Python for Experienced Java Developers

Dr. Jörg Richter

Table of Contents

Foreword

Welcome to "Python for Experienced Java Developers." This book represents a unique experiment at the intersection of human expertise and artificial intelligence. With decades of experience in various programming languages and particularly with Java since its version 1.1, I embarked on a journey to learn and master Python with the assistance of ChatGPT, an advanced AI language model.

When I began this project, I had no prior knowledge of the Python programming language. However, my extensive background in software development equipped me with a clear understanding of what I wanted to learn about Python. This book is the result of my collaboration with ChatGPT, crafted under my guidance and shaped by my expertise.

The process of creating this book was both innovative and intensive. It started with ChatGPT generating a proposed outline for each chapter. From there, ChatGPT provided detailed suggestions for each point in the outline. My role involved shortening, rearranging, and refining the text, correcting and enhancing Python examples, running and testing every line of code provided by ChatGPT, and often creating entirely new examples. Finally, I turned to ChatGPT again to make final corrections on the wording. This iterative process ensured that while ChatGPT had the first and last word on every chapter, the content in between required a significant amount of my effort and insight.

The result is the book I envisioned: a concise, focused guide that doesn’t waste pages on basics such as list manipulations, object orientation, or functional programming—concepts that every experienced Java programmer is already well-acquainted with. Instead, this book delves deeply into the unique features and capabilities that Python offers, providing a rich learning experience for seasoned developers.

I hope that other experienced software developers will appreciate the outcome of this experiment as much as I do. Whether you are looking to

add Python to your skill set or to deepen your understanding of its advanced features, I believe you will find this book both valuable and enjoyable.

Enjoy your journey into Python!

Part I
Setting Up Your Development Environment

Chapter 1
Installing Python

Other components, such as IDEs and development tools, depend on the presence of Python. Therefore, installing Python first ensures that these tools have the necessary runtime environment.
Here are the steps you need to follow:

1.1 Downloading Python

- Visit the official Python website at python.org.
- Navigate to the "Downloads" section.
- Choose the appropriate installer based on your operating system: Windows, macOS, or Linux.
- Select the desired Python version (Python 3.x is recommended for most projects, this book is based on Python 3.11.8).

1.2 Running the Installer

- Once the installer is downloaded, double-click to run it.
- Follow the prompts in the installation wizard.
- On Windows, ensure the option to add Python to PATH is selected. This allows Python to be accessible from the command line.
- On macOS and Linux, Python is typically installed by default. Ensure you are installing the desired version if multiple versions are available.

1.3 Verification

- After the installation is complete, open a terminal or command prompt.
- Run `python --version` to verify the installation.
- You should see the installed Python version displayed.

1.4 Updating Pip (Python Package Installer)

- Pip is the default package manager for Python, used to install additional libraries and packages. Further details about packages will be provided in a later chapter.
- Run `python -m pip install --upgrade pip` to ensure you have the latest version of pip installed.

Chapter 2
Choosing an IDE

In addition to specialized Integrated Development Environments (IDEs), it is worth noting that popular Java IDEs like IntelliJ IDEA and Eclipse also support Python development. In this book, we will primarily focus on using IntelliJ IDEA Community Edition (Version 2023.3.4) for Python development.
While IntelliJ IDEA Community Edition provides robust support for Python, it is important to recognize that there are IDEs specifically tailored for Python development.

Here are some of the most popular Python IDEs, all favored by AI developers for their extensive features and strong Python support.

PyCharm: Developed by JetBrains, PyCharm is a powerful IDE with a rich set of features tailored for Python development. PyCharm offers intelligent code completion, debugging tools, integration with version control systems, and support for popular AI libraries like TensorFlow, PyTorch, and scikit-learn.

Jupyter Notebook / JupyterLab: Jupyter Notebook is a web-based interactive computing environment that allows developers to create and share documents containing live code, equations, visualizations, and narrative text. Jupyter Notebook was initially developed by a team of researchers led by Fernando Pérez and Brian Granger. They began developing Jupyter Notebook in 2011 as part of the IPython project, an interactive computing environment for Python. It is widely used in the AI and data science community for prototyping, experimentation, and collaboration. JupyterLab is an enhanced version of Jupyter Notebook with a more comprehensive interface and additional features.

Google Colab: A free, cloud-based Jupyter notebook environment provided by Google. It allows developers to write and execute Python code in a browser-based interface, with access to powerful computing resources provided by Google’s infrastructure.

Spyder: An IDE specifically designed for scientific computing and data analysis, developed by the Spyder Project Contributors, an open-source community of developers. It provides features such as an interactive console, variable explorer, debugger, and support for plotting libraries like Matplotlib and seaborn. Spyder is popular among AI developers for its ease of use and integration with scientific computing libraries.

Visual Studio Code (VS Code): A lightweight and versatile IDE developed by Microsoft. VS Code offers extensive customization options and support for Python through plugins like Python and Pylance.

Atom: A customizable and open-source text editor developed by GitHub. It offers features such as syntax highlighting, code folding, and package management. Atom is popular among AI developers for its flexibility and support for plugins that extend its functionality for AI development.

Chapter 3
Create Your First Project

3.1 Project Creation

- Launch IntelliJ IDEA and create a new project.
- Select "Python" as the programming language.
- If this is your first Python project you will see a list of available plugins. Select the Python Community Edition plugin and install it.
- If you have more than one Python version installed, make sure that you choose the right version of the Python interpreter.
- Eventually select the creation of a repository for your VCS, for example GIT and leave all other parameters as they are.
- Press the Create button.

3.2 Select the Python Interpreter

- Navigate to *File → Project Structure → Project Settings → Project*.
- For *SDK*, select a Python SDK from the dropdown menu.
- Apply the changes and close the settings window.

3.3 Other Configurations

Additionally, you may want to consider other configurations such as version control management, configuring code style, and adjusting editor settings for features like code completion. These tasks can be configured in a manner similar to Java.

3.4 Create a Directory Structure

While Java projects typically adhere to a mandatory directory structure when using a Gradle-based build process, Python projects do not have such strict requirements. However, organizing your Python project into a structured directory layout can significantly enhance maintainability, collaboration, and ease of development. Here is a proposed directory structure that you will need to create manually.

```
project_name/
|- src/
|  |- package_name/
|  |  |- __init__.py
|  |  |- module1.py
|  |  |- module2.py
|- tests/
|  |- test_package_name/
|  |  |- test_module1.py
|  |  |- test_module2.py
|- docs/
|  |- conf.py
|  |- index.rst
|- data/
|  |- dataset1.csv
|  |- dataset2.json
|- README.md
|- LICENSE
|- requirements.txt
|- .gitignore.
```

`src/`: Contains the source code of the project, organized into packages and modules. Each package may contain submodules and related files. This will be explained in a later chapter.

`tests/`: Contains unit tests and test scripts for testing the functionality of the project. Tests should be organized to mirror the structure of the `src` directory.

`docs/`: Contains project documentation, like configuration files for documentation tools such as Sphinx (`conf.py`) or the main documentation file (`index.rst`).

`data/`: Contains data files or datasets used by the project. This directory may also include any generated or processed data files.

`README.md`: Provides an overview of the project, including installation instructions, usage examples, and other relevant information.

`LICENSE`: Contains the project's license file specifying the terms of use and distribution.

`requirements.txt`: The file lists project dependencies and their versions, enabling easy installation of dependencies using pip. The contents of this file will be explained in a later chapter.

`.gitignore`: Specifies files and directories to be ignored by the GIT version control systems.

3.5 Run a Test Program

As a final step to test that your IDE is properly set up, write a small 'Hello World' program and run it in the console window:

- In IntelliJ IDEA, select a directory in the `src` branch of your directory tree, do a right mouse click and select

  New → Python File.
- Enter a name for your Python file, for example `hello_world.py`.
- In the newly created Python file, enter the following code:

  ```
  print("Hello World")
  ```
- Right-click on the Python file in the project explorer and select:

  Run 'Hello_World'.
- After running the program, you should see `Hello World` printed in the console output window at the bottom of the IntelliJ IDEA interface.

Part II
The Python Language

Chapter 4
Syntax: Python vs Java

4.1 Basic Syntax Differences

In contrast to Java, Python syntax diverges from the C-like style. Here are some fundamental syntax differences between Python and Java:

Indentation vs. Braces: Python uses indentation to define blocks of code, such as loops, functions, and conditionals instead of delineating code with braces .

Semicolons: Python does not require semicolons at the end of statements. However, you can use semicolons to separate multiple statements on the same line.

Comments: Python supports single-line comments denoted by the `#` symbol and multi-line comments using triple quotes `'''` or `"""`.

```
# This is a single-line comment
"""
This is a
multi-line comment
"""
```

Strings: In Python, strings can be defined using single (`'`) or double (`"`) quotes interchangeably. The == operator checks whether the content of two strings is the same, whereas the `is` operator can be used to check whether two string variables refer to the same object in memory.

Line Continuation: In Python, unlike Java, continuing a statement on the next line typically requires the continuation character `\` at the line's end. However, an important exception exists: Within parentheses, brackets, or braces `()`, `[]`, `{}`, line continuation is implicit and doesn't necessitate any special syntax.

```
long_string = "This string spans \
two lines."
```

```
print("Hello",
      "World")
```

4.2 Data Types

Python supports various built-in data types, including:

`int`

Integer numbers without decimal points (e.g., 10, -5, 1000). In Python, there is no predefined limit on the size of integers like there is for `int` and `long` in Java. As long as your computer's memory can accommodate it, Python can handle very large integer values.

`float`

Floating-point numbers with decimal points (e.g., 3.14, -0.5, 2.718). They represent floating-point numbers with double precision according to the IEEE 754 standard, typically providing around 15 decimal digits of precision. There is no distinct type specifically for single precision floating-point numbers like in Java.

`complex`

Complex numbers with real and imaginary parts (e.g., 3+4j, -2-5j).

`str`

Strings representing immutable sequences of unicode characters.

String literals can be written with single or double quotes, allowing the use of the other quote type inside the string.

```
"He said: 'Hello!'"
'He said: "Hello!"'
```

As with comments, triple quotes for multiline string literals are supported as well.

String literals can contain arbitrary Unicode characters, either directly typed in or represented as code points or Unicode literals.

```
"Greek letter alpha: α’
"Greek letter alpha: \u03B1’
"Greek letter alpha: \N{GREEK SMALL LETTER ALPHA}’
```

There are special string literals called f-strings that evaluate expressions within braces `{}` and replace them with their values.

```
f"value of x = {x}"
```

`bool`

Boolean values representing `True` or `False`.

`bytes`

`bytes` objects can store immutable sequences of bytes, representing binary data ranging from 0 to 255.

`bytes` objects can be created using

- a `bytes` literal prefixed with `b`, for example `b"Hello"`, `b"x48x65"`
- the `bytes()` constructor who accepts a variety of input parameters, for example:

  ```
  bytes(65)
  bytes([72, 91])
  bytes("Hello", "utf-8") # String "Hello" in utf-8 encoding
  ```

Python has two additional numerical operators that Java does not have: `**` for exponentiation and `//` for floor division.

In Java, you would use

```
Math.pow(a, b)
```

for exponentiation and

```
Math.floor(a / b)
```

for floor division.

4.3 Variable Declarations

Python is a dynamically typed language, meaning that variables do not require explicit type declarations. Instead, the type of a variable is inferred at runtime based on the value assigned to it.

Therefore, variables in Python are declared by simply assigning a value to a name.

```
# Integer variable
my_integer = 10

# Float variable
my_float = 3.14

# String variable
my_string = "Hello World!"
```

This example also demonstrates that, unlike Java, static variables in Python can be defined outside of class definitions simply by declaring them in the top-level scope.

In Python, variables can dynamically change their type based on the assigned value at any time.

```
# Assigning with a string
my_variable = "Hello"
print(type(my_variable))  # <class 'str'>

# Reassigning with a boolean
my_variable = True
print(type(my_variable))  # <class 'bool'>
```

4.4 Control Flow Statements

4.4.1 if-else Statements

In Python, if-else statements function similarly to those in Java, with the distinction that they do not require braces for the condition and utilize indentations to differentiate the if and else blocks within the statement.

Conditions are written similarly to Java, but Python uses `and` and `or` instead of `&&` and `||` for logical operations.

```
if x > 0 and y > 0:
    print("both numbers are positive")
else:
    print("at least one number is non-positive")
```

In Python, there is no switch statement, but using `elif` statements (abbreviation for `else if`) serves a similar purpose.

```
if argument == 1:
    print("Case 1")
elif argument == 2:
    print("Case 2")
elif argument == 3:
    print("Case 3")
else:
    print("Default case")
```

In contrast to Java, in Python, the terms "truthy" and "falsy" are used to describe values that evaluate to `True` or `False` in a boolean context, respectively. These concepts are similar to Java, but there are some differences in how certain values are evaluated.

Truthy values: In Python, this includes non-zero numbers, non-empty sequences (lists, tuples, strings), and non-empty containers (dictionaries, sets) *(Sequences and containers will be discussed in Chapter 6: Python's Data Structures.)*. Additionally, objects with a `__bool__()` method returning `True` or a `__len__()` method returning a non-zero value are considered truthy *(Objects and methods will be discussed in Chapter 5: Object-Oriented Programming)*.

Falsy values: In Python, this includes `False`, `None`, numeric zero (`0`, `0.0`), empty sequences (`[]`, `()`, `""`), empty containers (`{}`), and objects with a `__bool__()` method returning `False` or a `__len__()` method returning zero.

```
x = []

if x:
    print("x is not empty")
else:
    print("x is empty")

# x is empty
```

In Python, variables defined within blocks of if statements and loops do not have restricted visibility and are accessible outside these blocks, unlike in Java:

```
x = 2

if x > 0:
    result = "x is positive"
else:
    result = "x is not positive"

print(result)  # x is positive
```

Python also supports a ternary operator, which provides a compact way to evaluate expressions based on a condition.

The syntax

```
x if condition else y
```

is equivalent to the Java's

```
condition ? x : y
```

Here is a short example:

```
x = 2

print("even" if x % 2 == 0 else "odd")  # even
```

4.4.2 `for` Loops

In Python, `for` loops can iterate over elements of sequences such as strings, lists, tuples, sets, etc *(Lists, tuples, and sets will be discussed in Chapter 6: Python's Data Structures.)*.

To emulate the behavior of a `for` loop that operates on integer indices, as commonly seen in languages like Java, Python provides the `range()` function. This function generates a sequence of integer numbers.

```
for fruit in ["apple", "banana", "cherry"]:
    print(fruit, end=" ")  # apple banana cherry

for char in "Hello":
    print(char, end=" ")  # H e l l o


for i in range(3, 10, 2): # (start, stop, step)
    print(i, end=" ")  # 3 5 7 9
```

The `end` parameter in the `print()` function specifies the character or string to print at the end of the output, replacing the default newline (`\n`).

4.4.3 `while` Loops

Except for the syntax, `while` loops work the same way as in Java.

```
x = 0
while x < 5:
    print(x)  # 0 1 2 3 4
    x += 1
```

4.4.4 Nested Blocks

In Python, the number of white spaces or tabs used for indentation is crucial for the interpreter to discern the code's structure and determine which statements belong to which blocks. Indentation can be achieved using either spaces or tabs.

While Python doesn't enforce a specific number of spaces for indentation, it is recommended to follow Python's official style guide, PEP 8, which

suggests using 4 spaces for each level of indentation.

Consistency in indentation within the same block is essential across the entire codebase. Mixing spaces and tabs for indentation is discouraged and may result in a `TabError`.

```
for i in range(3):
    print("Outer loop:", i)
    for j in range(2):
        print("Inner loop:", j)
    print("Back to outer loop")

   # This creates an IndentationError
   print("Still on outer loop")
```

4.4.5 `break`, `continue`, `pass`

Python provides control flow keywords such as break, continue, and pass to modify the behavior of loops and if-else statements.

`break`: Terminates the loop immediately.

`continue`: Skips the remaining code in the current iteration and moves to the next iteration of the loop.

`pass`: Acts as a placeholder and does nothing. It is commonly used when a statement is syntactically required but no action is needed.

```
for i in range(-20,20):
    if i < 5:
        continue

    print(i)  # 5 6 7 8 9 10

    if i >= 10:
        break

# This loop does nothing, but would not
# compile without the pass statement

for i in range(10):
    pass
```

In Python, loops (`for` and `while`) can have an optional `else` clause that is executed when the loop completes normally, without encountering a `break` statement. This construct can be particularly useful for tasks where you need to verify that the loop ran to completion:

```
for i in range(5):
    if i == 3:
        print("Value 3 found")
        break
else:
    print("Value 3 not found")

# Value 3 found

for i in range(5):
    if i == -1:
        print("Value -1 found")
        break
else:
    print("Value -1 not found")

# Value -1 not found
```

4.5 Functions

4.5.1 Definition and Invocation

In Python, you can define a function using the `def` keyword, followed by the function name and its parameters (if any).

Unlike Java, functions can be defined at the top-level scope outside of class definitions, akin to static methods in Java.

Additionally, unlike in Java, functions must be defined in the code before they can be called.

Similar to Java, parameters in Python are specified within the parentheses following the function name.

In Python, local variables are declared simply by assigning a value.

This means that the syntax for assigning to a global and a local variable is the same. To clarify that an assignment in a function body is for a global variable, you need to use the `global` keyword beforehand.

```
last_name = ""


def greet(name):
    global last_name
    last_name = name

    return f"Hello {name}!"


message = greet("John")

print(message)  # Hello John!
print(last_name)  # John
```

In Python, as in Java, methods are functions associated with objects. They are invoked using dot notation on an object of a specific type. For example, strings have built-in methods like `upper()` and `split()`.

```
string1 = "hello, world!"
print(string1.upper())  # HELLO, WORLD!

string2 = "a,b,c"
print(string2.split(","))  # ['a', 'b', 'c']
```

4.5.2 Unused Parameters

In Python, a common convention for marking unused parameters in functions is to name them _, __, ___, and so on, for each unused parameter.

```
def func(_, __, ___, x):
    print(x)


func(1, 2, 3, "Hello")  # Hello
```

4.5.3 Variable Argument Lists

Variable argument lists for functions and methods are also available in Python, indicated by the `*` symbol.

However, unlike Java, in Python, the function receives the parameters as a tuple, not as an array *(Tuples will be discussed in Chapter 6: Python's Data Structures.)*.

```
def func(*args):
    print(args)


func(1, "hello")  # (1, 'hello')
func()  # ()
```

4.5.4 Default Parameters and Keyword Arguments

You can specify default parameter values for functions. If a parameter is not provided when the function is called, it will use the default value.

Default values can only be assigned to parameters that appear at the end of the parameter list. Once a default value is assigned to a parameter, all subsequent parameters must also have default values.

In function calls, keyword arguments are supported, enabling you to specify arguments by their parameter names when calling a function.

It is important to note that if a function call has both positional and keyword arguments, positional arguments must be specified first in the function call.

```
def greet(name, age, city="Unknown"):
    print(f"Hello, {name}!")
    print(f"You are {age} years old.")
    print(f"You live in {city}.")


# Positional parameters only
greet("Alice", 30)
# Hello, Alice!
# You are 30 years old.
# You live in Unknown.

# Positional parameters followed by a
```

```
# keyword argument
greet("Bob", 25, city="New York")
# Hello, Bob!
# You are 25 years old.
# You live in New York.

# Invalid function call:
# Positional argument follows
# keyword argument
greet(name="Alice", 30)
```

Please note that default parameter values are evaluated only once when the function is defined, not every time the function is called.

Consider the following example:

```
def func(a=[]):
    a.append(1)
    return a

print(func())  # Output: [1]
print(func())  # Output: [1, 1]
```

When the function `func` is called for the first time, the default value of `a` (which is the empty list `[]` *(Lists will be discussed in Chapter 6: Python's Data Structures.)*) is used, and the `append(1)` operation modifies this list by adding `1` to it.

As a result, the first call to `func()` returns `[1]`.

However, when the function func is called for the second time, the default value of `a` (which is the same list as in the first call) is reused.

Therefore, the second call to `func()` returns [1, 1], as the list object modified in the previous call is still referenced by the default value of `a`.

In Python, the `/` symbol is used in function definitions to indicate that parameters preceding it are positional-only. This means they can only be specified by position and cannot be passed as keyword arguments. Parameters following the / are either positional-only or positional-or-keyword.

```
def func(a, b, c, /, d, e):
    print(a, b, c, d, e)


func(1, 2, 3, 4, 5)  # 1 2 3 4 5
func(6, 7, 8, 9, e=10)  # 6 7 8 9 10

# This does not work
# func(6, 7, c=8, d=9, e=10)
```

4.6 Data Type Conversion Functions

Python provides a set of built-in functions that are readily available for various common operations.

As a start, here are some built-in functions for data type conversion:

`int()`: Converts a given value to an integer. It can parse numeric strings or convert floating-point numbers to integers by removing the decimal part without rounding.

`float()`: Converts a given value to a floating-point number. It can parse numeric strings or convert integers to floating-point numbers.

`bool()`: Converts a given value to a boolean. It returns True if the value is considered "truthy" and False otherwise. For example, `bool(0)` returns `False`, while `bool(1)` returns `True`.

`str()`: Converts a given value to a string. It can convert numbers, booleans, or other data types to their string representations. This is equivalent to the `toString()` function in Java.

4.7 Top-Level Code

In contrast to Java, which requires all code to be encapsulated within classes and methods, Python allows top-level code to exist outside of any functions or classes. This top-level code is executed immediately when the module is imported or run as a script *(Modules and how to import them, as well as scripts, will be discussed in Chapter 10: Modules and Packages.*

For now, you can think of it as a process similar to importing packages in Java, and running a script is like executing a `main` *method in a Java program.).*

Top-level code in Python is executed in the following scenarios:

- When the module containing the code is run as a standalone script.
- During the import process, when the module is imported and cached into memory.

Here is an example to illustrate this concept *(The exact meaning of the* `__name__` *attribute will be discussed in Chapter 10: Modules and Packages.)*:

```
# example.py

print("This is top-level code")

def my_function():
    print("This is a function")

if __name__ == "__main__":
    print("This script is being run directly")
    my_function()
else:
    print("This module is being imported")
```

When you run `example.py` as a standalone script, the output will be:

```
This is top-level code
This script is being run directly
This is a function
```

When you run another script that imports `example.py` like this

```
import example

...
```

the output will be:

```
This is top-level code
This module is being imported
```

Chapter 5
Object-Oriented Programming

5.1 Classes and Objects

In Python, similar to Java, a class defines the attributes (data) and methods (behavior) that all objects of that class will have.

However, there are several key differences compared to Java:

Syntax: Instead of braces `{}`, Python uses indentations to denote the scope of classes.

Constructor: In Python, there is only one constructor method named `__init__()` that is used to initialize object attributes. To achieve functionality similar to having multiple constructors in Java, you can use default parameter values and variable-length argument lists.

Access Modifiers: While Python does not provide explicit access modifiers like Java (public, private, protected), it does offer conventions such as single (_) and double (__) underscore prefixes to indicate visibility. These conventions serve a similar purpose to access modifiers in Java and will be discussed in a later section.

Method Overloading: Python does not support method overloading based on different parameter types. Instead, you can utilize default parameter values and conditional logic within the method to achieve similar functionality.

Reference to Current Object: In Python, unlike Java, there is no `this` keyword. Instead, the constructor and every method within a class need to have at least one parameter, conventionally named `self`. Python automatically passes the object instance as the first argument when you call a method on an object.

Object Creation: In Python, as in Java, objects are created using the class name followed by the constructor parameters enclosed in parentheses.

Garbage Collection: Similar to the Java Virtual Machine (JVM), the Python interpreter utilizes a reference counting mechanism along with a cycle detection algorithm to manage memory.

Null reference: Python's `None` reference is equivalent to `null` in Java.

`object` class: As in Java, in Python, all classes ultimately inherit from a built-in class called `object`. This class provides default implementations for a variety of special methods, often referred to as dunder methods (double underscore methods). These dunder methods allow objects to interact with built-in language features, such as `__eq__` for the equality operator, `__str__` for the `str()` function, and `__init__` for object initialization.

Equality Operators: In Python, similar to Java, there are two equality operators for objects.

The == operator checks for equality of values, comparing the contents of two objects. On the other hand, the `is` operator checks for identity, determining whether two variables reference the exact same object in memory.

In Python the == operator invokes the `__eq__()` method, which by default compares values like the `is` operator. However, the behavior of `__eq__()` can be customized by overriding it, as is often done for many standard objects like numbers, strings, lists or sets.

```
class Pair:
    x = 0
    y = 0

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def print_xy(self):
        print(f"x = {self.x} y = {self.y}")

    def __eq__(self, other):
        if type(other).__name__ == "Pair":
            return (self.x == other.x and
                    self.y == other.y)
        else:
```

```
            return False


obj_1 = None
print(obj_1 is None)  # True

obj_1 = Pair(10)
obj_1.print_xy()  # x = 10 y = 0

obj_2 = Pair(10, 0)
obj_3 = obj_1
print(obj_2 == obj_1)  # True
print(obj_2 is obj_1)  # False
print(obj_3 is obj_1)  # True
print(obj_1 == 10)  # False
```

In contrast to Java, in Python attributes need to be added by any method at runtime, typically within an `__init__` method, simply by assigning them a value, as demonstrated by the `self.x` and `self.y` attributes in the previous example. These dynamically created attributes are referred to as instance variables because each instance of a class can have a distinct set of attributes.

Additionally, the `del statement` can be used to remove an instance variable:

```
class MyClass:

    def __init__(self):
        self.my_attribute = 42

    def remove_attribute(self):
        del self.my_attribute


obj = MyClass()
print(obj.my_attribute)  # 42

obj.remove_attribute()
print(obj.attribute)  # AttributeError
```

In Python, there are also class variables that are similar to static variables in Java. They are variables that are associated with the class itself rather than

with instances of the class.

Defined within the class definition but outside of any class methods, class variables are shared by all instances of the class.

Here is an example:

```
class MyClass:
    x = "Hello"

    def print_x(self):
        print(MyClass.x)


obj1 = MyClass()
obj2 = MyClass()

obj1.print_x()  # Hello
obj2.print_x()  # Hello

MyClass.x = "Bye"
obj1.print_x()  # Bye
obj2.print_x()  # Bye
```

Please note that in the example above `del MyClass.x` is possible as well, it removes the class attribute.

As in Java, accessing class attributes through instance objects is possible, but it should be avoided for clarity and consistency.

In the example above, the statement `obj1.x = "Good Morning"` is possible but would create an instance variable `obj1.x`, so `obj1.print_x()` would still print the old value of `x` and not "Good Morning".

5.2 Inheritance

In Python, a subclass of an existing class is defined by specifying the name of the parent class in parentheses after the subclass name.

Subclasses inherit all attributes and methods from their parent class, and they can access and use these inherited members as if they were defined

directly within the subclass itself. Unlike Java, Python does not enforce access restrictions on inherited members, meaning that the parent class cannot prevent them from being used by a subclass.

In contrast to Java, Python does not automatically invoke the constructor of the superclass from a subclass constructor.

Instead, if the subclass does not define its own constructor, Python automatically looks for a constructor in the superclass. If found, it is called automatically. However, if the subclass defines its own constructor, it must explicitly call the superclass constructor if desired, typically using `super().__init__()`. But this step is not mandatory, and failing to do so will not result in a compile error.

Subclasses can override methods of the parent class without any special annotation like `@override` in Java. When overriding methods in a subclass, the `super()` function can be used to call the overridden method in the superclass.

To determine whether an object is an instance of a class or one of its subclasses, you can utilize the `isinstance()` function.

Here is a code sample demonstrating the usage of all these concepts:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def sound(self):
        pass


class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color

    def sound(self):
        return "Meow!"


our_cat = Cat("Louis", "tabby")
```

```
print(our_cat.sound())  # Meow!
print(our_cat.name)  # Louis
print(our_cat.color)  # tabby

print(isinstance(our_cat, Cat))  # True
print(isinstance(our_cat, Animal)) # True
```

5.3 Multiple Inheritance

Multiple inheritance is a feature supported by Python, but its popularity varies among developers and is often a topic of debate.

Nevertheless, multiple inheritance is used frequently, even within Python's built-in libraries, to implement what is called a mixin in object-oriented programming: Small, reusable pieces of code that can be "mixed in" to multiple classes to provide common functionality. Therefore, we will briefly describe multiple inheritance here.

In Python, multiple inheritance is achieved by listing more than one parent class inside parentheses after the subclass name.

```
class Animal:
    def speak(self):
        print("Animal speaks")


class Bird:
    def fly(self):
        print("Bird flies")


class Parrot(Animal, Bird):
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name}, {self.name}")


parrot = Parrot("Polly")
parrot.speak()  # Polly, Polly
parrot.fly()    # Bird flies
```

In Python, the Method Resolution Order (MRO) determines the sequence in which methods are sought and invoked, especially in the context of multiple inheritance to address the so-called diamond problem.

The MRO is a list of classes that dictates the order in which Python searches for method implementations when a method is called on an object.

The MRO list is generated using the C3 linearization algorithm, which considers the inheritance hierarchy and the order of parent classes to establish the method resolution sequence. The resulting list defines the order in which Python scans for methods in the class hierarchy.

Notably, methods of classes positioned earlier in the parent class list of a subclass hold higher precedence compared to those listed later.

To explicitly call an overridden method in a class with lower hierarchy, you can use the `super()` function with a class name or explicitly specify the name of the class whose method you want to call.

Here is an example to demonstrate what the MRO looks like:

```
class A:
    def method(self):
        print("A’s method")


class B(A):
    def method(self):
        print("B’s method")


class C(A):
    def method(self):
        print("C’s method")


class D(B, C):
    def method(self):
        super().method()  # Calls B’s method
        C.method(self)  # Calls C’s method

        # Calls first class in MRO after C
        super(C, self).method()  # : Calls A’s method
```

```
d = D()
d.method()  # B’s method, C’s method, A’s method

# MRO as list of classes
print(type(d).__mro__)  # D, B, C, A, object
```

5.4 Access Modifiers

5.4.1 Using _ and __ Prefixes

Protected members of a class are conventionally indicated by prefixing their names with a single underscore (_).

While this doesn’t enforce strict protection, it signals to other developers that these members are intended for internal use within the class or its subclasses.

Private members in Python are denoted by prefixing their names with double underscores (__). This naming convention triggers name mangling, making it more difficult for external code to access or modify these members directly.

Mangling works as follows:

When a member of a class is prefixed with double underscores (__), Python internally renames the member by adding the class name as a prefix to the original member name.

This renaming process occurs during compilation, not runtime.

For example, if class `MyClass` has a function named `__func()`, this function will be renamed to `_MyClass__func()` during compilation, and references to the function in the code will be updated accordingly.

Attributes and methods surrounded by double underscores, such as `__name__` and `__init__()`, are not subject to name mangling. These are

typically part of Python's standardized data model and serve specific, predefined roles within the language.

```
class MyClass:
    def __init__(self):
        self._protected_var = 42

    def _protected_method(self):
        return "Protected"

    def __private_method(self):
        return "This is a private method"


obj = MyClass()

# These statements compile and run but
# are marked in the editor as
# "Access to protected member"
print(obj._protected_var)  # 42
print(obj._protected_method())  # Protected

# This code will fail with an AttributeError
print(obj.__private_method())
```

5.4.2 Setters and Getters via Decorators

Decorators in Python and annotations in Java serve somewhat similar purposes, but they operate differently and have distinct syntax and functionality *(Decorators will be discussed in Chapter 19: Meta-Programming.)*.

Python provides the decorators

- `@property`
- `@<attr>.setter`
- `@<attr>.deleter`

for defining getter, setter, and deleter methods for class attributes.

These decorators provide precise control over attribute access while preserving the illusion of direct attribute access.

The example below demonstrates their usage.

Please note that in this example, the method `radius` is overloaded, meaning there are two methods named `radius` with different signatures.

Python does not support method overloading in the sense of having multiple methods with the same name but different parameters. However, Python's property feature allows to define getters, setters, and deleters for class attributes, which can give the appearance of overloading.

This approach is commonly employed alongside property decorators to define getter, setter, and deleter methods for a single attribute.

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError(
                "Radius must be positive")
        self.__radius = value

    @radius.deleter
    def radius(self):
        del self.__radius


circle = Circle(5)

print(circle.radius)  # 5

circle.radius = 7
print(circle.radius)  # 7

del circle.radius

print(circle.radius)  # AttributeError
```

5.5 Classes as First Class Objects

In Python, everything is an object, including classes themselves. As such, classes are instances of the `type` class.

To obtain the class object of an object, you can use the built-in `type()` function.

```
x = 42
print(type(x))  # <class 'int'>
```

Of course, you can use the class definition itself to refer to the class object.

```
class MyClass:
    def __init__(self, value):
        self.x = value


print(MyClass)  # <class '__main__.MyClass'>
```

You can also use the `type()` function to dynamically create class objects by providing it with a class name, a tuple of base classes, and a dictionary of attribute and method names with their corresponding values *(Tuples and dictionaries will be discussed in Chapter 6: Python's Data Structures.)*.

```
def init_method(self, value):
    self.x = value


def print_all(self):
    print(self.a, self.x)


name = "MyClass"
base = (object,)
attributes_and_methods = {
    "__init__": init_method,
    "print": print_all,
    "a": 1}

MyClass = type(name, base, attributes_and_methods)

obj = MyClass(42)
```

```
obj.print()  # 1 42
```

It is also possible to dynamically add a method to a statically defined class.

```
class MyClass:
    x = 42


def print_x(self):
    print(self.x)


MyClass.print_x = print_x

obj = MyClass()
obj.print_x()  # 42
```

Finally, you can also add a method to an instance of a class instead to the class itself. For this purpose, it is important to understand the distinction between two types of methods in Python: bound and unbound.

A bound method is a method that is associated with a specific instance of a class. Since the instance (`self`) is passed as the first argument to the method, it can interact with the instance's attributes and other methods.

On the other hand, an unbound method is a standalone function that is not associated with any particular instance, similar to a static function in Java.

To add a function as a method to an instance, create a bound method using `types.MethodType`:

```
import types


def print_x(self):
    print(self.x)


class MyClass:
    def __init__(self, value):
        self.x = value
```

```
obj = MyClass(42)

obj.print = types.MethodType(print_x, obj)
obj.print_unbound = print_x

obj.print()  # 42
obj.print_unbound(obj)  # 42
```

The methods of a class that we have discussed so far are instance methods, meaning they are executed on an instance of a class.

In addition, there are two other types of methods: static methods and class methods, which are marked with the decorators *(Decorators will be discussed in Chapter 19: Meta-Programming.)* `@staticmethod` and @classmethod, respectively.

Similar to Java, a static method is a method that belongs to a class rather than any instance of the class. It does not require access to the class or its instances, so it neither takes a reference to the instance (`self`) nor to the class (`cls`) as a parameter.

They are typically used for utility functions that perform a task in isolation from the class or instance data.

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b


result = MathUtils.add(5, 3)
print(result)  # 8
```

Class methods are instance methods on the class object. This means they take a reference to the class object as their first parameter (`cls`) and can modify the class state that applies across all instances of the class.

They can be used for various purposes, such as setting configuration parameters of a class:

```
class TimeStamp:

    @classmethod
    def set_date(cls, year, month, day):
        cls.year = year
        cls.month = month
        cls.day = day


TimeStamp.set_date(year=2024, month=5, day=16)

time_stamp = TimeStamp()
print(time_stamp.year)  # 2024
```

Chapter 6
Python's Data Structures

Python offers similar collections and data structures as Java, albeit with some differences in implementation and syntax. Both languages provide common data structures such as list, maps (called dictionaries in Python) and sets, among others.
In this chapter, we will explore Python's native data structures, including lists, dictionaries, sets, tuples and more.

6.1 Lists

Similar to Java, Python lists allow you to store and manipulate collections of items in an ordered sequence.

Lists are mutable, meaning their elements can be modified after creation. Python lists are heterogeneous, capable of holding elements of any data type, including other lists or custom objects.

Like Java, the positions in a Python list are zero-indexed, meaning the first element of a list has the position zero. It is also possible to use negative indices starting with -1 from the end of the list.

Python supports various list operators and methods for manipulating lists. The code sample below demonstrates some common list operators and methods.

```
# Creating an empty list
my_empty_list = []

# Creating a list
my_list = [1, 2, 3, 4]

print(my_list[0])  # 1
print(my_list[-1])  # 4

print(my_list + [5, 6])  # [1, 2, 3, 4, 5, 6]
print([1, 2] * 3)  # [1, 2, 1, 2, 1, 2]

print(1 in my_list)  # True
```

```
print(7 not in my_list)  # True

print(len(my_list))  # 4

one, two, three, four = my_list
print(one, two, three, four)  # 1 2 3 4

first, *rest = my_list
print(first, rest)  # 1 [2, 3, 4]

index = my_list.index(3)
print(index)  # 2

my_list.append("Hello")
print(my_list)  # [1, 2, 3, 4, 'Hello']

del my_list[4]
print(my_list)  # [1, 2, 3, 4]

my_list.insert(2, 6)
print(my_list)  # [1, 2, 6, 3, 4]

top = my_list.pop()
print(top, my_list)  # 4 [1, 2, 6, 3]

my_list.reverse()
print(my_list)  # [3, 6, 2, 1]

my_list.sort()
print(my_list)  # [1, 2, 3, 6]

my_list.clear()
print(my_list)  # []

count = [1, 2, 2, 2, 3].count(2)
print(count)  # 3
```

In contrast to Java, Python does not have arrays. Instead, lists of a fixed length are used.

Multidimensional arrays can be replaced with nested lists:

```
nested_list = [[1, 2, 3], [3, 4, 6]]
print(nested_list[1][1])  # 4
```

For generating longer lists with predefined values, Python has a feature called list comprehension.

It generates the list elements from a `for` clause, and optionally, additional `for` or `if` clauses.

```
# List comprehension to generate a list
# of squared numbers
squares = [x ** 2 for x in range(1, 5)]

print(squares)  # [1, 4, 9, 16]

# List comprehension to generate squares
# of even numbers
even_squares = [x**2 for x in range(1, 10)
                if x % 2 == 0]

print(even_squares)  # [4, 16, 36, 64]

# List comprehension with two for statements
pairs = [f"{x}{y}" for x in range(1, 3)
         for y in range(4, 6)]

print(pairs)  # ['14', '15', '24', '25']
```

Finally, there is a highly versatile feature for creating sublists of a list, known as slicing.

The `::` operator is utilized in Python's slicing notation to generate a sublist of a list. Its syntax is as follows:

```
list[start:stop:step]
```

The three parameters can all be omittet and default to start of the list, end of the list and 1, respectively. When using a negative step, the defaults for start and stop are the list's last and first elements, respectively.

The slicing operation is generally safe from exceptions. It can also be used to make a (shallow) copy of a list by using the slicing operator only with default values (`[:]`).

Here are a few examples:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_indexed = my_list[::2]
print(even_indexed)  # [1, 3, 5, 7, 9]

odd_indexed = my_list[1::2]
print(odd_indexed)  # [2, 4, 6, 8, 10]

first_five = my_list[:5]
print(first_five)  # [1, 2, 3, 4, 5]

last_five = my_list[-5:]
print(last_five)

subsequence = my_list[2:7]
print(subsequence)  # [3, 4, 5, 6, 7]

reverse = my_list[::-1]
print(reverse)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

first_five_reversed = my_list[4::-1]
print(first_five_reversed)  # [5, 4, 3, 2, 1]

last_five_reversed = my_list[:-6:-1]
print(last_five_reversed)  # [10, 9, 8, 7, 6]

shallow_copy = my_list[:]
print(shallow_copy == my_list)  # True
print(shallow_copy is my_list)  # False

# This throws no exception!
print(my_list[-5000:5000:-300])  # []
```

Finally, it is possible to create reusable slice objects using the `slice()` function in Python:

```
slice_obj = slice(1, None, 2)  # 1::2

list_1 = [1, 2, 3, 4, 5]
list_2 = [10, 11]

print(list_1[slice_obj])  # [2, 4]
print(list_2[slice_obj])  # [11]
```

6.2 Tuples

In Python, a tuple is an ordered collection of elements, similar to a list. However, tuples are immutable, meaning their elements cannot be modified after creation.

Most list operations also work for tuples in a similar manner.

```
my_tuple = (1, 2, 3, 4, 5)

print(my_tuple[1])  # 2
print(my_tuple[-2])  # 4

print(my_tuple[1:4])  # (2, 3, 4)

new_tuple = my_tuple + (6, 7, 8)
print(new_tuple)  # (1, 2, 3, 4, 5, 6, 7, 8)

repeated_tuple = (1, 2, 3) * 2
print(repeated_tuple)  # (1, 2, 3, 1, 2, 3)

print(my_tuple.count(3))  # 1
print(my_tuple.index(4))  # 3

a, b, c, d, e = my_tuple
print(a, b, c, d, e)  # Output: 1 2 3 4 5
```

Additionally, it is possible to omit parentheses for tuple creation in assignment and return statements:

```
def func(x, y, z):
    return x, y, z


a = 1, 2, 3
b = func(4, 5, 6)

print(a)  # (1, 2, 3)
print(b)  # (4, 5, 6)
```

As tuples are immutable, they do not have a `sort()` method like lists do. Instead, the `sorted()` function can be used to sort both lists and tuples. This function returns a new list with sorted items, without modifying the original sequence.

```
my_tuple = (3, 1, 2)
sorted_list = sorted(my_tuple)
sorted_tuple = tuple(sorted_list)

print(my_tuple)  # (3, 1, 2)
print(sorted_list)  # [1, 2, 3]
print(sorted_tuple)  # (1, 2, 3)
```

Lastly, creating a tuple that consists of only one element requires an additional comma at the end to avoid ambiguity with a scalar value.

```
a = (1)
b = (1,)

print(a)  # 1
print(b)  # (1,)
```

For the same reason, you have to use the `tuple` constructor if you want to use comprehension for tuples:

```
no_tuple = (i for i in range(5))
print(no_tuple)  # <generator object <genexpr> ...

my_tuple = tuple(i for i in range(5))
print(my_tuple)  # (0, 1, 2, 3, 4)
```

6.3 Tuple Operations on Strings

Many operations that can be performed on tuples in Python can also be applied to strings. This is because both tuples and strings are immutable sequences, which means they share a variety of common behaviors and methods.

Here are some examples:

```
string = "xzaA"

print(string[-1])  # A
print(string[1:3])  # za

print(string.index("a"))  # 2
```

```
print(list(string))  # ['x', 'z', 'a', 'A']
print(3 * string)  # xzaAxzaAxzaA

sorted_list = sorted(string)
sorted_string = "-".join(sorted_list)

print(sorted_list)  # ['A', 'a', 'x', 'z']
print(sorted_string)  # A-a-x-z
```

6.4 Dictionaries

Dictionaries in Python serve as counterparts to maps in Java.

Like Java maps, dictionaries are unordered collections of items where each item is stored as a key-value pair.

They are mutable, allowing their elements to be modified after creation. Keys within a dictionary are unique, while values can be duplicated.

Dictionaries in Python are defined using braces `{}` and contain key-value pairs separated by a colon (`:`).

Unlike Java, both keys and values in a Python dictionary can have different data types.

```
name_of_locations = {

    "home": "New York",

    ("N35.6895", "E139.6917"): "Tokyo"
}
```

This code sample demonstrates various dictionary operations and methods:

```
# Creating a dictionary
my_dict = {"name": "John", "age": 30 }

# Accessing elements of the dictionary
print(my_dict["name"])  # John

# Adding a new key-value pair
my_dict["email"] = "john@example.com"
```

```
# Modifying a value
my_dict["age"] = 31

# Removing a key-value pair
my_dict["city"] = "New York"
del my_dict["city"]

# Iterating over the dictionary
for key, value in my_dict.items():
    print(key, ":", value)
# name : John
# age : 31
# email : john@example.com

# Check if a key exists in a dictionary
print("name" in my_dict)  # True
print("city" not in my_dict)  # True

# Comprehension: Create a dictionary
# mapping numbers to their cubes
# and filter even numbers
even_cubes = {x: x*x*x for x in range(1, 10)
              if x % 2 == 0}

print(even_cubes)  # {2: 8, 4: 64, 6: 216, 8: 512}
```

Finally, there is also a feature called dictionary packing/unpacking, which transforms a list of function parameters into a dictionary and vice versa.

```
# dictionary unpacking
def func_1(a=0, b=0, c=0, d=0):
    print(a, b, c, d)


my_dict = {"c": 3, "a": 1}
func_1(**my_dict)  # 1 0 3 0


# dictionary packing
def func_2(**args):
    print(args)


func_2(x=1, y=2)  # {'x': 1, 'y': 2}
```

Notice that when a function uses dictionary packing and is called without any parameters, it is automatically interpreted within the function's scope as a call with an empty dictionary `{}`.

```
def func(**args):
    print(args)


func(x=1, y=2)  # {'x': 1, 'y': 2}
func()  # {}
```

Using dictionary packing in combination with a variable argument list, it is possible to define a highly flexible signature for a function or method:

```
(*args, **kwargs)
```

This signature allows for an arbitrary list of positional parameters and an arbitrary set of key-value pairs, providing a universal interface for accepting various types of input.

```
def func(*args, **kwargs):
    print(args, kwargs)


func()
# () {}

func(42, "hello")
# (42, 'hello') {}

func(msg="hello", code=42)
# () {'msg': 'hello', 'code': 42}

func(42, msg="hello")
# (42,) {'msg': 'hello'}
```

6.5 Sets

In Python, similar to Java, sets are unordered and mutable collections of unique elements.

They are defined using braces `{}` that contain a list of elements. In contrast to Java, elements of a set in Python can be of different types.

```
set_1 = {42, 4.9, "Hello"}
print(set_1)  # {42, 4.9, 'Hello'}

# Duplicated elements are eliminated
set_2 = {1, 1, 2}
print(set_2)  # {1, 2}
```

The main operators for sets in Python include:

Union (`|`): Combines two sets, returning a new set with all unique elements from both sets.

Intersection (`&`): Finds the common elements between two sets, returning a new set.

Difference (`-`): Finds the elements that are only in the first set and not in the second set, returning a new set.

Symmetric Difference (`^`): Finds the elements that are present in only one of the sets, but not in both, returning a new set.

Membership (`in`): Checks if an element is present in the set.

Subset (`<=`), Superset (`>=`): Checks if a set is a subset / superset of another set.

Here are some examples for using these operators.

```
set_1 = {1, 2, 3, 4}
set_2 = {3, 4, 5, 6}

set_1.add(10)
print(set_1)  # {1, 2, 3, 4, 10}
set_1.remove(10)
print(set_1)  # {1, 2, 3, 4}

union_set = set_1 | set_2
print(union_set)  # {1, 2, 3, 4, 5, 6}

intersection_set = set_1 & set_2
```

```
print(intersection_set)  # {3, 4}

difference_set = set_1 - set_2
print(difference_set)  # {1, 2}

sym_difference_set = set_1 ^ set_2
print(sym_difference_set)  # {1, 2, 5, 6}

print(3 in set_1)  # True

subset_check = {1, 2} <= set_1
print(subset_check)  # True

superset_check = set_1 >= {1, 2}
print(superset_check)  # True
```

If necessary, you can create an immutable set using the `frozenset()` function, which converts a normal (mutable) set into an immutable set, preventing any further modifications.

```
my_set = {1, 2, 3, 4}
frozen_set = frozenset(my_set)

# This statement raises an AttributeError
frozen_set.add(5)
```

6.6 Equality

In Python, equality for lists, tuples, dictionaries, and sets is based on their contents:

- Lists and tuples are considered equal if they contain the same elements in the same order.
- Dictionaries are considered equal if they have the same key-value pairs.
- Sets are considered equal if they contain the same elements.

The elements are compared using the == operator to determine if they are equal.

6.7 Iterators and Generator Functions

In Python, an iterator is an object that implements two methods:

`__next__()`: This method returns the next item in the stream of data. If there are no more items, it raises a `StopIteration` exception *(Exceptions will be explained in Chapter 7: Exceptions.)*.

`__iter__()`: This method returns the iterator object itself. It enables an iterator to be used in a context where an iterable is expected, such as in a `for` loop.

Lists, tuples, dictionaries, and sets in Python implement only the `__iter__()` method, which is responsible for creating an iterator when called. These objects do not implement the `__next__()` method directly.

Instead, Python's `for` loops use the `iter()` function (which calls the object's `__iter__()` method) to obtain an iterator, and then repeatedly call `next()` on the iterator to fetch elements sequentially.

Conversely, functions like `list()`, `tuple()`, `dict()`, and `set()` convert iterators into their corresponding data structures. Here is an example for all this:

```
class MyIterator:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.limit:
            self.current += 1
            return self.current
        else:
            raise StopIteration


for x in MyIterator(4):  # 1 2 3 4
    print(x)

my_list = list(MyIterator(4))
print(my_list)  # [1, 2, 3, 4]
```

```
my_tuple = tuple(MyIterator(4))
print(my_tuple)  # (1, 2, 3, 4)

my_set = set(MyIterator(4))
print(my_set)  # {1, 2, 3, 4}

# You have to use enumerate for dict()
my_dict = dict(enumerate(MyIterator(4)))
print(my_dict)  # {0: 1, 1: 2, 2: 3, 3: 4}
```

Note that there is an alternative way to define iterators in Python by using a construct called a generator function.

Generator functions are regular functions that use a `yield` statement to provide the next element in the sequence.

```
def my_iterator(limit):
    for i in range(1, limit+1):
        yield i


for x in my_iterator(4):  # 1 2 3 4
    print(x)

print(list(my_iterator(4)))  # [1, 2, 3, 4]
```

Generator functions automatically manage their state between `yield` statements, enabling them to resume execution from where they left off. This simplifies the implementation of complex iteration logic.

6.8 Context Managers

Context managers in Python streamline resource management and controlled setup and cleanup actions. They are typically used with the `with` statement, which is similar to Java's try-with-resources statement.

Here is an example:

```
with open("example.txt", "r") as file:
    data = file.read()
```

In this example, the open function returns a context manager that manages the file resource *(File handling will be discussed in Chapter 8: Basic File Handling)*. The with statement ensures that the file is properly opened and closed, even if an exception occurs while reading the file.

You can create a custom context manager by defining `__enter__` and `__exit__` methods in a class.

Here is an example:

```
class MyContextManager:

    def __enter__(self):
        print("__aenter__ called")
        self.data = [1, 2, 3, 4, 5]
        return self

    def __exit__(self, exc_type, exc_value,
                 traceback):
        print("__aexit__ called")
        del self.data


with MyContextManager() as manager:
    print(manager.data)

# __aenter__ called
# [1, 2, 3, 4, 5]
# __aexit__ called
```

The `__enter__` method of a context manager is responsible for setting up the context and returning the object that will be used within the with statement block.

It is called automatically when entering the `with` block.

The `__exit__` method of a context manager is responsible for performing cleanup actions and handling any exceptions that occur within the context managed by the with statement.

It is called automatically when exiting the `with` block, whether due to normal execution or an exception.

In case an exception *(Exceptions will be discussed in Chapter 7: Exceptions.)* occurs during the execution of the `with` block, the `exc_type`, `exc_value`, and `traceback` parameters hold the class of the exception, the exception itself, and the exception's traceback attribute, respectively.

All parameters will be `None` if no exception occurred.

Chapter 7
Exceptions

Exception creation and handling share common concepts in both Python and Java, such as try-except blocks and exception classes.
In this chapter, we will explore these similarities and differences in more detail.

7.1 Standard Exceptions

Python comes with a variety of built-in exceptions that cover common error cases. Some of the commonly used standard exceptions include:

TypeError: Raised when an operation or function is applied to an object of inappropriate type.

ValueError: Raised when an operation or function receives a correct type but an inappropriate value.

ZeroDivisionError: Raised when the second operand of a division or modulo operation is zero.

IndexError: Raised when a sequence index is out of range.

KeyError: Raised when a dictionary key is not found.

FileNotFoundError: Raised when a file or directory is requested but cannot be found.

AttributeError: Raised when attempting to access an attribute or method that doesn’t exist or is not accessible within the context of the object.

These exceptions are raised by Python’s built-in functions, but they can also be raised by custom code.

```
def divide(x, y):
    if y == 0:
        raise ValueError("Zero Divisor")
```

```
    return x / y
```

In Python, most built-in exceptions inherit from the `Exception` class, which itself inherits from the `BaseException` class. The `BaseException` class has several standard attributes. Here are a few important ones:

`args`: This attribute is set automatically when an exception object is created. It contains a tuple of arguments passed to the exception constructor.

Typically, the `args` tuple contains one or more values that provide additional information about the exception. These values are often descriptive messages or data relevant to the specific error condition that caused the exception to be raised.

For example, when raising a ValueError, the args tuple may contain a message indicating why the value is considered invalid.

This is equivalent to the message string that can be passed as a parameter to the constructor of a Java exception.

`__traceback__`: This attribute is set automatically by the Python interpreter when an exception occurs. It contains the traceback associated with the exception.

A traceback in Python is similar to Java's stack trace. Both provide information about the sequence of function calls that led to the occurrence of an exception. It is a detailed report of the function calls, including file names, line numbers, and function names, leading up to the exception.

The traceback is printed to the console by default when an unhandled exception occurs. Explaining the details of the structure of a traceback is beyond the scope of this book.

`__context__`: This attribute is automatically set by the Python interpreter when an exception occurs within the context of handling another exception. It typically contains the exception object that was being handled when the current exception occurred.

An example of such a situation is when an exception is raised within an `except` block of a `try` - `except` statement that handles a different exception.

`__cause__`: This attribute is typically set programatically using the `from` keyword in the `raise` statement, as will be explained below.

7.2 Exception Handling

While the syntax for exception handling may differ between Java and Python, the underlying principles are similar. Exceptions propagate up the call stack until they are caught by an appropriate `try` - `except` block or propagate to the top level of the program, resulting in program termination.

However, a significant difference is that Python does not have the concept of checked exceptions like Java. In Python, all exceptions are considered unchecked, meaning you are not required to declare them in the method signature.

Here is a complete example.

```
def divide_numbers(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("Zero Division")
    except Exception as e:
        # All other exceptions,
        # 'as' is optional and ties the exception
        # to a variable
        print("Exception:", e)
    else:
        # Executed, if no error occurs
        print("Result:", result)
    finally:
        # Always executed
        print("Finished")


divide_numbers(10, 2)
# Result: 5.0
# Finished

divide_numbers(10, 0)
```

```
# Zero Division
# Finished

divide_numbers(10, "2")
# Exception: unsupported operand type(s) for /: ...
# Finished
```

7.3 Exception Chains

Exception chains, also known as exception nesting or chaining, refer to the concept of one exception being associated with another as its cause.

This typically occurs when an exception is raised within the handling of another exception.

As mentioned earlier, Python automatically sets the `__context__` attribute of an exception when it occurs within the context of handling another exception.

Additionally, you can explicitly establish a cause-and-effect relationship between exceptions using the `raise ... from ...` statement, which sets the `__cause__` attribute of the raised exception.

```
def divide(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        divisor_ex = ValueError("Divisor is zero.")
        cause = AttributeError("x={x}, y={y}")
        raise divisor_ex from cause


try:
    divide(1, 0)

except Exception as ex:
    print(f"ex: {type(ex)}")
    print(f"__cause__: {type(ex.__cause__)}")
    print(f"__context__: {type(ex.__context__)}")

# ex: <class 'ValueError'>
# __cause__: <class 'AttributeError'>
```

```
# __context__: <class 'ZeroDivisionError'>
```

7.4 Custom Exceptions

In Python, similar to Java, you can create custom exceptions by inheriting from the `Exception` class.

```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)
        self.error_code = 500


def failing_operation():
    raise CustomError("Function failed.")


try:
    failing_operation()
except CustomError as e:
    print(e)  # Function failed.
    print(e.error_code)  # 500
```

In contrast to Java, where the constructor of the Exception class typically accepts standard arguments like message or stack trace, the constructor of the base `Exception` class in Python only accepts a variable argument list (`*args`) and does not define standard arguments like message or stack trace explicitly.

These exceptions can then be accessed later through the `args` attribute of the raised exception.

Custom exceptions should be derived from `Exception` rather than its base class, `BaseException`. From the `BaseException` class exceptions such as `SystemExit` and `KeyboardInterrupt` are inherited which are typically reserved for system-level exceptions.

Chapter 8
Basic File Handling

This chapter covers fundamental file operations like opening, reading, and writing files, as well as exception handling during file operations.

8.1 Opening and Closing Files

Opening and closing a file in Python is similar to Java.

To open a file, you use the open() function. Once you're done working with the file, it's essential to close it using the close() method to release system resources.

```
file = open("example.txt", "r")
data = file.read()
# Close the file
file.close()
```

The first parameter in the open function specifies the file path. In this example, it is relative to the current working directory, where the Python script is executed.

Absolute paths are also possible by providing the complete path to the file. However, absolute paths may require platform-specific formatting depending on the operating system used.

This formatting typically involves specifying the path separators correctly for the operating system in use, but it is not directly handled by Python's core functionality.

The second parameter in the `open` function is the mode parameter. It determines how you want to access the file.

Some common options are read (`r`), write (`w`), append (`a`), binary (`b`), and text (`t`). Combinations like `rb` for reading a binary file are possible as well.

In Python, you can use the `with` statement to automatically handle the opening and closing of files. This is similar to the try-with-resources statement in Java. The `with` statement ensures that the file is properly closed after its suite finishes, even if an exception is raised during the execution of the code block.

```
with open("example.txt", "r") as file:
    data = file.read()
    print(data)
```

8.2 Reading from Files

Reading from files in Python is as straightforward as in Java, and there are multiple methods to achieve it:

Using the `read` method, a file can be read either completely or in chunks, which is the preferred method for binary files, as in Java.

```
with open("example.txt", "r") as file:
    contents = file.read()
# Process the contents

chunk_size = 1024  # Read 1 KB at a time
with open("example.bin", "rb") as file:
    while True:
        chunk = file.read(chunk_size)
        if not chunk:
            break
        # Process the chunk of data
```

For text files, `readline()` and `readlines()` can be used.

```
with open("example.txt", "r") as file:
    lines = file.readlines()
# Process the list of lines

with open("example.txt", "r") as file:
    while True:
        line = file.readline()
        if not line:
            break
        # Process the current line
```

Finally, Python file objects can be used as iterators to iterate over lines in a file directly within a loop.

```
with open("filename.txt", "r") as file:
    for line in file:
        # Process the current line
```

8.3 Writing to Files

The `write()` method allows you to write data directly to a file. You can use it to write strings and bytes. Any other data types must be converted before writing.

The file must have been opened in write mode (`w`) if you want to create or overwrite a file or in append mode (`a`) if you want to add data to an existing file without overwriting its content.

```
# Writing strings to a text file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a sample text.\n")

# Appending text to a text file
with open("example.txt", "a") as file:
    file.write("Appended line.\n")

# Writing binary data
data = b"\x48\x65\x6c\x6c\x6f"
with open("example.bin", "wb") as file:
    file.write(data)
```

The writelines() method is used to write a sequence of strings to a file.

Each string in the sequence is written to the file without adding any line separators, which must be added manually if needed.

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open("example.txt", "w") as file:
    file.writelines(lines)
```

8.4 Exception Handling

When reading or writing to a file in Python, some frequent exceptions are:

`FileNotFoundError:` Accessing a file that does not exist.

`PermissionError:` File operation without necessary permissions.

`IOError:` General exception for I/O-related errors.

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
except IOError:
    print("An I/O error occurred!")
```

Chapter 9
Functional Programming

In this chapter, we will explore functional programming in Python, covering its features, syntax, and applications.
While Python's support resembles Java, including higher-order functions, lambda expressions, and closures, Python's dynamic nature presents unique opportunities, as we will see.

9.1 First-Class Functions

In Python, functions can be treated just like any other data type, such as integers or strings. The following examples illustrate the main aspects of this feature.

Functions can be assigned to variables.

```
def greet(name):
    return f"Hi {name}!"


greet_function = greet
print(greet_function("Alice"))  # Hi Alice!
```

Functions can be passed as arguments to other functions.

```
def apply_operation(operation, x, y):
    return operation(x, y)


def multiply(x, y):
    return x * y


result = apply_operation(multiply, 3, 4)
print(result)  # 12
```

Functions can be returned as values from other functions.

```
def create_multiplier(factor):
    def multiplier(x):
```

```
        return x * factor

    return multiplier


triple = create_multiplier(3)
print(triple(5))  # 15
```

In Python, it is possible to pass a method of an object to a function as a parameter.

While this shares some similarities with method references in Java (denoted by the `::` operator), the syntax and implementation are different.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def multiply(self, other):
        return self.value * other


def use(function, obj):
    return function(obj)


my_object = MyClass(2)

result = use(my_object.multiply, 5)
print(result)  # 10
```

9.2 Lambda Expressions

In both Java and Python, lambda expressions are anonymous functions that can be used to create small, one-off functions without needing to explicitly define a separate function.

Lambda expressions in Python are defined using the keyword `lambda`, followed by a list of parameters separated by commas, a colon (`:`), and an expression that represents the return value of the function call.

Unlike Java, Python's dynamically typed nature means that you don't need to specify parameter types in lambda functions.

```
add = lambda x, y: x + y

print(add(3, 5))  # 8
```

Python lambdas can access variables from their enclosing lexical scope, creating closures that retain access to these variables even after the enclosing scope exits, unlike Java where final or immutable declarations are necessary.

```
def create_list_appender(x):
    return lambda y: x.append(y)


my_list = [1, 2]
add_to_my_list = create_list_appender(my_list)

add_to_my_list(3)
print(my_list)  # [1, 2, 3]
```

9.3 Built-in Higher-Order Functions

Lambda expressions are particularly handy with Python's built-in higher-order functions.

Here are examples showcasing `map()` and `filter()`. Note that in Python, they return iterators, unlike their Java counterparts.

```
numbers = [1, 2, 3, 4, 5]

squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # [1, 4, 9, 16, 25]

evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # [2, 4]
```

Part III
Managing Python Environments

Chapter 10
Modules and Packages

10.1 Overview

In Python, similar to Java, code is structured into a tree format of directories and code files, establishing a hierarchical namespace for accessing variables, functions, and classes within the code files. However, there are notable distinctions, as we will explore in this chapter.

Consider the following file tree example:

```
src/
|- main.py
|- utils/
|  |- __init__.py
|  |- helper.py
|- data/
|  |- __init__.py
|  |- data_processing.py.
```

This file tree contains examples of the following elements:

Scripts like `main.py` typically refer to standalone Python files that contains executable code meant to be run directly. They are the entry points to applications or workflows.

In a typical Python project structure, scripts are often either located at the top level of a project directory or in a directory named `scripts` or `bin` directly below the top level.

This separation helps to distinguish scripts, which are meant to be directly executable, from modules.

Modules like `helper.py` are Python files that contain reusable code, including variables, functions, and classes.

Each module can be imported and used in other modules or scripts.

Packages like `utils` are directories that contain Python modules and a special file named `__init__.py`. They provide a way to organize and namespace related modules.

The file names of modules must must end with the `.py` extension. Technically you can name your Python scripts without the .py extension, but doing so is not standard practice and may lead to confusion for other developers.

In Python, similar to Java, namespaces are hierarchical, consisting of modules and packages. When importing modules from packages, the dot notation is used to traverse the namespace hierarchy. For example, in the scenario described above, the `helper.py` module is referenced as `utils.helper` in an import statement.

In Python, there is no `package` statement; a module's package is solely determined by its position within the package hierarchy.

Therefore, in Python, unlike Java, two classes cannot share the same namespace if they are in different files. They must be in the same module file. For instance, both `utils.helper.class_1` and `utils.helper.class_2` need to be defined within the same module file, named `helper.py`.

10.2 Packages

In Python, there are regular packages and namespace packages. Regular packages have been available since both Python 2 and 3, while namespace packages are a feature introduced in version 3.3.

It is possible to use regular packages and namespace packages in parallel within a Python project.

Regular packages are directories that contain a file `__init__.py`. This file is executed when the corresponding package or parts of it are imported.

The `__init__.py` file can contain initialization code, such as defining variables, importing modules, or setting up resources needed by the package. Alternatively, it can be left empty.

It is executed only once per Python process, regardless of how many times the package is imported by different modules. Once it is executed, the module is cached in memory, and subsequent imports of the same package will reuse the already initialized module.

Namespace packages in Python enable multiple directories, ZIP files, or other file systems to collectively behave as a single Python package.

This feature is particularly useful for structuring large libraries or projects that may need to be distributed across different locations.

All package sources must not contain a `__init__.py`.

Python combines the contents of all directories or archives found within a single namespace, akin to Java packages.

It is possible to have namespace subpackages of regular packages and vice versa.

In contrast to Java Python does not throw an exception if a regular package exists twice or a namespace package finds the same module name in two different sources.

Instead it uses the first package or module it finds, ignoring other possible findings. As this can lead to an unexpected behaviour of the Python application, such situations should be avoided.

Similar to source sets in the Gradle build process, Python projects in IntelliJ IDEA can organize source code into different directories.

IntelliJ IDEA provides a project configuration feature where directories can be designated for different purposes.

This can be done by using the *Mark directory as* option in the context menu, accessed by right-clicking on a directory name in the project window.

Alternatively, you can navigate to *File → Project Structure → Project Settings → Modules → Sources* to configure directories.

The two most important markings are:

Sources Root: Indicates that all directories within it are considered top-level packages, and all Python files contained in this directory are considered part of the root package.

Excluded: Indicates that the files and directories in this package should be ignored when building and running the project.

The example below illustrates a directory structure with markings (S = Sources Root, E = Excluded) for a project that includes a legacy section utilizing regular packages and a newer section employing namespace packages, suitable for execution on two distinct operating system platforms.

With this configuration the project is ready to run on platform A using the packages

```
my_legacy_common_package
my_legacy_platform_package
my_package
```

To make the project ready to run on Platform B, you simply need to swap the Source Root and Excluded markings on the platform packages.

Notice that the `src` directory needs no marking to run the entry point `main.py` as this file is directly passed to the Python interpreter.

This example also illustrates the advantage of namespace packages. Both platform-dependent and platform-independent code can reside in the same namespace package, `my_package`. The legacy code requires two separate regular packages, `my_legacy_common_package` and `my_legacy_platform_package`

```
src/
|- main.py
|- legacy/
|  |- common (S)/
|  |  |- my_legacy_common_package/
|  |  |  |- my_common_module.py
|  |  |  |- __init__.py/
|  |- platform_A (S)/
```

```
|   |   |- my_legacy_platform_package/
|   |   |   |- my_platform_module.py
|   |   |   |- __init__.py/
|   |- platform_B (E)/
|   |   |- my_legacy_platform_package/
|   |   |   |- my_platform_module.py
|   |   |   |- __init__.py/
|- non_legacy/
|   |- common (S)/
|   |   |- my_package/
|   |   |   |- my_common_module.py
|   |- platform_A (S)/
|   |   |- my_package/
|   |   |   |- my_platform_module.py
|   |- platform_B (E)/
|   |   |- my_package/
|   |   |   |- my_platform_module.py.
```

Finally, for your reference, marking directories as source roots automatically adds them to the PYTHONPATH variable, which is a list of directories that the Python interpreter traverses when searching for modules or packages, similar to Java's classpath.

10.3 Importing Modules

Python supports various types of imports from modules, each serving specific purposes:

Importing Entire Modules: The most common type of import, where an entire module is imported using the `import` keyword. This makes all names defined in the module accessible via the module's namespace.

Importing Specific Items: Using the `from` keyword allows for the direct import of specific items into the current namespace. This facilitates selective access to only the necessary components, eliminating the need to prefix them with the module name. This practice enhances code readability and reduces namespace clutter.

Renaming Imported Items: With the `as` keyword, it is possible to rename imported items to avoid naming conflicts or provide more descriptive names.

Wildcard Imports: While Python supports wildcard imports to import all names from a module, it is generally discouraged due to potential namespace pollution and readability issues.

In the following sections, each type of import will be explained in detail. All example code will be based on the following module whose name is `my_module` and whose package is `my_package`.

```
# my_package/my_module

class MyClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, {self.name}!"


def my_function():
    return "Hello from my_module."


my_variable = "Hello, World!"
```

In Python, when you import from a package instead of a module, the objects are imported from the package's `__init__.py` file.

So the imports described in the following would also work with imports from `my_package` instead of `my_package.my_module` if the code above were to reside in the `__init__.py` file of `my_package`.

An example demonstrating the rationale behind this approach is provided at the end of the chapter.

10.3.1 Importing Entire Modules

You can import an entire module using the `import` keyword followed by the package and module name.

Once imported, you can access the contents of the module using dot notation.

```
import my_package.my_module

print(my_package.my_module.my_variable)
print(my_package.my_module.my_function())
obj = my_package.my_module.MyClass("Bob")
print(obj.greet())

# Hello, World!
# Hello from my_module.
# Hello, Bob!
```

10.3.2 Importing Specific Items

You can import specific items such as classes, functions, or variables from a module using the `from` keyword followed by the module name and `import` statement for the specific item(s) you want to import.

Once imported, you can directly reference the imported items without prefixing them with the module name.

```
from my_package.my_module import \
    my_variable, my_function, MyClass

print(my_variable)
print(my_function())
obj = MyClass("Charlie")
print(obj.greet())

# Hello, World!
# Hello from my_module.
# Hello, Charlie!
```

10.3.3 Renaming Imported Items

When importing items from a module, you can rename them using the `as` keyword followed by the desired alias. This allows for better code readability or resolving naming conflicts.

```
from my_package.my_module import \
    my_variable as renamed_variable, \
    my_function as renamed_function, \
    MyClass as RenamedClass
```

```
print(renamed_variable)
print(renamed_function())
obj = RenamedClass("Louis")
print(obj.greet())

# Hello, World!
# Hello from my_module.
# Hello, Louis!
```

10.3.4 Wildcard Imports

Wildcard imports, also known as star imports, allow importing all items from a module into the current namespace using the `*` symbol.

```
from my_package.my_module import *

print(my_variable)
print(my_function())

obj = MyClass("Lucy")
print(obj.greet())

# Hello, World!
# Hello from my_module.
# Hello, Lucy!
```

While convenient, wildcard imports can lead to namespace pollution and make it unclear which symbols are being imported, potentially causing conflicts or ambiguities in the code.

It is possible to restrict wildcard imports from a module by assigning the `__all__` variable with a list of items that will be imported using wildcards.

For example, placing

```
__all__ = ["MyClass"]
```

at the end of `my_package.my_module` would prevent `my_variable` and `my_function` from being imported with a wildcard.

However, these items would still be importable using other types of imports.

PEP 8, the official style guide for Python code, discourages the use of wildcard imports:

Wildcard imports (`from <module> import *`) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools.

10.4 Absolute vs Relative Imports

In Python, imports can be categorized into two types: absolute imports and relative imports.

Absolute imports specify the exact location of the module or package to import, starting from the top-level package. This type of import is similar to Java's import statements and provides a full path from the project's root directory to the target module or package.

Relative imports specify the location of the module or package relative to the current module. This type of import is useful when importing modules or packages within the same project or package hierarchy.

When using relative imports in Python, you can specify the number of levels up in the package hierarchy to traverse by prefixing the import path with one or more dots (.).

Each dot represents one level up in the hierarchy. For example, to import a module from the parent package, you use a single dot (`.`); to import from the grandparent package, you use two dots (`..`), and so on.

As an example, consider this project structure:

```
src/
|- main.py
|- package/
|  |- module
|  |- second_module
```

```
|  |- sub_package/
|  |  |- sub_module.py
|  |  |- sub_sub_package/
|  |  |  |- sub_sub_module.py
|  |- other_sub_package/
|  |  |- other_module.
```

If `main.py` wants to import a function from `sub_module.py`, it must use an absolute import because scripts are not part of any package and therefore cannot use relative imports.

```
# main.py

from package.sub_package.sub_module\
    import my_function
```

If `second_module` wants to import a function from the `module`, it can use a relative import using a single `.` to indicate that the module is located in the same package.

```
# second_module

from .module import my_function
```

If `sub_sub_module.py` wants to import functions from `sub_module.py` and `other_module.py`, it can do so by using `..` and `...` as prefixes, respectively.

```
# sub_sub_module

from ..sub_module \
    import my_function as func_1

from ...other_sub_package.other_module \
    import my_function as func_2
```

After discussing relative imports here, it is important to note that according to PEP 8, the Python style guide, absolute imports are generally recommended over relative imports. PEP 8 states:

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the

import system is incorrectly configured.

10.5 Package Interfaces

In Python, placing import statements within an `__init__.py` file or a module of a package makes those imports directly accessible for use in other modules or packages.

This behavior allows packages to provide a convenient interface for users to access commonly used objects directly from the package namespace.

For example, consider a package named mypackage with the following structure:

```
my_package/
|- __init__.py
|- my_module_1.py
|- my_module_2.py.
```

If `my_module_1.py` contains a function `my_function`, and `my_module_2.py` contains a class named `MyClass`, you can import these objects in `__init__.py`:

```
# __init.py__

from my_module_1 import my_function
from my_module_2 import MyClass
```

That way every script can directly import these items directly from the package:

```
from mypackage import my_function, MyClass
```

This allows for a cleaner and more intuitive API for users of the package, as they can access the objects they need directly from the package namespace.

10.6 The `__name__` Attribute in Python Modules

In Python, there is a special built-in attribute called`__name__` that is automatically defined for every module and script *(Please notice: The name attribute discussed here, should not be confused with* `__name__` *when referring to class objects in Python. The former pertains to module-level metadata, while the latter typically refers to the name of the class itself within the context of object-oriented programming.)*. It serves a crucial role in distinguishing between whether a module is being run as the main program or imported as a module into another program.

Main Program Execution: When a Python script is executed directly using `python script.py`, the `__name__` attribute for that script is set to `"__main__"`.

Module Import: When a Python script is imported as a module into a script or another module, its `__name__` attribute is set to the module's name.

The `__name__` attribute is commonly used in Python to control the execution of code based on whether the module is run as the main program or imported as a module.

Here is a simple example to illustrate its use:

```
# my_module.py

def script():
    print("Running as script.")


def imported():
    print(f"Module {__name__} imported.")


if __name__ == "__main__":
    script()
else:
    imported()
```

Running `my_module.py` directly would output

```
Running as script
```

whereas importing `my_module.py` elsewhere would output

```
Module my_module imported.
```

Chapter 11
Python Environments

11.1 Introduction to Python Environments

A Python environment, much like a Java Development Kit (JDK), encompasses the collection of resources and settings required for executing Python code and running Python-based applications.

It includes essential components such as the Python interpreter, libraries, dependencies, configuration settings, and runtime environment.

In summary, a Python environment provides the necessary infrastructure for creating, executing, and managing Python programs effectively.

There are two types of environments:

System-wide Environment: This is the default environment active when you run the Python interpreter in a newly created shell.

Virtual Environments: These provide a sandboxed environment where project-specific dependencies can be installed and managed independently of other Python projects. They can be activated either manually or automatically, for example, from your IDE when you open a project.

When you activate a virtual environment, any subsequent Python commands or scripts executed in the shell will use the Python interpreter and libraries associated with the activated virtual environment.

Conversely, deactivating a virtual environment restores the shell environment to its previous state, reverting to the system-wide Python interpreter and installed packages.

Virtual environments are typically created within a directory containing a specific structure that includes a separate Python interpreter and a set of directories for package installations and other resources.

Each virtual environment contains its own copy of the Python interpreter executable. This ensures that when the virtual environment is activated, it uses its own interpreter rather than the system-wide one.

Additionally, virtual environments include their own directories for installing Python libraries and packages. When you install a package within a virtual environment using `pip` or another package manager, the packages are installed into this isolated directory structure, separate from other environments.

Installation of packages and managing dependencies will be covered in the following chapter.

Virtual environments provide activation and deactivation scripts that modify the shell environment to use the virtual environment's Python interpreter and modify the shell's PATH variable to prioritize the virtual environment's binary directory for package execution. This allows you to switch between virtual environments seamlessly within a shell session.

Virtual environments are not inherently tied to a specific project.

While it is common practice to create a virtual environment for each project to manage project-specific dependencies, technically, the same virtual environment can be used across different projects.

11.2 Managing Virtual Environments

11.2.1 Available Tools

For creating and activating virtual environments in Python, several tools are available, each with its own advantages:

venv: This is the standard tool provided by Python itself. You can create a virtual environment using Python scripts from the `venv` module, which is included with Python by default.

Virtualenv: This popular third-party tool enhances the functionality provided by the `venv` module.

While it utilizes `venv` under the hood, Virtualenv adds extra features and flexibility to make the process more robust and user-friendly.

Conda: Conda is a popular package and environment management system, commonly utilized within the data science community. It serves as the package manager for the Anaconda Python distribution.

It provides functionality for creating and managing virtual environments, along with package installation and dependency management.

Pipenv: Pipenv is a popular tool that combines package management with virtual environment management. It creates a `Pipfile` to manage dependencies and a `Pipfile.lock` to lock dependency versions.

Poetry: Poetry is a modern dependency management tool and project manager for Python projects. It allows you to define project dependencies and settings in a `pyproject.toml` file.

Virtual environments can be created using these tools either from the command line or through an integrated development environment (IDE) like IntelliJ IDEA.

While there are scenarios where creating virtual environments from the command line may be necessary, such as when working on a remote server or when specific configurations are required, generally, using the IDE is the preferred approach.

This is because IDEs streamline the process, offer visual aids, and integrate seamlessly with the development workflow.

In the following, we will demonstrate how to manage virtual environments directly within IntelliJ IDEA. It’s worth noting that in IntelliJ IDEA, the term *SDK* (Software Development Kit) is often used interchangeably with *Virtual Environment* in the context of Python projects, as a virtual environment serves a similar purpose to an SDK in other languages.

IntelliJ IDEA has the capability to create virtual environments internally without the need for any external tools to be installed.

However, for the purposes of this demonstration, we will utilize Conda. Conda offers the advantage of creating environments for various Python versions without the necessity of pre-installing those versions on the computer.

You can download and install Conda from `conda.io`.

11.2.2 Creating a Virtual Environment

1. Navigate to
   File → Project Structure → Platform Settings → SDKs.
2. Click on the + icon and select *Python SDK* from the dropdown menu. In the dialog that appears, choose *Conda Environment* and keep the selection as *New environment*.
3. Specify the location for the new virtual environment. The location can be anywhere, but it is recommended to create a new directory within your project directory.
4. Select the desired Python version for the virtual environment.
   Keep in mind that you can only create virtual environments for Python versions supported by Anaconda, as Conda will download an Anaconda Python distribution.
5. Click *OK* to create the virtual environment.
   Conda will then download the selected Python interpreter and pre-install numerous modules so that you don’t have to install them manually later on.
6. After creating the environment, IntelliJ IDEA will assign it a default name. You can and should rename the virtual environment to something more descriptive.
   Note that this name is only used within IntelliJ IDEA, so Conda command-line tools will not show this name but the directory path instead.

11.2.3 Changing a Virtual Environment

1. Navigate to
   File → Project Structure → Project Settings → Modules

   and click on the *Dependencies* tab.
2. Under *Module SDK*, select the virtual environment you want to use from the dropdown menu and click *OK*.

From then on, the selected virtual environment will be used for running Python scripts within IntelliJ IDEA with standard settings.

Note that you can also change the virtual environment in the runtime configuration.

Each runtime configuration can run with a different virtual environment. This flexibility allows you to comfortably test your scripts against different Python versions, for example.

11.2.4 Deleting a Virtual Environment

1. Navigate to
   File → Project Structure → Platform Settings → SDKs.
2. Select the virtual environment you want to delete from the list of SDKs and click on the - icon.
3. Navigate to the directory where the virtual environment is located on your file system. Delete this directory manually.

Chapter 12
Using External Packages

In Python development, much like in Java, external packages are essential for extending the language's functionality beyond its standard library. These packages, also referred to as libraries or modules, are developed and maintained by the Python community, providing a diverse array of tools and utilities for tasks ranging from data manipulation and scientific computing to web development and beyond.

The term "package" can be somewhat misleading. Similar to a Java JAR file, an external package in Python is an archive that comprises Python code organized into packages and modules, along with other file types such as JSON, HTML, LICENSE, README, and metadata like version number, package dependencies, and author name.

It is important to note that the name of a package is a piece of metadata.

Consequently, an external package can contain more than one Python package, although this is uncommon. Additionally, the name of the external package might differ from the name of the contained Python package, which is a frequent occurrence.

For instance, consider the `Pillow` package, which is a fork of the Python Image Library but is named differently from the top-level package `PIL` it contains.

In contrast to Java, where external dependencies are often managed automatically by build tools like Gradle during the compilation process, Python requires external packages to be explicitly installed into the environment, either globally or within a virtual environment, before they can be used.

A module from an installed external package is then usable in Python code just like a module from an internal package, as described in Chapter 10: Modules and Packages.

The most common approach to installing external packages in Python is through package managers like pip, Conda, Pipenv, or Poetry, as well as built-in IDE tools like the Python Packages Tool in IntelliJ IDEA. These tools streamline the process of locating, downloading, and installing Python packages from online repositories.

Alternatively, you can install external packages from their source code or pre-built binaries by manually downloading them from the package's repository or website. Subsequently, these packages can be installed using the `setup.py` script or other installation mechanisms provided by the package.

This method is less common than using package managers, as it requires more manual effort and is prone to dependency resolution issues. Although necessary in certain cases, it will not be discussed in this book.

Instead, we will focus on the usage of pip and Conda, as they are the most widely used tools for managing Python packages and environments.

We will first explain how to use the package managers from a command line in IntelliJ IDEA's Python terminal as this is a good way to understand the principles of package management and then show how to do the same things using IntelliJ IDEA's Python Packages Tool.

12.1 Command Line Package Management

When you install Python libraries without an activated virtual environment using a packet manager, they are stored in the site-packages directory of the system-wide Python installation.

When you have a virtual environment activated, the Python libraries installed using a packet manager are stored within the site-packages directory specific to that virtual environment.

Therefore, before executing any package manager command from the command line, it is crucial to ensure that you are using the correct shell environment.

The easiest way to do this is to open a terminal program on your computer system to install packages system-wide or to open a terminal in IntelliJ IDEA, which activates the virtual environment configured for your project.

The commands for installing a package are very similar for pip and Conda. For example, to install the NumPy package, a fundamental package for scientific computing with Python, you would use one of these:

```
pip install numpy
conda install numpy
```

Here `numpy` is the package name for the NumPy package.

The package names in Python are not inherently hierarchical in the same way as Java packages, like `com.example.package`. They are typically using lowercase letters and hyphens to separate words, such as `numpy`, or `requests-toolbelt`, or `eyed3`.

pip and Conda fetch their libraries from online repositories.

The primary source for pip is the Python Package Index (PyPI), which is a repository of software packages developed and maintained by the Python community.

PyPI has a similar role for Python as Maven Central has for Java. It hosts thousands of Python packages that can be easily installed using pip.

Conda, on the other hand, has its own package repositories managed by Anaconda, Inc. These repositories contain a curated collection of packages optimized for use with Conda and are accessed when installing packages using the Conda package manager.

In addition to these primary sources, pip and Conda can also be configured to fetch packages from other repositories or local directories as needed.

Similar to Java, external packages in Python may have dependencies on other external packages. This dependency information is typically stored in

special files and evaluated by package management tools. When installing a package, these tools automatically install the required dependencies as well.

For example, when installing the data analysis package `pandas`, the `numpy` package is automatically installed as well.

Similar to Java, Python packages have version numbers and the version number of the latest version of a package can be queried from the repository, where the package is stored. Thus it is possible to install a special version of a package or upgrade a packet to the latest version:

```
pip install numpy==1.21.0
pip install --upgrade numpy

conda install numpy=1.21.0
conda update numpy
```

In Conda, it is also possible to update all installed packages at once:

```
conda update --all
```

Similar to Java, in Python only one version of a package can be installed in a specific environment at a time. This means that if multiple packages depend on different versions of the same package, it can lead to version conflicts and compatibility issues within the environment.

Therefore it is generally recommended to upgrade packages individually or in smaller groups to mitigate the risk of conflicts.

To list all installed packages along with their versions, use the following commands:

```
pip list
conda list
```

Both pip and Conda offer straightforward methods for uninstalling packages.

For example, to remove the NumPy package, you can use the following commands:

```
pip uninstall numpy
conda remove numpy
```

When uninstalling packages using pip or Conda, dependent packages that are no longer needed by any other installed packages will not be automatically uninstalled.

This is because package managers typically do not track dependencies beyond the initial installation.

However, you can use special commands to remove packages along with their dependencies.

For pip environments you can use:

```
pip-autoremove numpy
```

As `pip-autoremove` is not included with the standard Python installation or the pip package manager by default, you need to install it first:

```
pip install pip-autoremove
```

As of the time of writing this book, additionally, under Windows, after installing `pip-autoremove`, you may need to manually adjust the command for it to work properly.

See `https://github.com/invl/pip-autoremove/pull/44` for the details.

With Conda, you can use the `--prune` option to remove the package along with its unused dependencies:

```
conda remove --prune numpy
```

12.2 IntelliJ IDEA Package Management

IntelliJ IDEA offers a built-in Python Packages Tool that simplifies the process of managing external packages and dependencies within your Python projects.

You can access the tool by clicking on the corresponding icon, located in the lower part of the left menu bar by default.

The tool manages the virtual environment that is currently active. On the left side, it displays the packages currently installed along with their version numbers.

If a newer version is available for a package, the version number is shown as <*old version*> → <*new version*>, and clicking on the version number updates the package to the newest version.

There are also lists for available packages on PyPI and, if the virtual environment is a Conda environment, for Conda packages.

A search field at the top of the tool window filters the package names in all lists, making it easy to find a specific package.

Clicking on a package name shows the package description on the right side of the tool window.

At the top of this description, there is either an install button and a drop-down menu for selecting the version, or for installed packages, a vertical ellipsis icon that shows a button to uninstall the package.

It is also possible to manage the packages of all available virtual environments by navigating to

File → Project Structure → Platform Settings → SDKs

For a selected virtual environment, the *Packages* tab displays the currently installed packages and offers the same options to install, update, and uninstall packages as the Python Packages Tool.

12.3 Managing Project Dependencies

Managing dependencies is a crucial aspect of Python projects, ensuring smooth development, deployment, and collaboration.

One commonly used method for specifying project dependencies in Python is through the use of a `requirements.txt` file. Similar to the dependencies section in a Gradle script in Java, this file serves as a manifest of all the external packages and their respective versions required for your project to run.

While it is common practice to check a `requirements.txt` file into version control systems (VCS) to document project dependencies, it is generally discouraged to include the entire virtual environment directory in VCS. Instead, including only the `requirements.txt` file allows collaborators to recreate the virtual environment locally using this file and tools like pip or Conda.

While there are alternatives to `requirements.txt`, such as `Pipfile` (used by Pipenv) and `environment.yml` (used by Conda), this section will focus on the usage of `requirements.txt`.

It remains one of the most widely adopted methods for managing Python project dependencies across various tools and workflows, including Pip, virtualenv, and Docker, due to its familiarity to many Python developers.

12.3.1 Structure of `requirements.txt`

The `requirements.txt` file is typically located at the root directory of your project.

For each external package that your project relies on, the file contains a line starting with the package name followed by the version specification.

Here is an example:

```
pillow==10.2.0
numpy~=1.26.4
matplotlib>=3.6.0,<=3.8.3
requests
```

You can specify versions in various ways:

Exact Version: Pins the version to a specific release.

For example: `numpy==1.21.0`.

Version Range: Defines a range of acceptable versions using comparison operators.

For example: `numpy>=1.20.0,<1.22.0`.

Compatible Releases: Specifies that minor version updates are acceptable as long as they do not increment the leftmost non-zero digit.

For example: `numpy~=1.20.0` allows updating from 1.20.0 to 1.21.0 but not to 2.0.0

Latest Version: Indicates that your project should use the latest available version of a package. However, this approach can potentially introduce compatibility issues, so it is generally not recommended.

For example: `numpy`.

Comment lines starting with `#` are possible as well.

Additionally, there are other specifications possible in this file, such as referencing other requirements files, but this is beyond the scope of this book.

12.3.2 Creating `requirements.txt`

In addition to manually creating a `requirements.txt` file, you can utilize command-line tools such as pip or utilize IDE features like the one provided by IntelliJ IDEA, to automatically generate the file.

Before proceeding, please ensure that the virtual environment containing the dependencies you wish to record is activated.

To generate `requirements.txt` using pip, use the following command:

```
pip freeze > requirements.txt
```

This command will create a `requirements.txt` file in your project directory containing all the installed packages and their exact versions.

Alternatively you can create `requirements.txt` in IntelliJ IDEA by following these steps:

1. From the Tools menu, select *Sync Python Requirements...*.
2. In the opened dialog, you can specify the name of the file (by default, it is set to `requirements.txt`), the version type (such as no version, exact version, minimum version, or compatible releases), and a few other options like removing unused packages.
3. Click *OK* to create the file.

12.3.3 Installing `requirements.txt`

To install the dependencies specified in `requirements.txt` from the command line, use the following pip command:

```
pip install -r requirements.txt
```

This command instructs pip to install all the packages listed in the requirements.txt file along with their specified versions.

In IntelliJ IDEA, when you open `requirements.txt` or a Python file within a project containing `requirements.txt`, IntelliJ IDEA checks whether all the packages specified in `requirements.txt` are installed for the current virtual environment. If any packages are missing, a notification bar appears at the top of the editor, offering to install the missing requirements.

Note that for this the *Unsatisfied package requirements* inspection needs to be enabled, which is enabled by default. You can find this inspection under *Settings → Editor → Inspections*.

Chapter 13
Executing Python Code

Before exploring the creation of external code distributions, it is important to understand the process of compiling and running code in Python. Unlike Java, where a separate compilation step is required to generate bytecode, Python dynamically generates bytecode on-the-fly when executing a Python program.

Here is a brief overview of the process:

Compilation: The Python interpreter compiles the source code of a Python file into bytecode and stores it in a file with the same name but ending in .pyc.

Interpretation: The Python Virtual Machine (PVM) reads the bytecode instructions from the .pyc files and executes them sequentially.

When you run a Python script or load a module, the interpreter checks if there is already a compiled bytecode file (.pyc) corresponding to the source code (.py) and if the source code has not been modified since the bytecode was created.

If both conditions are met, the interpreter can reuse the existing bytecode file, skipping the compilation step and directly executing the bytecode.

Provided that the code in a .py file has been written in a platform-independent way (for example, by using a delimiter in file paths imported from a platform package), the bytecode in the .pyc file is platform-independent.

However, the .pyc code depends on the Python version, and Python makes no guarantee about bytecode compatibility between versions.

Additionally, it is possible to have modules written in languages other than Python, such as C++. These modules are called extension modules.

They typically consist of compiled binary code in a platform-specific format (for example, DLLs for Windows, shared libraries for Linux, macOS) and provide a certain set of functions and data structures in a format defined by Python.

This enables the Python interpreter to convert standard import statements and function calls from Python modules into native code calls of the extension module.

Integrating external modules into a Python project requires additional IDE and tool support, which is beyond the scope of this book.

Chapter 14
External Code Distribution

Unlike Java's JAR files, Python offers a range of packaging options for external distributions, each suited to different scenarios.
Describing all of them is beyond the scope of this book. Instead, we will focus on one set of configuration files and tools in the following sections, which should cover the most common scenarios for external distributions.

14.1 Python Code Package Types

Python code can be packaged into several types of archives, but the most common formats are sdist and wheel.

A wheel file contains pre-built artifacts for the target system, making installation straightforward by copying its contents to the appropriate directories and processing metadata such as dependencies. Typically, a wheel includes uncompiled .py files for Python modules, compiled extension modules for the target platform, and additional data files and metadata.

Since Python modules are compiled at execution time, different platforms require different wheels, mainly for extension modules or when compatibility with both Python 2 and 3 is necessary.

On the other hand, an sdist (source distribution) file contains the source code for extension modules, and may include configuration files like setup.py for setting up the distribution. When installing from an sdist file, extension modules are compiled during the installation process on the target platform.

While wheel files are preferred for production environments due to their pre-built nature, sdist files are useful when a suitable wheel is unavailable or for distributing test cases without bloating a production wheel. It is common practice to publish projects in both formats simultaneously on platforms like PyPI.

Additionally, there are specialized or less commonly used formats tailored to specific use cases or environments.

Conda Package (.conda): Conda packages are a distribution format used by the Conda package manager, typically associated with the Anaconda distribution. They contain contents similar to those found in a wheel file.

Docker Containers: Python Docker containers used for software distribution via Docker are typically based on official Python Docker images provided by Docker Hub or other reputable sources.

These official images serve as the foundation for Python Docker containers, containing the core Python runtime environment along with additional libraries or dependencies.

The following section will provide detailed instructions on how to create sdist and wheel packages.

14.2 Package Creation

Before creating a package file, ensure your project includes the key components: a well-organized structure, comprehensive documentation, thorough test scripts, specified dependencies, a README file, and a LICENSE.

The process of creating a package then involves two main steps:

First, creating the `setup.py` file to define package metadata and configuration, and then compiling the package to generate distributable package files.

There are various tools available for package creation in Python, with `distutils` and `setuptools` being among the most popular choices.

While `distutils` comes bundled with Python itself, this is not always the case for `setuptools`, and other tools definitely require separate installation.

It is important to install the chosen tool before writing the `setup.py` file because the file typically imports functions specific to the chosen tool, such as the `setup()` function in `setuptools` and `distutils`.

In our case, we will use `setuptools`.

It is worth mentioning that `setuptools` also supports other configuration files, such as `setup.cfg` or `pyproject.toml`, although these alternatives will not be explained in this book.

14.2.1 Creating `setup.py`

In this section, we will demonstrate the creation of a `setup.py` file.

We will cover two scenarios: One that is simple yet commonly encountered, and another that is more complex, showcasing some advanced features of `setup.py`. The simple scenario involves a package with a subpackage, structured as follows:

```
my_project/
|- src/
|  |- my_package/
|  |  |- my_sub_package/
|  |  |  |- __init.py__
|  |  |  |- my_sub_module_1
|  |  |  |- my_sub_module_2
|  |  |- __init.py__
|  |  |- my_module_1
|  |  |- my_module_2
|  |  |- my_module_3
|  |- setup.py.
```

A simple `setup.py` for this scenario could look like this:

```
from setuptools import setup

setup(
    name="python-simple-packaging-test",
    version="1.0.0",
    packages=[
        "my_package",
        "my_package.my_sub_package"
    ],
```

```
    license="MIT",
    author="Dr. Jörg Richter",
    author_email="python-book@nantoka.de",
    description="A simple packaging test"
)
```

In this simple form, the configuration parameters passed to the `setup()` function contain the name of the distribution (not to be confused with the name of the distributed package), the version number, the list of distributed packages, and some metadata (author, email address, description).

It is important to note that in the list of distributed packages, the subpackage needs to be explicitly listed. Otherwise, `setuptools` would only include the modules from the package but not those from the subpackage in the distribution archive.

Also, ensure that `setup.py` is placed in the top-level package directory, as this is where `setuptools` searches for packages by default.

Creating `setup.py` using IntelliJ IDEA is straightforward. Simply navigate to the *Tools* menu and select *Create setup.py*.

However, it is important to note that, at the time of writing this book, this feature works only if the top-level packages for the distribution are contained in the root directory of the project. Additionally, it does not support namespace packages.

Here is the structure of a more complex project.

```
my_project/
|- src_1/
|  |- package_1/
|  |  |- data/
|  |  |  |- data.json
|  |  |  |- webpage.html
|  |  |- sub_package/
|  |  |  |- __init.py__
|  |  |  |- sub_module_1
|  |  |  |- sub_module_2
|  |  |- __init.py__
|  |  |- module_1
```

```
|   |   |- module_2
|   |   |- module_3
|   |- package_2/
|   |   |- __init.py__
|   |   |- module_4
|   |- package_3/
|   |   |- __init.py__
|   |   |- module_5
|- src_2/
|   |- package_name_spc/
|   |   |- module_name_spc
|- src_3/
|   |- test_package/
|   |   |- test_module_1
|   |   |- test_module_2
|- MANIFEST.in
|- setup.py.
```

The project comprises code organized into three distinct root directories, one housing multiple top-level regular packages, while another accommodates a namespace package.

Furthermore, there is a dedicated package designed for testing purposes, containing modules specifically tailored for testing. These testing modules should be included in the sdist distribution but omitted from the wheel.

Please note that packaging configurations for the project are contained within a single `setup.py` file located in the project's top-level directory, along with an additional file named `MANIFEST.in`.

In this project, the `MANIFEST.in` file contains configurations specifically for adding the test package to the sdist distribution.

The comprehensive usage of `MANIFEST.in` and its available configurations are beyond the scope of this book.

Here is the contents of `setup.py` and `MANIFEST.in`, showing one possible configuration for packaging this project.

```
# setup.py
from setuptools import \
    setup, find_packages, find_namespace_packages
```

```
setup(
    name="python-complex-packaging-test",
    version="1.1.0",
    packages=find_packages("src_1") +
            find_namespace_packages("src_2"),
    package_dir={
        "": "src_1",
        "package_name_spc": "src_2/package_name_spc"
    },
    package_data={
        "package_1": ["data/*.json", "data/*.html"]
    },
    install_requires=[
        "matplotlib>=3.8.0",
        "numpy~=1.26.4",
    ],
    license="MIT",
    author="Dr. Jörg Richter",
    author_email="python-book@nantoka.de",
    description="A complex packaging test"
)

# MANIFEST.in

recursive-include src_3 *
```

The function `find_packages()` scans the specified directory recursively, identifying regular packages and subpackages based on the presence of `__init.py__` files.

It returns a list containing the names of these packages, eliminating the need to manually list each package in the `packages` parameter along with its subpackages.

`find_namespace_packages()` is similar to `find_packages()` but specifically designed for namespace packages. Unlike `find_packages()`, it treats every directory as a package, even if it does not contain an `init.py` file.

The `package_dir` parameter accepts a dictionary that maps package names to directory paths. It allows you to override the default directory structure expected by setuptools.

The keys in the `package_dir` dictionary are package names, and the values are directory paths relative to the location of `setup.py`. These directory paths are platform-independent, meaning they can be specified using forward slashes ("/") even on Windows systems.

The mapping `"": "src_1"` specifies that the contents of the `src_1` directory should be treated as top-level package.

The mapping `"package_name_spc": "src_2/package_name_spc"` specifies that the contents of `package_name_spc`, along with its subpackages, should be found in the directory `src_2/package_name_sp`.

The `package_data` parameter is used to specify additional files or patterns to include in the distribution package alongside Python modules.

These files can include data files, configuration files, templates, and more. The parameter accepts a dictionary where the keys are package names, and the values are lists of file patterns or file paths relative to each package directory.

The `install_requires` parameter is utilized to specify the dependencies, namely the packages required for your package to operate correctly. It accepts a list containing strings formatted similarly to the lines in a `requirements.txt` file.

The statement `recursive-include src *` in the `MANIFEST.in` file instructs the packaging tool to recursively include all files and directories within the `src` directory when creating the distribution package.

14.2.2 Building Packages

Once you have created the `setup.py` and, optionally, the `MANIFEST.in` file, you are ready to generate the distribution archives.

In IntelliJ IDEA, you can achieve this by selecting *Run setup.py Task...* from the *Tools* menu.

However, there are limitations to this approach: The `setup.py` file must be located in the top-level project directory, you can only create one distribution archive at a time, and the archive is generated by directly calling `setup.py`, which is discouraged by the developers of setuptools.

A more robust solution is to use the `build` package, distributed by the Python Packaging Authority (PyPA). This package is available on both PyPI and the Anaconda repository and can be installed using standard tools, as described in Chapter 10: Modules and Packages.

The benefit of using `build` is that it creates the package in an isolated environment, generating both an sdist and a wheel archive with just one command.

To do this, open a Python shell, navigate to the directory containing the `setup.py` file, and execute the following command:

```
python -m build
```

This command will create a directory named `dist` in the current directory, containing two files: an sdist file with a `.tar.gz` extension and a wheel file with a .whl extension. These files are then ready for testing and distribution.

When running the command, most temporary data is automatically cleaned up afterward. However, the `<package name>.egg-info` directory, containing metadata about the package, may persist. If you encounter issues due to outdated or incorrect metadata, manually delete this directory before rerunning the command.

After creating the package, it is advisable to conduct a quick "smoke test" to confirm its correct bundling and functionality.

To do this, set up a new project with a fresh virtual environment. Install the newly created package by opening a terminal and executing

```
pip install <package_file>
```

where `<package_file>` refers to either the wheel or the sdist file.

Once the package is installed in the virtual environment, you can proceed to write simple test code to ensure that the essential features are properly packaged and operational.

Additionally, especially for the sdist file, you can check the directory of the virtual environment to ensure that all additional files, such as the README or the LICENSE file, have been installed correctly.

14.2.3 Uploading Packages

Python's package manager, pip, relies on HTTP for downloading packages, enabling the upload of Python packages to any server that provides download URLs, such as AWS S3, Google Cloud Storage, Microsoft Azure Blob Storage, or a generic web server or file hosting service.

GitHub, in particular, offers a convenient platform for hosting code repositories and distributing software releases.

To begin, you can create a repository and upload your package code. Then, you can utilize the *Create a new release* link on the repository's homepage to upload the sdist or wheel file as a release. This allows you to upload various versions of your package under their respective version numbers.

Now, every user of your package can install it using the pip command:

```
pip install <package_download_url>
```

Additionally, users can add a dependency in a `requirements.txt` file or in the dependency list of the `install_requires` parameter in `setup.py` using the following syntax:

```
<package_name> @ <package_download_url>
```

To share your package widely, publish it on PyPI after creating an account. You will need tools like `twine`, which won't be detailed here.

Part IV
Advanced Python Techniques

Chapter 15
Writing Python Test Code

Among the most popular testing frameworks for Python, two frameworks stand out: `unittest` and `pytest`.
`unittest` is Python's built-in testing framework, drawing inspiration from the xUnit family of testing frameworks. It is sometimes referred to as PyUnit and shares similarities with JUnit for Java. Being included in every Python distribution, `unittest` is readily available and widely adopted, especially in projects that prioritize standardization and compatibility.

On the other hand, `pytest` is a third-party testing framework known for its simplicity, flexibility, and powerful features. It offers advanced capabilities such as fixtures, parameterized testing, and extensive plugin support, enabling thorough customization and seamless integration with other tools and libraries.

In this chapter, we will focus on `unittest`. Understanding `unittest` provides a strong foundation for mastering other testing frameworks, including `pytest`.

15.1 Structure of a Test Case Class

In `unittest`, a test case class inherits from `unittest.TestCase`. This base class provides various assertion methods for verifying expected outcomes.

Here is a basic example of a test case class:

```
import unittest


class MyTestCase(unittest.TestCase):

    def test_addition(self):
        result = 2 + 2

        self.assertEqual(result, 4)

    def test_check_positive_number(self):
        result = 5
```

```
        self.assertTrue(result > 0)

    def test_division_by_zero(self):
        with (self.assertRaises(ZeroDivisionError)):

            result = 5 / 0
```

Defining more than one test case class in a single module is also possible.

In contrast to JUnit, `unittest` relies on a naming convention rather than a decorator to identify methods as test cases. In `unittest`, any method whose name begins with `test` is treated as a test function, while any other method is considered an auxiliary function.

This naming convention also applies to the file names of modules within a package; failure to adhere to this convention may result in the modules not being recognized as containing tests when running all tests within a package.

15.2 Running Test Cases

Running Python test cases in IntelliJ IDEA is similar to running JUnit test cases. You can choose to run a specific test module or an entire test package directory using the following methods:

- Right-click on the test module or test package directory in the project window and select *Run Python Tests in...* from the context menu.
- Click on the gutter icon next to your test case or test class in the editor window.

The results of the test run will be presented similarly to JUnit test runs in the Run window, which is typically located at the bottom of the screen by default.

Alternatively, you can execute the tests from the command line using a terminal.

To run the tests of a specific module, enter:

```
python -m unittest <module_name>
```

To run all tests within a package directory, use the following command:

```
python -m unittest discover -s <path to directory>
```

`unittest` provides several command-line options to tailor the test execution process.

For instance, using `-v` increases the verbosity of the command's output, which can be particularly useful for pinpointing the cause of failures. Additionally, you can use the `--help` option to list all available options.

Please note that the discovery mechanism of the `unittest` test runner can handle only top-level namespace packages but will ignore all subpackages unless they are regular packages.

15.3 Test Organization

Similar to Java, it is considered good practice in Python to keep your test code closely integrated with the corresponding production code but within a separate directory. This means organizing test modules and packages in parallel with the structure of your application's source code.

For instance, if you have a module named `calculator.py` in a package named `utility` in your production code, you might create a corresponding test package named `test_utility`, which contains a module named `test_calculator.py`.

In `unittest`, to avoid code duplication, you have fixtures similar to those available in JUnit. These fixtures are methods of the `TestCase` class that you can override in your test class:

The `setUp()` method is called before each test method in a test case class and is used to set up any required resources or state for the test. Conversely, the `tearDown()` method is called after each test method and is used to clean up any resources allocated during testing.

In addition to per-test `setup` and `teardown`, `unittest` also provides class-level setup and teardown methods: `setUpClass()` and `tearDownClass()`. These methods are called once for the entire test case class, allowing you to perform setup and cleanup actions that are shared across multiple test methods.

Apart from defining test cases, the `unittest` module provides decorators that allow you to modify the behavior of individual test methods or entire test classes. These decorators are similar to the corresponding annotations used in JUnit for Java.

`@unittest.skip("<Reason for skipping>")`
This decorator marks a test method or a test class as skipped. When applied, the code inside the decorated method or class will not be executed, and the test case(s) will be marked as ignored in the result list.

You can provide an optional reason for skipping, which will be displayed in the test output.

`@unittest.expectedFailure`
This decorator marks a test method or all test methods of a test class as expected failures. If a test method marked with this decorator fails during execution, it will be marked as ignored in the result list. Conversely, if the test method succeeds, it will be marked as a failure.

This decorator is useful for temporarily marking tests that are known to fail due to issues such as unresolved bugs or incomplete functionality.

Finally, it is also possible to define a selection of test cases and their execution order using the `TestSuite` class provided by `unittest`.

Here is an example:

```
import unittest


class MyTestCase(unittest.TestCase):
    def test_add(self):
        self.assertEqual(2 + 2, 4)

    def test_subtract(self):
```

```
        self.assertEqual(5 - 3, 2)

    def test_divide(self):
        self.assertEqual(6 / 3, 2)


suite = unittest.TestSuite()
suite.addTest(MyTestCase("test_subtract"))
suite.addTest(MyTestCase("test_add"))
suite.addTest(MyTestCase("test_add"))

if __name__ == "__main__":
    print("Running tests")
    unittest.TextTestRunner(verbosity=2) \
        .run(suite)
```

This script constructs the suite and initiates a test runner capable of handling `TestSuite` objects, using the `TextTestRunner` provided by `unittest`.

You can then execute the script in a terminal using:

```
python my_test_suite.py
```

With the configured verbosity level 2 it will output the result of every test case.

Please note that the suite is configured to run `test_subtract` first, followed by `running test_add` twice, and `test_divide` is not run at all.

Also, the clause

```
if __name__ == "__main__":
```

prevents the test runner from executing if the script is loaded elsewhere, for example, when running a single test case in `MyTestCase` in a debugger while tracking down an issue.

Additionally, in a real project, you would keep the test class in a separate module and import the module in the test script that creates and runs the suite.

Finally, note that if you run this script from the context menu in IntelliJ IDEA, it will recognize it as a test file and run all three tests from `MyTestCase` exactly once.

So, to execute the suite, you will need to run it in a terminal.

15.4 Mocking and Patching

In contrast to Java, creating mock objects in Python and integrating them into the code under test is much simpler, thanks to Python's dynamically typed nature. The `unittest.mock` subpackage provides all the necessary tools for creating mocks and specifying their behavior. It also offers functionality for patching, allowing for the temporary substitution of attributes or functions within modules or classes.

Although the scope of this book does not allow for a comprehensive overview of the functionalities offered by `unittest.mock`, we will provide a brief introduction to some of its key features in this section.

15.4.1 Creating Mocks

With `unittest.mock`, the simplest way to create a mock object is by calling `Mock()`. The resulting object can then be invoked as a function or accessed as an object with attributes and methods.

There is no need to explicitly specify the names and return values of the functions it represents or the attributes it should contain. Whenever the mock object is called or queried, it dynamically creates another mock object and returns it.

Additionally, it is possible to specify the behavior of the mock object.

Its behavior as a function is determined by assigning a value to its `return_value` attribute, while its behavior for attribute queries is determined by assigning a value to the corresponding attribute.

Here is an example:

```
from unittest.mock import Mock

function_mock = Mock()
function_mock.return_value = 42

result_1 = function_mock("Life", "Universe")
result_2 = function_mock()

print(result_1, result_2)  # 42 42

object_mock = Mock()
object_mock.name = "Arthur Dent"
object_mock.luggage = "Towel"
object_mock.great_question = function_mock

result_3 = object_mock.name
result_4 = object_mock.luggage
result_5 = object_mock.great_question()

print(result_3, result_4, result_5)
# Arthur Dent Towel 42
```

As `result_1` and `result_2` show, a function mock can be called with an arbitrary number of arguments.

It is also possible to verify both the number of method calls and the parameters passed to those calls.

The assertions in the following example would be applicable to the scenario described above:

```
function_mock.assert_called()

function_mock.assert_any_call("Life", "Universe")

assert object_mock.great_question.call_count == 3
```

There is also a subclass of Mock called MagicMock, which offers useful default values for unconfigured attributes or functions when used with standard functions, for example:

```
int(MagicMock())  # returns 1
float(MagicMock())  # returns 1.0
MagicMock()[5]  # returns a MagicMock object
```

```
len(MagicMock())  # returns 0 !!!
```

Using `Mock` instead of `MagicMock` would raise an exception in these examples.

So in tests where a mock is needed for an arbitrary value that won't affect the test's outcome, `MagicMock` can simplify mock setup.

Finally there is an attribute called `side_effect` that you can specify for a mocked method in a mock.

It allows you to specify a function or an iterable that will be called or returned when the mock is called.

This attribute is useful for custom actions or varied return values based on test inputs or context.

This is how it works:

- If `side_effect` is set to a function, that function will be called with the same arguments as the mock, and its return value will be used as the return value of the mock call.
- If `side_effect` is set to an iterable (such as a list or a generator), the mock will return values from the iterable each time it is called.
- If `side_effect` is set to an exception, then when the mock is called, it will raise that exception instead of returning a value.

Here is an example:

```
from unittest.mock import Mock

func_1 = Mock(); func_2 = Mock(); func_3 = Mock()
func_1.side_effect = lambda x: x + 1
func_2.side_effect = range(1, 3)
func_3.side_effect = ValueError("Ooops")

assert func_1(10) == 11

assert func_2() == 1
assert func_2() == 2
```

```
try:
    func_3()
except ValueError as e:
    assert str(e) == "Ooops"
```

15.4.2 Patching

To isolate the code under test from its dependencies, such as external libraries, modules, or objects, `unittest.mock` provides a mechanism called patching.

While covering all aspects of this mechanism is beyond the scope of this book, we will at least cover the most important features.

A frequently occurring task when writing test cases is mocking HTTP requests. This can be easily achieved using the `@patch` decorator. As an example, consider the following function that performs a simple HTTP request:

```
# user_data.py

import requests

url = "https://api.example.com/users"


def get_user_list():
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()
    else:
        return None
```

To test the behavior of this function with both successful and failed requests, you need to patch the `requests.get` call, as illustrated in this example:

```
import unittest
from unittest.mock import patch, Mock
from user_data import get_user_list
```

```
@patch("user_data.requests.get")
class TestGetUserData(unittest.TestCase):

    def test_success(self, mock_get):
        data = ["John", "Jill", "Anna"]

        json_mock = Mock()
        json_mock.return_value = data

        response_mock = Mock()
        response_mock.status_code = 200
        response_mock.json = json_mock

        mock_get.return_value = response_mock

        result = get_user_list()
        self.assertEqual(result, data)

    def test_failure(self, mock_get):
        response_mock = Mock()
        response_mock.status_code = 404

        mock_get.return_value = response_mock

        result = get_user_list()
        self.assertIsNone(result)
```

This example demonstrates the use of the `@patch` decorator. The decorator can be applied to individual test functions or to the entire test class, which is equivalent to applying the decorator to every test function within the class.

The argument to @patch is the target specification, which specifies the attribute, function or object to be patched as a string representing the import path to the object.

In the example, `user_data.requests.get` specifies the `requests.get` function as the target, which has become a part of the `user_data` module through the import statement at the beginning of the module.

The `@patch` decorator then creates a mock object for the `requests.get` function and passes it as an additional parameter `mock_get` to the test methods.

Within the test method, a mock object is first created to simulate the return value of `requests.get`, and this mock object is then set as the `return_value` for the mocked `requests.get` function.

Please note that the target specified for `@patch` is `user_data.requests.get` and not `requests.get`.

In this example, the latter one would have worked as well but would have violated a general recommendation for patching: Patch where it is used and not where it is defined.

Ned Batchelder has written an excellent article explaining the reason for this rule on his blog at https://nedbatchelder.com. I encourage you to read the article for a deeper understanding of this principle.

https://tinyurl.com/y5k8y3ht

A more complex example is this simple repository, fetching data via network calls from an external source and buffering it in a database *(The database and remote data implementations are omitted as they are not needed for the test.)*:

```
# repository.py
from time import time
from model import Database
from model import RemoteData

_update_interval = 60.0  # seconds


class UserRepository:

    def __init__(self):
        self._database = Database()
        self._remote = RemoteData()
        self._last_update = 0.0

    def get_user_list(self):
        elapsed = time() - self._last_update
        if elapsed > _update_interval:
            self._update_database()

        return self._database.get_user_list()
```

```
    def _update_database(self):
        data = self._fetch_data()
        self._database.save_user_list(data)

        self._last_update = time()

    def _fetch_data(self):
        return self._remote.user_list()
```

To test the repository's behavior at startup in isolation, you need to replace the database and remote access. Though not strictly necessary, you may also want to replace the real system clock with a mocked clock, allowing you to control the time.

Here is an example of how you can do this:

```
from unittest import TestCase
from unittest.mock import patch, Mock

from repository import UserRepository


class TestUserRepository(TestCase):

    @patch("repository.time")
    @patch("repository.UserRepository._fetch_data")
    @patch("repository.Database")
    def test_repository_start(
            self, mock_db, mock_fetch, mock_time):

        current_time = 1709579635
        mock_time.return_value = current_time

        mock_db_inst = Mock()
        mock_db.return_value = mock_db_inst

        data = Mock()
        mock_fetch.return_value = data
        mock_db_inst.get_user_list.return_value = \
            data

        repo = UserRepository()
        user_list = repo.get_user_list()

        mock_fetch.assert_called()
```

```
        mock_db_inst.save_user_list.\
            assert_called_once_with(data)

        self.assertEqual(
            repo._last_update, current_time)

        self.assertEqual(user_list, data)
```

This example demonstrates a few important features of `@patch`:

You can attach more than one `@patch` decorator to a test function or class.

Each decorator adds an additional mock as a parameter to the test function. The decorators are stacked, meaning that the parameters appear in reverse order of the `@patch` statements.

The `@patch("repository.Database")` decorator creates a mock for a class, which is not the same as a mock for an instance of a class. A class mock can be called as a method, simulating the behavior of the `__init__` constructor of the class.

So, these two statements:

```
mock_db_inst = Mock()
mock_db.return_value = mock_db_inst
```

have the effect that `Database()` always returns `mock_db_inst`.

The `save` method of `Database()` is not mocked explicitly in the preparation of the test case. However, it is still possible to check whether or not it has been called using the statement:

```
mock_db_instance.save.\
    assert_called_once_with(data)
```

This is because calling a mock of an instance with any method automatically creates a method mock, which then creates an answer mock and returns it as a result of the method call.

`@patch("repository.UserRepository._fetch_data")` is an example of how to patch an internal method of a class. Since it is an internal function, the patch target is both where it is defined and where it is used. Therefore, the rule for patching targets mentioned earlier is not violated in this case.

Please also note that `@patch` cannot handle private member functions due to name mangling *(Name mangling is explained in Chapter 5: Object-Oriented Programming)*. If you replace the `_fetch_data` method with `__fetch_data`, the corresponding `@patch` statement will throw an `AttributeError` because, due to name mangling, the `__fetch_data` method does not exist at runtime.

As an alternative to the `@patch` decorator, `unittest.mock` provides a function `patch()`, which returns a context manager that can be used with a `with` statement.

Additionally, there is a function named `patch.object()`, which allows you to temporarily patch an attribute or method of an already instantiated object.

This function is especially useful when you want to patch instance variables created in an `__init__` function of a class, as these variables are not yet available at compile time and cannot be accessed with `@patch`.

Using `patch` and `patch.object`, the test class for `UserRepository` can be rewritten as follows:

```
from unittest import TestCase
from unittest.mock import patch, Mock

from repository import UserRepository


class TestUserRepository(TestCase):

    def test_repository_start(self):

        with patch("repository.time") as mock_time:
            current_time = 1709579635
            mock_time.return_value = current_time

            repo = UserRepository()
```

```
with patch.object(repo, "_database") \
        as mock_db, \
     patch.object(repo, "_remote") \
        as mock_rem:

    data = Mock()
    mock_rem.user_list.return_value\
        = data
    mock_db.get_user_list.return_value\
        = data

    user_list = repo.get_user_list()

    mock_rem.user_list.assert_called()

    mock_db.save_user_list.\
        assert_called_once_with(data)

    self.assertEqual(
        repo._last_update, current_time)

    self.assertEqual(user_list, data)
```

Please note the different setup of the test case: Instead of patching the class definitions first and then creating the object under test, the object under test is created first, and then the attributes of the object are patched afterward.

Also note that the statement `patch.object(repo, "_remote")` has the object as a patch target, in contrast to `patch` that has the import path of the class as the target.

15.5 Code Coverage Reports

As in Java, measuring code coverage in Python tests is crucial for understanding how much of your code is exercised by your tests.

There are several tools available for measuring code coverage in Python, with `coverage.py` being among the most popular ones, which will be described in the following.

Before using the tool you need to install the `coverage` package *(Installing packages is explained in Chapter 12: Using External Packages).*

Next, open a terminal and navigate to the directory where your tests are stored. Then, execute the following command:

```
coverage run -m unittest discover
```

This command will execute all tests found in the directory and its subdirectories, generating a binary result file named `.coverage`.

The command

```
coverage report
```

will then output a simple report containing the coverage percentages of all involved modules. Conversely, the command

```
coverage html
```

will create a directory named `htmlcov` that contains a webpage `index.html`, displaying a detailed coverage report.

Chapter 16
Parallelism and Concurrency

Parallelism and concurrency are often used interchangeably, but they represent distinct concepts:
Parallelism involves executing multiple processes simultaneously on multiple CPU cores or processing units. This means that multiple tasks are being performed at the same time, allowing for potentially increased throughput and performance.

Concurrency, on the other hand, refers to the ability of a system to execute multiple tasks within a single process simultaneously. While these tasks may appear to overlap in time, they are often managed by the operating system's scheduler, which switches between tasks rapidly, giving the illusion of simultaneous execution.

In Python programming, parallelism and concurrency play a critical role in building responsive, scalable, and efficient applications, particularly in scenarios involving I/O-bound and CPU-bound tasks:

CPU-bound tasks are those that primarily require processing power or computational resources to complete. These tasks typically involve heavy calculations, mathematical operations, or data manipulation that heavily utilize the CPU. Examples include sorting large datasets, performing complex mathematical computations, or running intensive algorithms.

I/O-bound tasks are those that primarily involve waiting for input/output operations to complete, such as reading from or writing to files, network operations, or interacting with databases. These tasks spend a significant amount of time waiting for external resources to respond and are not heavily CPU-intensive.

Python provides several packages and libraries to handle CPU-bound and I/O-bound tasks efficiently. In the following sections, we will explore the multiprocessing, threading, process / thread pools, and asyncio packages in depth.

16.1 Multiprocessing

Multiprocessing is a technique used in Python to execute multiple processes concurrently, taking advantage of multiple CPU cores or processors. These processes are separate instances of the Python interpreter running in their own memory space.

Starting a process involves defining the task to be executed in a function and passing this function to an instance of the `Process` class provided by Python's built-in `multiprocessing` module, optionally along with additional parameters.

You can start a process using the `start()` method, which launches the process and executes the target function. You can also use methods like `join()` to wait for a process to complete and `terminate()` or `kill()` to terminate a process prematurely.

Here is an example:

```
from multiprocessing import Process


def worker(name, age):
    print(f"{name} is {age} years old")


if __name__ == "__main__":
    args = ("Alice", 30)

    process = Process(target=worker, args=args)

    process.start()
    process.join()

#  Alice is 30 years old
```

Not all objects can be passed as process parameters in Python. Python uses a process called pickling to serialize and deserialize objects, allowing them to be passed as arguments to processes.

Pickling involves converting Python objects into a byte stream that can be transmitted across processes or stored in files, and then reconstructing the original objects from the byte stream.

However, not all objects can be pickled. For example, objects that represent open file handles, sockets, or database connections typically cannot be pickled. Each process needs to create these objects separately.

Please note that in the context of multiprocessing, each process created by `Process` will import the script module, including all its global variables and top-level code. Without the `if __name__ == "__main__":` guard, the top-level code would be executed in each child process again, leading to an infinite recursion of child process creation.

16.2 Inter-Process Communication

Multiple processes cannot communicate directly using global attributes because each process has its own separate memory space. Therefore, changes made to global attributes in one process are not visible to other processes.

For example, the following code will repeatedly print 42 for ten seconds and ignore any changes made to the global variable, which is set to 0:

```
from multiprocessing import Process
from time import sleep

i = 42


def worker():
    while True:
        print(i)


if __name__ == "__main__":

    process = Process(target=worker)

    process.start()
    i = 0
    sleep(10)
```

```
    process.kill()

# 42
# 42
# ...
```

Thus, Inter-Process Communication (IPC) is crucial for enabling communication between processes in a multiprocessing environment.

Python offers various IPC mechanisms, of which we will explore the basics of the following three:

Queues are thread and process-safe data structures used for communication between processes. They support multiple producers and consumers.

Pipes allow two processes to communicate by creating a pipe, which acts as a bidirectional communication channel. One process writes data to the pipe, and the other process reads data from the pipe.

Locks allow only one process at a time to acquire access to a shared resource, such as a log file.

16.2.1 Queues

The `Queue` class provided by the multiprocessing package facilitates smooth coordination among multiple processes, adhering to the FIFO (First-In, First-Out) principle. This enables one or more processes to produce and enqueue items, while others dequeue and consume them.

Here is an example *(In all examples in this chapter, the* `sleep()` *function is used as a placeholder for a blocking CPU-bound or I/O-bound task.)* where the main process distributes 15 jobs to a pool of three worker processes:

```
from multiprocessing import Process, Queue
from time import sleep


def worker(number, input_queue):
    while True:
        item = input_queue.get()
```

```
        if item == "Terminate":
            break
        print(f"Worker {number}: {item}")
        sleep(1)


if __name__ == "__main__":
    q = Queue()

    process_1 = Process(target=worker, args=(1, q))
    process_2 = Process(target=worker, args=(2, q,))
    process_3 = Process(target=worker, args=(3, q,))
    process_1.start()
    process_2.start()
    process_3.start()

    for i in range(15):
        q.put(i)
    q.put("Terminate")
    q.put("Terminate")
    q.put("Terminate")

    process_1.join()
    process_2.join()
    process_3.join()

# Worker 1: 0
# Worker 2: 1
# Worker 3: 2
# ...
```

16.2.2 Pipes

The `Pipe` class provided by the `multiprocessing` package facilitates two-way communication between two processes.

Leveraging lower-level operating system features, communication via pipes is typically significantly faster compared to queues.

Here is an example where two processes send the words "Ping" and "Pong" back and forth:

```
from multiprocessing import Process, Pipe
from time import sleep
```

```
def worker(my_pipe, ping_or_pong, start):
    if start:
        my_pipe.send(ping_or_pong)

    while True:
        word = my_pipe.recv()
        print(word)

        my_pipe.send(ping_or_pong)


if __name__ == "__main__":
    pipe_ends = Pipe()

    args_1 = (pipe_ends[0], "Ping", True)
    process_1 = Process(target=worker, args=args_1)

    args_2 = (pipe_ends[1], "Pong", False)
    process_2 = Process(target=worker, args=args_2)

    process_1.start()
    process_2.start()

    sleep(5)

    process_1.kill()
    process_2.kill()

# Ping
# Pong
# ...
```

16.2.3 Locks

The `multiprocessing` module provides a `Lock` class, which can be utilized to coordinate access to shared resources, like log files, among multiple processes.

In addition to providing `acquire()` and `release()` methods for this purpose, a `Lock` object can also be used as a context manager using the `with`

statement. This context manager automatically acquires and releases the lock when entering and exiting the `with` block, respectively.

Here is an example where a lock is used to ensure that the two-line outputs of several processes are kept together as groups:

```
from multiprocessing import Process, Lock
from time import sleep


def worker(number, print_lock):
    for i in range(5):
        with print_lock:
            print(f"Worker {number}")
            sleep(1)
            print(f"i = {i}")


if __name__ == "__main__":
    lock = Lock()

    processes = [
        Process(target=worker, args=(i, lock))
        for i in range(5)
    ]

    for process in processes:
        process.start()

    for process in processes:
        process.join()

# Worker 0
# i = 0
# Worker 2
# i = 0
# ...
```

16.2.4 Shared Memory

In addition to the communication mechanisms described here, shared memory is another concept used for IPC.

Shared memory allows multiple processes to share a region of memory, enabling them to exchange data more efficiently than other IPC methods like pipes or queues.

The shared memory implementations provided by the `multiprocessing` module rely on C Programming Language data types and byte streams.

While shared memory can offer high performance and efficiency in certain scenarios, it also introduces challenges related to synchronization, data consistency, and platform compatibility. As such, it will not be described in this book.

16.3 Threading

Just like in Java, threading in Python allows tasks to run concurrently within a single process. *(An important difference between Java and Python threading will be explained in the next section.)*

16.3.1 Creating and Running Threads

Python's built-in `threading` module provides a high-level interface for working with threads. Threads can be created by subclassing the `Thread` class and overwriting its `run()` method or by passing a target function to the `Thread` constructor. Once created, threads can be started, paused, resumed, and terminated using methods provided by the Thread class. Similar to processes, threads also have a `join()` function that allows you to wait for the thread to terminate.

Here is an example of a thread running a countdown timer:

```
from threading import Thread


class MyCountDownThread(Thread):
    def __init__(self, start):
        super().__init__()
        self.counter = start

    def run(self):
        while self.counter > 0:
```

```
            print(f"Counter = {self.counter}")
            self.counter -= 1

        print("Count Down Finished!")


def print_hello_world():
    print("Hello World!")


thread_1 = MyCountDownThread(100)
thread_1.start()
thread_1.join()

thread_2 = Thread(target=print_hello_world())
thread_2.start()
thread_2.join()

# Counter = 100
# ...
# Counter = 1
# Count Down Finished!
# Hello World!
```

16.3.2 Thread Synchronization Techniques

With threads, there is no need for Inter-Process Communication mechanisms like pipes or queues because threads within the same process share the same memory space. This means that data can be shared directly between threads without the overhead of communication between separate processes.

However, there is still a need for synchronizing access to shared data to prevent race conditions and ensure data integrity.

The `threading` package in Python offers a wide range of features for thread synchronization. In the following we will explore a few of these synchronization techniques and demonstrate how they can be used to manage concurrent access to shared data.

Locks

Locks behave similarly to the locks provided by the `multiprocessing` package.

Here is an example of synchronizing several threads writing to the same file:

```
from threading import Thread, Lock
from time import sleep

lock = Lock()


class FileWriter(Thread):
    def __init__(self, writer_id, output_file):
        super().__init__()
        self.id = writer_id
        self.file = output_file

    def run(self):
        for i in range(5):
            with lock:
                self.file.write(
                    f"Writer {self.id}: {i}\n"
                )
            sleep(1)


with open("my_file.txt", "w") as file:

    threads = [
        FileWriter(i, file)
        for i in range(5)
    ]

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

# my_file.txt:
# Writer 0: 0
# Writer 1: 0
# Writer 2: 0
# ...
```

Semaphores

Semaphores in Python work much like they do in Java. They are a way to control access to a shared resource by allowing only a certain number of threads to use it at the same time.

In addition to the `acquire()` and `release()` methods provided by the class `Semaphore`, a `Semaphore` object can also be used as a context manager with a `with` statement. This context manager automatically acquires a permit at the beginning and releases it at the end of the `with` block.

Here is an example of a pool of single-use threads where a maximum of three threads are allowed to run simultaneously:

```
from threading import Thread, Semaphore
from time import sleep

semaphore = Semaphore(3)

class Worker(Thread):
    def __init__(self, worker_id):
        super().__init__()
        self.id = worker_id

    def run(self):
        with semaphore:
            print(f"Worker: {self.id}")
            sleep(3)


threads = [
    Worker(i)
    for i in range(50)
]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

# Worker: 0
# Worker: 1
# Worker: 2
```

```
# ...
```

Condition Variables

Similar to Java, condition variables in Python provided by the `Condition` class, are synchronization primitives used to coordinate the execution of threads based on certain conditions. They are typically used in conjunction with locks, integrated into the condition variable itself, to control access to shared resources.

Threads can wait on a condition variable using the `wait()` method. When a thread calls `wait()`, it releases the associated lock and enters a blocked state until another thread notifies it.

Threads can be notified using the `notify()` or `notify_all()` methods.

When a thread calls `notify()`, it wakes up one waiting thread that is blocked on the condition variable. If multiple threads are waiting, it is not deterministic which thread will be awakened.

The `notify_all()` method wakes up all waiting threads.

After being awakened, the thread must reacquire the associated lock before it can proceed.

In addition to providing the `wait()`, `notify()`, and `notify_all()` methods, a `Condition` object can also be used as a context manager using the `with` statement. This context manager automatically acquires and releases the associated lock when entering and exiting the block of code, respectively.

It is important to note that `wait()`, `notify()`, and `notify_all()` must be called from within the block of a `with` statement.

The code below provides an example of using a condition variable, where a single producer irregularly produces items, requiring consumers to wait for new items.

```
from threading import Thread, Condition
from time import sleep
```

```
buffer = []

item_available = Condition()
signal_term = "Terminate"


class Producer(Thread):
    def run(self):
        for item in range(100):
            if item % 10 == 0:
                sleep(5)
            with item_available:
                print(f"Producer: Item {item}")
                buffer.insert(0, item)
                item_available.notify()

        with item_available:
            buffer.insert(0, signal_term)
            item_available.notify_all()


class Consumer(Thread):
    def __init__(self, consumer_id):
        super().__init__()
        self.id = consumer_id

    def run(self):
        while True:
            with item_available:
                if not buffer:
                    item_available.wait()
                item = buffer.pop()

                if item == signal_term:
                    buffer.insert(0, signal_term)
                    return

            print(f"Consumer {self.id}: Item {item}")
            sleep(5)


consumer_threads = [
    Consumer(consumer_id)
    for consumer_id in range(5)
]

for consumer_thread in consumer_threads:
```

```
    consumer_thread.start()

producer_thread = Producer()
producer_thread.start()

producer_thread.join()
for consumer_thread in consumer_threads:
    consumer_thread.join()

# Producer: Item 0
# ...
# Producer: Item 9
# Consumer 0: Item 0
# Consumer 1: Item 1
# ...
```

Event Objects

Event objects are synchronization primitives used to communicate between threads by signaling when a certain event has occurred. Threads can wait for an event to be set by another thread and then proceed accordingly.

Threads can call the wait() method to wait for the event to be set. If the event is already set, the thread continues execution immediately. If not, it waits until another thread sets the event using the set() method.

The set() method sets the event, allowing threads waiting for the event to proceed.

The clear() method resets the event, indicating that it is not set. Threads calling wait() after the event has been cleared will block until another thread sets the event again.

Here is an example where a thread provides a "printer access" after some delay:

```
from threading import Thread, Event
from time import sleep

printer_available = Event()

printer = None
```

```
class PrinterCreator(Thread):

    def run(self):
        sleep(3)

        global printer
        printer = lambda x: print(f"Printer: {x}")

        printer_available.set()


PrinterCreator().start()

printer_available.wait()
printer("Hello World")

# Printer: Hello World
```

16.4 Managing Concurrency with Pools

The `concurrent` package in Python offers a range of tools for managing concurrency and parallelism at a lower, and sometimes more platform-dependent, level. Providing a comprehensive description of this package is beyond the scope of this book.

However, we will focus on key components such as process and thread pools, as well as futures, provided by its `concurrent.futures` subpackage.

Thread and process pools are an important tool for managing concurrency because they pre-allocate a pool of threads or processes that remain alive and ready to execute multiple tasks until the pool is explicitly shut down. This approach avoids the overhead of creating and destroying threads or processes for each individual task.

They are similar to Java's ExecutorService interface but, unlike Java, support not only thread pools but also process pools.

Here is an example of using a process pool:

```
from concurrent.futures import ProcessPoolExecutor
from time import sleep


def square(x):
    sleep(2)
    return x * x


if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=2)\
            as executor:

        futures = [executor.submit(square, i)
                   for i in range(1, 6)]

        print("Everything submitted")

        for future in futures:
            result = future.result()
            print("Result:", result)

# Everything submitted
# Result: 1
# Result: 4
# Result: 9
# Result: 16
# Result: 25
```

Please note that the `if __name__ == "__main__":` guard is necessary here for the same reasons as explained in the multiprocessing examples.

The statement

```
ProcessPoolExecutor(max_workers=2)
```

creates a `ProcessPoolExecutor` object, which manages a pool of 2 worker processes for executing tasks. This executor can be used as a context manager within a `with` statement, handling the setup and teardown of the process pool.

Using the `submit` method of a `ProcessPoolExecutor` object, you can asynchronously submit a callable (like a function or lambda) along with its

arguments for execution in one of the worker processes.

If all worker processes are busy when a new task is submitted, it is queued until a worker becomes available. Once a worker is free, it picks up the next task from the queue and starts processing it.

The `submit` method returns a `Future` object immediately, representing the result of the asynchronous computation. This `Future` object can be used to monitor the task's status and retrieve its result once it is completed.

The most important methods of a `Future` object are:

`done()`: Returns `True` if the future has completed its computation or been cancelled, `False` otherwise.

`result()`: Returns the result of the computation if it is available. If the computation has not yet completed, this method will block until the result is available or until the optional `timeout` (in seconds) expires.

`exception()`: Returns the exception raised by the computation if it has failed. If the computation has not yet completed or has not raised an exception, this method will block until an exception is raised or until the optional `timeout` (in seconds) expires.

`cancel()`: Attempts to cancel the execution of the computation. Returns `True` if the cancellation was successful, `False` otherwise.

`add_done_callback(<callback function>)`: Adds a callback function to be called when the future completes its computation or is cancelled. The callback function will be executed in one of the processes from the process pool with the future object as its sole argument.

The above code functions similarly when

```
ProcessPoolExecutor(max_workers=2)
```

is replaced with

```
ThreadPoolExecutor(max_workers=2),
```

except that now all tasks are executed within threads running in the same process on a single CPU core.

The explanation for why thread pools in Python, unlike Java, cannot achieve true parallelism is provided in the following section.

16.5 Global Interpreter Lock (GIL)

The Global Interpreter Lock (GIL) in Python ensures that only one thread can execute Python bytecode at a time, even on multi-core systems with multiple CPU cores, in contrast to Java's concurrency model. Here is a detailed explanation of how it operates:

1. Thread Execution:
   When a Python program starts, the interpreter initializes the GIL, which acts as a mutex to control access to Python objects and memory.

   As the program executes, multiple threads may be created to perform concurrent tasks, such as I/O operations, computation, or data processing.

   Each thread operates independently and can execute Python bytecode instructions to perform its assigned tasks.
2. GIL Acquisition:
   When a thread wants to execute Python bytecode, it attempts to acquire the GIL.

   If the GIL is available (i.e., no other thread currently holds it), the thread acquires the GIL and proceeds with executing bytecode.

   If the GIL is already held by another thread, the requesting thread must wait until the GIL is released.
3. Exclusive Execution:
   Once a thread holds the GIL, it has exclusive access to Python objects and memory, allowing it to execute Python bytecode without

interference from other threads.

4. GIL Release:
   When a thread completes its task or reaches a point where it can voluntarily release the GIL, such as during I/O operations or long-running computations that do not require the GIL, it releases the lock to allow other threads to acquire it.

   For example, a thread performing file I/O operations may voluntarily release the GIL while waiting for data to be read from or written to a file.

   Similarly, a thread executing a time-consuming mathematical computation that does not involve shared Python objects may release the GIL to allow other threads to execute Python bytecode concurrently.

   This is typically handled automatically by the Python interpreter.

   Releasing the GIL allows other waiting threads to acquire it and continue executing Python bytecode, enabling concurrent execution of multiple threads.

5. Interleaved Execution:
   The Python interpreter includes a scheduler for managing the interleaved execution of threads. However, this scheduler operates within the constraints of the GIL, which allows only one thread to execute Python bytecode at a time.

   Despite this limitation, the scheduler ensures that all threads have a chance to acquire the GIL and execute their bytecode in an interleaved manner.

   The scheduler can also switch to another waiting thread, enabling it to execute its bytecode when a thread releases the GIL, such as during I/O operations or long-running computations.

All in all, this means that while multiple threads in Python can run concurrently, their performance is constrained by the Global Interpreter

Lock (GIL), which limits their ability to effectively utilize multiple CPU cores. This limitation significantly hinders parallelism and scalability, particularly in CPU-bound workloads.

As previously discussed in the section on multiprocessing, one effective approach to overcome the limitations imposed by the GIL in Python is to leverage multiprocessing instead of multithreading.

By utilizing separate processes, each with its own Python interpreter and GIL, multiprocessing enables true parallelism across multiple CPU cores without being constrained by the GIL.

Another approach to overcoming the limitations imposed by the GIL is through cooperative multitasking, which will be described in the next section.

16.6 Asynchronous Programming

16.6.1 Overview

Asynchronous programming is a programming paradigm that enables concurrent execution of multiple tasks without the need for explicit threading or multiprocessing.

`asyncio` is a Python library that simplifies asynchronous programming by providing high-level abstractions for managing asynchronous tasks.

It consists of the following components:

Coroutines: Special functions that run asynchronously within a thread. They are defined using the `async` keyword, indicating that the function is asynchronous and can be awaited for completion. Coroutines can start other coroutines and await their results.

Event loop: Coordinates the scheduling and execution of coroutines. Each thread typically has exactly one associated event loop, and coroutines scheduled by the event loop run within that thread.

Tasks: Objects that represent the execution of coroutines within the event loop. Tasks allow you to manage the execution of coroutines and await their results asynchronously.

Executors: Objects that manage the execution of blocking or CPU-bound tasks in separate threads or processes, allowing the event loop to remain responsive. Executors are typically created within a coroutine that asynchronously waits for the executor to finish and return a result.

Coroutines are lightweight concurrency units that can be cooperatively scheduled within a single thread. They typically have lower overhead compared to threads, as they don't involve context switching or OS-level thread management.

With the help of executors, coroutines provide a convenient way to work around the limitations imposed by the GIL for I/O-bound or CPU-bound tasks.

While `asyncio` offers many features for async programming, its full depth is beyond the scope of this book. However, in the next sections, you will get a good overview, so you will understand how asyncio works and how to use it effectively.

16.6.2 Creating and Running Coroutines

Creating just one coroutine and running it is very simple with `asyncio`.

Here is an example:

```
import asyncio


async def my_coroutine():
    print("Coroutine started")
    await asyncio.sleep(1)
    print("Coroutine resumed after 1 second")

    return 42

result = asyncio.run(my_coroutine())
print(result)
```

```
# Coroutine started
# Coroutine resumed after 1 second
# 42
```

The keywords `async def` mark the function `my_coroutine` as a coroutine, indicating that it cannot be directly invoked but must be scheduled for execution by the event loop.

Using `asyncio.run(my_coroutine())` initiates the creation of an event loop for the current thread. It then generates a task for `my_coroutine`, allows the event loop to begin executing the task, waits for its completion, and finally returns its result.

Please note that while `asyncio.sleep()` suspends the execution of the coroutine for the specified duration, allowing other coroutines to run in the meantime, using `time.sleep()` instead would have blocked the entire thread, preventing the event loop from processing other coroutines during this period.

The following example demonstrates a main coroutine initiating another coroutine, awaiting its completion, and then initiating the next coroutine.

```
import asyncio


async def my_coroutine(number, sleep_time):
    print(f"{number}: Falling asleep ...")
    await asyncio.sleep(sleep_time)

    print(f"{number}: Waking up ...")


async def main():
    await my_coroutine(1, 2)
    await my_coroutine(2, 8)

asyncio.run(main())

# 1: Falling asleep ...
# 1: Waking up ...
# 2: Falling asleep ...
# 2: Waking up ...
```

`await` creates a task from a coroutine, schedules it into the event loop of the thread, and then waits for its completion, eventually returning its result.

In this modification of the last example, the coroutines run concurrently, meaning the second coroutine is already scheduled while the first coroutine is suspended.

```
import asyncio


async def my_coroutine(number, sleep_time):
    print(f"{number}: Falling asleep ...")
    await asyncio.sleep(sleep_time)

    print(f"{number}: Waking up ...")
    return number


async def main():
    result = await asyncio.gather(
        my_coroutine(1, 2),
        my_coroutine(2, 8)
    )

    print(result)

asyncio.run(main())

# 1: Falling asleep ...
# 2: Falling asleep ...
# 1: Waking up ...
# 2: Waking up ...
# [1, 2]
```

`asyncio.gather`, similar to `await`, creates tasks for every coroutine passed, schedules them into the event loop of the thread, waits for the completion of all tasks, and returns their results in a list.

Finally, here is an example demonstrating the explicit creation of tasks:

```
import asyncio


async def my_coroutine(number, sleep_time):
    print(f"{number}: Falling asleep ...")
    await asyncio.sleep(sleep_time)

    print(f"{number}: Waking up ...")
    return number


async def main():
    task_1 = asyncio.create_task(my_coroutine(1, 3),
                                 name="Task 1")
    task_2 = asyncio.create_task(my_coroutine(2, 10),
                                 name="Task 2")
    task_3 = asyncio.create_task(my_coroutine(3, 10),
                                 name="Task 3")

    done, pending = await asyncio.wait(
        [task_1, task_2, task_3],
        return_when=asyncio.FIRST_COMPLETED)

    for task in done:
        name = task.get_name()
        result = task.result()
        print(f"Done: {name}, result = {result}")

    for task in pending:
        print(f"Pending: {task.get_name()}")

    task_2.cancel()
    task_3.cancel()

asyncio.run(main())

# 1: Falling asleep ...
# 2: Falling asleep ...
# 3: Falling asleep ...
# 1: Waking up ...
# Done: Task 1, result = 1
# Pending: Task 3
# Pending: Task 2
```

`asyncio.create_task` not only creates a task but also submits it to the event loop for processing.

`asyncio.wait` is a coroutine function that waits until a certain condition is fulfilled for a list of tasks. The default condition is "all tasks completed". In this example, the condition is already fulfilled when at least one task is completed.

The result of `asyncio.wait` is a tuple of two `Future` sets for the completed and not yet completed tasks.

Finally the main() coroutine cancels the two remaining tasks prematurely.

Up to this point, our examples have primarily relied on the implicit creation of event loops by functions such as `asyncio.run()`. However, there are scenarios where explicitly creating an event loop can be beneficial.

One reason is the need for more fine-grained control over the event loop's behavior and execution. Moreover, explicit event loop creation allows developers to implement custom monitoring, tracing, and debugging functionalities tailored to their specific requirements. Additionally, explicit event loop management can be useful in situations where multiple event loops need to coexist or when integrating `asyncio` with other event loop frameworks.

While providing examples for all potential use cases of explicit event loop creation is beyond the scope of this book, we will illustrate one example to demonstrate its practical application.

```
import asyncio


async def worker(worker_id):
    if worker_id >= 0:
        await asyncio.sleep(1)
        print(f"Worker {worker_id}: Finished")
    else:
        raise ValueError()


async def main():
    tasks = [
        asyncio.create_task(
            worker(i), name=f"Worker: {i}")
```

```
        for i in [1, -1, 2, -2, 3, -3]
    ]

    await asyncio.wait(tasks)


def custom_exception_handler(loop, context):
    task_name = context["future"].get_name()

    print(f"{task_name} raised an exception")


loop = asyncio.new_event_loop()
loop.set_exception_handler(custom_exception_handler)

loop.run_until_complete(main())

# Worker 1: Finished
# Worker 2: Finished
# Worker 3: Finished
# Worker: -3 raised an exception
# Worker: -2 raised an exception
# Worker: -1 raised an exception
```

In this example, a brief list of tasks is generated, with some tasks designed to raise exceptions.

A new event loop is initialized using `asyncio.new_event_loop()`, and a custom exception handler, which prints the name of the task that triggered the exception, is bound to the event loop.

Finally, the loop is started with `run_until_complete()`, which processes all tasks until completion.

16.6.3 Executors

In asynchronous programming with `asyncio`, coroutines are the building blocks of concurrent execution. Unlike threads, where the operating system handles context switching and scheduling, with coroutines, it is the responsibility of the programmer to ensure that a coroutine relinquishes control to the event loop in a timely manner.

Failure to do so can lead to blocking other coroutines, potentially causing delays or even deadlock in the application.

Here is an example where a coroutine inadvertently blocks the event loop, leading to degraded performance *(While there are faster ways to find Fibonacci numbers, we picked for demonstration purposes a method that requires a lot of calculations.)*:

```
import asyncio


def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)


async def fibonacci_async(n):
    result = fibonacci(n)
    print(f"fib({n}) = {result}")


async def hello():
    print("Hello!")


async def main():
    tasks = [
        asyncio.create_task(fibonacci_async(36)),
        asyncio.create_task(hello())
    ]

    await asyncio.wait(tasks)

asyncio.run(main())

# fib(36) = 14930352
# Hello!
```

Despite initiating both tasks consecutively, the second task begins only after the Fibonacci number calculation is complete.

A straightforward approach would be to transform the Fibonacci number calculation into a coroutine and incorporate an `asyncio.sleep()` statement within the recursive section of the calculations. This allows the coroutine to relinquish control back to the event loop periodically.

```
async def fibonacci(n):
    if n <= 1:
        return n
    else:
        await asyncio.sleep(0.001)  # 1ms sleep
        f_1 = await fibonacci(n-1)
        f_2 = await fibonacci(n-2)
        return f_1 + f_2
```

The drawback of this method is that the calculation process is noticeably slowed down, not only due to the delayed execution but also because it generates a large number of new tasks for the event loop.

A much better solution for this problem is offered by `asyncio`, which has the capability to delegate synchronous operations to separate threads or processes in a thread or process executor pool.

Here is an example:

```
import asyncio
from concurrent.futures import ProcessPoolExecutor


def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)


async def fibonacci_async(n):
    event_loop = asyncio.get_event_loop()

    with ProcessPoolExecutor(max_workers=1)\
            as executor:
        result = await event_loop.run_in_executor(
            executor, fibonacci, n)

        print(result)
```

```
async def hello():
    print("Hello!")


async def main():

    tasks = [
        asyncio.create_task(fibonacci_async(36)),
        asyncio.create_task(hello())
    ]

    await asyncio.wait(tasks)

if __name__ == "__main__":
    asyncio.run(main())

# Hello!
# 14930352
```

To execute a function in an executor, you use the `run_in_executor` method of the event loop. This method takes the executor pool to use, the function to run, and any parameters for the function.

The `run_in_executor` method then submits the function to the executor pool. The pool selects an idle thread or process to execute the function with the given parameters.

Note that the `run_in_executor` method accepts only positional and no keyword parameters. Additionally, you can set the parameter for the executor pool to `None`, in which case the default process pool of the event loop is used. By default, this is a thread pool, but it can be overwritten using the `set_default_executor()` method of the event loop.

16.6.4 Coroutine Synchronization

While coroutines themselves don’t inherently introduce race conditions, they often work with shared resources or perform I/O operations where coordination is necessary to prevent conflicts. The `Queue`, `Lock`, `Semaphore`,

and `Event` provided by `asyncio` help manage concurrent access to shared resources among coroutines.

Thread synchronization primitives like `Lock` and `Semaphores` block the entire thread until they are acquired or released. If used with coroutines, they may inadvertently block the event loop, preventing other coroutines from executing concurrently, defeating the purpose of asynchronous programming.

Therefore, `asyncio` provides its own set of synchronization primitives tailored for coroutine-based asynchronous programming.

Here is an example rewritten for coroutines, based on the example of using events from the thread synchronization section:

```
import asyncio

buffer = []

item_available = asyncio.Event()
signal_term = "Terminate"


async def producer():
    for item in range(100):
        if item % 10 == 0:
            item_available.set()
            await asyncio.sleep(5)

        print(f"Producer: Item {item}")
        buffer.insert(0, item)

    buffer.insert(0, signal_term)
    item_available.set()


async def consumer(consumer_id):
    while True:
        if buffer:
            item = buffer.pop()

            if item == signal_term:
                buffer.insert(0, signal_term)
                return
```

```
            print(f"Consumer {consumer_id}: Item {item}")
            await asyncio.sleep(5)
        else:
            item_available.clear()
            await item_available.wait()


async def main():
    tasks = [asyncio.create_task(producer())] + [
        asyncio.create_task(consumer(consumer_id))
        for consumer_id in range(12)
    ]

    await asyncio.gather(*tasks)

asyncio.run(main())

# Producer: Item 0
# ...
# Producer: Item 9
# Consumer 0: Item 0
# Consumer 1: Item 1
# ...
```

Note that with coroutines, there is no need to protect access to the buffer list with a lock. Unlike threads, neither the producer nor the consumer can be interrupted during access to the list due to a preemptive scheduler in a single-threaded process.

16.6.5 Asynchronous Context Managers

Asynchronous context managers are classes that can be used with an `async with` statement.

In contrast to synchronous context managers, they must implement the coroutines `__aenter__` and `__aexit__` instead of the `__enter__` and `__exit__` methods.

These coroutines are awaited by the `async with` statement, allowing for non-blocking execution during setup and cleanup.

Here is an example:

```
import asyncio


class MyAsyncContextManager:

    data = [1, 2, 3, 4, 5]

    async def __aenter__(self):
        print("__aenter__ called")
        return self

    async def __aexit__(self, exc_type,
                        exc_value, traceback):
        print("__aexit__ called")
        self.data = None


async def main():
    async with MyAsyncContextManager() as manager:
        print(manager.data)

asyncio.run(main())

# __aenter__ called
# [1, 2, 3, 4, 5]
# __aexit__ called
```

Chapter 17
HTTP Requests

Python provides several libraries for making HTTP requests, each with its own strengths and features. One of the most commonly used libraries is `requests`, which will be covered in this chapter.
Popular alternatives include `urllib` and `http.client`.

17.1 GET Requests

Making an HTTP GET request using the `get` function from the `requests` package is quite simple.

Here is an example *(Reqres.in is a REST-API service for mocking HTTP responses used in testing and prototyping applications.)*:

```
import requests

url = "https://reqres.in/api/users/1"

try:
    response = requests.get(url)

    if response.ok:
        print(f"Content Type: {
            response.headers["content-type"]}"
        )
        print(f"Contents: {response.text}")
    else:
        print(f"Error: {response.status_code}")
except Exception as ex:
    print(f"An exception occurred: {ex}")

# Content Type: application/json; charset=utf-8
# Contents: {"data":{"id":1, ...
```

The `get()` function of the `requests` module returns a `Response` object containing all data retrieved from the HTTP request in a structured format. The most important attributes of `Response` include:

`ok` Returns `True` if HTTP status code is less than 400, `False` otherwise.
`status_code:` The HTTP status code of the response.
`headers:` A dictionary containing the response headers.
`text:` The content of the response, in Unicode format.
`content:` The content of the response, in bytes.
`request:` An object providing information about the HTTP request
`url:` The URL of the response.

The `get` function in the requests library may raise exceptions under certain circumstances. Here are three of the most important exceptions that the `get` function may raise:

`ConnectionError:` Raised when a network connection error occurs, such as a DNS resolution failure, a refused connection, or a timeout.

`Timeout:` Raised when the request takes longer than the specified timeout period to complete.

`TooManyRedirects:` Raised when the request encounters too many redirects.

If you want to handle responses with an HTTP status code different from 200 by raising exceptions as well, you can call the `raise_for_status()` method of the `Response` class. This method raises a `HTTPError` exception if the response status code indicates an error.

17.2 Parsing JSON Responses

The `json()` method of the `Response` class parses the content of the response as a JSON string into a Python object using Python's built-in `json.loads()` function.

This function maps JSON data types to corresponding Python data structures as follows:

JSON object ({}): Mapped to a Python dictionary.
JSON array ([]): Mapped to a Python list.
JSON string: Mapped to a Python string.

JSON number (integer or float): Mapped to a Python `int` or `float`.
JSON true: Mapped to Python `True`.
JSON false: Mapped to Python `False`.
JSON null: Mapped to Python `None`.

Here is an example:

```
import requests

url = "https://reqres.in/api/users/1"

response = requests.get(url)

user_data = response.json()

print(
    user_data["data"]["first_name"],
    user_data["data"]["last_name"],
)

# George Bluth
```

Please note that the `requests` package does not include built-in support for parsing XML structured content.

Instead, you can parse XML contents using the built-in `xml` library, which provides SAX and DOM interfaces for XML parsing. Alternatively, you can utilize third-party libraries such as `ElementTree`.

However, detailed coverage of these parsing techniques is beyond the scope of this book.

17.3 Reading Streamed Contents

The `Response` object can also be used as a context manager in a `with` statement, which is useful when utilizing the `iter_content()` method to read the content sent by the HTTP server in chunks. This can be particularly helpful when downloading a large file.

Here is an example:

```
import requests

url = "http://speedtest.tele2.net/10MB.zip"
chunk_size = 1024

with requests.get(url, stream=True) as response:
    with open('large_file.zip', 'wb') as file:

        for chunk in response.iter_content(
                chunk_size=chunk_size
        ):
            print(f"Read {len(chunk)} bytes")
            file.write(chunk)

# Read 1024 bytes
# Read 1024 bytes
# Read 1024 bytes
# ...
```

In this example, the `stream` parameter of the `get` method is set to `True` to prevent reading the entire content of the response into memory at once. Instead, it allows the content to be iteratively read in smaller chunks using the `iter_content()` method.

17.4 Using Query Parameters

To include query parameters in a GET request, you can pass them as a dictionary to the `params` parameter of the `get` function.

Here is an example *(The examples below utilize* `httpbin.org`*, a testing site designed to mirror various HTTP requests, serving as a developer tool.)*:

```
import requests

url = "https://httpbin.org/get"

params = {
    "name": "Zaphod",
    "job": "President",
}

response = requests.get(url, params=params)

print(response.request.url)
```

```
print(response.json()["args"])

# https://httpbin.org/get?name=Zaphod&job=President
# {'job': 'President', 'name': 'Zaphod'}
```

17.5 Adding Custom Headers to Requests

Custom headers can be easily added to an HTTP request by passing a dictionary that maps header names to their respective values as an additional keyword parameter called `headers`.

Here is an example:

```
import requests

url = "https://httpbin.org/headers"

custom_headers = {
    "User-Agent": "CustomUserAgent",
    "Accept": "application/json",
}

response = requests.get(url, headers=custom_headers)

print(response.request.headers)

# {'User-Agent': 'CustomUserAgent',
# 'Accept-Encoding': 'gzip, deflate',
# 'Accept': 'application/json',
# 'Connection': 'keep-alive'}
```

Please note that the request contains additional headers like `Accept-Encoding` that have been added by the `requests` package.

17.6 POST Requests

For sending an HTTP POST request, the `requests` package provides a function called `post()`.

Here is an example that sends a POST request where the body has the format used by HTML forms:

```
import requests

url = "https://reqres.in/api/users"

data = {
    "name": "Arthur",
    "job": "Traveller"
}

response = requests.post(url, data=data)

print(response.text)

# {"name":"Arthur","job":"Traveller","id":"307" ...
```

The `post()` function supports various forms for the body of the request.

Here are two frequently used examples:

JSON Data: If you need to send JSON data, you can use the `json` parameter. This will automatically serialize the data to JSON format.

```
json_data = {"key1": "value1", "key2": "value2"}
response = requests.post(url, json=json_data)
```

Raw Data: For sending raw data in the body of the request, you can use the `data` parameter with a bytes or string object.

```
raw_data = b"raw data"
response = requests.post(url, data=raw_data)
```

To include a file in the body of a POST request, you can first read its contents into a variable and then send the variable's contents as raw data, following the method just described.

If you want to include files to be uploaded alongside the form data in the body of a POST request, you can utilize the `files` parameter. In its simplest form the `files` parameter is a dictionary mapping the form names of the files to upload to their contents in the form of an already openend file.

Here is an example:

```
import requests

url = "http://httpbin.org/post"

data = {
    "number_of_reports": 2
}

with open("report1.csv", "rb") as file1, \
        open("report2.csv", "rb") as file2:
    files = {
        "report1": file1,
        "report2": file2
    }

    r = requests.post(url, data=data, files=files)
    print(r.text)

# ...
#   "files": {
#     "report1": "John;Doe;42\r\nJane;Doe;56\r\n",
#     "report2": "John;Doe2;42\r\nJane;Doe2;56\r\n"
#   },
#   "form": {
#     "number_of_reports": "2"
#   },
# ...
```

It is possible to provide additional parameters like a different file name to be used by the HTTP request in a tuple passed as the value of the dictionary entries, but this aspect will not be covered here.

Also, note that the `post()` function does not automatically close the files. This responsibility falls on the calling code, which is managed here using a `with` statement.

To conclude this section, it is important to note that all the functionalities described for the `get()` function, such as parsing JSON responses, also apply to the `post()` function.

17.7 Miscellaneous Topics

In this final section of the chapter, we will cover various miscellaneous topics related to HTTP requests.

17.7.1 Other HTTP Methods

For each of the HTTP methods DELETE, HEAD, OPTIONS, PATCH, PUT, the requests package has a corresponding function that can be used in the same way as the already described functions `get()` and `post()`.

17.7.2 SSL Certificates

If you want to use locally stored SSL certificates with the requests library, you can pass the path to the certificate file as the value of the `cert` parameter when making the request. This parameter expects a string representing the path to the certificate file.

17.7.3 Authentication

The `requests` package offers extensive support for request authentication. However, providing an in-depth explanation of all its capabilities is beyond the scope of this book.

Nonetheless, here is a brief overview of the authentication possibilities:

To perform simple authentication in requests, you can utilize the `auth` parameter, which accepts a tuple containing the username and password.

You can enable Digest Authentication by passing the `auth` parameter with an instance of the `HTTPDigestAuth` class initialized with a username and password. This class is available in the `requests.auth` module.

OAuth2 is not directly supported by the `requests` package. However, there are other packages that provide OAuth2 support. For example, `requests_oauthlib` is a specific package that integrates OAuth support into the `requests` package, enabling developers to perform OAuth2 authentication using the familiar requests API. This package builds upon `OAuthLib`, handling OAuth-related tasks within the context of `requests`.

17.7.4 Coroutine Support

If you want to make HTTP requests within coroutines, it is crucial to perform the requests asynchronously to avoid blocking the event loop.

While using a thread or process executor is one approach, a more appropriate solution is to use asynchronous HTTP client libraries like `aiohttp`, which are designed for asynchronous programming and integrate seamlessly with `asyncio`.

Describing the functionality of `aiohttp` is beyond the scope of this book, but here is a short example that shows the basic functionality.

Please note that it uses `async with` statements, which ensure that the session and response created by the coroutine are automatically closed at the end.

```
import aiohttp
import asyncio

url = "https://reqres.in/api/users/1"


async def fetch_data():
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()


async def main():
    data = await fetch_data()
    print(data)

asyncio.run(main())

# {"data":{"id":1, ...
```

Chapter 18
Type Hinting

In Python, type hinting is a technique used to annotate code with information about the expected types of variables, function parameters, and return values. While Python is a dynamically typed language, meaning that variables do not have fixed types, type hinting allows developers to provide hints about the intended types of data used in their programs.
Using type hinting in Python is optional, but it can be beneficial in projects of all sizes, especially in large and complex codebases.

For instance, integrated development environments like IntelliJ IDEA offer seamless on-the-fly checks for type hint violations as code is being written. This immediate feedback loop aids in maintaining adherence to type hints, enhancing code quality and reliability.

Moreover, tools such as MyPy, though not explored within this book, facilitate static analysis of program code, diligently detecting deviations from specified type hints.

18.1 Basic Type Hinting

Type hinting in Python does use a different syntax compared to Java declarations but shares similarities. In Python, type hints are specified using `:` to denote types and `->` to indicate return types in function signatures.

Python supports various types and annotations, including built-in types such as int, float, str, bool, as well as more complex types like lists, tuples, dictionaries, and custom classes.

Here is an example:

```
from typing import List, Tuple, Dict, Any


def add(a: int, b: int) -> int:
    return a + b
```

```
def create_tuple() -> Tuple[str, int]:
    name: str = "John"
    age: int = 30
    return name, age


def to_string(data: List[Any]) -> List[str]:
    return [str(x) for x in data]


def to_dict(*data: int) -> Dict[str, int]:
    result: Dict[str, Any] = {}

    for i, num in enumerate(data):
        result[f"#{i}"] = num
    return result


def print_kw_args(**kwargs: float):
    for key in kwargs:
        print(f"{key}: {kwargs[key]}")


class Point:
    def __init__(self, x: float, y: float):
        self.x: float = x
        self.y: float = y


my_int: int = 5
my_float: float = 3.14
my_string: str = "Hello"
is_valid: bool = True
my_point: Point = Point(2.5, 4.7)


addition: int = add(3, 4)
my_list: List[str] = to_string(
    [42, "Hello World", True])
my_dict: Dict[str, int] = to_dict(1, 2, 3)
my_tuple: Tuple[str, int] = create_tuple()

print(addition)  # 7
print(my_list)  # ['42', 'Hello World', 'True']
print(my_dict)  # {'#0': 1, '#1': 2, '#2': 3}
print(my_tuple)  # ('John', 30)

print_kw_args(x=1.1, y=2.2, z=3.3)
```

```
# x: 1.1
# y: 2.2
# z: 3.3
```

Type aliases, usually written in CamelCase, provide descriptive names for complex types or combinations of types, thereby improving code readability.

```
from typing import List, Tuple

FirstName = str
LastName = str
UserName = Tuple[FirstName, LastName]


def print_first_names(*names: UserName):
    for name in names:
        print(name[0])


print_first_names(("John", "Doe"), ("Jane", "Doe"))

# John
# Jane
```

18.2 Advanced Type Hinting

18.2.1 Union Types

Union types allow you to specify that a variable can accept values of multiple types. This is useful when a variable or function parameter can have different types under different circumstances.

Union types are either expressed using the `Union` type constructor from the `typing` module or by using parentheses.

```
from typing import Union


def square_root_1(
        x: Union[int, float]
) -> float:
```

```
    return x ** 0.5


def square_root_2(
        x: (int, float)
) -> float:
    return x ** 0.5


assert square_root_1(2) == square_root_2(2.0)
```

18.2.2 Optional Types

Optional types indicate that a variable or function parameter can be of a specified type or `None`. This is helpful when dealing with optional values, such as function arguments that may be provided or omitted.

Optional types are represented using the `Optional` type constructor from the `typing` module.

```
from typing import Optional


def greet(name: Optional[str]) -> str:
    if name is None:
        return "Hello, Guest!"
    else:
        return f"Hello, {name}!"


print(greet(None))  # Hello, Guest!

print(greet("Arthur"))  # Hello, Arthur!
```

18.2.3 Generics

Similar to Java, creating a generic class, method, or function in Python by attaching a type variable serves to introduce a placeholder for a type within the context of that construct. This enables you to write generic code capable of operating on various types while upholding type safety. In Python, generics can be specified by defining a type variable between `[]`:

```
from typing import List, Optional


def first_element[T](items: List[T]) -> Optional[T]:
    if items:
        return items[0]
    else:
        return None


x: int = first_element([1, 2, 3, 4, 5])

y: float = first_element([1.1, 2.2, 3.3, 4.4, 5.5])
```

It is also possible to constrain the type variable to a specific type or a union type:

```
class Point[T: (int, float)]:
    def __init__(self, x: T, y: T):
        self.x: T = x
        self.y: T = y


class Vector[T: Point]:
    def __init__(self, point_1: T, point_2: T):
        self.point_1: T = point_1
        self.point_2: T = point_2


my_point_1: Point = Point(1.1, 2.2)
my_point_2: Point = Point(3.3, 4.4)

my_vector: Vector = Vector(my_point_1, my_point_2)
```

Type variables with constraints can be made reusable by utilizing the `TypeVar` constructor from the `typing` module:

```
from typing import TypeVar

Numeric = TypeVar("Numeric", int, float)


def square(x: Numeric) -> Numeric:
    return x*x
```

```
def cube(x: Numeric) -> Numeric:
    return x*x*x


print(square(2))  # 4

print(cube(3.3))  # 35.937
```

Please note that the first parameter of the `TypeVar` constructor must be identical to the name of the variable to which the result is assigned.

Finally, it is also possible to constrain a type variable to a single class or its subclasses:

```
from typing import TypeVar


class MyClass[T: (float, int)]:
    def __init__(self, value: T):
        self.value: T = value


class MySubclass(MyClass[int]):
    pass


T = TypeVar("T", bound=MyClass)


def print_value(x: T):
    print(x.value)


my_sub_class: T = MySubclass(42)

print_value(my_sub_class)  # 42
```

Chapter 19
Meta-Programming

Meta-programming is a programming technique where programs have the ability to manipulate or generate other code during runtime.
While Java's meta-programming capabilities are primarily limited to reflection and annotations, Python offers a wider range of powerful concepts and features for meta-programming.

Some of the key concepts and techniques available in Python will be explained in this chapter.

19.1 Dynamic Code Execution

Python provides two built-in functions, `exec()` and `eval()`, that enable dynamic code generation and execution at runtime.

The `exec()` function allows you to execute dynamically generated Python code.

Here is an example:

```
code = """
def greet(name):
    print(f"Hello {name}!")
"""

exec(code)

greet("Alice")  # Hello Alice!
```

The `eval()` function evaluates dynamically generated Python expressions stored in a string and returns the result.

Here is an example:

```
expression = "3 + 5 * 2"

result = eval(expression)
```

```
print(result)  # 13
```

Note that it is also possible to generate code objects from strings or other sources using the built-in `compile()` function, which can then be dynamically executed using `exec()` or `eval()`. Code objects are the intermediate, byte-compiled representations of Python code. They can be executed or evaluated, but not modified. However, this topic will not be covered here.

19.2 Decorators

Decorators in Python are higher-order functions that modify or enhance the behavior of other functions or methods.

In comparison to Java's annotations, which provide metadata about classes, methods, and other program elements, Python decorators directly modify the behavior of functions or methods.

While annotations in Java are primarily used for documentation and compile-time processing, decorators in Python are executed at runtime and can dynamically alter the behavior of functions.

Decorators are typically applied using the `@decorator_name` syntax directly above the function or method definition.

They take that function as input and return a modified or enhanced version of it. Additionally, they can perform actions before or after calling the original function, modify its arguments or return value, or even replace the original function entirely with a new implementation.

A decorator is invoked when the function or method it decorates is defined, not when it is called or invoked.

During the decoration process, the decorated function is replaced with the return value of the decorator. This occurs when the module containing the decorated function is loaded into memory.

Here is an example:

```
def log_function(func):
    def wrapper(*args, **kwargs):
        print(f"Function {func.__name__} called.")
        print(f"args: {args}")
        print(f"kwargs: {kwargs}")

        result = func(*args, **kwargs)

        print(f"Return value: {result}")

        return result

    return wrapper


@log_function
def add(a, b):
    return a + b


addition = add(3, b=5)
# Function add called.
# args: (3,)
# kwargs: {'b': 5}
# Return value: 8
```

Decorators in Python can accept parameters. When a decorator has parameters, it essentially becomes a function that returns another function, which acts as the actual decorator.

This means the function is first called with the value for the parameter, and then the returned decorator is applied to the decorated function or method.

Here is an example:

```
def repeat(n):

    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
```

```
    return decorator


@repeat(n=3)
def greet(name):
    print(f"Hello {name}!")


greet("Alice")
# Hello Alice!
# Hello Alice!
# Hello Alice!
```

Alongside function decorators, there are also class decorators. These functions take a class as input and return a modified or enhanced version of that class.

Here is an example:

```
def add_x(cls):

    def init(self, value):
        self.x = value

    def print_x(self):
        print(self.x)

    cls.__init__ = init
    cls.print_x = print_x

    return cls


@add_x
class MyClass:
    pass


obj = MyClass(42)
obj.print_x()  # 42
```

As a final note on decorators, it is worth mentioning that, similar to Java’s annotations, decorators in Python can be nested by stacking them above the

decorated function.

When nested, decorators are applied from the bottom up, meaning that the one closest to the function definition is applied first. Subsequently, the output function of one decorator becomes the input function of the next decorator.

19.3 Special Methods

Python provides a rich set of special methods, also known as "magic methods" or "dunder methods," that allow developers to customize the behavior of objects and classes.

These methods are invoked by the Python interpreter in response to specific operations or expressions involving objects. By implementing these methods, you can define custom behavior tailored to your classes and objects.

Due to space constraints, only a limited number of examples for special methods are provided in this book, showcasing their versatility in customizing object behavior.

19.3.1 Arithmetic Operations

Functions like `__add__`, `__sub__`, `__mul__`, and others help objects perform basic math operations like addition, subtraction, and multiplication. Similarly, methods such as `__iadd__` (for `+=`), `__isub__` (for `-=`) and `__imul__` (for `*=`) enable objects to support in-place operations.

Here is an example:

```
class Vector:
    def __init__(self, *args):
        self.__vector = args

    def __add__(self, other):
        return Vector(*self.__add_tuples(other))

    def __iadd__(self, other):
        self.__vector = self.__add_tuples(other)
```

```
        return self

    def __str__(self):
        return str(self.__vector)

    def __add_tuples(self, other):
        if isinstance(other, Vector):
            x = self.__vector
            y = other.__vector
            if len(x) == len(y):
                addition = tuple(
                    x[i] + y[i]
                    for i in range(len(x))
                )

                return addition
            else:
                raise ValueError("Not same length")
        else:
            raise ValueError("Not a vector")


v1 = Vector(1, 2, 3)
v2 = Vector(4, 5, 6)

result = v1 + v2
print(str(result))  # (5, 7, 9)

v1 += Vector(8, 9, 10)
print(str(v1))  # (9, 11, 13)
```

19.3.2 Object Comparison

In Python, you can customize the default behavior of comparison operators such as `==`, `<=`, `!=`, etc., by overriding the corresponding special methods:

- **`__eq__(self, other)`:** Equality (`==`)
- **`__ne__(self, other)`:** Inequality (`!=`)
- **`__lt__(self, other)`:** Less than (`<`)
- **`__le__(self, other)`:** Less than or equal to (`<=`)
- **`__gt__(self, other)`:** Greater than (`>`)
- **`__ge__(self, other)`:** Greater than or equal to (`>=`)

The only operators that are implemented for all classes in Python are `==` and `!=`.

`==` by default checks if the two objects are the same instance. If `!=` is not defined for a class, it is automatically derived from `__eq__` by inverting its result. If `__eq__` returns `True`, `__ne__` will return `False`, and vice versa.

Therefore, it is usually sufficient to override only `__eq__`.

If you try to use the other operators (<, <=, >, >=) on an object of a class that does not implement the corresponding special methods, Python will raise a `TypeError`.

Python is able to replace any of these operators by deriving it from its reverse operator. For example, it can replace `a <= b` with `b >= a`, but it cannot combine operators, such as replacing `a <= b` with `a < b` or `a == b`.

The following example shows a minimum set of special methods that need to be defined so that all comparison operators work:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return (
            self.x == other.x and self.y == other.y
        )

    def __lt__(self, other):
        return (
            self.x < other.x and self.y < other.y
        )

    def __le__(self, other):
        return (
            self.x <= other.x and self.y <= other.y
        )

    def __hash__(self):
        return hash((self.x, self.y))
```

```
point1 = Point(2, 3)
point2 = Point(5, 7)
point3 = Point(2, 3)

print(point1 == point3)  # True
print(point1 <= point3)  # True
print(hash(point1) == hash(point3))  # True

print(point1 == point2)  # False
print(point1 < point2)  # True
print(point2 >= point3)  # True
```

As demonstrated in the example above, it is a good practice in Python, similar to Java, to override the `__hash__` function if you have overridden the `__eq__` operator. You can use the built-in `hash()` function to compute a hash value.

The `hash()` function can handle only immutable types of objects. Immutable types are those whose state cannot be modified after creation. Examples of immutable types include integers, floats, strings, tuples, and frozensets.

Mutable types, such as lists, dictionaries, and sets, cannot be hashed directly because their state can change after creation.

19.3.3 Attribute Access

The `__getattr__` method is invoked when an attribute is accessed but not found. It can either raise an exception or return a value as the result of the attribute access.

The `__setattr__` method is invoked when an attribute is set. It can then set the attribute by calling the `__setattr__` method of its superclass. Directly setting the attribute within `__setattr__` is not possible as this would call `__setattr__` again, resulting in an infinite recursion loop.

Here is an example:

```
class MyClass:
    def __getattr__(self, name):
        return f"{name} not found"

    def __setattr__(self, name, value):
        print(f"{name} set to {value}")
        super().__setattr__(name, value)


obj = MyClass()
print(obj.unknown_attr)  # unknown_attr not found
obj.my_attr = 42  # my_attr set to 42
```

19.3.4 Indexing Protocol

In Python, protocols define a set of methods or attributes that objects can implement to achieve a specific interaction or behavior. They provide a flexible way to establish contracts between different parts of a program without relying on inheritance.

Protocols are not explicitly enforced by the language but serve as informal agreements or conventions. *(Please note that the Python standard PEP 544 introduces a more formal approach to protocols in Python. It introduces the concept of protocol classes, which are abstract base classes that specify the expected methods and attributes of objects that conform to a particular protocol. However, this formalization of protocols will not be discussed in this book.)*

In contrast, Java uses interfaces to achieve similar goals. Interfaces define a set of methods that classes must implement to adhere to a certain behavior or functionality.

While the terminology differs, the underlying concept of defining contracts for classes to follow remains consistent between Python's protocols and Java's interfaces.

Some examples of Python protocols include the iteration protocol (defined by the `__iter__` and `__next__` methods), the context management protocol (defined by the `__enter__` and `__exit__` methods), and the comparison

protocol (defined by methods like `__eq__`, `__lt__`, etc.). In the following section, we will explore the indexing protocol in Python, which allows objects to customize their behavior for indexing operations such as accessing elements using square brackets (`[]`).

The indexing protocol consists of several special functions:

`__getitem__(self, key):` Called when an item is accessed using square brackets. It takes a key as an argument and returns the corresponding value.

`__setitem__(self, key, value):` Called when an item is assigned a value using square brackets. It takes a key and a value as arguments and sets the value at the specified key.

`__delitem__(self, key):` Called when an item is deleted using the `del` statement with square brackets. It takes a key as an argument and removes the item at the specified key.

`__iter__(self):` Called when an object is iterated over using a loop or other iterable context. It returns an iterator object that allows sequential access to the elements of the object.

`__contains__(self, item):` Called when the `in` operator is used to check for membership in the object. It takes an item as an argument and returns `True` if the item is found in the object, `False` otherwise.

`__len__(self):` Called when the built-in `len()` function is used to determine the length of the object. It returns the number of items in the object.

In this example, all these methods are used to create an "array" that can handle assignments and access with lists of array indices:

```
class CustomArray:
    def __init__(self, size):
        self.__size = size
        self.__content = [None for i in range(size)]

    def print(self):
        print(self.__content)
```

```
    def __getitem__(self, key):
        if isinstance(key, tuple):
            return tuple(
                self.__content[i] for i in key
            )
        else:
            return self.__content[key]

    def __setitem__(self, key, value):
        if isinstance(key, tuple):
            for i in key:
                self.__content[i] = value
        else:
            self.__content[key] = value

    def __delitem__(self, key):
        if isinstance(key, tuple):
            for i in key:
                self.__content[i] = None
        else:
            self.__content[key] = None

    def __iter__(self):
        return iter(self.__content)

    def __contains__(self, item):
        return item in self.__content

    def __len__(self):
        return self.__size


my_array = CustomArray(5)
my_array.print()  # [None, None, None, None, None]
print(len(my_array))  # 5

my_array[0, 1, 2] = 42
my_array.print()  # [42, 42, 42, None, None]

my_array[0] = 1
my_array.print()  # [1, 42, 42, None, None]

print(my_array[3, 2, 1])  # (None, 42, 42)
print(my_array[1])  # 42

for j in my_array:
    print(j)
# 1 42 42 None None
```

```
print(42 in my_array)  # True
print(69 in my_array)  # False

del my_array[1, 2]
my_array.print()  # [1, None, None, None, None]

del my_array[0]
my_array.print()  # [None, None, None, None, None]
```

19.3.5 `__new__` and `__call__`

In Python, `__new__` and `__call__` are two powerful special methods that play crucial roles in object creation and instance behavior.

`__new__` is a class method responsible for creating a new instance of a class. It receives the arguments of the class constructor call and must return a new instance of the class. It is typically used in scenarios where immutable objects need to be initialized or when subclassing immutable types like `int` or `tuple`, as in these cases the already created object cannot be changed in the `__init__` method anymore.

Here is an example:

```
class TupleWithHash(tuple):
    def __new__(cls, *args):
        new_tuple = args + (hash(args),)

        return super().__new__(cls, new_tuple)


my_tuple = TupleWithHash(1, 2, 3, 4, 5)

print(my_tuple)
# (1, 2, 3, 4, 5, -5659871693760987716)
```

The `__call__` method allows an instance of a class to be called as if it were a function:

```
class Multiplier:
    def __init__(self, factor):
        self.__factor = factor
```

```
    def __call__(self, value):
        return value * self.__factor


double = Multiplier(2)
print(double(5))  # 10
```

19.4 Metaclasses

Metaclasses are classes whose instances are classes. They are constructs that Python uses to create class objects and instances, dictating how they are constructed and initialized.

The only metaclass that comes built-in with Python is `type`. It is the metaclass that Python uses by default to create new class objects.

In addition, you can define your own metaclasses to customize the class creation process. This allows you, for example, to automatically add or modify class attributes and methods, or to ensure that classes meet certain criteria, such as having specific attributes or methods.

Interestingly, the metaclass of `type` itself is `type`. This self-referential property illustrates the recursive nature of Python's type system, where type is both an instance and a subclass of itself.

In the following sections, we will explore how to define custom metaclasses, how they interact with the class creation process, and the role they play when creating an instance of a class.

19.4.1 Defining Metaclasses

When creating your own metaclass, it is generally recommended to inherit from `type`. This is because `type` is the default metaclass and provides the necessary infrastructure to create classes.

A metaclass is an ordinary class whose constructor parameters are as follows:

`cls`: The newly created class.
`name`: The name of the class being created.
`bases`: A tuple containing the base classes of the class being created.
`attr`: A dictionary mapping the names of the attributes and methods of the class being created to their values.

Here is an example of a metaclass that adds a version number to every class created with this metaclass:

```
class MyMetaClass(type):
    version = "1.0"

    def __init__(cls, name, bases, attr):
        attr["version"] = MyMetaClass.version
        super().__init__(name, bases, attr)


class MyClass(metaclass=MyMetaClass):
    x = 42


print(MyClass.version)  # 1
print(MyClass.x)  # 42
```

It is also possible to instantiate the metaclass directly. For example, the definition of `MyClass` in the example above could be accomplished like this:

```
MyClass = MyMetaClass(
    "MyClass",
    (object,),
    {"x": 42}
)
```

However, the resulting class is not identical. In the original example, the Python interpreter automatically adds attributes like `__module__`, containing the module name, which does not occur when the metaclass is instantiated directly.

19.4.2 Intercepting Object Creation

In Python, creating an instance of a class is essentially equivalent to invoking the `__call__` method of the class's metaclass.

The default `__call__` method of the built-in metaclass type is responsible for creating instances of classes.

When you call a class object, the `__call__` method of `type` is invoked behind the scenes, which in turn invokes the `__new__` and `__init__` methods of the class to create and initialize the instance.

This mechanism allows you to intercept the creation of class instances by defining a custom `__call__` method in a metaclass.

Here is an example implementing a singleton pattern:

```
class SingletonMeta(type):
    __instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in SingletonMeta.__instances:
            ins = super().__call__(*args, **kwargs)
            SingletonMeta.__instances[cls] = ins

        return SingletonMeta.__instances[cls]


class SingletonClass(metaclass=SingletonMeta):
    def __init__(self, value):
        self.value = value


singleton1 = SingletonClass(10)
singleton2 = SingletonClass(20)

print(singleton1.value)  # 10
print(singleton2.value)  # 10
print(singleton1 is singleton2)  # True
```

19.5 Class and Object Introspection

As in Java, Python provides built-in functions and modules like `inspect` that allow introspection of classes and objects at runtime.

Introspection enables developers to examine the structure, attributes, and methods of classes and objects dynamically, facilitating tasks like runtime debugging, documentation generation, and dynamic code generation.

19.5.1 Built-in Functions for Introspection

Python includes several built-in functions that are particularly useful for introspection:

`type(obj)`: Returns the class that the object is an instance of.

`dir(obj)`: Returns a list of the attributes and methods of an object.

`getattr(obj, name[, default])`: Returns the value of the named attribute of an object. If the attribute is not found, the default value is returned if provided; otherwise, an `AttributeError` is raised.

`hasattr(obj, name)`: Checks if the object has the named attribute.

`setattr(obj, name, value)`: Sets the named attribute on the object to the specified value.

`delattr(object, name)`: Deletes the named attribute from the object.

`isinstance(obj, class)`: Checks if an object is an instance or subclass of a class.

`issubclass(subclass, class)`: Checks if a class is a subclass of another class.

Here is an example:

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class MySubClass(MyClass):
    def __init__(self, x, y, z):
        super().__init__(x,y)
```

```
        self.z = z


my_subclass = MySubClass(1, 2, 3)

print(type(my_subclass))
# <class '__main__.MySubClass'>

print(dir(my_subclass))
# ['__class__', '__delattr__', ... 'x', 'y', 'z']

print(getattr(my_subclass, "x"))  # 1
print(hasattr(my_subclass, "z"))  # True

setattr(my_subclass, "has_towel", True)
print(my_subclass.has_towel)  # True

delattr(my_subclass, "has_towel")
print(hasattr(my_subclass, "has_towel"))  # False

print(isinstance(my_subclass, MyClass))  # True
print(issubclass(MySubClass, MyClass))  # True
```

Please note that the `dir()` function does not show all attributes of an object. It lists most of the attributes and methods associated with an object but may exclude certain special or dynamically added attributes.

Here is an example:

```
class MyClass:
    def __getattr__(self, name):
        if name == "dyn_attr":
            return "Dynamic Attribute"
        raise AttributeError(f"{name} not found")


obj = MyClass()

print(obj.dyn_attr)  # Dynamic Attribute

print(dir(obj))  # List does not contain dyn_attr
```

19.5.2 The `inspect` Module

The `inspect` module in Python, much like Java's reflection API, enables you to examine the structure and members of an object.

Explaining the complete functionality provided by this module would be beyond the scope of this book. Instead, here are two examples demonstrating frequently occurring tasks.

Logging Method Calls

This example shows how to automatically add logging to every method call of an object. Automatically wrapping methods can be generally useful when writing test code.

```
import inspect


def add_logging(obj):
    for name, value in inspect.getmembers(
            obj, inspect.ismethod):
        setattr(obj, name, add_logger(value))


def add_logger(meth):
    def wrapper(*args, **kwargs):
        print(f"*** Calling method: {meth.__name__}")
        return meth(*args, **kwargs)
    return wrapper


class MyClass:

    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(f"value = {self.value}")


my_object = MyClass(42)
add_logging(my_object)

my_object.print_value()
# *** Calling method: print_value
# value = 42
```

The function `inspect.getmembers()` retrieves all the members of an object. The object can be an instance, a class, a module, or any other type of object in Python. A member can be a method, attribute, property, or any other type of member that an object can have.

`inspect.getmembers()` returns a list of tuples, where each tuple contains two elements: the name of the member and the value of the member.

`inspect.getmembers()` takes as parameters the object to be inspected and, optionally, a predicate function that filters the members.

`inspect.ismethod()` is a predicate function that returns `True` if its parameter is a bound method.

The `inspect` module provides other predicate functions like

`inspect.isfunction()` for functions
`inspect.isbuiltin()` for built-in functions such as `__new__()`.

Mock Creation

This example demonstrates how to create a mock for a class by replacing all methods with a dummy method. It returns values based on the return type hint: `0` for `int`, `False` for `bool`, an empty string for `str`, and `None` for other types.

```
import inspect


def mock(cls):
    methods = {}

    for key, value in inspect.getmembers(
            cls, inspect.isfunction):
        methods[key] = mock_method(value)

    methods["__init__"] = lambda x: None

    mock_class = type(
        f"{cls.__name__}_mock",
        (cls,),
        methods
```

```
    )

    return mock_class()


def mock_method(method):
    signature = inspect.signature(method)
    return_annotation = signature.return_annotation

    def mocked_method(*args, **kwargs):
        signature.bind(*args, **kwargs)

        if return_annotation == int:
            return 0
        elif return_annotation == bool:
            return False
        elif return_annotation == str:
            return ""
        elif return_annotation is not None:
            return None

    return mocked_method


class MyClass:

    def __init__(self, value):
        self.value = value

    def add(self, second_value) -> int:
        self.value += second_value
        return self.value

    def is_positive(self) -> bool:
        return self.value > 0

    def as_tuple(self) -> tuple:
        return (self.value,)

    def __str__(self) -> str:
        return f"Value: {self.value}"


obj = MyClass(42)
mocked_obj = mock(MyClass)

print(obj.__class__)
# <class '__main__.MyClass'>
```

```
print(mocked_obj.__class__)
# <class '__main__.MyClass_mock'>

print(isinstance(mocked_obj, MyClass))  # True

print(obj.is_positive())  # True
print(mocked_obj.is_positive())  # False

print(obj.as_tuple())  # (42,)
print(mocked_obj.as_tuple())  # None

print(obj.add(1))  # 43
print(mocked_obj.add(1))  # 0

print(obj)  # Value: 43
print(mocked_obj)  # <empty string>
```

The `mock()` function takes a class object as an argument and creates a mock for that class. It does so by generating a subclass of the given class, where all methods of the parent class are overridden with mocked methods. The function then returns an instance of this subclass.

Here are a few points in this code that are worth explaining in detail:

The expression

```
inspect.getmembers(cls, inspect.isfunction)
```

returns all the methods of the class and its superclasses. The filter function used here is `inspect.isfunction()` instead of `inspect.ismethod()`, because methods are bound to an instance of a class, not to the class itself.

The statement

```
methods["__init__"] = lambda x: None
```

creates a constructor for the mock class that does nothing. This allows the `mock()` function to create an instance of the mock class using a parameterless constructor.

The `inspect.signature()` function returns a complex object that describes the signature of a method or function.

Its main components include a list of objects describing the parameters of the signature (`signature.parameters`) and, if specified by a type hint, the expected return type of the method or function (`signature.return_annotation`).

In the example above, `signature.bind()` is used to test whether a mock method is called with the correct parameters. This ensures that the mock implementation faithfully mimics the interface of the original method.

For instance, in the example above, calling the `add()` method of `mocked_obj` without arguments would cause `signature.bind()` to raise a `TypeError`.

More generally, the `signature.bind()` function returns a `BoundArguments` object, which represents the mapping between the passed arguments and the function's parameters.

This complex object contains several attributes, among them `arguments`, an `OrderedDict` (a dictionary where the keys maintain their order) that maps the parameter names of the original signature to the passed arguments.

By using `.apply_defaults()` on the `BoundArguments` object, it is also possible to fill in any missing arguments with their default values from the signature.

Here is an example:

```
import inspect


def my_method(x, y, z=42):
    pass


my_signature = inspect.signature(my_method)

bound_args = my_signature.bind(1, y=2)
print(bound_args.arguments)
```

```
# {'x': 1, 'y': 2}

bound_args.apply_defaults()
print(bound_args.arguments)
# {'x': 1, 'y': 2, 'z': 42}
```

About the Author

Dr. Jörg Richter has dedicated over three decades to the IT industry, where he has worn many hats including those of a team leader, software developer, and product manager.

He holds a PhD in Computer Science and a university diploma in Mathematics.

His journey into the world of programming began at the age of 16 when he wrote his first program in BASIC. This was a few years before IBM introduced its first personal computer, during an era when it was still common to use card punchers for programming mainframes.

For the past ten years, he has focused on developing Android applications.

Currently, he is exploring the exciting possibilities that Artificial Intelligence offers. He is particularly interested in how seasoned software developers can leverage AI to solve new challenges and create innovative solutions.

About the Cover

The cover features Hypoxis Serrata, also known as the African Star Grass. The graphic is based on a handcolored botanical copper engraving from 1804.