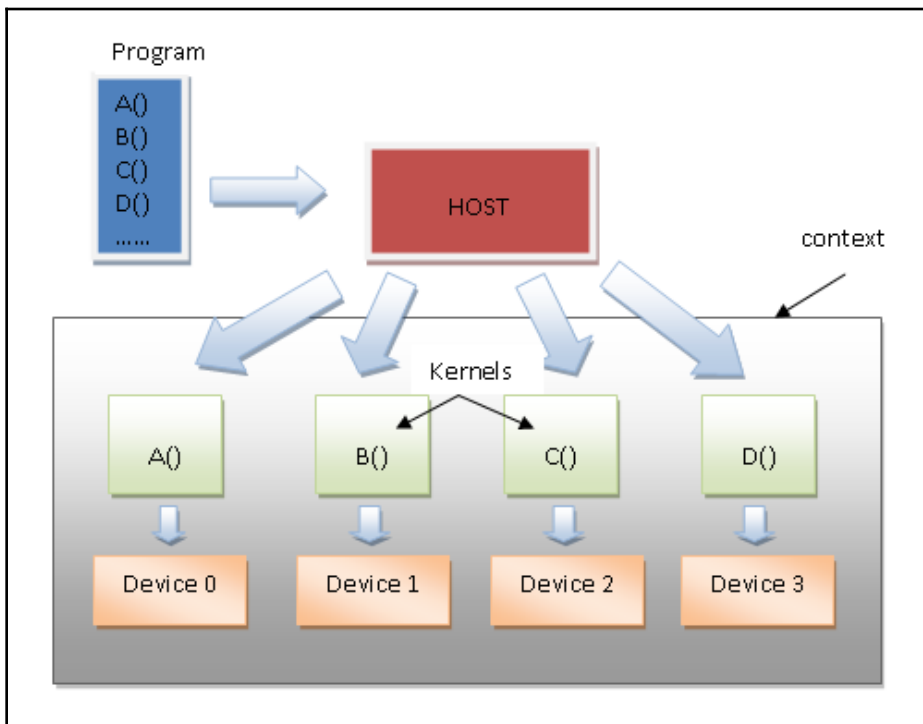
Programs can be called from applications. We have the execution queues that indicate the order in which the kernels are executed. However, in some cases, these can be launched without following the original order.

We can finally list the fundamental elements for developing an application with PyOpenCL:

- **Device:** This identifies the hardware in which the kernel code is to be executed. Note that the PyOpenCL application can be run on both CPU and GPU boards (as well as PyCUDA) but also on embedded devices such as **Field-Programmable Gate Arrays (FPGAs)**.

- **Program:** This is a group of kernels that has the task of selecting which kernel must be run on the device.
- **Kernel:** This is the code to execute on the device. A kernel is a C-like function, which means it can be compiled on any device that supports PyOpenCL drivers.
- **Command queue:** This orders the execution of kernels on the device.
- **Context:** This is a group of devices that allows devices to receive kernels and transfer data.

The following diagram shows how this data structure can work in a host application:



PyOpenCL programming model

Again, we observe that a program can contain more functions to run on the device and that each kernel encapsulates only a single function from the program.

How to do it...

In the following example, we show the basic steps to build an application with PyOpenCL: the task to be performed is the sum of two vectors. In order to have a readable output, we'll consider two vectors that each have 100 elements: each *i-th* element of the resulting vector will be equal to the sum of the *i-th* element of `vector_a`, plus the *i-th* element of `vector_b`:

1. Let's start by importing all the necessary libraries:

```
import numpy as np
import pyopencl as cl
import numpy.linalg as la
```

2. We define the size of the vectors to be added, as follows:

```
vector_dimension = 100
```

3. Here, the input vectors, `vector_a` and `vector_b`, are defined:

```
vector_a =
np.random.randint(vector_dimension, size=vector_dimension)
vector_b =
np.random.randint(vector_dimension, size=vector_dimension)
```

4. In sequence, we define platform, device, context, and queue:

```
platform = cl.get_platforms()[1]
device = platform.get_devices()[0]
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

5. Now, it's time to organize the memory areas that will contain the input vectors:

```
mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, \
hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, \
hostbuf=vector_b)
```

6. Finally, we build the application kernel by using the `Program` method:

```
program = cl.Program(context, """
__kernel void vectorSum(__global const int *a_g, __global const int
*b_g, __global int *res_g) {
    int gid = get_global_id(0);
    res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()
```

7. Then, we allocate the memory of the resulting matrix:

```
res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
```

8. Then, we call the kernel function:

```
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)
```

9. The memory space used to store the result is allocated in the host memory area (res_np):

```
res_np = np.empty_like(vector_a)
```

10. Copy the result of the computation into the memory area created:

```
cl._enqueue_copy(queue, res_np, res_g)
```

11. Finally, we print the results:

```
print ("PyOPENCL SUM OF TWO VECTORS")
print ("Platform Selected = %s" %platform.name )
print ("Device Selected = %s" %device.name)
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print (vector_a)
print ("INPUT VECTOR B")
print (vector_b)
print ("OUTPUT VECTOR RESULT A + B ")
print (res_np)
```

12. Then, we perform a simple check in order to verify that the sum operation is correct:

```
assert (la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

How it works...

In the following lines, after the relevant import, we define the input vectors:

```
vector_dimension = 100
vector_a = np.random.randint(vector_dimension, size= vector_dimension)
vector_b = np.random.randint(vector_dimension, size= vector_dimension)
```


Each vector contains 100 integer items, which are randomly selected through the `numpy` function:

```
np.random.randint(max_integer , size of the vector)
```

Then, we select the platform to achieve the computation by using the `get_platform()` method:

```
platform = cl.get_platforms()[1]
```

Then, select the corresponding device. Here, `platform.get_devices()[0]` corresponds to the Intel(R) HD Graphics 5500 graphics card:

```
device = platform.get_devices()[0]
```

In the following steps, the context and the queue are defined; PyOpenCL provides the method `context (device selected)` and `queue (context selected)`:

```
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

In order to perform the computation in the selected device, the input vector is copied to the device's memory:

```
mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, \
    hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, \
    hostbuf=vector_b)
```

Then, we prepare the buffer for the resulting vector:

```
res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
```

Here, the kernel code is defined:

```
program = cl.Program(context, """
__kernel void vectorSum(__global const int *a_g, __global const int *b_g,
__global int *res_g) {
    int gid = get_global_id(0);
    res_g[gid] = a_g[gid] + b_g[gid];}
""").build()
```

`vectorSum` is the name of the kernel, and the parameter list defines the data types of the input arguments and output data type (both are integer vectors). Inside the kernel body, the sum of two vectors is defined in the following steps:

1. *Initialize* the vector's index: `int gid = get_global_id(0).`
2. *Sum* the vector's components: `res_g[gid] = a_g[gid] + b_g[gid].`

In OpenCL (hence, in PyOpenCL), the buffers are attached to a context (<https://documentation.pyopencl.org/runtime.html#pyopencl.Context>), which are moved to a device once the buffer is used on that device.

Finally, we execute `vectorSum` in the device:

```
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)
```

To check the result, we use the `assert` statement. This tests the result and triggers an error if the condition is false:

```
assert(la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

The output should be as follows:

```
(base) C:\>python vectorSumPyopencl.py
```

```
PyOPENCL SUM OF TWO VECTORS
```

```
Platform Selected = Intel(R) OpenCL
```

```
Device Selected = Intel(R) HD Graphics 5500
```

```
VECTOR LENGTH = 100
```

```
INPUT VECTOR A
```

```
[45 46 0 97 96 98 83 7 51 21 72 70 59 65 79 92 98 24 56 6 70 64 59 0
 96 78 15 21 4 89 14 66 53 20 34 64 48 20 8 53 82 66 19 53 11 17 39 11
 89 97 51 53 7 4 92 82 90 78 31 18 72 52 44 17 98 3 36 69 25 87 86 68
 85 16 58 4 57 64 97 11 81 36 37 21 51 22 17 6 66 12 80 50 77 94 6 70
 21 86 80 69]
```

```
INPUT VECTOR B
```

```
[25 8 76 57 86 96 58 89 26 31 28 92 67 47 72 64 13 93 96 91 91 36 1 75
 2 40 60 49 24 40 23 35 80 60 61 27 82 38 66 81 95 79 96 23 73 19 5 43
 2 47 17 88 46 76 64 82 31 73 43 17 35 28 48 89 8 61 23 17 56 7 84 36
 95 60 34 9 4 5 74 59 6 89 84 98 25 50 38 2 3 43 64 96 47 79 12 82
 72 0 78 5]
```

```
OUTPUT VECTOR RESULT A + B
```

```
[70 54 76 154 182 194 141 96 77 52 100 162 126 112 151 156 111 117 152
 97 161 100 60 75 98 118 75 70 28 129 37 101 133 80 95 91 130 58 74 134
 177 145 115 76 84 36 44 54 91 144 68 141 53 80 156 164 121 151 74 35]
```

```
107 80 92 106 106 64 59 86 81 94 170 104 80 76 92 13 61 69 171 70 87
125 121 119 76 72 55 8 69 55 144 146 124 173 18 152 93 86 158 74]
```

There's more...

In this section, we have seen that the PyOpenCL execution model, like PyCUDA, involves a host processor that manages one or more heterogeneous devices. In particular, each PyOpenCL command is sent to the devices from the host in the form of source code that is defined through the kernel function.

The source code is then loaded into a program object for the reference architecture, the program is compiled into the reference architecture, and the kernel object that is relative to the program is created.

A kernel object can be executed in a variable number of workgroups, creating an n -dimensional computation matrix that allows it to effectively subdivide the workload for a problem in n -dimensions (1, 2, or 3) in each workgroup. In turn, they are composed of a number of work items that work in parallel.

Balancing the workload for each workgroup based on the parallel computing capability of a device is one of the critical parameters for achieving good application performance.

A wrong balancing of the workload, together with the specific characteristics of each device (such as transfer latency, throughput, and bandwidth), can lead to a substantial loss of performance or compromise the portability of the code when executed without considering any system of dynamic acquisition of information in terms of device calculation capacities.

However, the accurate use of these technologies allows us to reach high levels of performance by combining the results of the calculation of different computational units.

See also

More on PyOpenCL programming can be found at https://pydanny-event-notes.readthedocs.io/en/latest/PyConPL2012/async_via_pyopencl.html.

Element-wise expressions with PyOpenCL

The element-wise functionality allows us to evaluate kernels on complex expressions (which are made of more operands) into a single computational pass.

Getting started

The `ElementwiseKernel` (`context`, `argument`, `operation`, `name`, `optional_parameters`) method is implemented in PyOpenCL to handle element-wise expressions.

The main parameters are as follows:

- `context` is the device or the group of devices to which the element-wise operation will be executed.
- `argument` is a C-like argument list of all the parameters involved in the computation.
- `operation` is a string that represents the operation to perform on the argument list.
- `name` is the kernel's name that is associated with `ElementwiseKernel`.
- `optional_parameters` is not important in this recipe.

How to do it...

Here, we consider the task of adding two integer vectors again:

1. Start importing the relevant libraries:

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy as np
```

2. Define the context element (`context`) and the command queue (`queue`):

```
context = cl.create_some_context()
queue = cl.CommandQueue(context)
```

3. Here, we set the vector dimension and the space allocation for the input and output vectors:

```
vector_dim = 100
vector_a=cl_array.to_device(queue,np.random.randint(100,\
size=vector_dim))
vector_b = cl_array.to_device(queue,np.random.randint(100,\
size=vector_dim))
result_vector = cl_array.empty_like(vector_a)
```

4. We set `elementwiseSum` as the application of `ElementwiseKernel`, and then set it to a set of arguments that define the operations to be applied to the input vectors:

```
elementwiseSum = cl.elementwise.ElementwiseKernel(context, "int\n*a,\nint *b, int *c", "c[i] = a[i] + b[i]", "sum")
elementwiseSum(vector_a, vector_b, result_vector)
```

5. Finally, we print the result:

```
print ("PyOpenCL ELEMENTWISE SUM OF TWO VECTORS")
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print (vector_a)
print ("INPUT VECTOR B")
print (vector_b)
print ("OUTPUT VECTOR RESULT A + B ")
print (result_vector)
```

How it works...

In the first lines of the script, we import all the requested modules.

In order to initialize the context, we use the `cl.create_some_context()` method. This asks the user which context must be used to perform the calculation:

Choose platform:

```
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x1c0a25aecf0>
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x1c0a2608400>
```

Then, we need to instantiate the queue that will receive `ElementwiseKernel`:

```
queue = cl.CommandQueue(context)
```

Input and output vectors are instantiated. The input vectors, `vector_a` and `vector_b`, are integer vectors of random values obtained using the `random.randint` NumPy function. These vectors are then copied into the device by using the PyOpenCL statement:

```
cl.array_to_device(queue, array)
```

In `ElementwiseKernel`, an object is created:

```
elementwiseSum = cl.elementwise.ElementwiseKernel(context, \
    "int *a, int *b, int *c", "c[i] = a[i] + b[i]", "sum")
```



Note that all the arguments are in the form of a string formatted as a C argument list (they are all integers).

The operation is a C-like code snippet that carries out the operation, that is, the sum of the input vector elements.

The name of the function with which the kernel will be compiled is `sum`.

Finally, we call the `elementwiseSum` function with the arguments defined previously:

```
elementwiseSum(vector_a, vector_b, result_vector)
```

The example ends by printing the input vectors and the result obtained. The output looks like this:

```
(base) C:\>python elementwisePyopencl.py
```

```
Choose platform:
```

```
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x1c0a25aecf0>
```

```
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x1c0a2608400>
```

```
Choice [0]:1
```

```
Choose device(s):
```

```
[0] <pyopencl.Device 'Intel(R) HD Graphics 5500' on 'Intel(R) OpenCL' at 0x1c0a1640db0>
```

```
[1] <pyopencl.Device 'Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz' on 'Intel(R) OpenCL' at 0x1c0a15e53f0>
```

```
Choice, comma-separated [0]:0
```

```
PyOpenCL ELEMENTWISE SUM OF TWO VECTORS
```

```
VECTOR LENGTH = 100
```

```
INPUT VECTOR A
```

```
[24 64 73 37 40 4 41 85 19 90 32 51 6 89 98 56 97 53 34 91 82 89 97 2
 54 65 90 90 91 75 30 8 62 94 63 69 31 99 8 18 28 7 81 72 14 53 91 80
```

```

76 39 8 47 25 45 26 56 23 47 41 18 89 17 82 84 10 75 56 89 71 56 66 61
58 54 27 88 16 20 9 61 68 63 74 84 18 82 67 30 15 25 25 3 93 36 24 27
70 5 78 15]

```

INPUT VECTOR B

```

[49 18 69 43 51 72 37 50 79 34 97 49 51 29 89 81 33 7 47 93 70 52 63 90
99 95 58 33 41 70 84 87 20 83 74 43 78 34 94 47 89 4 30 36 34 56 32 31
56 22 50 52 68 98 52 80 14 98 43 60 20 49 15 38 74 89 99 29 96 65 89 41
72 53 89 31 34 64 0 47 87 70 98 86 41 25 34 10 44 36 54 52 54 86 33 38
25 49 75 53]

```

OUTPUT VECTOR RESULT A + B

```

[73 82 142 80 91 76 78 135 98 124 129 100 57 118 187 137 130 60 81 184
152 141 160 92 153 160 148 123 132 145 114 95 82 177 137 112 109 133
102 65 117 11 111 108 48 109 123 111 132 61 58 99 93 143 78 136 37 145
84 78 109 66 97 122 84 164 155 118 167 121 155 102 130 107 116 119 50
84 9 108 155 133 172 170 59 107 101 40 59 61 79 55 147 122 57 65
95 54 153 68]

```

There's more...

PyCUDA also has element-wise functionality:

```
ElementwiseKernel(arguments, operation, name, optional_parameters)
```

This feature has pretty much the same arguments as the function built for PyOpenCL, except for the context parameter. The same example this section, which is implemented through PyCUDA, has the following listing:

```

import pycuda.autotinit
import numpy
from pycuda.elementwise import ElementwiseKernel
import numpy.linalg as la

vector_dimension=100
input_vector_a = np.random.randint(100,size= vector_dimension)
input_vector_b = np.random.randint(100,size= vector_dimension)
output_vector_c = gpuarray.empty_like(input_vector_a)

elementwiseSum = ElementwiseKernel(" int *a, int * b, int *c",\
                                     "c[i] = a[i] + b[i]", " elementwiseSum ")
elementwiseSum(input_vector_a, input_vector_b,output_vector_c)

print ("PyCUDA ELEMENTWISE SUM OF TWO VECTORS")
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")

```

```
print (vector_a)
print ("INPUT VECTOR B")
print (vector_b)
print ("OUTPUT VECTOR RESULT A + B ")
print (result_vector)
```

See also

In the following link, you'll find interesting examples of PyOpenCL applications: <https://github.com/romanarranz/PyOpenCL>.

Evaluating PyOpenCL applications

In this section, we are doing a comparative test of performance between CPU and GPU by using the PyOpenCL library.

In fact, before studying the performance of the algorithms to be implemented, it is also important to understand the computational advantages offered by the computing platform you have.

Getting started

The specific characteristics of a computing system interfere with the computational time, and hence they represent an aspect of primary importance.

In the following example, we will perform a test in order to monitor performance on such a system:

- GPU: GeForce 840 M
- CPU: Intel Core i7 – 2.40 GHz
- RAM: 8 GB

How to do it...

In the following test, the calculation time of a mathematical operation, as the sum of two vectors with floating-point elements, will be evaluated and compared. To make the comparison, the same operation will be performed on two separate functions.

The first function is computed by the CPU only, while the second function is written by using the PyOpenCL library to use the GPU card. The test is performed on vectors with a size of 10,000 elements.

Here is the code:

1. Import the relevant libraries. Note the import of the `time` library to calculate the computation times, and the `linalg` library, which is a tool of linear algebra tools of the `numpy` library:

```
from time import time
import pyopencl as cl
import numpy as np
import deviceInfoPyopencl as device_info
import numpy.linalg as la
```

2. Then, we define the input vectors. They both contain 10000 random elements of floating-point numbers:

```
a = np.random.rand(10000).astype(np.float32)
b = np.random.rand(10000).astype(np.float32)
```

3. The following function computes the sum of the two vectors working on the CPU (host):

```
def test_cpu_vector_sum(a, b):
    c_cpu = np.empty_like(a)
    cpu_start_time = time()
    for i in range(10000):
        for j in range(10000):
            c_cpu[i] = a[i] + b[i]
    cpu_end_time = time()
    print("CPU Time: {0} s".format(cpu_end_time - cpu_start_time))
    return c_cpu
```

4. The following function computes the sum of the two vectors working on the GPU (device):

```
def test_gpu_vector_sum(a, b):
    platform = cl.get_platforms()[0]
    device = platform.get_devices()[0]
    context = cl.Context([device])
    queue = cl.CommandQueue(context, properties=\
cl.command_queue_properties.PROFILING_ENABLE)
```

5. Within the `test_gpu_vector_sum` function, we prepare the memory buffers to contain the input vectors and the output vector:

```
a_buffer = cl.Buffer(context, cl.mem_flags.READ_ONLY \
                       | cl.mem_flags.COPY_HOST_PTR, hostbuf=a)
b_buffer = cl.Buffer(context, cl.mem_flags.READ_ONLY \
                       | cl.mem_flags.COPY_HOST_PTR, hostbuf=b)
c_buffer = cl.Buffer(context, cl.mem_flags.WRITE_ONLY, b.nbytes)
```

6. Still, within the `test_gpu_vector_sum` function, we define the kernel that will computerize the sum of the two vectors on the device:

```
program = cl.Program(context, """
__kernel void sum(__global const float *a,\
                  __global const float *b,\
                  __global float *c){
    int i = get_global_id(0);
    int j;
    for(j = 0; j < 10000; j++){
        c[i] = a[i] + b[i];}
}""").build()
```

7. Then, we reset the `gpu_start_time` variable before starting the calculation. After this, we calculate the sum of two vectors and then we evaluate the calculation time:

```
gpu_start_time = time()
event = program.sum(queue, a.shape, None, a_buffer, b_buffer, \
                    c_buffer)
event.wait()
elapsed = 1e-9*(event.profile.end - event.profile.start)
print("GPU Kernel evaluation Time: {0} s".format(elapsed))
c_gpu = np.empty_like(a)
cl._enqueue_read_buffer(queue, c_buffer, c_gpu).wait()
gpu_end_time = time()
print("GPU Time: {0} s".format(gpu_end_time - gpu_start_time))
return c_gpu
```

8. Finally, we perform the test, recalling the two functions defined previously:

```
if __name__ == "__main__":
    device_info.print_device_info()
    cpu_result = test_cpu_vector_sum(a, b)
    gpu_result = test_gpu_vector_sum(a, b)
    assert (la.norm(cpu_result - gpu_result)) < 1e-5
```

How it works...

As explained previously, the test consists of executing the calculation task, both on the CPU via the `test_cpu_vector_sum` function, and then on the GPU via the `test_gpu_vector_sum` function.

Both functions report the execution time.

Regarding the testing function on the CPU, `test_cpu_vector_sum`, it consists of a double calculation loop on 10000 vector elements:

```
cpu_start_time = time()
for i in range(10000):
    for j in range(10000):
        c_cpu[i] = a[i] + b[i]
cpu_end_time = time()
```

The total CPU time is the difference between the following:

```
CPU Time = cpu_end_time - cpu_start_time
```

As for the `test_gpu_vector_sum` function, you can see the following by looking at the execution kernel:

```
__kernel void sum(__global const float *a,
                 __global const float *b,
                 __global float *c){
    int i=get_global_id(0);
    int j;
    for(j=0;j< 10000;j++){
        c[i]=a[i]+b[i];}
```

The sum of the two vectors is performed through a single calculation loop.

The result, as can be imagined, is a substantial reduction in the execution time for the `test_gpu_vector_sum` function:

```
(base) C:\>python testApplicationPyopencl.py
```

```
=====
OpenCL Platforms and Devices
=====
Platform - Name: NVIDIA CUDA
Platform - Vendor: NVIDIA Corporation
Platform - Version: OpenCL 1.2 CUDA 10.1.152
Platform - Profile: FULL_PROFILE
=====
```

```
Device - Name: GeForce 840M
Device - Type: GPU
Device - Max Clock Speed: 1124 Mhz
Device - Compute Units: 3
Device - Local Memory: 48 KB
Device - Constant Memory: 64 KB
Device - Global Memory: 2 GB
Device - Max Buffer/Image Size: 512 MB
Device - Max Work Group Size: 1024
=====
Platform - Name: Intel(R) OpenCL
Platform - Vendor: Intel(R) Corporation
Platform - Version: OpenCL 2.0
Platform - Profile: FULL_PROFILE
-----
Device - Name: Intel(R) HD Graphics 5500
Device - Type: GPU
Device - Max Clock Speed: 950 Mhz
Device - Compute Units: 24
Device - Local Memory: 64 KB
Device - Constant Memory: 64 KB
Device - Global Memory: 3 GB
Device - Max Buffer/Image Size: 808 MB
Device - Max Work Group Size: 256
-----
Device - Name: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Device - Type: CPU
Device - Max Clock Speed: 2400 Mhz
Device - Compute Units: 4
Device - Local Memory: 32 KB
Device - Constant Memory: 128 KB
Device - Global Memory: 8 GB
Device - Max Buffer/Image Size: 2026 MB
Device - Max Work Group Size: 8192

CPU Time: 39.505873918533325 s
GPU Kernel evaluation Time: 0.013606592 s
GPU Time: 0.019981861114501953 s
```

Even if the test is not computationally expansive, it provides useful indications of the potential of a GPU card.

There's more...

OpenCL is a standardized cross-platform API for developing applications that exploit parallel computing in heterogeneous systems. The similarities with CUDA are remarkable, including everything from the memory hierarchy to the direct correspondence between threads and work items.

Even at the programming level, there are many similar aspects and extensions with the same functionality.

However, OpenCL has a much more complex device management model due to its ability to support a wide variety of hardware. On the other hand, OpenCL is designed to have code portability between products from different manufacturers.

CUDA, thanks to its greater maturity and dedicated hardware, offers simplified device management and higher-level APIs that make it preferable, but only if you are dealing with specific architectures (that is, NVIDIA graphic cards).

The pros and cons of the CUDA and OpenCL libraries, as well as the PyCUDA and PyOpenCL libraries, are explained in the following sections.

Pros of OpenCL and PyOpenCL

The pros are as follows:

- They allow the use of heterogeneous systems with different types of microprocessors.
- The same code runs on different systems.

Cons of OpenCL and PyOpenCL

The cons are as follows:

- Complex device management
- APIs not fully stable

Pros of CUDA and PyCUDA

The pros are as follows:

- APIs with very high abstraction levels
- Extensions for many programming languages
- Huge documentation and a very large community

Cons of CUDA and PyCUDA

The cons are as follows:

- Supports only the latest NVIDIA GPUs as devices
- Reduces heterogeneity to CPUs and GPUs

See also

Andreas Klöckner has made a series of lectures on GPU programming with PyCuda and PyOpenCL available at <https://www.bu.edu/pasi/courses/gpu-programming-with-pyopencl-and-ptycuda/> and https://www.youtube.com/results?search_query=pyopenCL+and+ptycuda.

GPU programming with Numba

Numba is a Python compiler that provides CUDA-based APIs. It has been designed primarily for numerical computing tasks, just like the NumPy library. In particular, the numba library manages and processes the array data types provided by NumPy.

In fact, the exploitation of data parallelism, which is inherent in numerical computation involving arrays, is a natural choice for GPU accelerators.

The Numba compiler works by specifying the signature types (or decorators) for Python functions and enabling the compilation at runtime (this type of compilation is also called *Just In Time*).

The most important decorators are as follows:

- `jit`: This allows the developer to write CUDA-like functions. When encountered, the compiler translates the code under the decorator into the pseudo-assembly PTX language, so that it can be executed by the GPU.
- `autojit`: This annotates a function for a *deferred compilation* procedure, which means that the function with this signature is compiled exactly once.
- `vectorize`: This creates a so-called **NumPy Universal Function (ufunc)** that takes a function and executes it in parallel with vector arguments.
- `guvectorize`: This builds a so-called **NumPy Generalized Universal Function (gufunc)**. A `gufunc` object may operate on entire sub-arrays.

Getting ready

Numba (release 0.45) is compatible with Python 2.7 and 3.5 or later, as well as NumPy versions 1.7 to 1.16.

To install `numba`, it is recommended as per `pyopenc1` to use the Anaconda framework, so, from the Anaconda Prompt, just type the following:

```
(base) C:\> conda install numba
```

In addition, to use the full potential of `numba`, the `cuda-toolkit` library must be installed:

```
(base) C:\> conda install cuda-toolkit
```

After that, it's possible to verify whether the CUDA library and GPU are properly detected.

Open the Python interpreter from the Anaconda Prompt:

```
(base) C:\> python
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>
```

The first test entails checking whether the CUDA library (`cuda-toolkit`) is properly installed:

```
>>> import numba.cuda.api
>>> import numba.cuda.cudadrv.libs
>>> numba.cuda.cudadrv.libs.test()
```

The following output shows the quality of the installation, where all the checks returned a positive result:

```
Finding cublas from Conda environment
  located at C:\Users\Giancarlo\Anaconda3\Library\bin\cublas64_10.dll
  trying to open library... ok
Finding cusparse from Conda environment
  located at C:\Users\Giancarlo\Anaconda3\Library\bin\cusparse64_10.dll
  trying to open library... ok
Finding cufft from Conda environment
  located at C:\Users\Giancarlo\Anaconda3\Library\bin\cufft64_10.dll
  trying to open library... ok
Finding curand from Conda environment
  located at C:\Users\Giancarlo\Anaconda3\Library\bin\curand64_10.dll
  trying to open library... ok
Finding nvvm from Conda environment
  located at C:\Users\Giancarlo\Anaconda3\Library\bin\nvvm64_33_0.dll
  trying to open library... ok
Finding libdevice from Conda environment
  searching for compute_20... ok
  searching for compute_30... ok
  searching for compute_35... ok
  searching for compute_50... ok
True
```

In the second test, we verify the presence of a graphics card:

```
>>> numba.cuda.api.detect ()
```

The output shows the graphic card found and whether it is supported:

```
Found 1 CUDA devices
id 0 b'GeForce 840M' [SUPPORTED]
      compute capability: 5.0
      pci device id: 0
      pci bus id: 8

Summary:
  1/1 devices are supported
True
```

How to do it...

In this example, we provide a demonstration of the Numba compiler using the `@guvectorize` annotation.

The task to execute is matrix multiplication:

1. Import `guvectorize` from the `numba` library and the `numpy` module:

```
from numba import guvectorize
import numpy as np
```

2. Using the `@guvectorize` decorator, we define the `matmul` function, which will perform the matrix multiplication task:

```
@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
             '(m,n), (n,p)->(m,p) ')
def matmul(A, B, C):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
```

3. The input matrices are 10×10 in size, while the elements are integers:

```
dim = 10
A = np.random.randint(dim, size=(dim, dim))
B = np.random.randint(dim, size=(dim, dim))
```

4. Finally, we call the `matmul` function on the previously defined input matrices:

```
C = matmul(A, B)
```

5. We print the input matrices and the resulting matrix:

```
print("INPUT MATRIX A")
print(":\n%s" % A)
print("INPUT MATRIX B")
print(":\n%s" % B)
print("RESULT MATRIX C = A*B")
print(":\n%s" % C)
```

How it works...

The `@guvectorize` decorator works on array arguments, taking four arguments in order to specify the `gufunc` signature:

- The first three arguments specify the types of data to be managed and arrays of integers: `void(int64[:, :], int64[:, :], int64[:, :])`.
- The last argument of `@guvectorize` specifies how to manipulate the matrix dimensions: `(m,n) , (n,p) -> (m,p)`.

Then, the matrix multiplication operation is defined, where A and B are the input matrices and C is the output matrix: $A(m,n) * B(n,p) = C(m,p)$, where *m*, *n*, and *p* are the matrix dimensions.

The matrix product is performed through three `for` loops along with the matrix indices:

```
for i in range(m):
    for j in range(p):
        C[i, j] = 0
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
```

The `randint` NumPy function is used here to build the input matrices of 10×10 dimensions:

```
dim = 10
A = np.random.randint(dim, size=(dim, dim))
B = np.random.randint(dim, size=(dim, dim))
```

Finally, the `matmul` function is called with these matrices as arguments, and the resultant C matrix is printed out:

```
C = matmul(A, B)
print("RESULT MATRIX C = A*B")
print(":\n%s" % C)
```

To execute this example, type the following:

```
(base) C:\>python matMulNumba.py
```

The result shows the two matrices given as input and the matrix resulting from their product:

INPUT MATRIX A

:

```
[8 7 1 3 1 0 4 9 2 2]
[3 6 2 7 7 9 8 4 4 9]
[8 9 9 9 1 1 1 1 8 0]
[0 5 0 7 1 3 2 0 7 3]
[4 2 6 4 1 2 9 1 0 5]
[3 0 6 5 1 0 4 3 7 4]
[0 9 7 2 1 4 3 3 7 3]
[1 7 2 7 1 8 0 3 4 1]
[5 1 5 0 7 7 2 3 0 9]
[4 6 3 6 0 3 3 4 1 2]]
```

INPUT MATRIX B

:

```
[2 1 4 6 6 4 9 9 5 2]
[8 6 7 6 5 9 2 1 0 9]
[4 1 2 4 8 2 9 5 1 4]
[9 9 1 5 0 5 1 1 7 1]
[8 7 8 3 9 1 4 3 1 5]
[7 2 5 8 3 5 8 5 6 2]
[5 3 1 4 3 7 2 9 9 5]
[8 7 9 3 4 1 7 8 0 4]
[3 0 4 2 3 8 8 8 6 2]
[8 6 7 1 8 3 0 8 8 9]]
```

RESULT MATRIX C = A*B

:

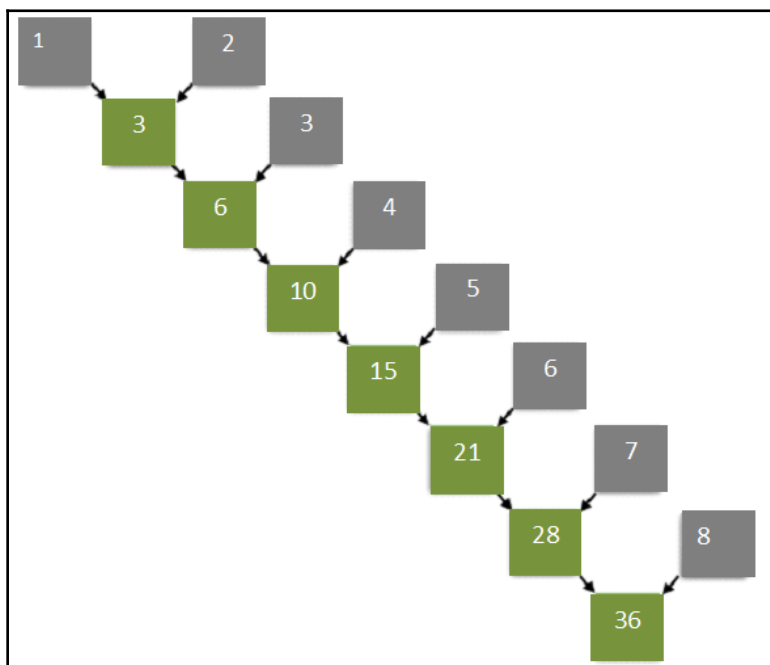
```
[225 172 201 161 170 172 189 230 127 169]
[400 277 289 251 278 276 240 324 295 273]
[257 171 177 217 208 254 265 224 176 174]
[187 130 116 117 94 175 105 128 152 114]
[199 133 117 143 168 156 143 214 188 157]
[180 118 124 113 152 149 175 213 167 122]
[238 142 186 165 188 215 202 200 139 192]
[237 158 162 176 122 185 169 140 137 130]
[249 160 220 159 249 125 201 241 169 191]
[209 152 142 154 131 160 147 161 132 137]]
```

There's more...

Writing an algorithm for a reduction operation using PyCUDA can be quite complex. For this purpose, Numba provides the `@reduce` decorator for converting simple binary operations into *reduction kernels*.

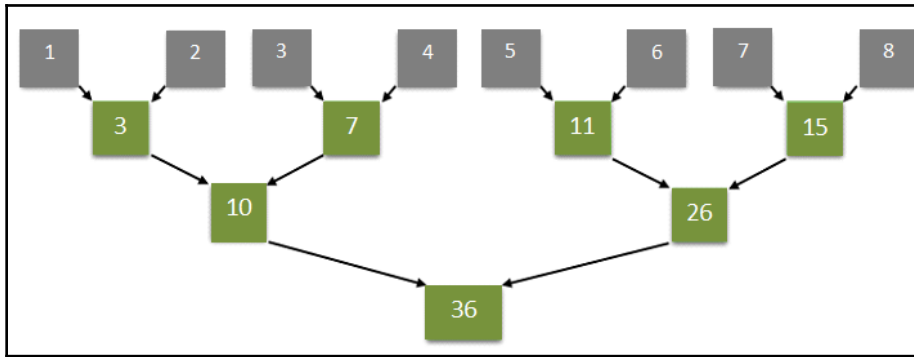
Reduction operations reduce a set of values to a single value. A typical example of a reduction operation is to calculate the sum of all the elements of an array. As an example, consider the following array of elements: 1, 2, 3, 4, 5, 6, 7, 8.

The sequential algorithm operates in the way shown in the diagram, that is, adding the elements of the array one after the other:



Sequential sum

A parallel algorithm operates according to the following schema:



Parallel sum

It is clear that the latter has the advantage of shorter execution time.

By using Numba and the `@reduce` decorator, we can write an algorithm, in a few lines of code, for the parallel sum on an array of integers ranging from 1 to 10,000:

```
import numpy
from numba import cuda

@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(10000, dtype=numpy.int64)) + 1
print(A)
got = sum_reduce(A)
print(got)
```

The previous example can be performed by typing the following command:

```
(base) C:\>python reduceNumba.py
```

The following result is provided:

```
vector to reduce = [ 1 2 3 ... 9998 9999 10000]
result = 50005000
```

See also

In the following repository, you can find many examples of Numba: <https://github.com/numba/numba-examples>. An interesting introduction to Numba and CUDA programming can be found at <https://nyu-cds.github.io/python-numba/05-cuda/>.

9 Python Debugging and Testing

This last chapter will introduce two important software engineering topics—debugging and testing—that are important steps in the software development process.

The first part of the chapter is focused on code debugging. A bug is a mistake in a program and can cause different problems that may be more or less serious depending on the situation. To encourage programmers to search for bugs, special software tools are used, called **debuggers**; using these software tools, we have the ability to find errors or malfunctions within a program by taking advantage of specific debugging functions, an activity that exists precisely for identifying the portion of software affected by a bug.

In the second part, the main topic is *software testing*: it is a process used to identify deficiencies of *correctness*, *completeness*, and *reliability* in a software product that is being developed.

In this context, we will, therefore, examine the three most important Python tools for debugging code in action. These are `winspdb-reborn`, which involves debugging with a visualization tool; `pdb`, the debugger from the Python standard library; and `rpdb`, where `r` stands for remote, meaning that it is code debugging from a remote machine.

Regarding software testing, we will examine the following tools: `unittest` and `nose`.

These are frameworks for developing unit tests, whereby the unit is the minimum component of a program within an independent operation.

In this chapter, we will cover the following topics:

- What is debugging?
- What is software testing?
- Debugging using Winpdb Reborn
- Interacting with `pdb`
- Implementing `rpdb` for debugging
- Dealing with `unittest`
- Application testing using `nose`

What is debugging?

The term *debugging* indicates the activity of identifying the portion of code in which one or more errors (bugs) are detected in software following its use.

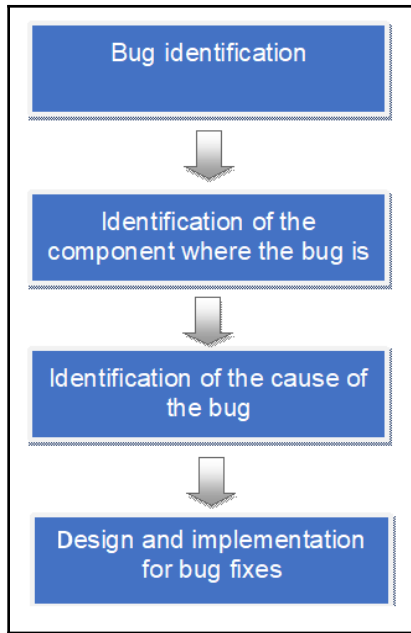
The error can be localized during the testing phase of the program; that is when it is still in the development phase and is not yet ready to be used by the end-user, or during the use of the program by the latter. After finding the error, the debugging phase ensues and identifies the software part in which the error lies, which is sometimes very complex.

Nowadays, this activity is supported by specific applications and debuggers, which show the execution to the programmer using step-by-step software instructions, allowing the viewing and analysis of the inputs and outputs of the program itself at the same time.

Before these tools were available for the activity of identifying and correcting errors (and even now, in the absence of them), the simplest (but also least effective) techniques for code inspection were printing a file or printing the instructions on the screen that the program was executing.

Debugging is one of the most important operations for the development of a program. It is often extremely difficult due to the complexity of the software that is being developed. It is even delicate due to the risk of introducing new errors or behaviors that are not in line with those desired in the attempt to correct those for which the activity was undertaken.

Although the task of perfecting software using debugging is unique every time and constitutes a story in itself, some general principles are always applicable. In particular, in the context of software applications, it is possible to recognize the following four *debugging phases*, summarized in the following diagram:



Debugging phases



Of course, Python offers the developer numerous debugging tools (see <https://wiki.python.org/moin/PythonDebuggingTools> for a list of Python debuggers). In this chapter, we will consider Winpdb, Reborn, rpdb, and pdb.

What is software testing?

As mentioned in the introduction of this chapter, software testing is a process used to identify deficiencies of correctness, completeness, and reliability in a software product that is being developed.

With this activity, we, therefore, want to ensure the quality of the product by searching for defects, or a sequence of instructions and procedures that, when executed with particular input data and in particular operating environments, generate malfunctions. A malfunction is a behavior of the software that is not expected by the user; therefore, it is different from the specifications and from the implicit or explicit requirements defined for such applications.

The purpose of testing is, therefore, to detect defects through malfunctions, so as to minimize the probability of such malfunctions occurring in the normal use of the software product. Testing cannot establish that a product functions correctly under all possible conditions of execution, but it can highlight defects under specific conditions.

In fact, given the impossibility of testing all the input combinations and the possible software and hardware environments in which the application may be operating, the probability of malfunctions cannot be reduced to zero, but it must be reduced to a minimum in order to be acceptable to the user.

A particular type of software testing is the unit test (which we will learn about in this chapter), the purpose of which is to isolate each part of a program and show its correctness and completeness in the implementation. It also promptly brings out any defects so that they can be corrected easily before integration.

Furthermore, the unit test lowers the costs—in terms of time and resources—of identifying and correcting defects, compared to achieving the same result by performing tests on the entire application.

Debugging using Winpdb Reborn

Winpdb Reborn is one of the most important and well-known Python debuggers. The major strength of this debugger is managing the debugging of thread-based code.



Winpdb Reborn is based on the RPDB2 debugger, while Winpdb is the GUI frontend to RPDB2 (see: <https://github.com/bluebird75/winpdb/blob/master/rpdb2.py>).

Getting ready

The most commonly used way to install Winpdb Reborn (*release 2.0.0 dev5*) is via `pip`, so from your console, you need to type the following:

```
C:\>pip install winpdb-reborn
```

Also, if you have not already installed wxPython in your Python distribution, then you need to do so. wxPython is a cross-platform GUI toolkit for the Python language.



For Python Version 2.x, please refer to <https://sourceforge.net/projects/wxpython/files/wxPython/>. For Python Version 3.x, wxPython is automatically installed as a dependency via `pip`.

In the next section, we will examine the main features and the graphical interface of Winpdb Reborn through a simple example of its use.

How to do it...

Suppose we want to analyze the following Python application, which uses the `threading` library. An example that is very similar to the following example is already described in the *How to define a thread subclass* section of Chapter 2, *Thread-Based Parallelism*. In the following example, we use the `MyThreadClass` class to create and manage the execution of three threads. Here is the entire code to debug:

```
import time
import os
from random import randint
from threading import Thread

class MyThreadClass (Thread):
    def __init__(self, name, duration):
        Thread.__init__(self)
        self.name = name
        self.duration = duration
    def run(self):
```

```
        print ("---> " + self.name + \
              " running, belonging to process ID "\
              + str(os.getpid()) + "\n")
        time.sleep(self.duration)
        print ("---> " + self.name + " over\n")
def main():
    start_time = time.time()
    # Thread Creation
    thread1 = MyThreadClass("Thread#1 ", randint(1,10))
    thread2 = MyThreadClass("Thread#2 ", randint(1,10))
    thread3 = MyThreadClass("Thread#3 ", randint(1,10))

    # Thread Running
    thread1.start()
    thread2.start()
    thread3.start()

    # Thread joining
    thread1.join()
    thread2.join()
    thread3.join()

    # End
    print("End")

    #Execution Time
    print("--- %s seconds ---" % (time.time() - start_time))

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

Let's have a look at the following steps:

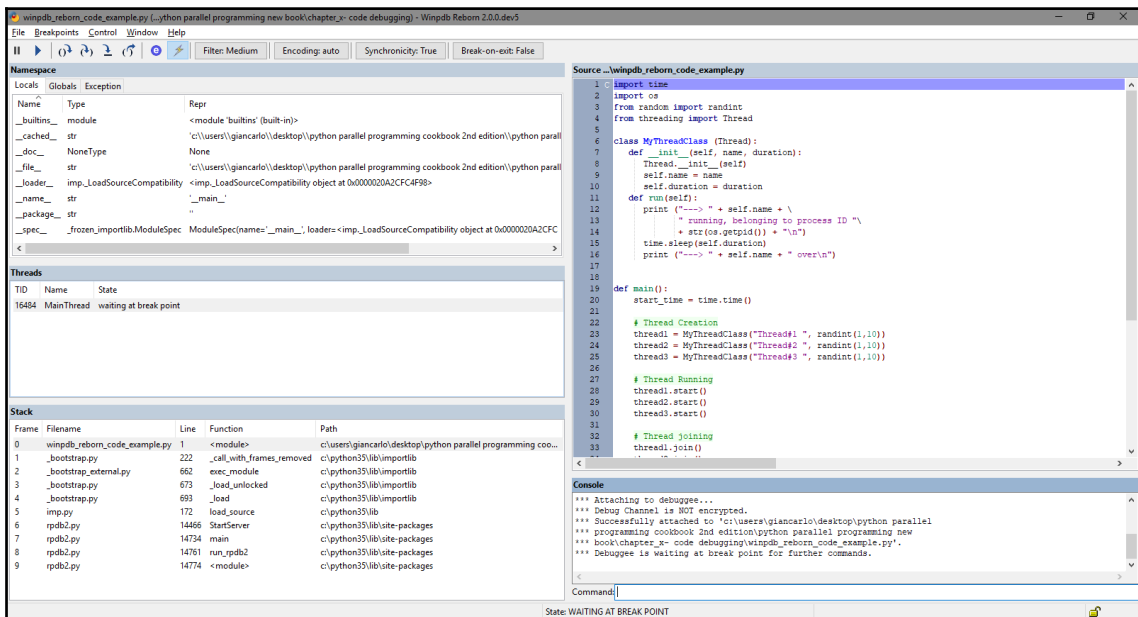
1. Open your console and type in the name of the folder containing the sample file, `winpdb_reborn_code_example.py`:

```
python -m winpdb .\winpdb_reborn_code_example.py
```



This works on macOS as well, but you have to use a framework build of Python. If you are using Winpdb Reborn with Anaconda, simply use `pythonw` instead of `python` to launch a Winpdb Reborn session.

2. If the installation was successful, then the Winpdb Reborn GUI should open:



Winpdb Reborn GUI

3. As you can see in the following screenshot, we have inserted two breakpoints (using the **Breakpoints** menu), in both line 12 and line 23 (highlighted in red):

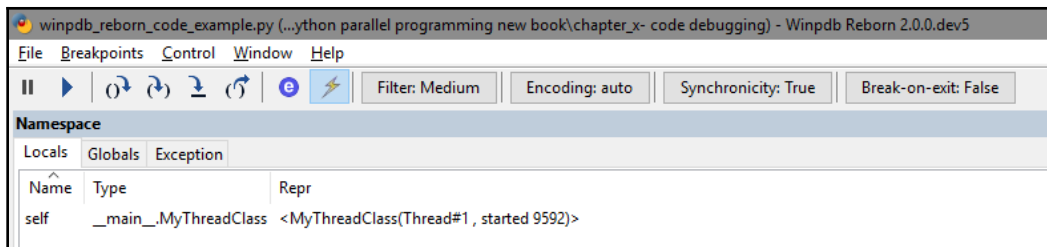
```
Source ...\\winpdb_reborn_code_example.py
1  import time
2  import os
3  from random import randint
4  from threading import Thread
5
6  class MyThreadClass (Thread):
7      def __init__(self, name, duration):
8          Thread.__init__(self)
9          self.name = name
10         self.duration = duration
11     def run(self):
12         print ("Thread " + self.name + \
13             " running, belonging to process ID " +
14             + str(os.getpid()) + "\n")
15         time.sleep(self.duration)
16         print ("---> " + self.name + " over\n")
17
18
19 def main():
20     start_time = time.time()
21
22     # Thread Creation
23     thread1 = MyThreadClass("Thread#1 ", randint(1,10))
24     thread2 = MyThreadClass("Thread#2 ", randint(1,10))
25     thread3 = MyThreadClass("Thread#3 ", randint(1,10))
26
```

Code breakpoints



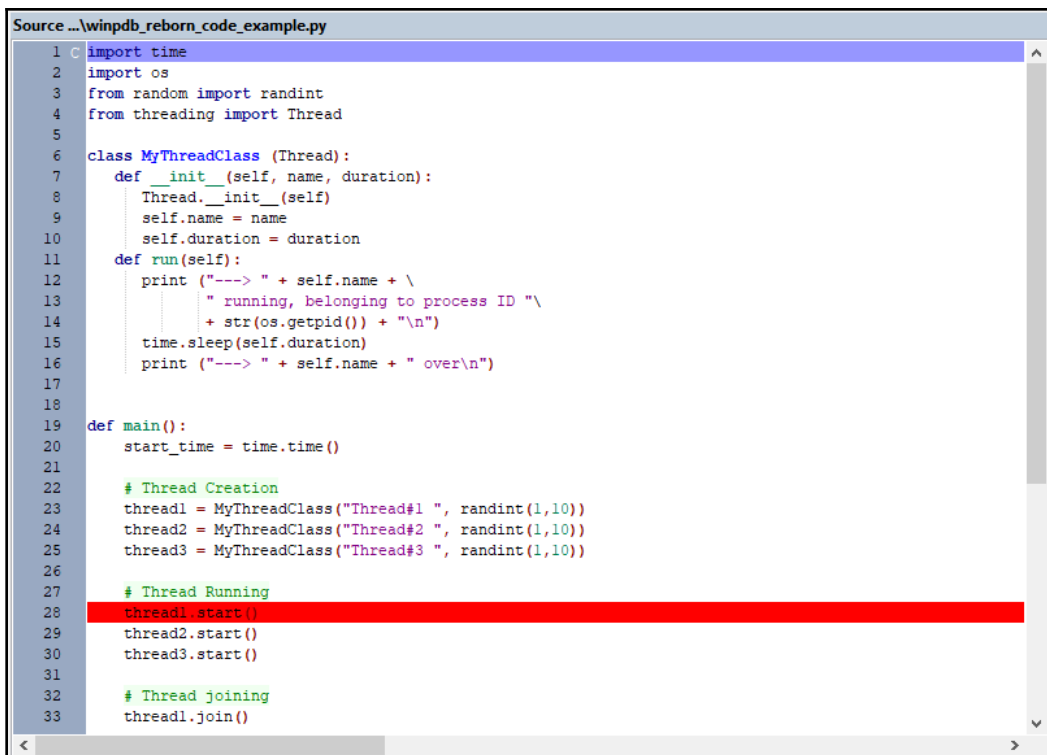
To learn about what a breakpoint is, move on to the *There's more...* section of this recipe.

4. Remaining in the **Source** window, we place the mouse on line 23, where we have inserted the second breakpoint, and press the *F8* key, and then the *F5* key. The breakpoint allows the code to be executed up to the selected line. As you can see, **Namespace** indicates that we are considering the instance of the `MyThreadClass` class, with `thread#1` as an argument:



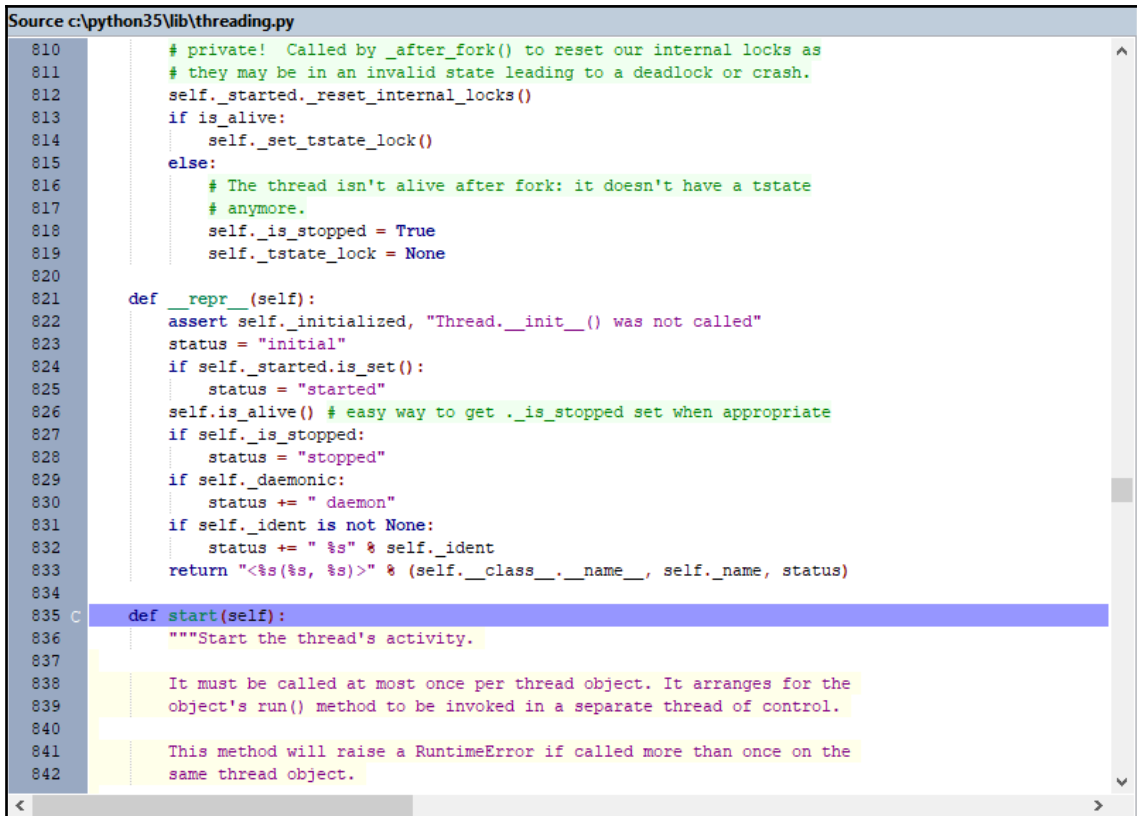
Namespace

- Another fundamental feature of the debugger is the **Step Into** capability, which is the ability to inspect not only the code being debugged but also the library functions and the subroutines called for execution.
- Before you start to delete the previous breakpoints (**Menu | Breakpoints | Clear All**), insert the new breakpoint on line 28:



Line 28 breakpoint

7. Finally, press the *F5* key and the application will be executed up to the breakpoint of line 28.
8. Then, press *F7*. Here, the source window no longer shows our sample code, but rather, the `threading` library we are using (see the next screenshot).
9. Therefore, the **Breakpoints** functionality, together with that of **Step Into**, not only allow the debugging of the code in question but also allow the inspection of all the library functions and any other subroutines used:



```
Source c:\python35\lib\threading.py
810     # private! Called by _after_fork() to reset our internal locks as
811     # they may be in an invalid state leading to a deadlock or crash.
812     self._started._reset_internal_locks()
813     if is_alive:
814         self._set_tstate_lock()
815     else:
816         # The thread isn't alive after fork: it doesn't have a tstate
817         # anymore.
818         self._is_stopped = True
819         self._tstate_lock = None
820
821     def __repr__(self):
822         assert self._initialized, "Thread.__init__() was not called"
823         status = "initial"
824         if self._started.is_set():
825             status = "started"
826         self.is_alive() # easy way to get ._is_stopped set when appropriate
827         if self._is_stopped:
828             status = "stopped"
829         if self._daemon:
830             status += " daemon"
831         if self._ident is not None:
832             status += " %s" % self._ident
833         return "<%s(%s, %s)>" % (self.__class__.__name__, self._name, status)
834
835     def start(self):
836         """Start the thread's activity.
837
838         It must be called at most once per thread object. It arranges for the
839         object's run() method to be invoked in a separate thread of control.
840
841         This method will raise a RuntimeError if called more than once on the
842         same thread object.
```

Line 28 Source window after executing Step Into

How it works...

In this first example, we have become familiar with the Winpdb Reborn tool. This debugging environment (like every environment in general) allows you to stop program execution at precise points, inspect the execution stack, the contents of the variables, the status of the objects created, and much more.

To use Winpdb Reborn, just take a note of the following basic steps:

1. Set some breakpoints in the source code (the **Source** window).
2. Inspect the functions through the **Step Into** function.
3. View the status of the variables (the **Namespace** window) and the execution stack (the **Stack** window).

The breakpoints are set by simply double-clicking the desired line with the left mouse button (you will see the selected line underlined in red). As a general warning, it is inadvisable to have multiple commands on the same line; otherwise, it will not be possible to associate breakpoints with some of them.

When you use the right mouse button, you can selectively *disable breakpoints* without removing them (the red highlighting will disappear). To remove all the breakpoints instead, use the **Clear All** command, which is present in the **Breakpoints** menu, as mentioned previously.

When the first breakpoint is reached, it is good to keep an eye on the following views in the point of the program that is being analyzed:

- The **Stack** view shows the contents of the execution stack, where all the instances of various methods that are currently suspended appear. Typically, the one at the bottom of the stack is the main method and the one at the top of the stack is the method containing the breakpoint that has been reached.
- The **Namespace** view shows the local variables of the method and allows you to inspect the values. If the variables refer to objects, then it is possible to find out the unique identifier of the object and inspect its status.

In general, the execution of a program can be managed with different modes associated with the icon (or the *Fx* keys) present on the Winpdb Reborn command bar.

Finally, we'll point out the following important execution methods:

- **Step Into** (*F7* key): This resumes the execution of the program one line at a time, and invocations of library methods or subroutines.
- **Return** (*F12* key): This allows you to resume execution at the exact point where the **Step Into** function was activated.
- **Next** (*F6* key): This resumes the execution of the program one line at a time without stopping in any methods invoked.
- **Run to Line** (*F8* key) This runs the program until it stops (waiting for new commands) at the indicated line.

There's more...

As you saw in the Winpdb Reborn GUI screenshot, the GUI is divided into five main windows:

- **Namespace:** In this window, the names of entities are displayed, which are various variables and identifiers that are defined by the program and used in the source file.
- **Threads:** The current thread of the execution is shown, and it is characterized by the **TID** (short for **Thread IDentification**) fields, the name of the thread, and the thread status.
- **Stack:** This is where the execution stack of the program to be analyzed is shown. Stacks are also known as **Last In, First Out (LIFO)** data structures, as the last element inserted is the first to be removed. When a program calls a function, the called function must know how to return the calling control, so the return address of the calling function is entered into the program execution stack. The program execution stack also contains the memory for the local variables used at each invocation of the function.
- **Console:** This is a command-line interface, thus allowing a textual interaction between the user and Winpdb Reborn.
- **Source:** This window shows the source code to debug. By scrolling along the lines of code, it is also possible to insert the breakpoints by pressing *F9* once you are on the line of code of interest.

The breakpoint is a very basic debugging tool. In fact, it allows you to run a program, but with the possibility of interrupting it at the desired point or when certain conditions occur, in order to acquire information on a running program.

There are multiple debugging strategies. Here, we list some of them:

- **Reproduce the error:** Identify the input data that caused it.
- **Simplify the error:** Identify the simplest possible data that caused it.
- **Divide and rule:** Perform the main proceeding in step-over mode until the anomaly occurs. The method that caused it is the last performed before it was possible to find the problem, so we can re-debug by doing step-in into that particular invocation, and proceeding again with step-over following the method's instructions.
- **Proceed consciously:** During debugging, you constantly compare the current values of the variables with the ones you would expect.
- **Check all the details:** Don't overlook the details while debugging. It is best to make a note if you notice any discrepancies in the source code.
- **Correct the errors:** Correct the error only if you are sure you have understood the problem well.

See also

A good Winpdb Reborn tutorial can be found
at <http://heather.cs.ucdavis.edu/~matloff/winpdb.html#usewin>.

Interacting with pdb

`pdb` is a Python module for performing interactive debugging.

The main features of `pdb` are as follows:

- The use of breakpoints
- Interactive processing of the source code line by line
- Stack frame analysis

The debugger is implemented through the `pdb` class. For this reason, it can be easily extended with new features.

Getting ready

No installation of `pdb` is required because it is part of the Python standard library. It can be launched with the following main use pattern:

- Interacting with the command line
- Using the Python interpreter
- Inserting a directive (that is, a `pdb` statement) in the code to debug

Interacting with the command line

The simplest method is simply passing the name of your program as input. For example, for the `pdb_test.py` program, this is as follows:

```
class Pdb_test(object):
    def __init__(self, parameter):
        self.counter = parameter

    def go(self):
        for j in range(self.counter):
            print ("--->", j)
        return

if __name__ == '__main__':
    Pdb_test(10).go()
```

By executing from the command line, `pdb` loads the source file to be analyzed and stops its execution at the first statement found. In this case, the debug stops at line 1 (that is, at the definition of the `Pdb_test` class):

```
python -m pdb pdb_test.py
> .../pdb_test.py(1) <module>()
-> class Pdb_test(object):
(Pdb)
```

Using the Python interpreter

The `pdb` module can be used in interactive mode by using the `run()` command:

```
>>> import pdb_test
>>> import pdb
>>> pdb.run('pdb_test.Pdb_test(10).go()')
> <string>(1) <module>()
(Pdb)
```

In this case, the `run()` statement is from the debugger and will stop the execution before evaluating the first expression.

Inserting a directive in the code to debug

For a long-running process, where the problem occurs much later in the program execution, it would be much more convenient to start the debugger within the program using the `set_trace()` directive:

```
import pdb

class Pdb_test(object):
    def __init__(self, parameter):
        self.counter = parameter
    def go(self):
        for j in range(self.counter):
            pdb.set_trace()
            print ("--->", j)
        return

if __name__ == '__main__':
    Pdb_test(10).go()
```

`set_trace()` can be called at any point in the program to debug. For example, it can be called based on conditions, exception handlers, or a specific branch of control instructions.

In this case, the output is as follows:

```
-> print ("--->", j)
(Pdb)
```

The code run stops, exactly after the `pdb.set_trace()` statement completes.

How to do it...

To interact with `pdb`, you need to use its language, which allows you to move around the code, examine and modify the values of the variables, insert breakpoints, or move through stack calls:

1. Use the `where` command (or alternatively, the compact form, `w`) to view which line of code is running and the call stack. In this case, this is on line 17 in the `go()` method of the `pdb_test.py` module:

```
> python -m pdb pdb_test.py
-> class Pdb_test(object):
(Pdb) where
c:\python35\lib\bdb.py(431)run()
-> exec(cmd, globals, locals)
    <string>(1)<module>()
(Pdb)
```

2. Inspect the lines of code near the current location (indicated by an arrow) by using `list`. In the default mode, 11 rows are listed around the current one (five before and five after):

```
(Pdb) list
1 -> class Pdb_test(object):
2   def __init__(self, parameter):
3     self.counter = parameter
4
5   def go(self):
6     for j in range(self.counter):
7       print ("--->", j)
8     return
9
10  if __name__ == '__main__':
11    Pdb_test(10).go()
```

3. If `list` receives two parameters, then they are interpreted as the first and last lines to be displayed:

```
(Pdb) list 3,9
3 self.counter = parameter
4
5 def go(self):
6   for j in range(self.counter):
7     print ("--->", j)
8   return
9
```

4. Use up (or u) to move to older frames on the stack and down (or d) to move to more recent stack frames:

```
(Pdb) up
> <string>(1) <module>()
(Pdb) up
> c:\python35\lib\bdb.py(431)run()
-> exec(cmd, globals, locals)
(Pdb) down
> <string>(1) <module>()
(Pdb) down
> ....\pdb_test.py(1) <module>()
-> class Pdb_test(object):
(Pdb)
```

How it works...

The debugging activity is carried out following the flow of the running program (tracing). In each line of code, the coder displays the operations performed by the instructions in real time and the values recorded in the variables. In this way, the developer can check that everything is working properly or identify the cause of a malfunction.

Each programming language has its own debugger. However, there is no valid debugger for all programming languages because each language has its own syntax and grammar. The debugger executes the step-by-step source code. Therefore, the debugger must know the rules of the language, like the compiler.

There's more...

The most useful `pdb` commands, along with their short forms, to keep in mind while working with the Python debugger are as follows:

Command	Action
args	Prints the argument list of the current function
break	Creates a breakpoint (requires parameters)
continue	Continues program execution
help	Lists the commands (or help) for a command (as a parameter)
jump	Sets the next line to be executed
list	Prints the source code around the current line

<code>next</code>	Continues execution until the next line in the current function is reached or returns
<code>step</code>	Executes the current line, stopping at the first possible occasion
<code>pp</code>	Pretty-prints the value of the expression
<code>quit</code> or <code>exit</code>	Aborts from <code>pdb</code>
<code>return</code>	Continues execution until the current function returns

See also

You can find out more about `pdb` by watching this interesting video tutorial: <https://www.youtube.com/watch?v=bZZTeKPRSLQ>.

Implementing `rpdb` for debugging

In some cases, it is appropriate to debug code in a remote location; that is, a location that doesn't reside on the same machine in which we run the debugger. For this purpose, `rpdb` was developed. This is a wrapper on `pdb` that uses a TCP socket to communicate with the world outside.

Getting ready

The installation of `rpdb` first requires the main step of using `pip`. For Windows OS, just type the following:

```
C:\>pip install rpdb
```

Then, you need to be sure that you have a working **telnet** client enabled on your machine. In Windows 10, if you open Command Prompt and type `telnet`, then the OS will respond with an error as it is not present by default in the installation.

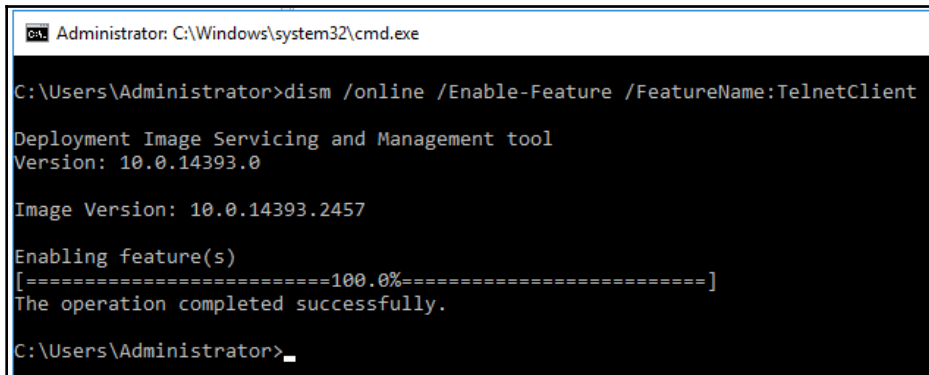
Let's see how to install it with a few simple steps:

1. Open Command Prompt in administrator mode.
2. Click on the Cortana button and type `cmd`.
3. In the list that appears, right-click on the Command Prompt item and select **Run as Administrator**.

4. Then, when running Command Prompt as an administrator, type the following command:

```
dism /online /Enable-Feature /FeatureName:TelnetClient
```

5. Wait a few minutes until the installation finishes. If the process is successful, then you will see this:



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\Administrator>dism /online /Enable-Feature /FeatureName:TelnetClient

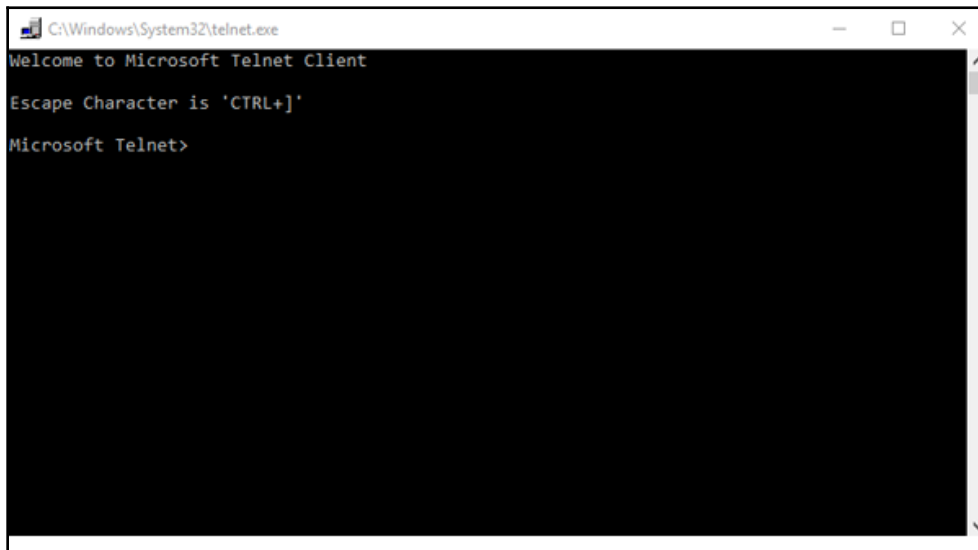
Deployment Image Servicing and Management tool
Version: 10.0.14393.0

Image Version: 10.0.14393.2457

Enabling feature(s)
[=====100.0%=====]
The operation completed successfully.

C:\Users\Administrator>_
```

6. Now, you can use telnet directly from the prompt. By typing `telnet`, the following window should appear:



```
C:\Windows\System32\telnet.exe

Welcome to Microsoft Telnet Client

Escape Character is 'CTRL+]'

Microsoft Telnet>
```

In the following example, let's see how to run a remote debug with `rpdb`.

How to do it...

Let's perform the following steps:

1. Consider the following sample code:

```
import threading

def my_func(thread_number):
    return print('my_func called by thread N°
                {}'.format(thread_number))

def main():
    threads = []
    for i in range(10):
        t = threading.Thread(target=my_func, args=(i,))
        threads.append(t)
        t.start()
    t.join()

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

2. To use `rpdb`, you need to insert the following lines of code (just after the `import threading` statement). In fact, these three lines of code enable the use of `rpdb` via a remote client on port 4444 with an IP address of 127.0.0.1:

```
import rpdb
debugger = rpdb.Rpdb(port=4444)
rpdb.Rpdb().set_trace()
```

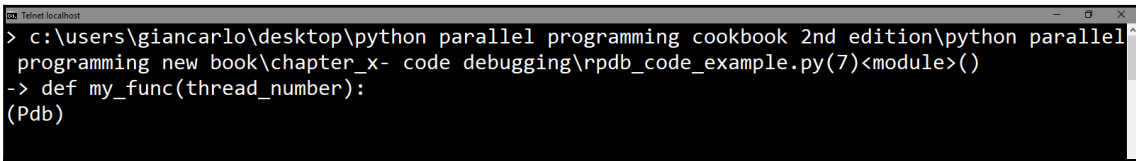
3. If you run the sample code after inserting these three lines of code that enable the use of `rpdb`, then you should see the following message on Python Command Prompt:

```
pdb is running on 127.0.0.1:4444
```

4. Then, you can switch to debug the sample code remotely by making the following telnet connection:

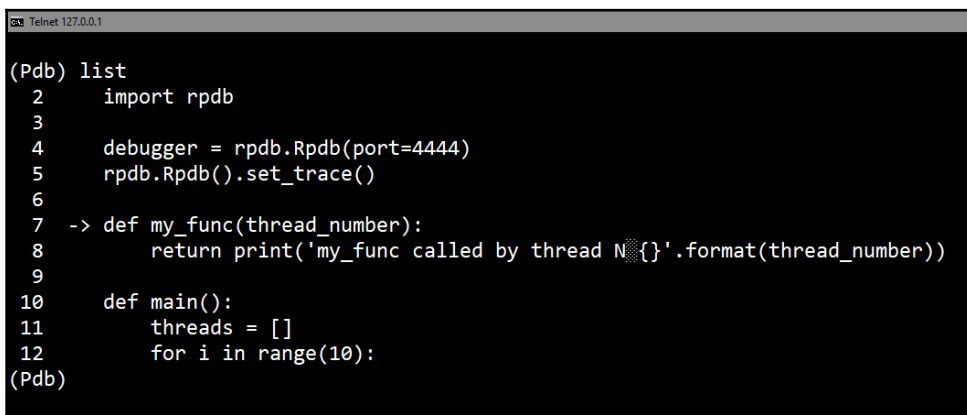
```
telnet localhost 4444
```

5. The following window should open:



```
Telnet localhost
> c:\users\giancarlo\desktop\python parallel programming cookbook 2nd edition\python parallel
programming new book\chapter_x- code debugging\rpdb_code_example.py(7)<module>()
-> def my_func(thread_number):
(Pdb)
```

6. In the sample code, note the arrow in line 7. The code is not running, it is just waiting for an instruction to execute:



```
Telnet 127.0.0.1
(Pdb) list
2     import rpdb
3
4     debugger = rpdb.Rpdb(port=4444)
5     rpdb.Rpdb().set_trace()
6
7 -> def my_func(thread_number):
8     return print('my_func called by thread N{}'.format(thread_number))
9
10    def main():
11        threads = []
12        for i in range(10):
(Pdb)
```

7. For example, here, we execute the code and type the next statement repeatedly:

```
(Pdb) next
> c:\users\giancarlo\desktop\python parallel programming cookbook
2nd edition\python parallel programming new book\chapter_x- code
debugging\rpdb_code_example.py(10)<module>()
-> def main():
(Pdb) next
> c:\users\giancarlo\desktop\python parallel programming cookbook
2nd edition\python parallel programming new book\chapter_x- code
debugging\rpdb_code_example.py(18)<module>()
-> if __name__ == "__main__":
(Pdb) next
> c:\users\giancarlo\desktop\python parallel programming cookbook
2nd edition\python parallel programming new book\chapter_x- code
debugging\rpdb_code_example.py(20)<module>()
-> main()
(Pdb) next
my_func called by thread N 0
my_func called by thread N 1
```

```
my_func called by thread N 2
my_func called by thread N 3
my_func called by thread N 4
my_func called by thread N 5
my_func called by thread N 6
my_func called by thread N 7
my_func called by thread N 8
my_func called by thread N 9
--Return--
> c:\users\giancarlo\desktop\python parallel programming cookbook
2nd edition\python parallel programming new book\chapter_x- code
debugging\rpdb_code_example.py (20) <module> () ->None
-> main()
(Pdb)
```

Once the program is finished, you can still run a new debug section. Now, let's see how `rpdb` works in the next section.

How it works...

In this section, we'll see how to simply move through the code by using the `next` statement, which continues execution until the next line in the current function is reached or returned.

To use `rpdb`, follow these steps:

1. Import the relevant `rpdb` library:

```
import rpdb
```

2. Set the `debugger` parameter, which specifies the telnet port to connect to in order to run the debugger:

```
debugger = rpdb.Rpdb(port=4444)
```

3. Call the `set_trace()` directive, which makes it possible to enter into debugging mode:

```
rpdb.Rpdb().set_trace()
```

In our case, we placed the `set_trace()` directive immediately after the `debugger` instance. In reality, we can place it anywhere in the code; for example, if conditions are satisfied, or within a section managed by an exception.

The second step, instead, consists of opening Command Prompt and launching `telnet` by setting the same port value specified in the `debugger` parameter definition within the sample code:

```
telnet localhost 4444
```

It is possible to interact with the `rpdb` debugger by using a small command language that allows movement between calls to the stack, to examine and to modify the values of the variables and control the way in which the debugger executes its own program.

There's more...

The list of commands with which you can interact with in `rpdb` can be displayed by typing the `help` command from the `Pdb` prompt:

```
> c:\users\giancarlo\desktop\python parallel programming cookbook 2nd
edition\python parallel programming new book\chapter_x- code
debugging\rpdb_code_example.py(7) <module>()
-> def my_func(thread_number):
(Pdb) help
```

```
Documented commands (type help <topic>):
```

```
=====
```

```
EOF      c      d      h list q rv undisplay
a cl debug help ll quit s unt
alias clear disable ignore longlist r source until
args commands display interact n restart step up
b condition down j next return tbreak w
break cont enable jump p retval u whatis
bt continue exit l pp run unalias where
```

```
Miscellaneous help topics:
```

```
=====
```

```
pdb exec
```

```
(Pdb)
```

Among the most useful commands, this is how we insert the breakpoints in the code:

1. Type `b` and the line number to set a breakpoint. Here, a breakpoint is set in lines 5 and 10:

```
(Pdb) b 5
Breakpoint 1 at c:\users\giancarlo\desktop\python parallel
programming cookbook 2nd edition\python parallel programming new
book\chapter_x- code debugging\rpdb_code_example.py:5
(Pdb) b 10
Breakpoint 2 at c:\users\giancarlo\desktop\python parallel
programming cookbook 2nd edition\python parallel programming new
book\chapter_x- code debugging\rpdb_code_example.py:10
```

2. It is sufficient to type the `b` command to display the list of breakpoints implemented:

```
(Pdb) b
Num Type Disp Enb Where
1 breakpoint keep yes at c:\users\giancarlo\desktop\python parallel
programming cookbook 2nd edition\python parallel programming new
book\chapter_x- code debugging\rpdb_code_example.py:5
2 breakpoint keep yes at c:\users\giancarlo\desktop\python parallel
programming cookbook 2nd edition\python parallel programming new
book\chapter_x- code debugging\rpdb_code_example.py:10
(Pdb)
```

At each new breakpoint added, a numeric identifier is assigned. These identifiers are used to enable, disable, and interactively remove breakpoints. To disable a breakpoint, use the `disable` command, which tells the debugger not to stop when that line is reached. The breakpoint is not forgotten but is ignored.

See also

You can find a lot of information on `pdb`, and then on `rpdb`, on this site: <https://github.com/spiside/pdb-tutorial>.

In the next two sections, we will look at some Python tools that are used for the implementation of unit tests:

- `unittest`
- `nose`

Dealing with unittest

The `unittest` module is provided with the standard Python library. It has an extensive set of tools and procedures for performing unit tests. In this section, we'll briefly see how the `unittest` module works.

A unit test consists of two parts:

- The code to manage the so-called *test system*
- The test itself

Getting ready

The simplest `unittest` module can be obtained via the `TestCase` subclass, to which the appropriate methods must be rewritten or added.

A simple `unittest` module can be composed as follows:

```
import unittest

class SimpleUnitTest(unittest.TestCase):

    def test(self):
        self.assertTrue(True)

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

To run the `unittest` module, you need to include `unittest.main()`, while we have a single method, `test()`, which fails if `True` is ever `False`.

By executing the preceding example, you get the following result:

```
-----
Ran 1 test in 0.005s

OK
```

The test was successful, thus giving the result, `OK`.

In the following section, we go into more detail about how the `unittest` module works. In particular, we want to study what the possible outcomes of a unit test are.

How to do it...

Let's see how we can characterize the results of a test with this example:

1. Import the relevant module:

```
import unittest
```

2. Define the `OutcomesTest` class, which has the `TestCase` subclass as its argument:

```
class OutcomesTest(unittest.TestCase):
```

3. The first method we define is `testPass`:

```
def testPass(self):  
    return
```

4. Here is the `TestFail` method:

```
def testFail(self):  
    self.failIf(True)
```

5. Next, we have the `TestError` method:

```
def testError(self):  
    raise RuntimeError('test error!')
```

6. Finally, we have the main function, with which we recall our procedure:

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

How it works...

In this example, the possible outcomes of a unit test by `unittest` are shown.

The possible outcomes are as follows:

- **ERROR:** The test raises an exception other than `AssertionError`. There is no explicit way to pass a test, so the test status depends on the presence (or absence) of an exception.

- **FAILED:** The test is not passed and an `AssertionError` exception is raised.
- **OK:** The test is passed.

The output is as follows:

```
=====
ERROR: testError (__main__.OutcomesTest)
-----
Traceback (most recent call last):
  File "unittest_outcomes.py", line 15, in testError
    raise RuntimeError('Errore nel test!')
RuntimeError: Errore nel test!

=====
FAIL: testFail (__main__.OutcomesTest)
-----
Traceback (most recent call last):
  File "unittest_outcomes.py", line 12, in testFail
    self.failIf(True)
AssertionError

-----
Ran 3 tests in 0.000s

FAILED (failures=1, errors=1)
```

Most tests affirm the truth of a condition. There are different ways of writing tests that verify a truth, depending on the perspective of the author of the test and whether the desired result of the code is verified. If the code produces a value that can be evaluated as true, then the `failUnless ()` and `assertTrue ()` methods should be used. If the code produces a false value, then it makes more sense to use the `failIf ()` and `assertFalse ()` methods:

```
import unittest

class TruthTest(unittest.TestCase):

    def testFailUnless(self):
        self.failUnless(True)

    def testAssertTrue(self):
        self.assertTrue(True)

    def testFailIf(self):
        self.assertFalse(False)

    def testAssertFalse(self):
```

```

        self.assertFalse(False)

if __name__ == '__main__':
    unittest.main()

```

The result is as follows:

```

> python unittest_failwithmessage.py -v
testFail (__main__.FailureMessageTest) ... FAIL

=====
FAIL: testFail (__main__.FailureMessageTest)
-----
Traceback (most recent call last):
  File "unittest_failwithmessage.py", line 9, in testFail
    self.failIf(True, 'Il messaggio di fallimento va qui')
AssertionError: Il messaggio di fallimento va qui

-----
Ran 1 test in 0.000s

FAILED (failures=1)
robby@robby-desktop:~/pydev/pymotw-it/dumpscripts$ python unittest_truth.py
-v
testAssertFalse (__main__.TruthTest) ... ok
testAssertTrue (__main__.TruthTest) ... ok
testFailIf (__main__.TruthTest) ... ok
testFailUnless (__main__.TruthTest) ... ok

-----
Ran 4 tests in 0.000s

OK

```

There's more...

As mentioned previously, if a test raises an exception other than `AssertionError`, then it is treated as an error. This is very useful for discovering errors that occur while you are editing code for which a matched test already exists.

There are circumstances, however, in which you would want to run a test to verify that certain code actually produces an exception. For example, in cases when an invalid value is passed as an attribute of an object. In such cases, `failUnlessRaises()` makes the code clearer than capturing the exception in your code:

```
import unittest

def raises_error(*args, **kwds):
    print (args, kwds)
    raise ValueError\
        ('Valore non valido:'+ str(args)+ str(kwds))

class ExceptionTest(unittest.TestCase):
    def testTrapLocally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Non si vede ValueError')

    def testFailUnlessRaises(self):
        self.assertRaises\
            (ValueError, raises_error, 'a', b='c')

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

The results for both are the same. However, the result for the second test, which uses `failUnlessRaises()`, is shorter:

```
> python unittest_exception.py -v
testFailUnlessRaises (__main__.ExceptionTest) ... ('a',) {'b': 'c'}
ok
testTrapLocally (__main__.ExceptionTest) ... ('a',) {'b': 'c'}
ok

-----
Ran 2 tests in 0.000s

OK
```

See also

More information on Python testing can be found at <https://realpython.com/python-testing/>.

Application testing using nose

`nose` is an important Python module for defining unit tests. It allows us to write simple test functions using subclasses of `unittest.TestCase` but also, classes of tests that are *not* subclasses of `unittest.TestCase`.

Getting ready

Install `nose` by using `pip`:

```
C:\>pip install nose
```

The source package can be downloaded and installed at <https://pypi.org/project/nose/> by following these steps:

1. Unzip the source package.
2. `cd` to the new directory.

Then, enter the following command:

```
C:\>python setup.py install
```

One of the strengths of `nose` is automatically collecting tests from the following:

- Python source files
- Directories and packages found in the working directory

To specify which tests to run, pass the relevant test names on the command line:

```
C:\>nosetests only_test_this.py
```

The test names specified may be file or module names, and may optionally indicate the test case to run by separating the module or filename from the test case name with a colon. Filenames may be relative or absolute.

Some examples are as follows:

```
C:\>nosetests test.module
C:\>nosetests another.test:TestCase.test_method
C:\>nosetests a.test:TestCase
C:\>nosetests /path/to/test/file.py:test_function
```

You may also change the working directory, where `nose` looks for tests, by using the `-w` switch:

```
C:\>nosetests -w /path/to/tests
```

Note, however, that support for multiple `-w` arguments is now deprecated and will be removed in a future release. However, it is possible to get the same behavior by specifying the target directories without the `-w` switch:

```
C:\>nosetests /path/to/tests /another/path/to/tests
```

Further customization of test selection and loading is possible through the use of plugins.

The test result output is identical to that of `unittest`, except for the additional features, such as error classes, and plugin-supplied features such as output capture and assert introspection.

In the next section, we look at testing a class using `nose`.

How to do it...

Let's perform the steps that follow:

1. Import the relevant `nose.tools`:

```
from nose.tools import eq_
```

2. Then, set the `TestSuite` class. Here, the methods of the class are tested by the `eq_` function:

```
class TestSuite:
    def test_mult(self):
        eq_(2*2, 4)
    def ignored(self):
        eq_(2*2, 3)
```

How it works...

A unit test can be developed independently by the developer, but it is good practice to have a standard product such as `unittest` and adhere to a common test practice.

As you can see from the following example, the test method was set by using the `eq_` function. This is similar to `assertEquals` by `unittest`, which verifies that the two parameters are equal:

```
def test_mult(self):
    eq_(2*2, 4)
def ignored(self):
    eq_(2*2, 3)
```

This testing practice, despite good in intentions, has obvious limitations, such as not being able to be repeated over time (for example, when a software module changes) for so-called **regression tests**.

Here is the output:

```
C:\>nosetests -v testset.py
testset.TestSuite.test_mult ... ok
```

```
-----
Ran 1 tests in 0.001s
```

```
OK
```

In general, testing is not able to identify all the errors in a program and the same is true for unit testing, which, by analyzing individual units by definition, cannot identify integration errors, performance problems, and other system-related problems. In general, unit testing is more effective when used in conjunction with other software testing techniques.

Like any form of testing, even unit testing cannot certify the absence of errors, but can only *highlight* their presence.

There's more...

Software testing is a combinatorial mathematics problem. For example, each Boolean test requires at least two tests, one for the true condition and one for the false condition. It can be shown that, for each functional code line, three to five lines of code are required for a test. It is therefore unrealistic to test all possible input combinations of any non-trivial code without a dedicated test case generation tool.

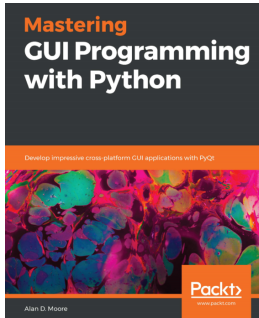
To achieve the desired benefits from a unit test, a strict sense of discipline is required throughout the development process. It is essential to keep track not only of the tests that have been developed and performed but also of all the changes made to the functional code of the unit in question and all the other units. The use of a version control system is essential. If a later version of a unit fails a test that it previously passed, then a version control system allow you to highlight the code changes that have occurred in the meantime.

See also

A valid tutorial on nose is available at <https://nose.readthedocs.io/en/latest/index.html>.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering GUI Programming with Python

Alan D. Moore

ISBN: 9781789612905

- Get to grips with the inner workings of PyQt5
- Learn how elements in a GUI application communicate with signals and slots
- Learn techniques for styling an application
- Explore database-driven applications with the QtSQL module
- Create 2D graphics with QPainter
- Delve into 3D graphics with QOpenGLWidget
- Build network and web-aware applications with QtNetwork and QtWebEngine



Expert Python Programming - Third Edition

Tarek Ziade, Michał Jaworski

ISBN: 9781789808896

- Explore modern ways of setting up repeatable and consistent development environments
- Package Python code effectively for community and production use
- Learn modern syntax elements of Python programming such as f-strings, enums, and lambda functions
- Demystify metaprogramming in Python with metaclasses
- Write concurrent code in Python
- Extend Python with code written in different languages
- Integrate Python with code written in different languages

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

Advanced Message Queuing Protocol (AMQP) 196
agglomeration 26
Alltoall
 used, for collective communication 140, 141, 142
Amazon Web Services (AWS) 218
Amdahl's law 30
application testing
 nose, using 331, 332, 333
applications
 building, with PyOpenCL 276, 277, 278, 279, 280, 281
assignments 35
asyncio
 dealing with 177, 179, 180
 reference link 166
 tasks, reference link 177
 used, for handling coroutines 166, 167, 168, 170, 171
 used, for managing event loop 160, 162, 163, 164, 165
 used, for manipulating tasks 173, 174, 175, 176
AWS Lambda 246
AWS Step Functions 248

B

barrier
 using, for thread synchronization 81, 82
billiard package
 reference link 197
broadcast
 used, for collective communication 131, 132, 133, 134
buffered mode 126

C

cache coherency 17
Cache-Only Memory Architecture (COMA) 18
Celery
 reference link 202
 used, for distributed task management 194, 196, 197, 199
 using 200, 202
 Windows setup 197
chain topology
 about 207
 implementing 208, 209, 210
class definition techniques
 reference link 101
classes 40
client-server applications
 about 183
 client-server architecture 184
 client-server communications 185
 TCP/IP connection 185, 186
cloud community 217
cloud computing architecture 214
cloud computing platforms 218
cloud computing services
 about 215
 Infrastructure as a Service (IaaS) 216
 Platform as a Service (PaaS) 216
 Software as a Service (SaaS) 215
cloud computing
 about 212, 213
 characteristics 212
cluster of workstations 20
clusters
 fail-over cluster 20
 high-performance computing cluster 20
 load balancing cluster 20

- collective communication
 - Alltoall, using 140, 141, 142
 - broadcast, using 131, 132, 133
 - gather function, using 137, 138, 139, 140
 - scatter function, using 134, 135, 137
- comments 35
- communication
 - optimizing 146, 147, 148, 150, 152
- communicator function 120
- Compute Unified Device Architecture (CUDA)
 - about 253
 - advantages 293
 - disadvantages 293
 - programming, reference link 257
- concurrent.futures module
 - using 154, 155, 157
 - working 157, 159
- condition
 - using, for thread synchronization 73, 74, 75, 77
- container 236
- coroutines in Python
 - reference link 173
- coroutines
 - handling, with asyncio 166, 167, 168, 170, 171
- CUDA model and interface, programming guide
 - reference link 257
- current thread
 - determining 55, 56, 57

D

- daemons 94
- data types
 - about 35, 37
 - dictionaries 35
 - lists 35
 - tuples 35
- data-parallel model
 - parallel program, designing 25
- data
 - exchanging, queue used 101, 102, 103, 104
- deadlock problems
 - avoiding 126, 128, 129, 130
- debuggers 302
- debugging
 - about 303

- strategies 314
- Winpdb Reborn, using 305, 306, 307, 309, 311, 312, 313, 314
- devices 251
- distributed applications, types
 - about 183
 - client-server applications 183
 - multi-level applications 186
- distributed computing 182, 183
- distributed memory system
 - about 15, 18, 19
 - advantages 18
 - features 19
- Distributed Shared Memory (DSM) 17
- distributed task management
 - with Celery 194, 196, 197, 198, 199
- distribution models, cloud computing architectures
 - about 216
 - cloud community 217
 - hybrid cloud 217
 - private cloud 217
 - public cloud 217
- Django
 - reference link 229
- Docker
 - installing, for Windows 231
- dynamic mapping
 - about 27, 28
 - decentralize 28
 - global algorithms 27
 - hierarchical manager/worker 28
 - local algorithms 27
 - manager/worker 28

E

- efficiency 29, 30
- element-wise expressions
 - with PyOpenCL 283, 284, 285
- event loop
 - about 160, 161
 - managing, with asyncio 160, 162, 163, 164, 165
 - reference link 161
- event source 161
- event

using, for thread synchronization 77, 79, 80
exceptions 41

F

Field-Programmable Gate Arrays (FPGAs) 276
files
 managing 42
finite state automaton 167
finite state machine 167
First-In, First-Out (FIFO) 101
flow control 38
Flynn's taxonomy
 about 10
 data flow 10
 instruction flow 10
 Multiple Instruction Multiple Data (MIMD) 14, 15
 Multiple Instruction Single Data (MISD) 13
 Single Instruction Multiple Data (SIMD) 13
 Single Instruction Single Data (SISD) 10
function 39
Function as a Service (FaaS) 236
futures
 dealing with 177, 179, 180

G

gather function
 used, for collective communication 137, 138, 140
Global Interpreter Lock (GIL) 45
glue language, Python
 reference link 32
Google App Engine
 reference link 229
GPU programming
 about 252, 253
 CUDA 253
 OpenCL 253
 with Numba 293, 294, 295, 296, 297, 298
Graphics Processing Unit (GPU)
 about 250
 architecture 252
Gustafson's law 31

H

help function 32, 33
Heroku
 reference link 229
heterogeneous architecture 20, 21
heterogeneous computing 251
heterogeneous programming
 with PyCUDA 257, 258, 259, 260, 261, 262
High-Performance Computing (HPC) 119
host 251
hybrid cloud 217
Hypervisor 235

I

Infrastructure as a Service (IaaS) 216
Inter-Process Communication (IPC) 187

L

Lambda function 247
Last In, First Out (LIFO) 313
layers, multi-level applications
 Business Logic Layer (BLL) 186
 Data Access Layer (DAL) 186
 Presentation Layer (PL) 186
libraries
 importing 42
Linux Containers (LXC) 230
list comprehensions 43
lock-acquire execution 64
lock-release execution 64
lock
 used, for thread synchronization 61, 62, 63, 64
 versus RLock 69
locked state 66

M

mapping 27
mapping problem 27
massively parallel processing (MPP) 20
memory access, shared memory system
 Cache-Only Memory Architecture (COMA) 18
 No Remote Memory Access (NoRMA) 17
 Non-Uniform Memory Access (NUMA) 17
 Uniform Memory Access (UMA) 17

- memory management
 - implementing, with PyCUDA 263, 264, 265, 266, 267, 268, 269, 271
- memory organization
 - about 15
 - cluster of workstations 20
 - distributed memory system 18, 19
 - heterogeneous architecture 20, 21
 - in MIMD architecture 15
 - massively parallel processing (MPP) 20
 - shared memory system 16, 17
- message brokers
 - reference links 202
- Message Passing Interface (MPI)
 - about 23, 116
 - advantages 119
 - reference link 117
 - structure 117, 118
- message passing model 23
- Microservice architecture
 - in Docker 234, 236
 - in virtual machine 234, 236
- mpi4py Python module
 - using 119, 120, 121, 122
- mpich
 - installation link 117
- multi-level applications 186
- Multiple Instruction Multiple Data (MIMD) 14, 15
- Multiple Instruction Single Data (MISD) 13
- multiprocessing documentation
 - reference link 88
- multiprocessing
 - implementation 49
- multithread model 23
- multithreaded programming 50
- mutex 72

N

- National Institute of Standards and Technology (NIST) 213
- No Remote Memory Access (NoRMA) 17
- Non-Uniform Memory Access (NUMA) 17
- nose
 - reference link 334
 - used, for testing application 331, 332, 333

- NP-complete 27
- Numba
 - reference link 253, 301
 - using, for GPU programming 293, 294, 295, 296, 297, 298
- NumPy Generalized Universal Function (gufunc) 294
- NumPy Universal Function (ufunc) 294

O

- objects
 - exchanging, pipes used 105, 106, 107, 108
- Open Computing Language (OpenCL)
 - about 253, 292
 - cons 292
 - pros 293
 - reference link 276
- OpenShift
 - reference link 229
- original class 61

P

- parallel computing
 - need for 9
- parallel processing
 - versus distributed processing 183
- parallel program
 - agglomeration 26
 - Amdahl's law 30
 - designing 25
 - dynamic mapping 27
 - efficiency 29
 - Gustafson's law 31
 - mapping 27
 - performance, evaluating 28, 29
 - scaling 30
 - speedup 29
 - task assignment 26
 - task decomposition 26
- parallel programming models
 - about 22
 - data-parallel model 24, 25
 - message passing model 23
 - multithread model 23
 - shared memory model 22

- pdb
 - command line, interacting with 315
 - directive, inserting in code to debug 316
 - features 314
 - interacting with 314, 317, 318, 319
 - Python interpreter, using 316
 - reference link 319
- PEP380
 - reference link 167
- pip
 - installing 44
 - updating 44
 - used, for installing Python packages 44
 - using 44
- pipes
 - reference link 108
 - used, for exchanging objects 105, 106, 108
- Platform as a Service (PaaS) 216
- point-to-point communication
 - implementing 122, 123, 124, 125, 126
- Pool class
 - apply() method, reference link 112
 - apply_async() method 112
 - map() method, reference link 112
 - map_async() method 112
- pooling technique 155, 159
- Portable Operating System Interface (POSIX) 23
- private cloud 217
- process pool
 - reference link 115
 - using 112, 113, 115
- process
 - about 45, 46, 47
 - defining, in subclass 99, 100, 101
 - executing, in background 93, 94, 95
 - killing 96, 98
 - naming 91, 92, 93
 - spawning 88, 89, 90, 91
 - synchronization examples, reference link 112
 - synchronizing 108, 109, 111
 - threads 48
 - versus threads 46
- producer-consumer problem 101
- public cloud 217
- PyCUDA
 - dealing with 254, 255
 - element-wise functionality 286
 - execution model 256
 - reference link 253, 257
 - used, for implementing memory management 263, 264, 265, 266, 267, 268, 269, 271
 - using, for heterogeneous programming 257, 258, 259, 260, 261, 262
 - working 255
- PyOpenCL applications
 - evaluating 287, 288, 289, 290, 291
- PyOpenCL
 - about 272, 273, 282
 - advantages 292
 - disadvantages 292
 - programming, reference link 282
 - reference link 254, 276
 - used, for building applications 276, 277, 278, 279, 280, 281
 - using, for element-wise expressions 283, 284, 285
 - working 274
- Pyro4
 - with RMI 202, 203, 205, 206, 207
- Python 3.5.0
 - reference link 53
- Python 3.7.4 documentation
 - reference link 91
- Python application
 - dockerizing 230, 232, 233, 234
- Python debuggers
 - reference link 304, 305
- Python parallel programming
 - about 45
 - processes 45, 47, 48, 49
 - threads 45, 46
- Python socket module
 - using 187, 189, 190, 191
- Python sockets
 - reference link 194
- Python testing
 - reference link 331
- Python version 2.x
 - reference link 306
- Python

- about 31, 32
- assignments 35
- classes 40
- comments 35
- data types 35, 37
- exceptions 41
- features 31, 32
- files, managing 42
- flow control 38
- function 39
- help function 32, 33
- libraries, importing 42
- list comprehensions 43
- multiprocessing module 88
- package installation, pip used 44
- scripts, executing 43
- strings 37
- syntax 34
- PythonAnywhere
 - reference link 229
 - web applications, developing with 218, 219, 221, 222, 223, 226, 228

Q

- queue module
 - using, for thread communication 83, 84, 85
- queue
 - about 101
 - used, for exchanging data 101, 102, 104
 - using, reference link 105

R

- RabbitMQ
 - installation link 196
- ranks 120
- reduction kernels 299
- reduction operation
 - about 299, 300
 - using 143, 144, 145
- register 11
- Remote Method Invocation (RMI)
 - with Pyro4 202, 203, 205, 206, 207
- RLock
 - used, for thread synchronization 66, 68
- rpdb

- implementation, for debugging 319, 320, 321, 322, 323, 325
- reference link 325

S

- scaling 30
- scatter function
 - used, for collective communication 134, 136, 137
- semaphores
 - used, for thread synchronization 69, 70, 71
- serverless computing
 - about 236, 237, 238, 240, 242, 245, 247
 - issues 248
 - limitations 248
 - working 246
- shared memory model 22
- shared memory system
 - about 15, 16, 17
 - features 17
- Single Instruction Multiple Data (SIMD)
 - about 13
 - numerical supercomputers 14
 - vectorial machines 14
- Single Instruction Single Data (SISD)
 - about 10, 11, 12
 - central memory unit 11
 - CPU 11
 - I/O system 11
- Single Program Multiple Data (SPMD) 121
- SISD, CPU operations
 - decode 11
 - execute 11
 - fetch 11
- socket
 - communicating, through phases 193
- sockets, types
 - about 193
 - datagram sockets 193
 - raw socket 193
 - stream sockets 193
- Software as a Service (SaaS) 215
- software testing 305
- speedup 29
- stream sockets 193, 194

- Streaming Multiprocessors (SMs) 251
- Streaming Processors (SPs) 252
- strings 37
- Symmetric Multiprocessors (SMPs) 17
- synchronization primitive
 - barrier 109
 - condition 108
 - event 108
 - lock 108
 - RLock 109
 - semaphore 109
- synchronous mode 126
- syntax 34

T

- task assignment 26
- task decomposition
 - about 26
 - domain decomposition 26
 - functional decomposition 26
- tasks
 - manipulating, with `asyncio` 173, 174, 175, 176
- TCP/IP client-server architecture 186
- telnet client 319
- thread communication
 - with queue module 83, 84, 85
- Thread IDentification (TID) 313
- thread subclass
 - defining 57, 58, 59, 60
- thread synchronization
 - with barrier 81, 82
 - with condition 73, 74, 75, 77
 - with event 77, 79, 80
 - with lock 61, 62, 63, 64
 - with RLock 66, 68

- with semaphores 69, 70, 71
- thread
 - about 45, 46, 47, 48, 49, 51, 52
 - defining 53, 54, 55
 - versus processes 46
- threading module 52, 53

U

- Uniform Memory Access (UMA) 17
- unittest module
 - dealing with 326, 327, 329, 330
- unlocked state 66

V

- Virtual Machine Manager 215

W

- warp scheduler 263
- warps 263
- web applications
 - developing, with `PythonAnywhere` 218, 219, 221, 222, 223, 226, 228
- web2py
 - reference link 229
- webhooks
 - reference link 202
- Windows
 - Docker, installing for 231
- Winpdb Reborn
 - about 305
 - references 305
 - used, for debugging 305, 306, 307, 309, 311, 312, 313, 314
- WS web console
 - reference link 237