Python Mini Reference 2023

A Quick Guide to the Modern Python Programming Language for Busy Coders

Harry Yoon

Version 1.1.1, 2023-05-14

Copyright

Python Mini Reference:

A Quick Guide to the Python Programming Language

Published: October 2022

Harry Yoon
San Diego, California

Preface

Python is a dynamic language.

It means many different things. It means that the language is more flexible. It means that Python is an easier language to program with. You can quickly write a simple program without having to go through too much "rituals", compared to many other programming languages. Python scripts tend to require less boilerplate code. Python gives you more freedom.

On the other hand, it also means that the language is less safe, and more error prone. It means that it is harder to build a larger system with Python. It means that software written in Python is generally harder to maintain over a longer period of time.

The key to using Python effectively is to understand this tradeoff. Python can be an ideal language, for instance, for quick prototyping, or "scripting". On the flip side, Python is not as much used for the "enterprise software" development.

Python is a general purpose programming language. Python is used in many different application areas, from system administration tasks to web application development. Python is now one of the most widely used programming languages for scientists, who have traditionally been using more high-level tools like Matlab. Python provides an easier "upgrade" path to these "non-professional" programmers. Now, Python is becoming *the* language for machine learning and data science.

Python is beginner-friendly. In fact, it seems to be the most favorite first language for beginning programmers, even more so than JavaScript.

It should be noted, however, that Python is *not* a simple language. Over the past 30 years or so of its history, it has gone through a lot of big and small changes. It is still easy to get started with programming in Python.

Nonetheless, once you reach a certain point, say, from the advanced beginners to intermediate level, its complexity can be overwhelming.

This book will give you a broad and *sanitized* overview of the Python language, as of its most recent incarnation (e.g., 3.10 and 3.11). This book can be useful to anyone who has been dabbling with Python without solid foundation. It can also be useful to the people who have experience with other programming languages and want to get some quick overview of the language.

This book is written as an (informal) language reference. The goal of this book is not to teach you how to program effectively in Python, but rather to provide a concise summary of the language grammar. If you have some basic programming knowledge, you can read this book more or less from beginning to end, and you should be able to get the overall understanding of the Python programming language quickly, regardless of your particular programming background.

The book is written for a broad audience, but one caveat is that there are a fair amount of cross references, unlike in the books written in a tutorial style. If you have no prior experience with programming in Python or any similar language, then you may find it a little difficult to go through some parts of the book. This book is not for complete beginners. It skims through some elementary concepts in the beginning, for the sake of brevity, so that we can focus more on the intermediate and advanced level topics.

This book is not an authoritative language reference. For that, we recommend the readers to refer to the official language specification.

Dear Readers:

Please read b4 you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it's unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Chapter 1. Introduction

This book will give you an overview of the Python language grammar.

It appears that the line between the programming language proper and the standard library is becoming more blurred these days. Although it is not our goal to go through the Python standard library (which is well beyond the scope of this book), we will touch upon a few important concepts from the standard library modules that are considered more or less part of the language.

This "reference" starts with the most boring part of Python. ☺ If you plan to read the book from beginning to end, say, rather than using it just as a quick reference, then you can skip the first few chapters in your first reading, and come back to them later when needed.

The term "program" means different things in different contexts, and across different programming languages. We start with somewhat formal explanations of what a Python program is, and how Python programs are executed, e.g., in the Python interpreter.

Python programs are logically organized into packages and modules, which we take a look at next. A *module* corresponds to a file in a physical file system, and modules, and including package modules, are the basic units of code reuse and sharing in Python.

Next, we briefly go through some of the lexical elements of the Python source code. There are some small variations across different programming languages, but their lexical compositions are rather similar, and Python is no different. This part can be skipped in your "reading".

Generally speaking, a program consists of *code* and *data.* Code refers to instructions. Data in Python is represented by *objects.* The object in Python is a fundamental component. Everything which we deal with in

a Python program is objects. We start the main part of the book by introducing various important concepts related to the objects.

Although Python uses a dynamic type system, the types still play the foundational roles in the Python programming language. We first go through some of the basic builtin types in Python. Python includes quite a few builtin types, and this reference touches on only some of them. As indicated, this is generally true across all topics discussed in this reference. Completeness is not the goal of this *mini reference*.

Python also includes a few builtin compound types such as list and dictionary, which are important components of any non-trivial programs. We briefly go through these types in the next chapter. Advanced types, e.g., functions and classes, in particular, are explained in detail later in the book.

As with any imperative programming language, Python has expressions and statements. An expression prescribes how to compute a value using other expressions and values. Python supports most of the common operators and expressions found in other imperative languages. If you are coming from other procedural programming language background, you can skim through most of this part.

A statement is an instruction. Statements control the flow of a program to attain the desired goal. In Python, there are two kinds of statements, simple statements and compound statements. Compound, or complex, statements can "include" other simple or compound statements.

Simple statements include expression statement, assignment statement, `assert` statement, `pass` statement, `del` statement, `return` statement, `raise` statement, `break` statement, `continue` statement, `global` statement, and `nonlocal` statement. Python's compound statements include `if` statement, `while` statement, `for` statement, `try` statement, `with` statement, `match` statement, function `def` statement, `class` definition, and other coroutine-related statements.

One of the most significant changes to Python for the last 30+ years has been the addition of structural pattern matching to the language, as of Python 3.10 (2021). Pattern matching was first popularized by functional programming languages like Haskell, and it is now becoming more and more widely available across many different programming languages like C#, rust, swift, scala, and (yes) even Java. ☺

As we more adopt this feature, as a community, pattern matching will likely change how we program in Python in the coming years. Although it is currently supported only in the context of the `match` statement, it is conceivable, and in fact expected, that pattern matching will be available more broadly across the language in the near future, considering its simplicity, elegance, and power.

A function definition is a compound statement. We dedicate its own chapter to the function definition. This chapter also includes other function-related topics such as lambda expressions and decorators.

Another compound statement, the class definition, is explained next. Other class-related concepts such enums and data classes are described in this chapter as well. We also provide an informal introduction to the object oriented programming styles in Python, including multiple inheritance.

Special kinds of functions, generators and coroutines, are separately discussed in the last part of the book, in the Coroutines chapter. We briefly touch on the (high-level) asynchronous programming style in Python using the new `async` and `await` keywords. In the last section, we provide a simple async programming example, namely, the producer-consumer problem.

As stated, we do not exhaustively cover every topic in Python in this "mini reference", including the (low-level) multi-thread and multi-process programming, etc. However, it goes through all *the essential language features* that any intermediate to advanced level Python programmers should be familiar with.

Chapter 2. Python Programs

A complete Python program can be passed to the Python interpreter:

- With the *-c* command line option with a program text,
- With the *-m* command line option with a module name,
- As a file name passed as the first command line argument, or
- From the standard input.

The Python interpreter executes the input file, or the input text, as a complete program, in the namespace of a special module, `__main__`.

When the file or standard input is a terminal, or when the Python interpreter is invoked without an argument, the interpreter enters the interactive mode. The Python interpreter reads and executes one statement at a time (simple or compound) in the interactive mode instead of executing a complete program.

Here are a few examples:

```
$ python -c "print('Hello World!')"    ①
```

① A complete Python program is executed using the *-c* flag.

```
$ python main.py                       ①
```

① The Python script is passed to the interpreter as a command line argument.

```
$ python -m hello_world                ①
```

① The Python module `hello_world` is executed through the *-m* flag.

```
$ echo "print('huh?')" | python        ①
```

① The Python code from the stdin (via Unix pipe).

```
$ python <<< "print('a')"              ①
```

① Using the Unix shell "here string" syntax.

```
$ python << EOF                        ①
> print("hello")
> print("world")
> EOF
```

① The "here doc".

```
$ python                               ①
```

① Invoking the python command without an argument starts the REPL.

```
$ python -                             ①
```

① Ditto. The Python interpreter ignores the flags following the dash `-`.

```
$ python -i                            ①
```

① It also starts the interactive shell. The *-i* flag can be combined with other flags, e.g., *-m*, *-c*, etc.

We will use the Unix shell for illustration, throughout the book, when relevant. Depending on your platform, some commands may not work exactly as indicated here.

2.1. File/Text Input

A complete Python program is a series of zero or more statements, with optional newlines between the statements. A complete program with this form is expected in the following situations:

- When a Python program is read from a file or from an input string,
- When an imported module is parsed, and
- When a string argument is passed to the builtin `exec` function.

As indicated, the main module of a running program is always called `__main__` in Python.

2.2. Interactive Mode

The Python interpreter in the interactive mode (aka Python REPL) does not execute a complete program. Instead, it reads and executes one statement at a time. Each input comprises a code block in the namespace of `__main__`.

A top-level compound statement must be followed by an extra newline in the interactive mode, which is used by the parser to detect the end of the input.

```
$ python                                  ①
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on
linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>                                       ②
>>> for levels in ("White", "Black"):     ③
...     print(levels)                     ④
...                                       ⑤
White                                     ⑥
```

```
Black
>>>                                    ⑦
>>> exit()                             ⑧
$                                      ⑨
```

① A command that starts the Python REPL. This might be different on your system.

② The Python REPL prompt, `>>>` . We just pressed Enter in this example.

③ The first line of a `for` compound statement.

④ The Prompt has changed from `>>>` to `...` . The statements in the `for` "suite" need to be indented.

⑤ We press Enter one more time to signal the end of the compound statement to the Python interpreter.

⑥ The output, displayed in two lines. Note that the output lines are not preceded by the REPL prompts.

⑦ The Python REPL waits for another command/statement. We press Enter again in this example.

⑧ You can exit the Python REPL by calling the builtin `exit` or `quit` functions, or by issuing the EOF signal (e.g., CTRL+D or CMD+D).

⑨ Back to the Unix shell.

Chapter 3. Program Execution

3.1. Code Blocks

A Python program is executed in a series of "execution frames". A piece of Python program that is executed as a unit in an execution frame is called a "code block".

For example, modules and function/class definitions are code blocks, among other things. An entire program passed to the interpreter, e.g., as *-c* option or as a file argument, is a code block (more precisely, the top-level code block). So is an imported module.

The code blocks can be nested.

3.2. Name Binding

In Python, all values are "objects", and objects are often referred to by their "names". Names are introduced by name binding operations.

For example, `import` statements, function and class definitions, and assignment expressions and statements bind names within their enclosing code blocks. Function parameters also bind names, within the function definition block. If a name is bound in a block, it is a "local variable" of that block by default.

If a name is bound at the module level, it is also a "global variable". The `import` statement of the form `from XYZ import *` can only be used at the module level. It binds all names defined in the imported module that do not begin with an underscore `_`. All names imported this way are global (as well as local with respect to the importing module).

3.3. Scope

A name is accessible within a particular part of a program, e.g., within the code block where the name is bound. This is called a "scope". The scope of a local variable bound in a code block includes that block.

When a name is used in a code block, the smallest enclosing scope is used to find the binding of the name. It is called the "name resolution". If the name cannot be resolved, then an exception is raised. Depending on the context, `NameError`, or more specifically `UnboundLocalError`, may be raised.

3.3.1. Function definition block

A function `def` statement defines a code block. If a name is bound in a function block, then its scope includes this function block and any other blocks that are contained/nested therein.

3.3.2. Class definition block

In a class definition, the scope of the names defined in a class block is limited to that class block only, *excluding* those of the enclosed methods. The local variables that are not bound in the class block are looked up in the global namespace.

3.3.3. The `global` declaration

If a `global` statement is used for a name in a block, then the uses of the name refer to its binding in the top-level namespace, which includes those of the code block module and the `builtins` module, as well as the builtins namespace.

3.3.4. The `nonlocal` declaration

If a `nonlocal` statement is used for a name in a code block, the name

refers to the one bound in the smallest enclosing function scope. If the name binding is not found at compile time, then a `SyntaxError` exception is raised

3.4. Scope Examples (Optional)

Here's a single module (e.g., a Python source file named *my_module.py*) that demonstrates the function and class blocks and the scopes of the variables defined in those blocks. (Note: The program/module is split into three segments for formatting reasons.)

my_module.py

```
1  x, w = 1, 2                          ①
2
3  def var_demo():                      ②
4      y = 3                            ③
5      print("A", x, w, y)              ④
6      def inner_function():            ⑤
7          w = 22                       ⑥
8          nonlocal y                   ⑦
9          y = 33                       ⑧
10         print("B", x, w, y)          ⑨
11     inner_function()                 ⑩
12     print("C", x, w, y)              ⑪
```

① The variables `x` and `w` are *local* to the module, and hence they are *global* variables. Their scope includes the entire module, lines 1-28.

② A function definition. It creates a block, lines 4-12.

③ A name binding introduces the name to the current scope. Hence, `y` is local to this function, and its scope is the same block, lines 4-12.

④ At this point, we cannot easily tell what this `print` function will print out, e.g., without going through/understanding the entire program. `x` and `w` are global, and hence their values might have

changed somewhere in the code by the time this statement is executed. The current value of `y` is `3`.

⑤ An inner function definition. It creates a nested scope.

⑥ A new variable binding. `w` is a local variable to the `inner_function` function, and its scope is the function block, lines 7-10.

⑦ The `nonlocal` statement asserts that we are going to use the variable `y` from the outer function scope, lines 4-12.

⑧ We are assigning a new value `33` to this non-local variable `y`. Without the `nonlocal` declaration, this statement could have introduced a new variable `y` into the local scope, lines 7-10.

⑨ At this point, `x` is a global variable, `w` is a local variable (local to `inner_function`), and `y` is a non-local variable (but, local to `var_demo`).

⑩ Calling `inner_function` might have (potentially) changed the values of the global `x` and nonlocal `y`, but it should have no effect on the global `w`.

⑪ It prints the global `x` and `w`, and local `y` (local to `var_demo`).

```
14 class VarDemo:                          ①
15     z = 5                               ②
16     global w                            ③
17     w = 222                             ④
18     print("D", x, w, z)                 ⑤
19     def inner_method(self):             ⑥
20         print("E", x, w, self.z)        ⑦
21         self.z = 10                     ⑧
22         print("F", x, w, self.z)        ⑨
```

① A `class` definition creates a new scope (within the global scope).

② `z` is a local variable in the class block. Its scope is lines 15-18, excluding the enclosed function/method blocks, e.g., lines 19-22.

③ This `global` statement declares that the name `w` is the same one defined in the global scope.

④ This assignment changes the value of the global `w` to `222`. Without the `global` declaration, this statement could have introduced a new name `w` to the local scope, lines 15-18.

⑤ At this point, `x` and `w` refer to the global variables (first bound in line 1), and `z` refers to the local variable.

⑥ An inner method definition. It creates a new scope within the *global scope* (not within the scope of `VarDemo`).

⑦ Global `x` and global `w`. `self.z` refers to the class variable `z` (because an instance variable named `z` does not (yet) exist).

⑧ This creates a new instance variable `z`, which shadows the class variable `z` defined in line 15.

⑨ At this point, `self.z` refers to the instance variable `z`. Class vs instance variables are discussed in more detail in a later chapter, Classes.

```
25 if __name__ == "__main__":              ①
26     x = 10                              ②
27     var_demo()                          ③
28     x = 20                              ④
29     obj = VarDemo()                     ⑤
30     obj.inner_method()                  ⑥
```

① The `if` statement does not create a new scope. All names declared/referred to within the suite of this `if` statement are global.

② `x` refers to the global `x` (line 1). This value is reset to `10` at this point.

③ Calling `var_demo` will execute all statements in the function object created by the function definition, lines 4-12.

④ The value of the global `x` now set to `20`.

⑤ We create an instance of `VarDemo` and bind it to a global variable `obj`.

⑥ Calling `inner_method` on `obj` will execute the statements defined in the function block, e.g., lines 20-22 in this example.

This is a somewhat artificial example, but hardly more complicated than the "real world" programs, in terms of the nested scopes and what not. If this code does not make sense, you can come back to it later after going through the rest of the book.

If you run the program,

1. It will first bind the global `x` and `w` to `1` and `2`, respectively. Line 1.
2. It will create a function object for `var_demo`. Lines 3-12.
3. It will execute the statements in the `class` definition. Lines 15-20.
    a. The new local variable `z` is bound to `5`, line 15.
    b. The global variable `w` is assigned a new value `222`, line 17.
    c. This `print` function will use, `x == 1` (line 1), `w == 222` (line 17), and `z == 5` (line 15), and hence its output is, *D 1 222 5*.
    d. Then, a function object for `inner_method` is created, lines 19-20.
4. Next, the `if` compound statement is executed. Lines 23-28.
    a. A new value `10` is assigned to the global `x`, line 24.
    b. When we call the function `var_demo`, `x == 10` and `w == 222`, line 25.
        i. The local variable `y` is bound to `3`, line 4.
        ii. The `print` call of line 5 therefore prints out *A 10 222 3*.
        iii. A function object for `inner_function` is created, lines 6-10.
        iv. And, we call this function, line 11.
            A. When we execute the `print` call statement, line 10, the

values of global `x`, local `w`, and non-local `y`, are 10 (line 24), 22 (line 7), and 33 (line 9), respectively. Hence, the output will be *B 10 22 33*.

v. The `print` call of line 12 will print out *C 10 222 33* because the global `x` is `10` (line 26), the global `w` is `222` (line 17), and the local `y` is `33` (line 9).

c. The global `x` is set to `20` here, line 28.

d. We call the constructor `VarDemo`, line 29, to create a new object with name `obj`. The `obj` object shares the class variable `z` with the `VarDemo` class, whose value is `5`.

e. When we call `inner_method` on `obj`, `x == 20` and `w == 222`, line 30.

i. Hence, the print function call (line 20) prints out *E 20 222 5*.

ii. On the other hand, the second print function call (line 22) prints out *F 20 222 10* because `self.z` now refers to the instance variable `z`, whose value is `10` (line 21).

Here's the complete output:

```
D 1 222 5
A 10 222 3
B 10 22 33
C 10 222 33
E 20 222 5
F 20 222 10
```

3.5. Program Start and Termination

As indicated, the main module for a script, `__main__`, is a code block.

The Python interpreter starts a program by executing the main code block, which can subsequently invoke other code blocks, which can in

turn invoke other code blocks, and so forth. When the execution of a code block is completed, it returns the control to its surrounding code block which invoked the current code block.

When the execution of the main module code block is done, the program terminates.

3.6. Exceptions

The normal flow of a program, via the code block call chain, can be bypassed using the exception handling mechanism. An exception can be explicitly "raised" under an exceptional or error condition using the `raise` statement. The Python interpreter can also raise an exception when it detects a run-time error.

A raised exception may be handled by the surrounding, or any of the upstream, code blocks. When an exception is not handled in any of the code blocks, the interpreter terminates execution of the program, in the non-interactive mode. In the REPL mode, it simply returns to its interactive main loop. In either case, it prints a stack traceback, except when the exception is `SystemExit`.

Exception handlers are specified with the `try` statement. As of Python 3.11, a new `try - except*` syntax is also supported in addition to `try - except`.

There are two basic builtin exception types in Python, `BaseException` and `Exception`, which roughly correspond, for instance, to the "runtime exception" and "checked exception" of Java, respectively. Likewise, Python now includes (3.11+) two builtin exception group types, `BaseExceptionGroup` and `ExceptionGroup`.

Exception handling will be further discussed, later in the book, in the contexts of the `try` and `raise` statements.

Chapter 4. Packages & Modules

4.1. Modules

Python code is organized into "modules", which are associated with namespaces containing other Python *objects*. Python code in a module can access the code in a different module through the process of "importing". For example, a module can be imported using the `import` statement.

4.1.1. The `import` statement

The `import` statement performs the following operations:

- It first searches for the named module, and
  - If the module is found, then
    - It creates, and initializes, a module object, and it binds the object to a name in the local scope.
  - If the named module is not found, then
    - A `ModuleNotFoundError` exception is raised.

For example,

```
>>> import math                    ①
>>> type(math)                     ②
<class 'module'>                   ③
```

① An `import` statement.

② We will take a look at Python's type system throughout this book. The builtin `type` function returns the type of the given object.

③ The type of a module object is `module`.

The internal implementations of the `import` statement, as well as other importing related utility functions, are provided in the standard library module, `importlib`.

4.2. Packages

To help organize modules, Python has a concept of `packages`. A package is a special kind of module, and it provides a namespace/naming hierarchy. Technically, packages are modules that contains a `__path__` attribute.

If the Python interpreter is invoked with a script, then a package corresponds to a directory on a file system and a regular non-package module corresponds to a Python source file. Like file system directories, packages are organized hierarchically, and packages may contain other packages, as well as regular modules.

Every module has a name. Subpackage names are separated from their parent package name by a dot (`.`). A module's `__name__` attribute is set to the fully-qualified name of the module. Module's qualified names are used to uniquely identify the modules in the Python import system.

There are two kinds of packages in Python, "regular packages" and " namespace packages".

4.2.1. Regular packages

A regular package is typically implemented as a directory containing an *__init__.py* file. When a regular package is imported, the *__init__.py* files of the package and all of its parent packages are implicitly executed. The objects defined in each package's *__init__.py* file are bound to names in the package's namespace.

4.2.2. Namespace packages

An implicit namespace package is a (virtual) composite package that can include subpackages from different locations in the file system (e.g., not constrained by the directory hierarchy).

With namespace packages, there is no package *__init__.py* file. A namespace package may correspond to multiple directories unlike the regular packages. For example, when the packages `parent/one` and `parent/two` are imported with no physical directory corresponding to the `parent` module, Python automatically creates a namespace package for the top-level `parent` package.

4.3. Package Relative Imports

Relative imports use leading dots (`.`). A single leading dot indicates a relative import, starting with the current package. Two leading dots indicate a relative import of the parent to the current package, etc.

Absolute imports may use either the `import X.Y` or `from X import Y` syntax, but relative imports may only use the second form. For instance, from a given Python script/module `xyz` in a folder *bbb*,

aaa/bbb/xyz.py

```
from . import efg                ①
from .hij import klm             ②
from .. import mno               ③
from ..opq.qrs import stu        ④
```

① The module `efg` corresponds to a file *aaa/bbb/efg.py*.

② The module `klm` corresponds to a file *aaa/bbb/hij/klm.py*.

③ The module `mno` corresponds to a file *aaa/mno.py*.

④ The module `stu` corresponds to a file *aaa/opq/qrs/stu.py*.

Chapter 5. Python Source Code

The Python interpreter reads program text as Unicode code points. The encoding of a Python source code file is "UTF-8" by default. The source file can also be given an explicit encoding declaration. If an encoding is declared, the encoding name must be recognized by Python. If the text cannot be decoded, then a `SyntaxError` exception is raised.

The Python interpreter first breaks a source file into "tokens", which are then fed into the parser.

5.1. Line Structure

Python programs are "line-based". A Python source code consists of " logical lines".

5.1.1. Logical lines

A statement is normally contained in a logical line, whose end is represented by the NEWLINE token. A compound statement can be in multiple logical lines if that is permitted by the Python grammar.

A logical line is constructed from one or more "physical line".

5.1.2. Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In Python source files and strings, any one of the three standard line termination sequences can be used, `\n`, `\r\n`, or `\r`, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

5.1.3. Comments

A comment starts with a hash character (#) and it continues until the end of the physical line. Comments are mostly ignored by the syntax. A comment can signify the end of a logical line, that is, a NEWLINE token is generated unless the implicit line joining rules are invoked.

5.1.4. Blank Lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored.

5.1.5. Physical line joining

Explicit joining

Two or more physical lines may be joined into a logical line using backslash characters (\).

When a physical line ends with a backslash, it is joined with the following line forming a single logical line, deleting the backslash and the following end-of-line character.

- A backslash is not allowed anywhere else on a line outside a string literal or a comment.
- A backslash cannot split a token across physical lines except for string literals.
- A backslash does not continue a comment.

For example,

```
>>> a, b = \
... 10, 20
```

This is one logical line, `a, b = 10, 20`, although it is written on two physical lines. Note that no other characters, including white spaces, are allowed after the backslash other than one of the three line termination sequences.

Implicit joining

Expressions in parentheses (`(…)`), square brackets (`[…]`), or curly braces (`{…}`), as well as triple-quoted strings (`"""…"""` or `'''…'''`), can be split over more than one physical line without using backslashes.

The implicitly continued lines can carry comments except for triple-quoted multiline strings. The indentation of the continuation lines is not important. Blank continuation lines are allowed.

For instance,

```
>>> fruits = [                      ①
... "manzana",     # apple          ②
... "naranja",     # orange
... ]                                ③
>>> print(                           ④
...                                  ⑤
... fruits
... )                                ⑥
['manzana', 'naranja']
```

① The start of an implicit joining.

② Comments are allowed.

③ The end of a logical line.

④ The start of another implicit joining.

⑤ An empty physical line.

⑥ The end of a logical line. This statement is equivalent to `print(fruits)` in one physical line.

Note that since parentheses can be used to group expressions, practically any expression can be split into multiple physical lines.

5.1.6. Indentations

In Python, the indentations of the logical lines are used to determine the grouping of statements.

The indentation level of a line is computed based on the leading whitespace at the beginning of the line. The total number of spaces preceding the first non-blank character determines the line's indentation.

The Python lexical analyzer generates INDENT and DEDENT tokens, using a stack, based on the indentation levels of consecutive logical lines, which are then used to group logical lines into statements.

5.2. Tokens

Besides NEWLINE, INDENT and DEDENT, Python has the following categories of tokens: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*.

5.3. Identifiers and Keywords

The syntax of identifiers, or names, in Python is based on the Unicode standard. Identifiers are unlimited in length and they are case-sensitive. Within the ASCII range (U+0001..U+007F), the names can include the alphanumeric characters and the underscore (_). The names cannot start with digits. Outside the ASCII range, the `unicodedata` module classifies the characters which are valid in identifiers.

5.3.1. Keywords

The following names are used as reserved words, or keywords of the language, and they cannot be used as ordinary identifiers.

- `False True None and as assert async await break class continue def del elif else except from finally for global if import in is lambda nonlocal not or pass raise return try while with yield`

5.3.2. Reserved classes of identifiers

In addition to keywords, certain classes of identifiers have special meanings to Python. These classes are identified by the patterns of leading and trailing underscore characters:

`_`

The special identifier _ is used in the interactive interpreter to store the result of the last evaluation.

`_xyz`

Not imported by `from module import *`.

`__xyz__`

System-defined names, informally known as "dunder names". These names are defined by the interpreter and its implementation (including the standard library).

`__xyz`

Class-private names. When used in a class definition, these names are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes.

5.4. Literals

"Literals" are a particular class of tokens that have special meanings to the lexer, other than identifiers and other special tokens. Usually, literals are constant values of some built-in types.

5.4.1. String and bytes literals

String and bytes literals can be enclosed in matching single quotes (`'`) or double quotes (`"`). They can also be enclosed in matching groups of three single (`'''`) or double (`"""`) quotes.

The backslash (`\`) character is used within string and bytes literals to "escape" characters that otherwise have special meanings, such as newline, backslash itself, or the quote characters.

Bytes literals, instances of the `bytes` type, are always prefixed with `'b'` or `'B'`. They may only contain ASCII characters. Both string and bytes literals may optionally be prefixed with a letter `'r'`/`'R'`. Such literals are called raw strings and treat backslashes as literal characters.

5.4.2. Formatted string literals

A *formatted string literal*, or *f-string* for short, is a string literal with prefix `'f'` or `'F'`. These strings may contain replacement fields, which are variables or expressions enclosed in curly braces `{}`. A format specifier may be added, after an optional colon `:`.

While other string literals always have constant values, f-strings are really expressions evaluated at run time.

An equal sign `=` can be added after the expression and before the colon and format string, if any. When it is included, the result string will have the expression text, the `=` sign, and the evaluated value, concatenated.

For example,

```
>>> a = 10 / 7
>>> f"{a}"                                  ①
'1.4285714285714286'
>>> f"a = {a:.5f}"                          ②
'a = 1.42857'
>>> f"{a=}", f"{10/7=:.5f}"                 ③
('a=1.4285714285714286', '10/7=1.42857')
```

① A simple f-string.

② The replacement field has a format specifier, `.5f`, after `:`.

③ The equal sign prints out the expression text, followed by `=`, before the value.

5.4.3. String literal concatenation

We can concatenate string expressions using the `+` operator. As a shortcut, in case of adjacent string literals, as well as bytes literals, they can be concatenated without using the `+` operator. They can be delimited by (possibly empty) whitespaces, and they have the same effect as the string concatenation using the `+` operator.

Literal concatenation can use different quoting styles for each operand. F-string literals may be concatenated with plain string literals. For instance,

```
>>> a = "hello"
>>> (                                       ①
... "hi" "hi"                               ②
... f"{a} world"                            ③
... """bye bye"""                          ④
... )
'hihihello worldbye bye'
```

① The parenthesized expression is used here to put the string literals

in multiple physical lines, for illustration.

② `"hi" "hi"` is equivalent to `"hi" + "hi"`.

③ An f-string literal can be concatenated with other string literals. Again, the `+` operator is optional.

④ A triple quoted string literal, which can span multiple physical lines, is concatenated with the adjacent formatted string literal, without the `+` operator in this example.

5.4.4. Numeric literals

There are three types of numeric literals: integers, floating point numbers, and imaginary numbers.

Integer literals must start with digits. Literals that start with `0x`/`0X`, `0o`/`0O`, and `0b`/`0B` are hexadecimal, octal, and binary numbers, respectively. Decimal numbers cannot start with the digit `0` except for the number `0`.

Underscores (`_`) are ignored for evaluating the value of an integer literal. Integer literals cannot start with an underscore, and multiple consecutive underscores are not allowed. For example,

```
0b_010          ①
0O000_010       ②
0x0F            ③
1_000_000       ④
```

① A binary integer literal representing a decimal number `2`.

② An integer literal in base 8, representing a decimal number `8`.

③ A hexadecimal literal equivalent to a decimal number `15`.

④ A decimal number (base 10). Note that decimal integer literals cannot start with `0`.

Floating point literals include a period (`.`) and/or exponent symbol (`e` or `E`). The integer and exponent parts are interpreted using base 10.

Examples of floating point number literals:

```
100.
03.14
.0101
0_0_0e0_0_0
31.415_927E-10
```

An imaginary literal, a floating point literal with a suffix `j` or `J`, yields a complex number with a real part of `0.0`. An expression `1j * 1j` yields a complex number `(-1+0j)`.

To create a complex number with a nonzero real part, one can use an expression that adds a floating point number to an imaginary number literal. For instance,

```
100j                        ①
03.14J                      ②
10E10 + 0.0001E-100j        ③
10 - 0.314_159e+10J         ④
```

① An imaginary literal.

② Another imaginary literal.

③ An expression comprising real and imaginary number literals.

④ Another expression whose type is `complex`.

5.5. Compound Type Literals

The literal syntax for tuples, lists, sets, and dictionaries are discussed in the Builtin Compound Types chapter.

5.6. Operators

The following tokens are operators in Python:

```
+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

5.7. Delimiters

The following tokens, other than whitespaces, serve as delimiters in the Python grammar:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
```

The following tokens, the augmented assignment operators, perform operations, and they are also lexical delimiters.

```
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

Chapter 6. Objects

In Python, data is represented by *objects*. An object has an *identity*, *type*, and *value*. An object may be referred to by a "name" (or, "reference"). An object has one and only one invariant identity, throughout its lifetime, but it can have zero, one, or more names/references at any given moment.

Python is a dynamically and loosely typed programming language. Although the type system is at the heart of the Python programming language, and the types play a crucial role in Python programs, they become relevant primarily at run time, and only indirectly. (Python's type system is often called "duck typing".) In fact, one can create a new type at run time and one can even change the type of an object at run time (although that is not generally a common practice).

The values of objects of some types can change. These objects are said to be "mutable". For example, builtin data types like lists and dictionaries are mutable types. On the other hand, for some other types, one cannot *directly* modify the values of the objects once they are created. They are called "immutable". For example, numbers, strings, and tuples are immutable types. (Note that the (effective) values of immutable objects can still change.)

6.1. Identities

The "identity" is a rather abstract concept, but Python includes a builtin `id` function that returns a unique value which can be used as the identity of a given object. In fact, the `id` function "defines" the identities of the objects. (In CPython implementation, it returns the given object's memory address.)

For example,

```
>>>                                                  ①
>>> a, b, c = 3, "the ring", ['a', 'b', 'c']       ②
>>> id(a), id(b), id(c)
(546842556784, 546841138928, 546841139008)         ③
>>> x = 3
>>> id(a), id(x), id(3)
(546842556784, 546842556784, 546842556784)         ④
```

① This prompt >>> indicates that we are in the REPL.

② We often use this "multiple assignment" syntax in this book. The expression list on the right hand side evaluates to a tuple. Each of its items is then assigned to `a`, `b`, `c`, through "tuple unpacking".

③ The names `a`, `b`, and `c` all have different `id` values. They refer to different objects.

④ The names `a` and `x` and the object `3` all have the same `id` values. `a` and `x` reference the same object, `3`.

Objects' identities can also be compared with the builtin `is` and `is not` operators. `x is y` evaluates to `True` if and only if `x` and `y` are the same object, that is, according to the `id` function. Note that the identity equality (`is`) between two objects implies their value equality (`==`).

Using the same example above,

```
>>> a is a, a is b, a is c      ①
(True, False, False)
>>> a is x, a == x              ②
(True, True)
```

① `(a is a) == True`. The names `a`, `b`, and `c` refer to different objects.

② `(a is x) == True` and `(x is a) == True`. Likewise, `a == x` and `x == a`.

6.2. Attributes

An object is associated with a set of "attributes". An attribute is a name that refers to (other) objects. When the object that it refers to is a function, the attribute is called a *method* (of the original object). Otherwise, it is called the data attribute (e.g., often known as "fields", or data members, in other programming languages).

An attribute of an object can be accessed using the attribute reference syntax (or, the "dot notation").

```
>>> class A: pass               ①
...
>>> a = A()                     ②
>>> a.magic_number = 42         ③
>>> a.magic_number              ④
42
>>> def f(): pass               ⑤
...
>>> a.empty_method = f          ⑥
>>> a.empty_method()            ⑦
```

① The `class` statement creates a new type, `A`.

② Calling `A()`, which is called a constructor, returns an instance object of the type `A`. The object is bound to a name `a` in this example.

③ Adds a data attribute, named `magic_number`, to `a`, and binds it to an (immutable) number object `42`.

④ `a.magic_number` refers to the object `42`, whose value is `42`. (Note that, for simple types, objects' identities and values are closely tied.)

⑤ The `def` statement creates a new function object, `f`.

⑥ The new attribute `a.empty_method` now refers to the object `f`.

⑦ The method call syntax invokes the function `f`.

The built-in `dir` function can be used

- To find all names in the current scope, or
- To list the attributes associated with a given object.

In the current context, with the above example,

```
>>> dir()                       ①
['A', '__annotations__', ... '__spec__', 'a', 'f']
>>> dir(A)                      ②
['__class__', ... '__weakref__']
>>> dir(a)                      ③
['__class__', ... '__weakref__', 'empty_method',
'magic_number']
>>> type(a)                     ④
<class '__main__.A'>
```

① Calling `dir()` without any argument returns the names in the current scope, as a list of strings, lexically ordered. Note that there are three names that we just introduced, `A`, `a`, and `f`, in addition to some system-defined names.

② Calling `dir(cls)` on a class object returns the attributes of the class `cls` and all of its base classes (including `object`).

③ Calling `dir(obj)` on a general (non-class) object returns a combined list of its own attributes (for `obj`) and the attributes returned by `dir()` for its type (and all base types).

④ In this example, the type of `a` is `A`. `magic_number` and `empty_method` are `a`'s own attributes, and the rest come from `A`.

Note that the `dir` function calls the object's `__dir__` method, if it exists.

```
>>> a.__dir__()                 ①
['magic_number', 'empty_method', '__module__', ...
```

```
 '__class__']
```

① This returns the same result as `dir(a)` (modulo the order).

One can overwrite a (user-defined) class's `__dir__` method to return a different list from the `dir()` function call.

6.3. Types

Every object in Python is associated with a "type" besides its identity and value. The type of an object affects the object's behavior. For instance, the type of an object defines all possible values that the object can have. Furthermore, the type of an object determines the set of operations that the object supports. (Note that objects (of most types) in Python are malleable, so to speak, and their behavior can change during the execution of the program, regardless of their designated types/type names. Cf. Duck typing.)

As another example, the objects of immutable types are "reused" by the Python interpreter, when possible. That is, certain operations that are supposed to return new objects, or compute new values, for immutable types may actually return a reference to any existing object *with the same type and value*. On the contrary, objects are never "reused" for mutable types in these situations. For example,

```
>>> def f():
...   a, b = (1, 2), [3, 4]
...   print(id(a), id(b))
...   p, q = (1, 2), [3, 4]
...   print(id(p), id(q))
...
>>> f()
510220577408 510220577600
510220577408 510221371520
```

Note that the `id`s of the tuples `a` and `p`, immutable objects, are the same whereas those of the lists `b` and `q`, mutable objects, are different. `a` and `p` point to the same object. On the other hand, `b` and `q` point to two different objects.

6.4. Builtin `type` Function

Python provides a builtin `type` function that returns the type/class of a given object. A `type` is, in fact, a "class object" that creates a type. One of its constructor functions takes an object as its argument and returns the object's type object (e.g., possibly "reused").

For example,

```
>>> type, type(type), type(type(type))          ①
(<class 'type'>, <class 'type'>, <class 'type'>)
>>> type(10), type('hello'), type([1, 2])       ②
(<class 'int'>, <class 'str'>, <class 'list'>)
```

① The type of `type` is `type`. ☺

② The types of objects `10`, `'hello'`, and `[1, 2]` are `int`, `str`, and `list`, respectively.

6.5. Mutable vs Immutable Types

In Python, the mutability/immutability is a characteristic of a type. An object of a mutable type is mutable, and an object of an immutable type is immutable.

One cannot directly change the value of an immutable object. However, that does not mean that the object is truly immutable. For example, an attribute of an immutable object references an object, and that object may still be mutable. And, the overall effective value of the immutable object may change.

This is generally true for compound types. For instance, for immutable container types like tuples, they can still contain mutable elements, and the overall value of a tuple can still change (although its value still may not be directly updated).

For example,

```
>>> a, b = (1, 2), [3, 4]           ①
>>> a[0] = 10                       ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> b[0], b[1] = 30, 40             ③
>>> a, b                            ④
((1, 2), [30, 40])
>>> x = a, b                        ⑤
>>> x                               ⑥
((1, 2), [30, 40])
>>> x[1] = [3000, 4000]             ⑦
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> x[1][0], x[1][1] = 300, 400     ⑧
>>> x                               ⑨
((1, 2), [300, 400])
```

① `a` references a tuple `(1, 2)` whereas `b` references a list `[3, 4]`.

② Tuples are immutable, and hence one cannot change its elements.

③ In contrast, one can change a list's items since lists are mutable.

④ `a` and `b` references the same objects as before, but `b`'s value has changed.

⑤ The comma-separated expression list on the right-hand side evaluates to a tuple object.

⑥ `x` refers to a tuple, an immutable object.

⑦ One cannot directly change its value since `x` is "immutable".

⑧ However, one can change the value of an item of a collection, or the value of an attribute of an object, if it is mutable. In this example, the second item of `x` happens to reference a mutable object, a list `[30, 40]`. Hence, we can change its value.

⑨ The effective value of the "immutable" `x` has indeed changed (although the items of `x` are still the *same* objects).

6.6. Constructors

All types in Python, built-in or user-defined, include constructor functions, or *constructors* for short. The name of the type is the same as the name of the constructor function. Or, more precisely, types/classes are "callable". A constructor, when called, returns an instance object of the given type/class.

For some builtin types such as tuples and lists, one can also use the literal syntax to create instances of those types as we explain in a later chapter, Builtin Compound Types.

```
>>> class A: pass               ①
...
>>> type(A)                     ②
<class 'type'>
>>> a = A()                     ③
>>> type(a)                     ④
<class '__main__.A'>
```

① A `class` statement creates a new class object, `A` in this example.

② The type of type `A` is `type`.

③ `A`, a class object, is callable.

④ Calling the constructor returns an instance object of type `A`.

6.7. Boolean Context

(Almost) all objects in Python have "truth values", which depend on their types and values by default. When an object is used in an `if` or `while` condition or as an operand of a Boolean operation, its truth value is returned. (The `and` and `or` operators work slightly differently.)

The truth value of an object is `True` by default. It is `False` in the following cases:

- If the object/its class has a `__bool__` method defined, and
  - This method returns `False`, or
- If the object does not have a `__bool__` method but the object/its class has a `__len__` method, and
  - This method returns `0`.

For built-in types, the truth values of the following objects are `False`:

- `None` and `False`.
- The "zero value" objects of all numeric types, e.g., `0`, `0.0`, `0j`, etc.
- Empty collections, e.g., `""`, `()`, `[]`, `{}`, etc.

The truth value of an object of a custom type is always `True` by default, regardless of the values of their attributes. In order to give a different behavior, e.g., having either `True` or `False` based on their internal states, we will need to implement the `__bool__` method. This method, by default, does not exist in a custom type.

If an object, or more precisely its class, implements this method, then it is called in the boolean context. For example,

```
class BiggerThan8:
    def __init__(self, zzz):
```

```
        self.zzz = zzz
    def __bool__(self):
         return True if self.zzz >= 8 else False
```

```
>>> z1, z2 = BiggerThan8(5), BiggerThan8(10)
>>> bool(z1), bool(z2)
(False, True)
```

The builtin `bool` function returns the Boolean value of the argument (object or expression, or name). This code showcases another (subtle) power of "duck typing". As long as the object `z` is of a type that has the method `__bool__(self)`, it works as expected. In other statically typed programming languages, this kind of functionality is provided through extra constructs, like interfaces, traits, and prototypes, etc.

6.8. Lifetime of an Object

In Python, we can create objects, e.g., using constructor functions, but there is no way to explicitly destroy them. Any objects that are unreachable, e.g., because they do not have any valid names referring to them, may be "garbage collected" by the Python runtime.

The `del` statement can be used to remove/unbind a name from a given object. Furthermore, Python's standard library module `gc` can used to explicitly control various aspects of the garbage collection process.

Note that some objects may reference system resources such as files and windows, which are controlled by the operating system. When such an object is garbage collected, those system resources are automatically released. To explicitly release those resources, one can add a cleanup logic (e.g., calling `file.close()`) in the `finally` block of the `try` statement. Or, one can include such logic in the `__exit__` method of a context manager, to be used with the `with` statement.

Chapter 7. Simple Types

A number of types are built into the Python programming language. They are called the "builtin types". Types can be classified into simple vs compound types. Python's builtin compound types are described in the next chapter.

7.1. None

The `NoneType` simple type has a single value, `None`, which generally indicates an absence of a (meaningful) value.

`None` has a number of uses in Python. For example, a function that does not explicitly return a value is considered to be returning `None`. That is, the value of a function call expression that does not return any concrete values like `100` or anything else is `None`.

```
>>> type(None)              ①
<class 'NoneType'>
>>> bool(None)              ②
False
>>> def f(): pass           ③
...
>>> print(f())              ④
None
```

① `None` is the only value of the type `NoneType`.

② The Boolean/truth value of `None` is `False`.

③ A function that does not return any value. *It returns none.*

④ The Python REPL does not print the value of an expression if it is `None`. We use the `print` function to explicitly print the value of the `f()` expression.

7.2. NotImplemented

The NotImplemented type is another singleton type, and it likewise has a single value, NotImplemented.

```
>>> type(NotImplemented)
<class 'NotImplementedType'>
>>> def f(): return NotImplemented       ①
...
>>> f()
NotImplemented
>>> if NotImplemented:                   ②
...     print("Not implemented")
...
<stdin>:1: DeprecationWarning: NotImplemented should not be
used in a boolean context
Not implemented
```

① NotImplemented is sort of an invalid value, and it cannot be used in numerical or comparison operations, for instance.

② The Boolean value of NotImplemented used to be True. It still is, as of this writing. But, its use has been deprecated, and if you use NotImplemented in a Boolean context, Python will raise a warning or error.

7.3. Ellipsis

An ellipsis literal, ... (or, Ellipsis), can be used in a few different places in Python programs where no specific value is needed (e.g., as a placeholder).

```
>>> type(...), bool(...)             ①
(<class 'ellipsis'>, True)
>>> Ellipsis is ...                  ②
```

```
True
```

① The Boolean value of ... is `True`.

② `Ellipsis` and `...` are synonyms. They point to the same singleton object.

7.4. Numbers

All builtin numeric types inherit from `numbers.Number` from the `numbers` module. Numeric objects are immutable. In fact, the objects of the simple types in Python are all immutable.

One thing to note is that `bool` is a numerical type in Python.

7.4.1. Integers (`numbers.Integral`)

The `numbers.Integral` type defines a set of integers.

`int`

The `int` type in Python represent (arbitrary-precision) integers.

`bool`

The `bool` type represent the truth values `False` and `True`. Their numeric values are `0` and `1`, and their string values are `"False"` and `"True"`, respectively.

```
>>> import numbers
>>> issubclass(int, numbers.Integral)
True
>>> issubclass(bool, int)
True
>>> isinstance(True, int), isinstance(10, int)
(True, True)
```

7.4.2. Real numbers (`numbers.Real`)

`float`

The `float` type, a subtype of `numbers.Real`, represents a set of double precision floating point numbers.

```
>>> import numbers
>>> issubclass(float, numbers.Real)
True
>>> isinstance(1.5, float), isinstance(1.2E-5, float)
(True, True)
```

7.4.3. Complex numbers (`numbers.Complex`)

`complex`

The `complex` type represents a set of all pairs of double precision floating numbers, namely, the real and imaginary parts.

```
>>> import numbers
>>> issubclass(complex, numbers.Complex)
True
>>> c = 1 + 2j
>>> isinstance(c, complex), isinstance(0.1 + .5j, complex)
(True, True)
>>> c.real, c.imag                ①
(1.0, 2.0)
```

① The `complex` type has attributes `real` and `imag`, which return the real and imaginary parts of the complex number object, respectively.

Chapter 8. Compound Types

Objects can be built from other objects. Likewise, types can be built from other types. They are called the complex or "compound types".

Python's builtin compound types such as tuples and lists, which we will discuss in this chapter, and the user-defined compound types, which we will discuss later in the book, are built using other simple and complex types as building blocks.

Python's builtin compound types fall into three categories,

- Sequences, such as `tuple` and `list`,
- Sets, such as `set`, and
- Mappings, such as `dict`.

We will go through each of these three categories in this chapter after a brief discussion on how to construct `tuples`, `lists`, `sets`, and `dicts`.

8.1. Tuples, Lists, Sets, and Dictionaries

The builtin collection types, tuples, lists, sets, and dictionaries, can be constructed in a few different ways.

8.1.1. Constructors

First, we can use type constructors, `tuple`, `list`, `set`, and `dict`.

```
>>> tuple()                ①
()
>>> list()                 ②
[]
>>> set()                  ③
set()
```

```
>>> dict()                  ④
{}
```

① Calling a `tuple()` type constructor function without any arguments creates an empty tuple, whose literal representation is `()`.

② Calling a `list()` constructor. It creates an empty list, `[]`.

③ This constructor call `set()` creates an empty set. Note that there is no literal syntax for an empty set.

④ This call `dict()` creates an empty dictionary, `{}`.

These constructors also accept arguments of an iterable type, for their initial elements. For instance,

```
>>> tuple((1, 2)), tuple([1, 2])
((1, 2), (1, 2))
>>> list((1, 2)), list([1, 2])
([1, 2], [1, 2])
>>> set((1, 2)), set([1, 2])
({1, 2}, {1, 2})
>>> dict([('a', 1), ('b', 2), ('c', 3)])       ①
{'a': 1, 'b': 2, 'c': 3}
```

① An iterable of two-element iterables.

In addition, the `dict` constructor supports a special keyword argument syntax. `dict` can also be initialized with other mapping object. For instance,

```
>>> x = dict(a = 1, b = 2)
>>> x
{'a': 1, 'b': 2}
>>> y = dict(x)
>>> y
{'a': 1, 'b': 2}
```

8.1.2. Literal syntax

Another way to create builtin collections is to use the literal syntax,

- Either by explicitly listing the elements, or
- By providing "instructions" as to how to generate the element list, which is called a *comprehension*.

Here are some examples of the explicit literal syntax:

```
(1, 2, 3)                    ①
[1, 2, 3]                    ②
{1, 2, 3}                    ③
{"a": 1, "b": 2, "c": 3}     ④
```

① This literal creates a `tuple` of three integer elements, `1`, `2`, and `3`, enclosed in a pair of parentheses (`(…)`). For a one-element tuple, a trailing comma is required. E.g., `(1,)`.

② A `list` literal, enclosed in a pair of square brackets (`[…]`).

③ A `set` literal, enclosed in a pair of curly braces (`{…}`).

④ This literal creates a `dict` of three key-value pairs, `("a", 1)`, `("b", 2)`, and `("c", 3)`. Dictionary literals use the same curly braces (`{…}`) as sets. The literal `{}` represents an empty dictionary.

8.1.3. Comprehensions

Lists, sets, and dictionaries (but not tuples) can also be constructed using the comprehension syntax. The comprehension consists of the following general components:

- An expression, followed by
- At least one `for` clause, and
- Zero or more `if` or (nested) `for` clauses.

The `for` clause has the following general syntax:

- The `for` keyword, or `async for`,
- The target variable list, and
- The `in` keyword, followed by
- An iterable, or an `async` iterable, respectively.

8.2. Sequences

Sequences are ordered sets indexed by non-negative integers. The `i`-th item of a sequence `a` is selected by `a[i]`. The index runs from 0 to `len(a) - 1`. `len` is a built-in function that returns the number of items in a sequence.

"Slicing" a sequence produces another sequence of the same type, called a slice. For example, `a[i:j]` selects all items with index between `i` (inclusive) and `j` (exclusive). The index of a slice starts from `0` regardless of how it is sliced.

8.2.1. Sequence unpacking

An expression list evaluates to a `tuple` object. When the resulting tuple is assigned to a single variable, it is called "tuple packing":

```
>>> x = 1, 'a', True              ①
>>> x
(1, 'a', True)
```

① "Tuple packing". The type of `x` is `tuple`. The three values on the right-hand side have been "packed" into one variable.

On the flip side, a sequence type object can be "unpacked" to multiple variables.

```
>>> p, q, r = ['a', 'b', 'c']      ①
>>> p, q, r
('a', 'b', 'c')
```

① A single list with three items on the right-hand side has been "unpacked" to three variables, `p`, `q`, and `r`.

Normally, the number of variables must be the same as that of elements in the sequence. Python, however, supports a special syntax for the "remaining elements". For example,

```
>>> p, *q = [1, 2, 3, 4]          ①
>>> p, q
(1, [2, 3, 4])
```

① At most one variable in the unpacking syntax can be marked with `*`. This variable gets all the remaining elements from the sequence after each leading or trailing element has been mapped to the corresponding variable. Note that the type of `q` is `list`.

8.3. Immutable Sequences

Objects of immutable sequence types cannot change once created.

8.3.1. Strings

A `string` is a sequence of values that represent Unicode code points. Python does not have a character type. The built-in function `ord` converts a code point from its string form to an integer in the range `0 - 0x10FFFF`. On the other hand, `chr` converts an integer in the range `0 - 0x10FFFF` to the corresponding `length-1` string object.

Strings can be concatenated with the `+` operator. Adding two strings returns a *new* string. Strings are truly immutable in Python. The string

type includes a number of builtin methods like `count`, `index`, `find`, `replace`, `format`, `join`, `split`, `startswith`, `endswith`, `upper`, `lower`, `isalnum`, `isalpha`, `islower`, `isupper`, and `isspace`.

8.3.2. Tuples

A `tuple` is an immutable sequence of zero, one, or more arbitrary Python objects within a pair of parentheses. The *length* of a tuple, that is, the number of elements, or items, in the tuple, can be computed using the Python builtin function, `len`.

```
>>> len((False, "Hello", 1_000_000))
3
```

8.3.3. Bytes sequences

Python does not have a byte type. Instead, it has a `bytes` sequence type, which is an immutable array of 8-bit bytes.

8.4. Mutable Sequences

Items of a mutable sequence can be changed after they are created, e.g., using the assignment or `del` (delete) statements.

8.4.1. Lists

The `list` type is another important builtin data type in Python. A list is a mutable sequence of zero, one, or more objects. A list can be modified using the builtin `list` methods such as `append`, `insert`, `pop`, and `remove`, etc. The order of the elements in a list can be updated using the methods, `reverse` and `sort`.

As with tuples, the `len` builtin function can be used to get the number of elements in a list. For instance,

```
>>> len([1.0, 2.0, 3.0])
3
```

8.4.2. List comprehension

A `list` can be created by using the list comprehension syntax, in addition to the explicit literal syntax. For example,

```
>>> [ x*x for x in range(5) ]                  ①
[0, 1, 4, 9, 16]
>>> [ x*x for x in range(5) if x % 2 == 0 ]    ②
[0, 4, 16]
```

① This expression evaluates to `[0, 1, 4, 9, 16]`.

② The `if` clause can be used to filter elements.

The list comprehension, as well as the list literal expression, yields a *new* list object every time they are evaluated.

8.4.3. Byte arrays

`bytearray` objects are like `bytes` sequences, but they are mutable. A `bytearray` object is created by the built-in `bytearray` constructor.

8.5. Set Types

The set types represent unordered sets of unique, immutable objects. (Elements of set types are comparable to the keys of the mapping types.) The items in a set cannot be indexed using the subscription syntax. But they can be iterated over, as with other sequence types. (That is, a set is an `iterable`.) The `len` function returns the number of items in a set.

8.5.1. Sets

A `set` represents a mutable set. They can be created by the built-in `set` constructor, as illustrated earlier.

8.5.2. Set comprehension

A `set` can be created using the set comprehension syntax, in addition to the explicit set literal syntax. For example,

```
>>> type({1, 3, 5, 7})                ①
<class 'set'>
>>> { x*2 for x in range(5) }         ②
{0, 2, 4, 6, 8}
```

① A set literal, `{1, 3, 5, 7}`.

② This comprehension expression evaluates to `{0, 2, 4, 8, 6}`.

A set comprehension, as well as the literal expression with a comma-separated list of elements, returns a *new* mutable set object each time they are evaluated.

8.5.3. Frozen sets

The built-in `frozenset` constructor creates a `frozenset` object, which is an immutable set.

8.6. Mappings

Mapping types represent ordered or unordered sets of objects indexed by arbitrary index sets, or keys. The built-in function `len` returns the number of items in the given mapping object. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`. This can be used, for example, as the target of assignments or `del` statements.

8.6.1. Dictionaries

A dictionary is the builtin mutable mapping type in Python. They represent finite sets of objects indexed by the values of any type that support constant hash values. Dictionaries preserve insertion order. That is, keys will be produced in the same order they were added sequentially to the dictionary.

8.6.2. Dictionary comprehension

A new `dict` object can be created using the dictionary comprehension syntax, in addition to the dictionary literal syntax by listing elements. For example,

```
>>> { str(x): x**2 for x in range(4) }          ①
{'0': 0, '1': 1, '2': 4, '3': 9}
```

① Note the expression syntax (`k:v`) before the `for` clause.

8.6.3. Element insertion/deletion

```
>>> d = {"a": 1}
>>> d['b'] = 2                  ①
>>> del(d['a'])                 ②
>>> d
{'b': 2}
```

① Overwrites an existing element if the element exists with a given key. Otherwise, it inserts a new element.

② The `del` statement is described later in the book.

Chapter 9. Expressions

An "expression" comprises operators and operands and it evaluates to a value. A value is (trivially) an expression.

In Python, the building blocks of an expression are called "atoms". Atoms include lexical tokens like identifiers and literals, and they can be nested. Expressions enclosed in parentheses, square brackets or curly braces are also syntactically atoms.

At the next level up, the following "primaries" represent the most tightly bound operations:

Attribute References

A primary followed by a period and a name, called the attribute reference, is also a primary. An object's attribute reference can be customized by overriding its `__getattr__` method.

Subscriptions

Subscription, or indexing, of a sequence (e.g., string, tuple, or list) or mapping object (e.g., dictionary) selects an item from the given collection. User-defined objects can support subscription by implementing a `__getitem__` method.

Slicings

A slicing operation selects a range of items in a sequence object. Slicings may be used as expressions or as targets in assignment or `del` statements. The primary is indexed (using the same `__getitem__` method as the subscription) with a key that is constructed from the slice list, as follows.

- If the slice list contains at least one comma, the key is a `tuple` containing the conversion of the slice items.
- Otherwise, the conversion of the lone slice item is the key.

Calls

A "call" calls a callable object (e.g., a `function` or `class`) with a (possibly empty) series of arguments. All objects having a `__call__` method are callable such as built-in functions, methods of built-in objects, class objects, methods of class instances, and user-defined functions. All argument expressions are evaluated before the call is attempted.

There are a few different kinds of expressions in Python. An expression can comprise one or more other (sub-)expressions, e.g., combined with operators.

- Arithmetic conversions.
- Arithmetic operations, unary and binary.
- Bitwise operations, unary and binary.
- Shifting operations.
- Comparisons (`==`, `!=`, `>`, `>=`, `<`, `<=`)
- Boolean operations (`not`, `and`, `or`).
- Assignment expressions.
- Conditional expressions.

Lambda expressions and `await` expressions are described later.

9.1. Expression Lists

An expression list is a series of one or more expressions, separated by commas. A trailing comma is optional, except for an expression list comprizing a single expression. The expressions in an expression list are evaluated from left to right.

An expression list of one or more expressions yields a tuple object with the corresponding number of elements. For instance,

```
>>> 50 + 50, 100 * 2            ①
(100, 200)                      ②
```

① The Python interpreter first computes `50 + 50`, which is evaluated to `100`, and then it computes `100 * 2`, which is evaluated to `200`.

② The value of this expression list is a tuple, e.g., as represented by a tuple literal `(100, 200)` in this case.

```
>>> 1 / 2                       ①
0.5
>>> 1 / 2,                      ②
(0.5,)
```

① The value of an expression `1 / 2` is `0.5`.

② The value of an expression list `1 / 2,` is `(0.5,)`. The trailing comma is required for single-expression expression lists.

The "iterable unpacking" syntax, using the asterisk `*` operator, may be used in the expressions of the `iterable` types in an expression list.

```
>>> a, b, c = 1, [2, 3, 4], 5       ①
>>> a, b, c
(1, [2, 3, 4], 5)
>>> a, *b, c                        ②
(1, 2, 3, 4, 5)
>>> *b,                             ③
(2, 3, 4)
```

① `b` is a list, an iterable type.

② Unpacking `b`. The elements of `b` is now a part of the resulting tuple.

③ Unpacking in an expression list comprising a single iterable expression. Note that the trailing comma is required.

9.2. Evaluation Order

As a general rule, Python evaluates expressions from left to right, based on the operator precedence rules.

When evaluating an assignment, the right-hand side is evaluated before the left-hand side. In an "augmented assignment" expression, however, the left hand side target is evaluated first before the right hand side. Then the final result is assigned back to the left hand side target.

```
>>> a, b = 1 + 2, 2 * 5             ①
>>> a, b
(3, 10)
>>> a += 1 + 2 * 3                  ②
>>> a
10
```

① The expressions, `1 + 2` are `2 * 5` evaluated first (e.g., from left to right) and the resulting value (a tuple) is "unpacked" and assigned to `a` and `b`, respectively.

② In this augmented assignment, the (current) value of `a` is evaluated first, which is `3`. The right hand side is next evaluated to `7` (since the multiplication `2 * 3` has a higher precedence than the addition `+`). `3 + 7` is then evaluated and its result is assigned back to `a`.

9.3. Assignment Expressions

An assignment expression assigns an expression on the right hand side of the assignment operator `:=` (informally called the "walrus" operator) to a name on the left hand side, and it returns its value (which is the same as that of the target).

Assignment expressions are commonly used in compound statements where the result of an expression evaluation needs to be retained, e.g.,

so that it can be used in the statement suite. For instance,

```
>>> import random
>>> def zero_or_not():
...     return random.randint(0, 4)
...
>>> while r := zero_or_not():              ①
...     print("Not zero:", r)               ②
...
Not zero: 1
Not zero: 1
Not zero: 4
```

① The `while` clause depends on the result of the function call, `zero_or_not()`.

② The *same value* is used in each iteration. Hence, the value was previously stored in the variable `r`.

9.4. Conditional Expressions

In addition to the conditional statement, Python also supports the conditional expression, through the `if - else` expression syntax. In the expression `x if C else y`, comprising three (sub-) expressions, `C`, `x`, and `y`, if `C` evaluates to `True`, then the value of the expression is `x`. Otherwise, its value is `y`.

For instance,

```
family = input("What is your family name? ")
print(f"You are {'a reptile' if family == 'Python' else
'nobody'}!")
```

Python's `if` expression corresponds to the ternary operator `? :` in other C-style languages. An expression `x if C else y` is roughly

equivalent to `C ? x : y` in those languages.

Note that *only two* of these tree expressions are evaluated regardless of the value of `C`. The condition `C` is always evaluated first, and if it true, then `x` evaluated, but not `y`. Otherwise, `y` is evaluated, but not `x`.

9.5. Arithmetic Conversions

When a binary arithmetic operation is performed,

- If either argument is a complex number, the other is converted to complex, and
- Otherwise
  - If either argument is a floating point number, the other is converted to floating point, and
  - Otherwise, both arguments are integers and Python does no automatic conversion.

9.6. Arithmetic Operations

9.6.1. Unary arithmetic operators

The unary `-` (minus) operator yields the negation of its numeric argument. The unary `+` (plus) operator yields its numeric argument unchanged. In either case, if the argument does not have the proper type, a `TypeError` exception is raised.

All unary arithmetic and bitwise operations have the same priority.

9.6.2. Binary arithmetic operators

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, in which case they are

multiplied together, or one argument must be an integer and the other must be a sequence, in which case sequence repetition is performed. A negative repetition factor yields an empty sequence.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. Division of floats or integers yields a float, while floor division of integers results in an integer. Division by zero raises a `ZeroDivisionError` exception.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. A zero right argument raises a `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., `3.14 % 0.7` equals `0.34` (since `3.14` equals `4 * 0.7 + 0.34`). The modulo operator always yields a result with the same sign as its second operand.

The floor division and modulo operators are connected by the following identity: `x == (x //y ) * y + (x % y)`. Floor division and modulo are also connected with the built-in function `divmod`: `divmod(x, y) == (x // y, x % y)`.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers, in which case they are added together, or both be sequences of the same type, in which case the sequences are concatenated. The `-` (subtraction) operator yields the difference of its arguments.

9.6.3. The power operator

The power operator (`**`) takes two arguments and it works the same way as the built-in `pow` function. That is, `x ** y` is equivalent to `pow(x, y)`, which yields the value `x` raised to the power of `y`.

For example,

```
>>> 2 ** 3                    ①
8
>>> 2 ** -3                   ②
0.125
>>> -2 ** 0.5                 ③
-1.4142135623730951
>>> (-2) ** 0.5               ④
(8.659560562354934e-17+1.4142135623730951j)
>>> 10 ** 0                   ⑤
1
>>> 0.0 ** 2                  ⑥
0.0
>>> 0.0 ** -2                 ⑦
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: 0.0 cannot be raised to a negative power
```

① `2 ** 3` is the same as `2 * 2 * 2`, which is an integer `8`.

② `2 ** -3` is the same as `1.0 / (2 * 2 * 2)`, which yields a `float` number even though both operands are `int`.

③ `-2 ** 0.5` is the same as `- (2 ** 0.5)`, which is `-1` times the square root of `2`.

④ On the other hand, a fractional power over a negative number returns a complex number.

⑤ `x ** 0` yields `1` regardless of `x`. Likewise, `x ** 0.0` is always `1.0`.

⑥ `0 ** x`, for integer and float `x`, yields `0` and `0.0`, respectively. `0.0 ** x` always returns `0.0` regardless of the type and value of `x`.

⑦ Zero (`0` or `0.0`) to the power of a negative number raises a `ZeroDivisionError` exception.

9.7. Bitwise Operations

9.7.1. Unary bitwise operator

The unary `~` (inversion) operator only applies to integral numbers. It yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. If the argument does not have the proper type, a `TypeError` exception is raised.

9.7.2. Binary bitwise operators

- The bitwise AND (`&`) operator yields the bitwise AND of its integer arguments.
- The bitwise XOR (`^`) operator yields the bitwise exclusive OR of its integer arguments.
- The bitwise OR (`|`) operator yields the bitwise (inclusive) OR of its integer arguments.

```
>>> 0b0 & 0b0, 0b0 & 0b1, 0b1 & 0b0, 0b1 & 0b1
(0, 0, 0, 1)
>>> 0b0 ^ 0b0, 0b0 ^ 0b1, 0b1 ^ 0b0, 0b1 ^ 0b1
(0, 1, 1, 0)
>>> 0b0 | 0b0, 0b0 | 0b1, 0b1 | 0b0, 0b1 | 0b1
(0, 1, 1, 1)
```

9.7.3. The shift operators

Both left shift (`<<`) and right shift (`>>`) operators accept integers as arguments. They shift the first argument to the left `<<` or to the right `>>` by the number of bits given by the second argument.

- A right shift by `n` bits is defined as floor division by `pow(2,n)`.
- A left shift by `n` bits is defined as multiplication with `pow(2,n)`.

9.8. Boolean Operations

An expression that evaluates to a `bool` value, either `True` or `False`, is called a Boolean expression. Hence `True` is a (trivial) Boolean expression, and so is `False`.

9.8.1. The `not` operator

The operator `not` yields

- `True` if its argument is false, or
- `False` otherwise.

9.8.2. The `and` operator

The `and` operation `x and y` first evaluates *x*, and

- If *x* evaluates to false, its value is returned (which may not be Boolean).
- Otherwise, *y* is evaluated, and the resulting value is returned.

Note that the type of the overall expression is effectively the union of the types of `x` and `y` (not necessarily a Boolean type). That is, the type of the `and` expression is that of `x` if `x`'s Boolean value is false. Otherwise, it is the type of `y`.

9.8.3. The `or` operator

The `or` operation `x or y` first evaluates *x*, and

- If *x* evaluates to true, then its value is returned (which again may not be `True`).
- Otherwise, *y* is evaluated and the resulting value is returned.

The type of the overall expression `x or y` is likewise the union of the types of `x` and `y`. That is, the type of the `or` expression is that of `x` if `x`'s Boolean value is true. Otherwise, it is the type of `y`.

9.9. Comparisons

Comparisons yield boolean values: `True` or `False`. Comparisons in Python can be chained arbitrarily. For example, `x < y <= z` is a valid expression unlike in many other C-style languages, and it is more or less equivalent to `x < y and y <= z`. Or more precisely, it is equivalent to `x < (t := y) and t <= z`, using a temporary variable `t`. Note that `y` is evaluated only once.

In general, if `a`, `b`, `c`, ..., `y`, `z` are expressions and `op1`, `op2`, ..., `opN` are binary comparison operators, then `a op1 b op2 c … y opN z` is semantically equivalent to `a op1 b and b op2 c and … y opN z`, except that each expression is evaluated at most once.

9.9.1. Identity comparisons

The operators `is` and `is not` test for an object's identity: `x is y` is true if and only if *x* and *y* are the same object. The object's identity is determined using the `id` function. `x is not y` yields the inverse truth value.

9.9.2. Value comparisons

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the values of two objects. The objects do not need to have the same type.

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing comparison dunder methods such as `__lt__` and `__gt__`, etc.

The default behavior for equality comparison (`==` and `!=`) is based on the identity of the `objects`. Hence, the equality comparison of instances with the same identity results in equality, and the equality comparison of instances with different identities results in inequality, which may be counter-intuitive in many cases. In general, user-defined types will need to customize their comparison behavior.

9.9.3. Membership test operations

For collections, the operators `in` and `not in` test for membership. `x in s` evaluates to `True` if `x` is a member of `s`, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and `set` types support this. For `dict`, the `in` operator tests whether the dictionary has a given key.

For the `string` and `bytes` types, `x in y` is `True` if and only if `x` is a substring of `y`. The operator `not in` is defined to have the inverse truth value of `in`. For example,

```
>>> 2 in [1, 2, 3, 2]
True
>>> 'c' not in { 'a', 'b', 'd'}
True
>>> "k1" in { "k1": "v1", "k2": "v2" }
True
>>> "world" in "hello world"
True
```

For user-defined classes which define the `__contains__` method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

Chapter 10. Simple Statements

Statements, unlike expressions, are primarily used for side effects, e.g., to control program flows or to generate outputs. A simple statement comprises a single logical line. Several simple statements may occur on a single line separated by semicolons. We will discuss the compound statements in the next chapter.

Simple statements include:

- Expression statement,
- Assignment statement,
- `assert` statement,
- `pass` statement,
- `del` statement,
- `return` statement,
- `raise` statement,
- `break` statement,
- `continue` statement,
- `global` statement, and
- `nonlocal` statement.

The `import` statement was explained in the beginning of the book, Package and Modules. The `yield` statement is described in the context of the coroutines, or more specifically, the `yield` expressions.

10.1. Expression Statement

In Python, an expression, or an expression list in general, can be used as a statement, e.g., solely for its side effects. This is called an

expression statement, and it evaluates the expression list and discards its result. The most common use case is to call a function that returns `None`. Another common use case is using a constant string expression for documentation purposes.

For example,

```
def a():
  "Nada"                        ①
  return "Something"

a()                             ②
```

① The function definition `a()` includes an expression statement, which is a string literal.

② Calling the function `a()` is an expression, but it is written as a statement here. The function's return value, `"Something"`, is discarded.

10.2. Assignment Statement

Assignment is one of the simple statements in Python. Assignment statements are primarily used to bind a new name, or rebind an existing name, to an object. They are also used to modify attributes or items of a mutable object.

The left-hand side of an assignment statement can be

1. A new name or a name referring to an object, or
2. An attribute or item of a mutable object.

An assignment statement evaluates the expression, or the expression list, on the right hand side, from left to right, and it assigns the single resulting object to the corresponding target list on the left hand side,

from left to right. If the target list is a single target with no trailing comma, the object is assigned to that target. (As indicated, the assignment of an expression list to a target list is a combination of the tuple packing and sequence unpacking.)

In Python, we can also bind multiple names to a single object in one statement. For example,

```
>>> x = y = "dragon"                ①
>>> p = q = ['a', 'b']              ②
>>> id(x), id(y)
(140593323269680, 140593323269680)
>>> id(p), id(q)
(140593324864960, 140593324864960)
```

① The variables `x` and `y` refer to the same object.

② `p` and `q` refer to the same object.

10.2.1. Augmented Assignment Statements

An augmented assignment statement is the combination of a binary operation and an assignment statement. The following operators are used in augmented assignment: `+=`, `-=`, `_=`, `@=`, `/=`, `//=`, `%=`, `_*=`, `>>=`, `<<=`, `&=`, `^=`, and `|=`.

Unlike normal assignments, an augmented assignment evaluates the target on the left-hand first, and then the expression list on the right-hand side. It then performs the binary operation and writes back the result to the target.

Python's assignment statement does not make a copy of the source object, unlike in some other programming languages. It merely binds, or rebinds, a name to the object. For copying, the standard library `copy` module provides functions for both shallow and

> deep copying.

10.3. The pass Statement

The pass statement is used as a placeholder, e.g., in places where the Python grammar requires a statement. When this statement is executed, nothing happens. pass is a null operation. It is primarily used during development, and it serves no other purposes.

For example,

```
def find_numbers(arg):
    pass        # To be implemented     ①
```

① The function definition syntactically requires at least one statement.

```
class OrdinalNumber:
    pass        # Placeholder           ①
```

① The same with the class definition. Clearly, these comments like `"Placeholder"` are not needed since the use of the pass statements generally implies that they are placeholders.

10.4. The return Statement

A return statement is only allowed syntactically within a function definition. When executed, it leaves the current function call. A return statement can include an optional argument, an expression list. If present, the expression list is returned to the caller as the value of the function call. If not, the None return value is assumed.

```
>>> def f():
...     return 1, 2, 3      ①
```

```
...
>>> f()
(1, 2, 3)                    ②
```

① The function `f` returns an expression list with three expressions.

② The value of the function call expression `f()` is a tuple, e.g., through the tuple packing.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before leaving the function.

10.5. The `raise` Statement

A `raise` statement raises, or throws, an exception or an exception group:

- If an argument is provided,
  - If it evaluates to an instance of `BaseException` or its subtype, it raises the exception/exception group object.
  - If the expression is a class, then it creates an instance of the class constructed with no argument.
- If no argument is provided,
  - If an exception is active in the current scope, it re-raises this exception.
  - Otherwise, it is a `RuntimeError`.

When a new exception is raised when another exception is currently active, the new exception can be tied to the existing exception using the `__cause__` attribute. This is known as the "exception chaining". This can be accomplished by attaching the `from` clause at the end of the `raise` statement.

```
>>> class UnusualException(Exception): pass     ①
...
>>> raise UnusualException                      ②
Traceback (most recent call last):              ③
  File "<stdin>", line 1, in <module>
__main__.UnusualException
```

① A new exception type should inherit from `Exception(BaseException)`.

② An exception that inherits from `BaseException` can be *raised.*

③ The Python REPL catches the exception. Normally, we will need the `try - except` statement to catch the raised exceptions. Otherwise, the program will crash.

```
>>> try: raise UnusualException
... except: raise                               ①
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.UnusualException
```

① Re-raising the current exception.

```
>>> try:
...     raise UnusualException("Unusual")
... except BaseException as ex:                 ①
...     raise UnusualException("Really unusual") from ex ②
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.UnusualException: Unusual

The above exception was the direct cause of the following
exception:
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.UnusualException: Really unusual
```

① The `as` clause is used to name an exception.

② Exception chaining using the `raise - from` syntax.

10.6. The `break` Statement

A `break` statement can be used in a `for` or `while` loop, only directly included, and not as part of a function of class definition within the loop. It terminates the innermost enclosing loop. If the loop has an `else` clause, it is skipped.

```
>>> for i in range(3):                  ①
...     try:
...         print(i)
...         if i == 1:
...             break                   ②
...         print(i)
...     finally:
...         print("finally")
... else:
...     print("huh?")                    ③
...
0
0
finally
1                                       ④
finally                                 ⑤
```

① A `for` loop with `range` function.

② A `break` statement in this `for` loop.

③ The `else` clause is not executed when the loop is terminated with a

`break` statement.

④ The `break` statement is executed after printing the first `1`. The second print statement is skipped.

⑤ The `finally` clause is still executed before exiting the loop.

10.7. The `continue` Statement

Just like `break`, a `continue` statement can only be used directly in a `for` or `while` loop. It continues with the next cycle of the innermost enclosing loop, without executing the rest of the statements in the current loop.

```
>>> for i in range(3):                  ①
...     try:
...         print(i)
...         if i == 1:
...             continue                ②
...         print(i)
...     finally:
...         print("finally")
... else:
...     print("huh?")
...
0
0
finally
1                                       ③
finally                                 ④
2                                       ⑤
2
finally
huh                                     ⑥
```

① A `for` complex statement.

② A `continue` statement in this `for` loop.

③ The `continue` statement is executed after printing the first `1`. The second print function call is skipped.

④ The `finally` clause is still executed before continuing with the next iteration.

⑤ The next iteration continues.

⑥ The `else` clause is executed when the loop is normally terminated.

10.8. The `global` Statement

The `global` and `nonlocal` statements were briefly discussed earlier in the context of code blocks and scopes.

The `global` statement, the keyword `global` followed by a comma-separated list of names, indicates that the declared names, in the current code block, are to be interpreted as globals (e.g., the current module scope). The names listed in a `global` statement cannot be used in the same code block before that `global` statement.

Note the asymmetry between name binding and use in Python. There is no way to bind a value to a global variable without using the `global` declaration in a non-global code block. On the other hand, to access/read the value of a global variable, the `global` declaration is not needed.

```
>>> def f():
...     print(x)                ①
...     global y                ②
...     y = 4                   ③
...     print(y)
...
>>> x, y = 1, 2                 ④
>>> f()                         ⑤
```

```
1
4
```

① No `global` declaration is needed to access the (later-defined) global variable, `x`, in the function scope. Note that the variable `x` is used within the function `f` without being declared first. This is called a " free variable".

② This `global` declaration allows the global variable, `y`, to be assignable.

③ *After* the `global` statement, we can assign a value to the global variable. Now, `y` is bound to a different object `4` in this example.

④ The `global` variables, `x` and `y`, are declared.

⑤ The function call `f()` uses these global variables.

10.9. The `nonlocal` Statement

The `nonlocal` statement works in a similar way as the `global` statement. The names declared in a `nonlocal` statement refer to the *previously bound* variables in the innermost enclosing scope (excluding globals).

Some examples were given earlier, in the Scope Examples section. Here's another example:

```
>>> def a():
...     x, y = 1, 2                ①
...     def b():                   ②
...         print(x)               ③
...         nonlocal y             ④
...         y = 4                  ⑤
...         print(y)
...     b()
...
```

```
>>> a()
1
4
```

① `x` and `y` are variables local to the code block of the function definition for `a`.

② The start of a new (nested) code block.

③ `x` refers to the variable `x` declared outside the local scope (for `b`).

④ In order to be able to assign a new value to the non-local variable `y`, the variable needs to be declared as `nonlocal`.

⑤ The variable `y` refers to the one defined in the scope of `a`, but outside the scope of `b`.

10.10. The `del` Statement

A `del` statement deletes the listed target name(s). Deletion of a target list recursively deletes each target, from left to right. If a target name is unbound, a `NameError` exception will be raised.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block.

For example,

```
>>> x = 1
>>> def f():
...     global x
...     del x                          ①
...
>>> print(x)
1
>>> f()                               ②
>>> print(x)                          ③
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

① This `del` statement deletes the global variable `x`.

② Calling `f()` will execute the `del` statement.

③ Attempting to use a deleted/unbound name will raise a `NameError` exception.

10.11. The `assert` Statement

An `assert` statement is a convenient way to insert debugging assertions into a program. There are two forms.

10.11.1. The basic form

```
assert <expression>
```

This is equivalent to:

```
if __debug__:
    if not <expression>: raise AssertionError
```

Note that the values for the built-in variables, such as `__debug__`, are determined when the Python interpreter starts and they cannot be modified. The variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The `assert` statement generates no code when optimization is requested at compile time.

10.11.2. The extended form

```
assert <expression1>, <expression2>
```

is equivalent to

```
if __debug__:
    if not <expression1>: raise AssertionError(<expression2>)
```

For example,

```
>>> a = 1, 2, 3,                          ①
>>> assert a == (1, 2, 3)                 ②
>>> assert a == [1, 2, 3]                 ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

① An expression list evaluates to a tuple.

② This `assert` statement succeeds, and hence there is no output.

③ This assertion fails and it throws an `AssertionError` exception.

Chapter 11. Compound Statements

A compound statement comprises other statements, and it affects or controls the execution of those statements in some way. A compound statement can span multiple (logical) lines.

A compound statement consists of one or more "clauses". A clause consists of a "header" and a "suite". The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword (e.g., `if`, `else`, `try`, `except`, etc.), and it ends with a colon `:`. A suite is a group of one or more statements controlled by the clause. A suite can be

- One or more semicolon-separated simple statements on the same line as the header, following the colon, or
- It can be one or more one-level more indented simple or compound statements on subsequent lines.

Python uses, as in many imperative programming languages, the common control flow statements such as the `if` statement, `while` statement, and `for` statement. They are all compound statements in Python. The `for` statement is often used with the builtin `range` and `enumerate` functions.

The `try` statement, along with the simple `raise` statement, is used for exception handling and/or for providing cleanup code. Another compound statement, the `with` statement, which is closely related to the context manager, is used to provide initialization and finalization code for a series of statements.

The new `match` statement (new as of 3.10) is discussed later in a broader context of pattern matching. Likewise, two other important compound statements, the function `def` statement and the `class`

definition statement, are discussed in their own chapters. Finally, coroutines and other related statements are explained separately in the last chapter of the book. `async`/`await` is Python's (relatively) new construct for high-level asynchronous programming.

11.1. The `if` - `elif` - `else` Statement

The `if` statement is used for conditional execution. It selects at most one of the suites by evaluating the `if`/`elif` expressions one by one, from top to bottom, until one is found to be `True`. Then the corresponding suite is executed (and no other part of the `if` compound statement is executed or evaluated). If all expressions are `False`, then the suite of the `else` clause, if present, is executed.

For example,

```
1 from datetime import datetime
2
3 weekday = datetime.today().isoweekday()  ①
4
5 if weekday == 7:                         ②
6   println("Today is Sunday!")
7 elif weekday == 6:                       ③
8   println("Today is Saturday!")
9 else:                                    ④
10  println("Just another day. :(")
11
12 println("Regardless, let's learn some programming!")
```

① `datetime`'s `isoweekday` method returns integer values, `1` for Monday, `2` for Tuesday, and `7` for Sunday, etc.

② This `if` statement comprises three clauses, `if` (lines 5-6), `elif` (lines 7-8), and `else` (lines 9-10). The `if` expression checks if today is Sunday. If so, it will print out *today is Sunday!* and the control moves to the end of the `if` statement, e.g., line 11.

③ If today is not Sunday, it is then tested against Saturday (6). If this expression evaluates to `True`, the statements in the suite (a single statement, line 8, in this example) are executed, and the execution of the `if` statement terminates.

④ In all other cases, this `if` statement will end up printing *Just another day. :(*. After executing the `if` statement (regardless of which clause has been executed), the program execution moves to the next statement, line 12, in this example.

Note that, unlike in many C-style languages, Python's `if` statement syntax does not require parentheses around the Boolean expressions in the `if` and `elif` lines. Parentheses can still be used for expression grouping purposes.

11.2. The `while` - `else` Statement

The `while` statement is used for repeated execution, similar to the `for` statement. The `while` statement's execution depends on an expression. It tests the expression at every start of iteration and, as long as an expression evaluates to `True`, it executes the first suite. If the expression turns `False` (which may be the first time it is tested), then the suite of the `else` clause, if present, is executed and the loop terminates.

For both `while` and `for` statements,

- A `break` statement in the `while`/`for` suite terminates the loop without executing the `else` clause's suite.
- A `continue` statement in the `while`/`for` suite skips the rest of the compound statement and goes back to testing the expression.

Python 3.0 was first released in 2008, and it now has a yearly release schedule (since 2020?). Let's print that information out:

```
>>> year = 2007
>>> while (year := year + 1) and (year < 3000):  ①
...     if year < 2018:                           ②
...         if year == 2008:
...             print(f"Python 3 was first released in {year
}.")
...         continue                              ③
...     elif year == 2022:                        ④
...         print("Python 3.11 is to be released at the end of
2022.")
...         break                                 ⑤
...
...     version = f"3.{year - 2011}"
...     print(f"Python {version} in {year}.")     ⑥
...
Python 3 was first released in 2008.
Python 3.7 in 2018.
Python 3.8 in 2019.
Python 3.9 in 2020.
Python 3.10 in 2021.
Python 3.11 is to be released at the end of 2022.
```

① This `while` statement is effectively an infinite loop, and its termination is controlled by the `break` statement. The condition `year < 3000` is added as a precaution, e.g., to avoid a runaway iteration in case there is a bug in the code, etc. The assignment expression `year := year + 1`, which always yields `True` in the Boolean context (since `year > 2007 > 0`), is used here, for illustration.

② When `year < 2018`, we just `continue` unless `year == 2008`.

③ The `continue` statement.

④ When `year == 2022`, we terminate the loop.

⑤ This `break` statement terminates the `while` loop in this example.

⑥ Note that this message is printed only when `2018 <= year < 2022`.

11.3. The `for - in - else` Statement

The `for-in-else` statement is used to iterate over the elements of a sequence (such as a string, tuple, or list) or other iterable object, including a dictionary. It has the following syntax:

```
for <target> in <expression_list>:
    <for_suite>
else:
    <else_suite>
```

The `<expression_list>` is evaluated first, and only once. An iterator is created as a result. The suite, `<for_suite>`, is then executed once for each item provided by the iterator. Each item in turn is assigned to the `<target>` using the standard rules for assignments, and then the `<for_suite>` is executed with the value of the current `<target>`. When the items are exhausted, that is, when the iterator raises a `StopIteration` exception, the <else_suite> in the `else` clause, if present, is executed, and the loop terminates.

Note that when the `<expression_list>` returns an empty sequence, the `<target>` variable is never assigned.

For example,

```
>>> features = {                                    ①
...     "3.11": "the exception group",
...     "3.10": "the match statement",
...     "3.9": "the dictionary union operator",
...     "3.8": "the assignment expression",
... }
>>> for k in features:                              ②
...     print(f"""Python {k} includes many new features
...         such as {features[k]}.""")
...
```

```
Python 3.11 includes many new features
        such as the exception group.
Python 3.10 includes many new features
        such as the match statement.
Python 3.9 includes many new features
        such as the dictionary union operator.
Python 3.8 includes many new features
        such as the assignment expression.
```

① A `dict` is an `iterable`.

② A `for` statement, iterating over a dictionary. Note that the order is preserved.

11.3.1. The `range` function

Python's `for in` statement can be used just like C's classic `for` loop, with the help of the builtin `range` function, which generates an integer sequence.

The `range` function can be called in three different ways:

```
range(start, end, step)          ①
range(start, end)                ②
range(end)                       ③
```

① It specifies `start` (inclusive), `end` (exclusive), and `step`, the last of which represents an increment or decrement in the sequence. All arguments are integers, and `step` cannot be zero.

② If the `step` is omitted, its default value is `1`.

③ In this case, `start`/`step` use default values, `0`/`1`, respectively.

For example, let's try adding even integers from 0 to 100, using the `for` statement:

```
>>> _sum = 0
>>> for i in range(0, 100+1, 2):
...     _sum += i
...
>>> print(f"Sum of even numbers from 0 to 100 is {_sum}")
Sum of even numbers from 0 to 100 is 2550
```

There is, in fact, a builtin `sum` function which adds all elements in a given sequence.

```
>>> sum(range(0, 101, 2))
2550
```

11.3.2. The `enumerate` function

The above two use cases, e.g., iterating over items in an arbitrary sequence and iterating over an integer sequence, can be more or less combined with the builtin `enumerate` function.

The `enumerate` function takes an `iterable` as an argument and returns an `iterable` of indexed pairs of the original sequence. That is, given `[item1, item2, item3]`, it returns an iterable of three elements, `(0, item1)`, `(1, item2)`, and `(2, item3)`. The `enumerate` function is not as flexible as `range`, but it takes an optional second argument `start`. Otherwise, the index is 0-based, by default.

Here's an example:

```
>>> name = "python"                          ①
>>> for i, v in enumerate(name, 3):          ②
...     print(i, v, end = ', ')
... else:                                    ③
...     print()
...
```

```
3 p, 4 y, 5 t, 6 h, 7 o, 8 n,          ④
>>>
```

① A string is an immutable sequence type, which is an `iterable`.

② We use the unpacking syntax to assign the index and value, from each iteration of the `enumerate()` function call, to two separate variables, (`i, v`).

③ The `else` clause is always executed at the end of the loop unless the loop is abnormally terminated, e.g., by `break` statement, etc. In this example, we add a new line, at the end of the loop.

④ Otherwise, this line would not have ended with a new line.

11.4. The `try` Statement

As of Python 3.11+, Python's exception framework has been extended to allow programs to raise and handle multiple unrelated exceptions simultaneously. First, a new standard exception type, the `ExceptionGroup`, which represents a group of unrelated exceptions have been introduced. Second, a new syntax `except*` has been added for handling ExceptionGroups.

The classic `try - except` compound statement has the following general syntax:

```
try:
    <suite>
except <except_expression>:              ①
    <suite>
except <except_expression> as <name>:    ②
    <suite>
except:                                  ③
    <suite>
else:                                    ④
    <suite>
```

```
finally:                                    ⑤
    <suite>
```

① An expression that evaluates to an `Exception` or `ExceptionGroup` type. The `ExceptionGroup` types were introduced in Python 3.11.

② A `try` statement can have zero, one, or more `except` clauses.

③ The "catch all" clause.

④ An optional `else` clause.

⑤ An optional `finally` clause. Note that, when there is no `except` clause, `finally` is required.

Since Python 3.11+, the `try` clause can be followed by one or more `except*` clauses instead of `except`.

```
try:
    <suite>
except* <exception_group>:                  ①
    <suite>
except* <exception_group> as <name>:        ②
    <suite>
except* (<ex1>, <ex2>) as <name>:           ③
    <suite>
else:
    <suite>
finally:
    <suite>
```

① An expression that evaluates to an `Exception` or `ExceptionGroup`.

② A `try` statement can have one or more `except*` clauses. Note that, unlike `except`, `except*` cannot be used as a "catch all" clause. That is, the exception/exception group expression is always required.

③ A quick shorthand way to create an "exception group" from the existing exception types.

The `except` clauses specify exception handlers for specific `BaseException` or `BaseExceptionGroup` types. On the other hand, the `except*` clauses specify exception handlers for a set of `BaseException` types contained in a `BaseExceptionGroup` type. `except`/`except*`, `else`, `finally` are all optional, but at least one of `except`/`except*` or `finally` is required. Note that `except` and `except*` cannot be mixed in one `try` statement.

When no exception occurs in the `try` clause, no exception handler is executed.

When an exception occurs in the `try` suite, a search for an exception handler is started. In case of the `except` clauses, this search inspects the `except` clauses, and evaluates the expressions, in order, from top to bottom, until one is found that matches the exception. An expression-less `except` clause, which matches any exception, must be last, if present.

In case of the `except*` clauses, the clause matches the exception if the exception or any of the exceptions in the exception group is the same or or a base type of the given exception. Unlike in the case of `except` clauses, more than one `except*` clauses can match the exception/exception group. Hence, control flow statements like `break` are not allowed in `except*` clauses.

If no `except` clause matches the exception, or if `except*` clauses do not fully handle all exceptions in the exception group, the search for a handler(s) for the exception, or the remaining unhandled exceptions, continues in the surrounding code and on the invocation stack.

When a matching except clause is found, the exception is assigned to the target specified after the `as` keyword in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This

means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

The optional `else` clause is executed if the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue`, or `break` statement was executed. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

If `finally` is present, it specifies a 'cleanup' handler. The `try` clause is executed, including any `except` and `!else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `!finally` clause is executed. If there is a saved exception it is re-raised at the end of the `!finally` clause. If the `!finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `!finally` clause executes a `return`, `break` or `continue` statement, the saved exception is discarded.

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try`...`finally` statement, the `finally` clause is also executed 'on the way out.'

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `!return` statement executed in the `!finally` clause will always be the last one executed.

An exception can be "re-raised". For example,

```
>>> try:
...     sum = 100 + unknown_name
... except:
```

```
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'unknown_name' is not defined
>>>                 ①
```

① The Python REPL handles the error, and it does not crash. It waits for the next user command.

In the context where an active `Exception` is present, the `raise` statement re-raises the current exception/error. This example code is more or less equivalent to the following, which explicitly raises the current exception.

```
>>> try:
...     sum = 100 + unknown_name
... except NameError as ex:
...     raise ex
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<stdin>", line 2, in <module>
NameError: name 'unknown_name' is not defined
```

11.5. The `with` Statement

The `with` compound statement is used to wrap the execution of a group of statements with methods defined by a context manager. For example, it can be used to encapsulate the common `try`...`except`...`finally` usage patterns. It has the following syntax:

```
with Expression as Target:          ①
    Suite
```

① The `as` clause is optional.

The `Expression` must evaluate to a context manager type, which supports `__enter__` and `__exit__` methods. The execution of the `with` statement proceeds as follows:

- The context `Expression` is evaluated to obtain a context manager.
- The context manager's `__enter__` is loaded first.
- The context manager's `__exit__` is loaded next.
- The `__enter__` method is invoked.
- The return value from `__enter__` is assigned to `Target`, if specified.
- The statements in the `Suite` is executed.
- Finally, the `__exit__` method is invoked.

For example,

```
>>> import pathlib
>>> path = pathlib.Path("hello.py")
>>> with (c := path.open("w")) as file:          ①
...     file.write("print('Hello World!')")      ②
...                                              ③
21
>>> type(c)                                      ④
<class '_io.TextIOWrapper'>
>>> dir(c)                                       ⑤
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__enter__', '__eq__',
'__exit__', ...]
```

① The `path.open` function returns a context manager. Its `__enter__` method returns a `file` object, if successful. *For illustration only,* we assign the context manager to a variable `c` using the walrus

operator. (The context managers cannot generally be "reused".)

② All statements included in the `with` statement suite are executed, e.g., until it encounters an error, if any.

③ Before terminating the `with` statement, regardless of the error conditions, the context manager's `__exit__` is guaranteed to be called. Normally, the "cleanup code" is implemented in the `__exit__` method. In the case of the `file` operations, `file.close()` is called if `file` has been (successfully) opened, among other things.

④ An internal type, `_io.TextIOWrapper`, is a class manager.

⑤ This object includes `__enter__` and `__exit__` methods, as expected.

The `with` header can include more than one expression-target item. With more than one expression-target pair, the context managers are processed as if multiple `with` statements were nested, from left to right.

For example,

```
>>> with open("a.txt") as f1, open("b.txt") as f2:
...     print(f1.read())
...     print(f2.read())
...
```

This statement is equivalent to the following:

```
>>> with open("a.txt") as f1:
...     with open("b.txt") as f2:
...         print(f1.read())
...         print(f2.read())
...
```

Chapter 12. Pattern Matching

Python's `match - case` statements, which was first introduced in 3.10, can now be used where complex `if - elif - else` statements may have been required. Pattern matching is essentially a generalization of the comparison expressions, which yield Boolean values.

12.1. The `match` - `case` Statement

The `match` statement has the following general syntax:

```
match SubjectExpression:
    CaseBlock
    ...
    CaseBlock
```

Unlike other compound statements, the `match` statement cannot be written in one physical line. That is, a newline is always needed after the colon. The *SubjectExpression* can be either a named expression or one or more star named expressions (comma separated). When an expression list is used, it is evaluated to a tuple first before the match process starts.

At least one *CaseBlock* is required. That is, for instance, we cannot just use a placeholder like the `pass` statement in the body of the `match` statement. Each *CaseBlock* has the following forms:

```
case Pattern:
    Suite
```

Or

```
case Pattern Guard:
    Suite
```

We go through all different pattern categories in the next section that are currently supported in Python. *Guard* is an optional Boolean condition. If included, the case clause matches only if the pattern matches and the condition is also satisfied.

In the simplest usage, the `match` statement can be used more or less like the traditional `switch` statement in C. For instance,

```
>>> color = Color.BLUE
>>> match color:
...     case Color.RED:
...         print("Color red detected")
...     case Color.BLUE:
...         print("Blue wins!")
...     case _:
...         print("We don't care about other colors :)")
...
Blue wins!
```

The `case` clauses do not require `break` statements, unlike in C's `switch - case` statement. They do not "fall through".

In this example, the first case does not match since `color != Color.RED`. Then the `match` statement matches the next case because `color == Color.BLUE`, and hence the statement(s) in the matched case is executed.

The wildcard expression `_` is used for "catch all". That is, the `case _` clause is more or less equivalent to the *default* case in the C-style switch statement. If there is no matching `case`, then the `match` statement terminates without executing any suites. If the catch-all case `case _` is

specified, then the statements within this default case are executed.

12.2. Patterns

12.2.1. The wildcard pattern

As indicated at the end of the previous section, the special pattern `_` matches any subject expression. This is the simplest pattern, and because it matches any expression, it can only be used in the last case clause in the `match` statement unless a guard is used.

```
>>> match 5:
...     case _:
...         print("I always match!")
...
I always match!
```

12.2.2. Literal patterns

Matching literal patterns is effectively equivalent to comparison operations between the subject expression and the literal patterns. Most of the constant literal expressions are supported, including Booleans, integer and float numbers, and strings. Complex numbers (which are not constant literals) also belong to this category.

For example,

```
>>> arr = (False, 30, 0.5 + 2j, """hello""", None)
... for val in arr:
...     match val:
...         case False:                              ①
...             print(f"{val} matched False")
...         case 30:                                 ②
...             print(f"{val} matched 30")
```

```
...         case 0.5 + 2j:                          ③
...             print(f"{val} matched 0.5 + 2j")
...         case "hello":                           ④
...             print(f"{val} matched 'hello'")
...         case _:                                 ⑤
...             print("Matched none")
...
False matched False
30 matched 30
(0.5+2j) matched 0.5 + 2j
hello matched 'hello'
Matched none
```

① This case matches if `val is False`.

② This case matches if `val == 30`

③ This case matches if `val == 0.5 + 2j`

④ This case matches if `val == "hello"`

⑤ The catch-all pattern. Note that the literal `None` in the tuple `arr` would have matched a case if the case expression was `None`.

12.2.3. Value patterns

Another class of patterns are names/variables that refer to values, including objects' attributes. For example,

```
>>> class X: pass
...
>>> x = X()
>>> x.A, x.B = 5, 50                    ①
>>> match 1 + 4:                        ②
...     case x.A:                       ③
...         print("Matched x.A")
...     case x.B:
...         print("Matched x.B")
...
```

```
Matched x.A
```

① The object `x` now has two (additional) attributes `A` and `B` with values `5` and `50`, respectively.

② The subject expression evaluates to `5`.

③ This case matches since `x.A == 5`.

12.2.4. Group patterns

A pattern enclosed in parentheses is also a pattern, in fact, the same pattern. For instance,

```
>>> match True:
...     case (True):
...         print("True is true no matter what")
...
True is true no matter what
```

12.2.5. Capture patterns

A capture pattern comprises a name. When a match occurs, it binds the subject value to the specified name. For instance,

```
>>> match 1 + 2:
...     case _sum:                      ①
...         print(f"sum is {_sum}.")
...
sum is 3.
```

① The capture pattern always succeeds. The name `_sum` is bound to the value of the subject expression, `3`.

The name cannot be an underscore `_`, and the capture pattern always

matches the subject expression. Capture patterns can be used as part of other patterns such as the OR patterns or sequence patterns.

12.2.6. OR patterns

We can use the vertical bars `|` to include alternative patterns in one case. For example,

```
>>> today = "Saturday"
>>> match today.lower():
...     case "saturday" | "sunday":     ①
...         print("It is a weekend!")
...     case _:
...         print("Just a weekday.")
...
It is a weekend!
```

① This case matches if the subject expression is equal to either one of the alternative patterns. In this example, `today.lower() == "saturday"`, hence this case ends up matching. The patterns in the OR pattern is tested from left to right.

12.2.7. AS patterns

The matched expression of the OR pattern can be given a name using the `as` keyword. For example,

```
>>> import random
>>> match random.randint(0, 3):
...     case 0 | 2 as e:
...         print(f"Got {e}. We are even now.")
...     case 1 | 3 as o:
...         print(f"Got {o}. That is odd!")
...
Got 1. That is odd!
```

12.2.8. Sequence patterns

The sequence patterns match structures and values of a sequence expression, e.g., tuples, lists, and strings.

A sequence pattern uses either parentheses `()` or square brackets `[]`, and it includes one or more subpatterns, separated by commas, each to be matched against an element or elements of the subject expression. When only one subpattern is used in parentheses, a trailing comma is required. Otherwise, it is considered a group pattern.

A subpattern can belong to any one of the pattern categories. In addition, at most one "star pattern" can be used, using an asterisk `*` followed by a capture pattern or a wildcard pattern. The star pattern matches the "remaining elements". If there is no star subpattern, all elements in a pattern must match the corresponding elements in the subject sequence. (Hence, their lengths should be the same to match.)

Here's a recursive implementation of the builtin `len` function, using pattern matching:

```
>>> def seq_length(seq):
...     match seq:
...         case []:                    ①
...             return 0
...         case [_, *y]:               ②
...             return 1 + seq_length(y)
...
```

① This is a fixed length pattern. It matches a sequence with zero elements.

② This is a variable length pattern, with two subpatterns, a wildcard pattern for the first element and the star capture pattern for the rest. This case will match any sequence with at least one element. We ignore the specific value of the first element, in this example, and

use the rest, a subsequence, to call the function recursively. Note that the length of a one-element sequence is 1.

Here's another example.

```
>>> for p in [(0, 0), (0, 2), (3, 3), (2, 4)]:
...     match p:                                ①
...         case (0, 0):                        ②
...             print("I'm the origin!")
...         case (x, 0):                        ③
...             print(f"I'm on the x-axis, x = {x}")
...         case (0, y):                        ④
...             print(f"I'm on the y-axis, y = {y}")
...         case (x, y) if x == y or x == -y:   ⑤
...             print(f"I'm on a diagonal, ({x}, {y})")
...         case (x, y):                        ⑥
...             print(f"I'm just a random point. ;(")
...
I'm the origin!
I'm on the y-axis, y = 2
I'm on a diagonal, (3, 3)
I'm just a random point. ;(
```

① In this example, the subject expression is a tuple.

② This pattern matches the tuple with specified elements, `0` and `0`. Note that the pattern need not use parentheses (just because the subject expression is a tuple in this example). Parentheses and square brackets are interchangeable, except for the one-element sequence pattern, as we alluded above.

③ This pattern comprises a capture subpattern and a literal pattern. This will match any two-element tuple with its second element `0`, and `x` will be bound to the first element of the tuple.

④ Similarly, this fixed-length pattern comprises a literal pattern for the first element and a capture subpattern for the second element. This will match any two-element tuple with the form `(0, y)`, and `y` will

be bound to the second element of the tuple.

⑤ The pattern `(x, y)` will match any two-element sequence since both subpatterns are capture patterns. This particular case includes a guard conditional expression. Besides matching the pattern, the guard expression must evaluate to `True`. Otherwise, the pattern as a whole is considered not a match. Note that we can use the captured names in the guard expression.

⑥ Since we do not use the captured name in this example, this pattern is equivalent to the catch-all wildcard pattern for all valid two-element sequences. The wildcard pattern `_`, if included as the last case, would have caught all expressions which are not a two-element sequence.

12.2.9. Mapping patterns

The mapping types can be used as patterns as well. It works in a similar manner to the sequence pattern, except that the subject expression is a mapping, e.g., a dictionary, and the subpatterns involve keys and values of the mapping elements.

For example,

```
>>> m = {"k1": 1, "k2": 2, "k3": 3}
>>> match m:
...     case {"k1": 1, "k2": 2}:
...         print("Exactly matched the 'k1' and 'k2'
elements.")
...     case {"k1": v1, "k2": v2}:
...         print(f"For key 'k1', the value is {v1}, for 'k2',
the value is {v2}.")
...     case {"k1": 1}:
...         print("Exactly matched the 'k1' element.")
...     case {"k1": v1}:
...         print(f"For key 'k1', the value is {v1}.")
...     case {}:
```

```
...         print("An empty mapping pattern matches all
objects of a mapping type.")
...
Exactly matched the 'k1' and 'k2' elements.
```

In this example, every case is "matchable" to the given subject expression, a dictionary `m`. Hence, the first case ends up matching, and the `match` statement terminates. The mapping pattern also supports a " double star subpattern", for "the rest" of the elements in a mapping expression. For example, `{"k1": v1, **rest}`.

The sequence and mapping patterns can be nested.

12.2.10. Class patterns

The class patterns are another category of patterns that Python's `match` statement supports. A class pattern represents a class and its positional and keyword arguments. The keyword argument uses the equal sign `=` instead of the colon `:`, which is used in the mapping pattern.

Here's an example,

```
>>> class A:
...     def __init__(self):
...         self.x, self.y = 1, 2
...
>>> a = A()
>>> match a:
...     case A(x = 1, y = 2):
...         print("Exactly matched the attributes 'x' and
'y'.")
...     case A(y = 2):
...         print("Matched 'y'. The value of 'x' is ignored.")
...     case A(x = u):
...         print(f"As long as 'a' is of type A, it always
matches. The captured value of 'x' is {u}.")
```

```
...     case A():
...         print("Ditto. But, we ignore both values. In this
case, the parentheses is optional.")
...
Exactly matched the attributes x and y.
```

The patterns of the four cases, in this example, are again all compatible with the subject expression, an object `a`. Hence, it matches *the first case,* and the `match` statement terminates. The positional argument patterns, e.g., `A(1)` or `A(u, v)`, etc., can be used as well when the type's constructor function has positional parameters.

If any positional patterns are present in the `case` clauses, they are converted to keyword patterns first, using the `__match_args__` attribute of the class.

This attribute must be a tuple type of string elements. Furthermore, the length of the tuple must be equal to, or bigger than, the number of the positional arguments. Each positional argument is then converted to the keyword using the `__match_args__` tuple, by mapping the index of the argument to the corresponding tuple element. All keywords mapped this way must be unique.

If this mapping succeeds, the resulting keywords are used for pattern matching using the keyword patterns. For instance, the class `A` in the above example has the following `__match_args__` attribute:

```
>>> A.__match_args__
('x', 'y')
```

Hence, the positional patterns, `A(1)` and `A(u, v)`, for instance, are translated to the keyword patterns, `A(x = 1)` and `A(x = u, y = v)`, respectively.

Chapter 13. Functions

13.1. Function Definition

A function definition is a compound statement that defines a new function.

- A `def` function definition defines a user-defined function.
- When a function definition statement is executed, it creates a function object and it binds the function name in the current local namespace to the function object.
- The function definition does not execute the function body. It gets executed every time the function object is called.
- If the first statement in the function body is a constant string literal expression, then the literal is used as the value of the function object's `__doc__` attribute. Hence, it becomes the function's *docstring*.

For example, the following statement, when executed, creates a function object named `add` which takes two arguments, `p1` and `p2`, and returns one value.

```
>>> def add(p1, p2):                ①
...     "Adds p1 and p2"            ②
...     return p1 + p2              ③
...
>>> add.__doc__                     ④
'Adds p1 and p2'
>>> add(1, 2)                       ⑤
3
```

① The `add` function takes two arguments. Note that there is no way to specify the return values in Python. See below.

② A docstring. Note that an `f-string` expression does not work as a docstring.

③ We can only infer the complete function signature by reading the function implementation.

④ The docstring is stored in the function object's `__doc__` attribute.

⑤ Calling this function with `1` and `2`. It returns `3`.

13.2. Function Parameters

A function can be defined with zero, one, or more parameters. There are three kinds of function parameters in Python.

Positional only parameters

For this type of parameters, when the function is called, the corresponding arguments need to be provided in the exactly the same positions as they are defined in the parameter list (e.g., as in C/C++ and some other C-style languages). In order to define positional only parameters, we use a separator `/` (forward slash). Any parameter preceding this optional separator is positional-only.

Keyword only parameters

For this type of parameters, they do not have the fixed positions in the parameter list, and the corresponding arguments need to be provided using the `parameter=value` syntax (known as the " keyword arguments"). To define *keyword only* parameters, we use a separator `*` (asterisk). (Or, the varargs arguments are also used as a separator for this purpose. See below.) The parameters following this optional separator, if any, are keyword-only.

Positional or keyword parameters

By default, for all other parameters, they have fixed positions and they can be used either with the positional argument syntax (in the corresponding positions) or with the keyword argument syntax.

Here are examples:

```
def fa(p2): pass                    ①
def fb(p1, /, p2): pass             ②
def fc(p2, *, p3): pass             ③
def fd(p1, /, p2, *, p3): pass      ④
```

① `p2` is a "normal" function parameter. It can be used either as a positional argument (`fa(v2)`) or keyword argument (`fa(p2 = v2)`).

② In this example, `p1` is a positional-only parameter.

③ For function `fc`, `p3` can be only used as a keyword arguments.

④ `p1` and `p2`, but not `p3`, can be used as positional arguments. `p2` and `p3`, but not `p1`, can be used as keyword arguments.

13.3. Optional Parameters

The function parameters can have default values, in the form of `parameter = expression`. Those parameters are said to be optional. In a function call, if no argument value is explicitly provided for an optional parameter, its default value is used.

- Any keyword-only parameter can be made optional.
- Only a consecutive list of last one or more positional parameters (positional-only or otherwise), before `*`, if present, can be also made optional.
- The default parameter values are evaluated *only once,* from left to right, when the function definition is executed. The same objects are then used for any subsequent function calls. When the default value objects are mutable, this can lead to an unexpected result since the *default values* can be effectively different across different function calls.

For example,

```
>>> def f(arr = []):                ①
...     arr.append(1)
...     return list(arr)
...
>>> f(), f(), f()                   ②
([1], [1, 1], [1, 1, 1])
```

① In this example, the default value of `arr` is set to `[]`, a list object, which is mutable.

② The expressions in an expression list are evaluated from left to right. Each call `f()` leads to a different result.

Every time we call the same function `f`, relying on the default value of the optional parameter `arr`, it behaves differently. One way around this is to set the *real* default value within the function. For instance,

```
>>> def f(arr = None):              ①
...     if arr == None:             ②
...         arr = []
...     arr.append(1)
...     return list(arr)
...
>>> f(), f(), f()                   ③
([1], [1], [1])
```

① This is idiomatic. When the same default value is required for a mutable parameter across multiple function calls, which is generally the case, we set the default value to `None`.

② Then, in the function body, if the argument value is `None`, then we set the argument with the real default value, `[]` in this example.

③ Each function call `f()` now returns the same result.

13.4. "Varargs" Functions

An optional (at most one) positional varargs parameter can be included in a function definition as the last parameter before the keyword-only parameters, if any. Syntactically, this parameter name is preceded by `*`. When a varargs argument is present, the keyword-only parameter separator `*` is not needed.

When this function is called, all arguments before this varargs argument should be specified and they should use the positional syntax. A tuple including any excess positional arguments (which could be empty) is assigned to the positional varargs argument.

For example,

```
def f(
    a,                          ①
    *args,                      ②
    b):                         ③
    print(a, args, b)
```

① Either positional argument or keyword argument syntax can be used for `a`. But, when the varargs argument is used in a function call, all preceding arguments, including `a` in this example, should use the positional syntax.

② A (positional) varargs argument. Although it is not required, it is conventional to use the name `args` for the varargs argument in Python.

③ `b` is a keyword-only parameter since it is preceded by the varargs parameter.

This function `f` can be called in a number of different ways. For instance,

```
f(1, b = "baby")                ①
f(b = "boy", a = 5)             ②
f(10, 20, 30, b = "girl")       ③
```

① The value of `a` is `1` in the function body. This function call will print out `1 () baby`, including an empty tuple for `*args`.

② This function call will print out `5 () boy`.

③ This will print out `10 (20, 30) girl`.

We can also include (at most one) keyword varargs parameter in the function parameter list. Its name should be preceded by `**`, and it can be used only as the last parameter, either after `*` or `*args` and other keyword-only parameters, if any.

Any excess keyword arguments that are not explicitly specified in a function call is included in the keyword-only varargs argument, as an ordered map.

For example, for a function defined as follows,

```
def f(
    a,                          ①
    /, *,
    b = "rice",                 ②
    **kwargs):                  ③
    print(a, b, kwargs)
```

① `a` is a positional-only parameter.

② `b` is a keyword-only parameter, with a default value `"rice"`, in this example.

③ `kwargs` is a keyword varargs parameter, as indicated by the prefix `**`. It is conventional to use the name `kwargs`, e.g., `**kwargs`, for this kind of keyword-based varargs parameters in Python.

```
f(1)                              ①
f(5, b = "wheat")                 ②
f(10, c = "oats", d = "hops")     ③
```

① The output will be `1 rice {}`, including an empty dict for `**kwargs`.

② The output will be `5 wheat {}`.

③ This function call will print out `10 rice {'c': 'oats', 'd': 'hops'}`.

The more flexible varargs functions tend to include both `*args` and `**kwargs`, often placed together at the end of the parameter list (e.g., with no additional keyword-only parameters).

13.5. Function Call

13.5.1. Calls

A function call is an expression. A call expression (with parentheses) *calls* any `callable` object, not just a function, with zero, one, or more (positional and/or keyword) arguments. Arguments are separated by commas. A trailing comma may optionally be added if at least one argument is included.

13.5.2. Callable

All objects having a `__call__` method are *callable*. The following objects are also callable:

- Built-in and user-defined functions,
- Methods of built-in objects,
- Class objects, and

- Methods of instance objects of a class.

For example,

```
>>> class X: pass                          ①
...
>>> x = X()                                ②
>>> callable(X), callable(x)               ③
(True, False)
>>> x()                                    ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'X' object is not callable
>>> X.__call__ = lambda self: 42           ⑤
>>> callable(x)                            ⑥
True
>>> dir(x)                                 ⑦
['__call__', ...]
>>> x()                                    ⑧
42
```

① A new type `X` is created with a `class` statement.

② `X` is a class object, and hence it is callable. It returns an instance object of type `X`, that is, `x` in this example.

③ The builtin `callable()` function can be used to check if an object is callable. `x` is not callable.

④ Trying to call a non-callable object, with the parenthesis call syntax, will raise a `TypeError` exception.

⑤ One can define the `__call__` method in its type, `X`. Lambda expressions are discussed in a later section, Lambda. Note that this particular Lambda function takes one argument named `self`. This is required since `__call__` is a method.

⑥ Now that its type `X` has a `__call__` method defined, `x` is *callable*.

⑦ As we can easily verify, `x` now has a `__call__` method attribute, inherited from `X`.

⑧ Calling `x()` now calls the `X.__call__` dunder method, which happens to simply return `42` in this example.

13.5.3. Positional vs keyword arguments

As we have seen earlier, function parameters can be defined to be positional-only, positional-or-keyword, or keyword-only, in this order, separated by `/` and `*`, if needed.

When we call a function, or any callable, we can provide positional arguments to positional-only and positional-or-keyword parameters, and keyword arguments to positional-or-keyword and keyword-only parameters.

In addition,

- All positional arguments should come before all keyword arguments, if any.
- Each of the positional arguments should be placed in the position of the corresponding positional parameters.
- Valid arguments should be provided for all non-optional parameters, positional or keyword.

Some examples were given earlier, in the Function Definition section.

13.6. Lambda Expressions

A Lambda *expression* evaluates to an anonymous function object. Here's the general syntax:

```
lambda <parameters>: <expression>
```

The above form is more or less equivalent to a (named) function object defined with:

```
def <name>(<parameters>):
    return <expression>
```

Note that

- Function objects created with lambda expressions can be called "in place", and
- The Lambda expressions cannot contain statements or annotations.

For example,

```
>>> (lambda x: x + 1)(10)              ①
11
>>> a = [1, 2, 3]
>>> list(map(lambda x: x * 2, a))      ②
[2, 4, 6]
```

① A Lambda expression can be assigned to a variable, or it can be called at the point of definition, as shown in this example.

② Lambda expressions are often used as one-time use function-type arguments to higher-order functions (HOFs), for instance. In this example, `map` is a builtin function that accepts a function object as its first argument.

13.7. `map`, `filter`, and `reduce`

Although we do not go through all builtin functions in this mini reference, let's review the builtin `map` and `filter` functions as well as the `reduce` function from the `functools` module. They are generally called the *high-order functions* (HOFs) because they take as arguments,

and/or return, functions.

The map, filter, and reduce functions, possibly with different names, are a few of the most commonly used high-order functions across different programming languages, functional or imperative.

It is a rather common practice to use anonymous functions, i.e., Lambda expressions, for their function arguments for these HOF functions since the function arguments are often used only once.

13.7.1. The map function

The builtin map function takes arguments of function and iterable types, and it applies the function to each element in the given iterable. For example,

```
>>> list(map(lambda x: x**2, [1, 2, 3]))       ①
[1, 4, 9]
```

① The map function returns a map object. (That is, map is a constructor.) We use the list constructor to convert the returned map to a list.

13.7.2. The filter function

The builtin filter function likewise takes a function and an iterable, and it returns a filtered list based on the given function. For example,

```
>>> list(filter(                                ①
...     lambda x: True if x%2==0 else False,    ②
...     [1, 2, 3, 4]))
[2, 4]
```

① The same with filter. It is a constructor function.

② It applies the function to each element in the given iterable, and if its

Boolean value is `True` (e.g., `3`, `'hello'`, etc.), it is included in the result. Otherwise (e.g., `None`, `[]`, etc.), it is filtered out.

13.7.3. The `functools.reduce` function

The `reduce` function of the `functools` module works in a similar manner, but it applies the given function *cumulatively* to all elements in the given iterable. Here's an example:

```
>>> from functools import reduce
>>> reduce(lambda s, a: s + a, [1, 2, 4], 0)   ①
7
```

① The third argument is optional, and if it is provided, it is used as the first item in the "reduction" operation. In this example, the `reduce` function applies the given lambda function *iteratively* for each element in the list `[1, 2, 4]`. That is, it computes, `0 + 1` (which is `1`), `1 + 2` (which is `3`), `3 + 4`, which is `7`, the final result.

13.8. Function Decorators

A decorator is a function, or more generally a `callable`, that takes a function/callable and returns a function/callable of the same type. Syntactically, a decorator is used with a target function definition, as a prefix, and it transforms the target function and returns the transformed function. In other words, the function decorator becomes effectively a part of the target function definition.

Here's the general syntax:

```
@decorator_function
def another_function(args):
    pass
```

This is semantically equivalent to the following:

```
def another_function(args):
    pass

another_function = decorator_function(another_function)
```

The `decorator_function` is a function:

```
def decorator_function(f1):
    # transform f1 to f2,
    # or otherwise create f2 based on f1.
    return f2
```

The key to using decorators, besides the syntactic convenience, is

- The *decorator_function* function provides common functionalities across multiple different functions, and
- These decorated functions, such as `another_function`, will always be used as decorated/transformed in the program.

13.8.1. Built-in decorators

The `@property` decorator

A "property" in Python is an abstract construct that behaves like a data attribute of an object, e.g., with a getter, setter, and deleter, and a docstring, etc.

A property can be created using the `property` constructor function. Alternatively, the `@property` decorator can be used to easily define new properties, or modify existing ones, in a class definition. For example,

```
>>> class X:
...     def __init__(self):
...         self._a = 10
...     @property                          ①
...     def a(self):
...         "I am the 'a' property."       ②
...         return f"Magical {self._a}"
...     @a.setter                          ③
...     def a(self, value):
...         self._a = value
...     @a.deleter                         ④
...     def a(self):
...         print("Urghhh, I am being deleted...")
...         del self._a
...
```

① This creates a property named `a`. The decorated method is used as a "getter" for the property `a`.

② This method's docstring is used as that of the property `a`.

③ This sets the decorated method the setter of `a`. That is, `a` can now be used on the left-hand side of an assignment statement or assignment expression.

④ When the `del` statement is used on the property `a`, the decorated method is called.

Let's try using an object of type `X`:

```
>>> x = X()
>>> help(X.a)                              ①
Help on property:

    I am the 'a' property.
(END)
>>> x.a                                    ②
'Magical 10'
```

```
>>> x.a = 100                                    ③
>>> x.a
'Magical 100'
>>> del x.a                                      ④
Urghhh, I am being deleted...
>>> x.a                                          ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in a
AttributeError: 'X' object has no attribute '_a'. Did you
mean: 'a'?
```

① `X` has a property `a`, and its docstring is used in the builtin `help` function.

② This expression invokes the `a(self)` method decorated with `@property`.

③ This assignment statement uses the `a(self, value)` method decorated with `@a.setter`.

④ This `del` statement invokes the `a(self)` method decorated with `@a.deleter`.

⑤ The property, or a virtual data attribute, `a`, no longer exists at this point.

The `@staticmethod` decorator

A "static method" of a class in Python is similar to those in other OOP languages like C++, Java, and C#. It is essentially an independent function although it is defined in a class definition.

Syntactically, static methods can be called on the class or on an instance, but they do not receive an implicit first argument, e.g., `self`.

The `@staticmethod` decorator converts a function within a class definition to be a static method. For example,

```
>>> class Y:
...     @staticmethod                    ①
...     def f1():
...         print("I am a static method")
...
>>> Y.f1()                              ②
I am a static method
>>> y = Y()
>>> y.f1()                              ③
I am a static method
```

① This decorator makes the decorated method a static method.

② The static method, syntactically, can be called on the class object.

③ Or, on the instance object. Without the `@staticmethod` decorator, a general function defined in a class definition cannot use the method call syntax on an instance object.

The `@classmethod` decorator

A "class method" of a class in Python is a method defined on a class object, not on an instance. Normal methods receive the instance object as an implicit first argument. In contrast, a class method receives the class object as the implicit first argument.

Syntactically, class methods can be called on the class or instance, just like static methods, but they receive the class as its first argument (even when they are called on an instance object). If a class method is called for a derived class, the derived class object is passed as the implied first argument.

The `@classmethod` decorator converts a function to be a class method. For example,

```
>>> class Z:
```

```
...     @classmethod                    ①
...     def f2(cls):
...         print(f"Class method: {cls.__name__}")
...
>>> Z.f2()                              ②
Class method: Z
>>> z = Z()
>>> z.f2()                              ③
Class method: Z
>>> class Z2(Z): pass
...
>>> Z2.f2()                             ④
Class method: Z2
```

① The `@classmethod` decorator makes the decorated method a class method.

② Without the decoration, this would have been a type error.

③ By default, methods defined in a class is an instance method, which takes the instance object as the implicit first argument. For class methods, the implicit first argument is the class object.

④ The implicit `class` object argument is now the subclass `Z2`.

Chapter 14. Classes

14.1. Class Definition

A `class` definition is a compound statement that defines a callable `class` object. When executed, it executes the statements in the class suite, and it creates an object of the type `type` in memory as specified by the `class` definition.

The general syntax is as follows:

```
class MyType(BaseType1, BaseType2):
    Suite
```

The keyword `class` starts the statement, followed by the new class name (e.g, `MyType` in this example) and an optional list of one or more base classes in the parentheses (e.g., `BaseType1`, `BaseType2`, etc.). Python supports "multiple inheritance".

When no base class list is specified, the class inherits from the ultimate base class `object`. In such a case, the parentheses after the class name can be omitted. That is,

```
class Orange:
    pass
```

This definition is equivalent to the following:

```
class Orange(object):
    pass
```

- A `class` definition creates a local namespace, nested within the

global namespace, and the statements in the class's suite are executed in the newly created execution frame.

- A `class` object is then created, inherited from the base classes as (explicitly or implicitly) specified in the base class list, and using the newly created local namespace for the class's `__dict__` attribute.
- The class name, as defined in the `class` definition statement, is then bound to this `class` object in the original local namespace (which can be the global namespace).

14.2. Classes and Instances

Python's `class` is just an object (with some special features). But, in Python programming, it plays the role of classes in other "class-based" object oriented programming (OOP) languages. If nothing else, using classes can help create more modular and more structured Python programs, whose components can be more easily reusable, etc. In many problem domains, thinking in terms of "classes" and "objects" (in the sense of the OOP) can be rather natural and intuitive.

Just like function definitions, a class definition includes a series of statements. Most class definitions primarily include the definitions of variables (e.g., "class variables") and functions (e.g., "static methods", "class methods", and "instance methods").

When a class definition is read/executed by the Python interpreter, these statements are executed. For example,

```
>>> class X(object):
...     if True:
...         print(True)
...     else:
...         print(False)
...
```

```
True                                    ①
```

① This is the output of the `if` statement. Unlike the function definitions, Python executes all statements in the `class` definition scope while creating a `class` object.

A `class` definition is somewhat similar to a "module". These class suite statements are only executed for the first time when the `class` statement is executed (e.g., when the Python interpreter reads the statement). When we *use* the "class object" (e.g., to create an instance object of that class), these statements are not executed. For instance,

```
>>> x = X()                             ①
>>>                                     ②
```

① We "call" the class object to create an instance of that class. The "function" `X()` for class `X` used this way is a *constructor function*.

② The statements in the class definition is not executed.

14.2.1. Class objects

Python's `class` statement creates a class object in memory (just like the `def` statement creates a function object). A `class` object supports the "attribute references" syntax, e.g., through the dot notation, just like any other Python objects.

```
>>> class SoSimple:
...     one_name = "simple"
...
>>> type(SoSimple)                      ①
<class 'type'>
>>> SoSimple.one_name                   ②
'simple'
>>> SoSimple.one_number = 666           ③
>>> SoSimple.one_number
```

```
666
>>> dir(SoSimple)                    ④
['__class__', ... '__weakref__', 'one_name', 'one_number']
```

① The type of a class object is `type`.

② The attribute included in the class definition.

③ We can add any additional attributes to a class object, just like any other objects.

④ Note that a class object comes with a number of predefined attributes, all of which start and end with double underscores (`__`), aka "dunder".

This type of attributes of a class correspond to the static variables (or, static fields) and the static methods in other OOP programming languages.

14.2.2. Class variables

A class object (e.g., defined by the `class` statement) plays two roles, among others. First, as we have seen before, it is the *constructor* for the instance objects of the given class/type. Second, it holds the common variables across all instances of the class. In fact, the class variables are *shared* by all instance objects, as we just mentioned.

```
>>> class Car:
...     brand = "GM"                  ①
...
>>> car1, car2 = Car(), Car()
>>> car1.brand, car2.brand           ②
('GM', 'GM')
>>> Car.brand = "Ford"
>>> car1.brand, car2.brand           ③
('Ford', 'Ford')
```

① `Car.brand` is a class variable. It belongs to the class object.

② You can access it from an instance of the class.

③ But, the variable points to the same object, `Car.brand`.

14.2.3. Constructors

In addition to the attribute references, a `class` object supports the " instantiation operation". A `class` object in Python is a constructor for the objects of the given class/type.

Using the same example from earlier,

```
>>> s = SoSimple()
>>> type(s)
<class '__main__.SoSimple'>
>>> s.one_name
'simple'
>>> s.one_number
666
```

Note that the instance object, `s`, includes the ad-hoc attribute, `one_number`, as well as `one_name`, which is part of the original class definition. One can add any additional attributes to a given instance object as well:

```
>>> s.one_address = "Playa"
>>> s.one_address
'Playa'
>>> dir(s)
['__class__', ... '__weakref__', 'one_address', 'one_name',
'one_number']
```

Note that the instance `s` includes more or less the same predefined

attributes in the original `SoSimple` class as well as other attributes added *later* to this class object `SoSimple`, and those specific to the instance object `s` itself. We can also delete an attribute defined in a class or in an instance using the `del` statement.

```
>>> del type(s).one_number
>>> dir(s)
['__class__', ... '__weakref__', 'one_address', 'one_name']
```

The instance object `s` no longer has the attribute, `one_number`, after executing the `del` statement on the class attribute `SoSimple.one_number`. (Note that, syntactically, `type(s).x` and `SoSimple.x` both refer to the same class attribute `x`.) Likewise, we can delete the instance attribute `one_address` via `del s.one_address`.

14.2.4. The `__init__` function

Note that a class object is not only used as a "template" when creating an instance object of that class, but they essentially share the same attributes.

```
>>> s = SoSimple()
>>> s.one_name
'simple'
>>> SoSimple.one_name = "not simple any more"
>>> SoSimple.one_name
'not simple any more'
>>> s.one_name
'not simple any more'
```

Class instantiation (or, instantiating an instance object of a class) can be customized by overwriting the `__init__` method of the class. This method is automatically called, e.g., by the Python interpreter, after an instance has been created.

```
>>> class SoEasy:
...   def __init__(self):
...     self.one_title = "programmer"
...
>>> s = SoEasy()
>>> s.one_title
'programmer'
>>> dir(s)
['__class__', ... '__weakref__', 'one_title']
```

Note the function signature. The `__init__` method has at least one parameter. The first parameter always refers to the instance object just created, which is always named `self`.

An instance object for the type `SoEasy`, in this example, has an attribute, `one_title`, automatically attached to it. This is because we create this name/attribute and attach it to the `self` object in the `SoEasy.__init__` method.

14.2.5. Instance objects

An *instance object* of a class includes all the attributes defined in the class, and it can include other instance-specific attributes. Attribute references can be used to refer to those attributes of an instance object. There are two kinds of attributes, the data attributes, or fields or variables, and the methods.

As mentioned, an instance object, just like everything else in Python, has attributes, namely, the data attributes and the method attributes. Initially, most of its attributes come from its type, and those added in the `__init__` method, when it is created. But, as with other kinds of custom objects (including functions, classes, etc.), new attributes can be added to the instance objects.

Instance objects have methods that correspond to the functions in a

class definition. All functions that take an instance object as its first argument (e.g., `self`) are, by definition, "methods", and Python allows the object method calling syntax for these functions. For example,

```
>>> class Ship:
...     def fly(self):
...         print("I cannot fly. Only spaceships can fly.")
...
>>> s = Ship()
>>> s.fly()
I cannot fly. Only spaceships can fly.
```

Here, we call the method `fly()` on the instance object, `s`. This is equivalent to the function call:

```
>>> Ship.fly(s)             ①
I cannot fly. Only spaceships can fly.
```

① Note the function argument in this call.

In fact, `<instance>.f(…)` is just a "syntactic sugar" for the more normal function call syntax `<class_name>.f(self, …)`. (Note the difference in the parameter list.) This works as long as the first argument of the function, `self`, is an object of the given type/class.

14.2.6. Instance variables

Although we can add any attributes to an instance object in Python, it is conventionally done in the `__init__` method. Then, all instance objects of the class will have the same set of (not shared) attributes.

```
>>> class Pet:
...     def __init__(self):
...         self.kind = "dog"
```

```
...
>>> pet1, pet2 = Pet(), Pet()
>>> pet1.kind, pet2.kind
('dog', 'dog')
```

In the initializer, the parameter `self` refers to the instance object which has been just created by *calling* the class object. In the case of `pet1`, for instance, `self` and `self.kind` refer to `pet1` and `pet1.kind`, respectively. Likewise, for the `pet2` instance, `self` and `self.kind` refer to `pet2` and `pet2.kind`, respectively.

The attribute `kind` is an instance variable, and it belongs to a specific instance object, and they are not shared across different instances. In addition, the class object does not have that attribute:

```
>>> Pet.kind
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Pet' has no attribute 'kind'
```

Note the syntax to define an instance variable in a class definition. Instance variables are defined on the `self` object. Hence, as a corollary, in a class definition, you can only define instance variables within an instance method.

All user-defined types are mutable. All objects of user-defined types are mutable.

14.2.7. Instance methods

As indicated, the most common statements included in `class` definitions are generally the `def` statements to define functions. A function defined in a class is an attribute of a class, and it is also an attribute of any instance object of that class. And, it can be used with

the method syntax on the object as long as the type of its first argument is the same class.

Unlike the class variables, the class functions have two kinds, besides the instance methods:

- One that is just a function (except for the dotted name syntax), which is called the "static method", and
- The other which is a part of a class object and which can access the class variables. This is called the "class method". The first argument of a class method is the class object itself.

As we have seen earlier, the builtin decorators, `@staticmethod` and `@classmethod`, can be used to specifically declare one or the other kind of class-level methods. Otherwise, by convention, all other methods in a class definition should be instance methods, that is, their first function parameter must be an instance of that class, `self`.

14.3. Object Oriented Programming

A `class` definition statement is used in Python to create a "template" for objects of that class. A `class` defines a custom type, how to create an instance object of that type, and how to access the object, among other things.

A `class` always implicitly "inherits" from the builtin type `object` in Python, either directly or indirectly. This is true for any builtin or custom types. A `class` can directly inherit from another type, which is in turn a subtype of `object`. Python supports inheritance from more than one direct base class.

14.3.1. Data encapsulation

In Python, there is no such things as real "private attributes", data or methods. Data hiding in an instance object is supported via

conventions, as with many other features in Python.

A name prefixed with an *underscore* `_` is treated as "private". That is, by convention, we do not directly access the members of other class objects or instance objects if their names start with one or more underscores. Such attributes are considered an implementation detail, and they are not part of the "public API".

Python *does* have some minimal support for name hiding, however. When a name of a variable or a function/method in a class *starts with at least two underscores and ends with at most one underscore,* then Python modifies the attribute's name. It is called the "name mangling" although it does not truly "mangle" the names (e.g., as in C++). Regardless, using the mangled names should be avoided, even within your own programs. (Note that the "dunder names" are not mangled, or modified, since they end with two underscores.)

14.3.2. Magic methods

The `__init__` method is a special method, as indicated. This particular method is used to provide any initialization code for the newly instantiated instance object. This method is automatically called, by the Python runtime, on the newly created instance object, if it is overwritten in the object's class.

The base type `object` has the following attributes:

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
```

Since every type inherits, directly or indirectly/transitively, from this base class `object`, these attributes are always available to all types, builtin or user-defined. Some of those data attributes might be empty, and some of those method attributes might have empty implementations. Also, some attributes may not show in the `dir()` output since the `__dir__` method of a class/object can be customized.

The `__doc__` attribute stores the docstring of the type, if any. Otherwise, it is an empty string. As with functions, if the first statement in the class definition suite is a constant string literal expression, then it is automatically used as the class's docstring.

(Python's builtin `help` function uses the object's docstring as part of their automatically generated help message.)

The `__str__` method is used when an object is used in a string context. When a `__str__` method is not implemented in a class, it falls back to the `__repr__` method, if present. For lower-level customization, or for displaying debug information, etc., one can overwrite the `__repr__` method. For example,

```
>>> class A:
...     def __str__(self):
...         return "A from __str__()"
...
>>> class B:
...     def __repr__(self):
...         return "B from __repr__()"
...
>>> class C:
...     def __str__(self):
...         return "C from __str__()"
...     def __repr__(self):
...         return "C from __repr__()"
...
>>> a, b, c = A(), B(), C()
```

```
>>> print(f"""\
... a = {a}
... b = {b}
... c = {c}""")
a = A from __str__()                    ①
b = B from __repr__()                   ②
c = C from __str__()                    ③
```

① Since `a`'s type `A` has the `__str__` method implemented, it is called in the string context.

② Since `b`'s type `B` has the `__repr__` method, but not `__str__` method, the `__repr__` method is called.

③ Likewise, `c`'s `__str__` method is called in the string context.

On the other hand,

```
>>> print([a, b, c])                    ①
[<__main__.A object at 0x7093ad81f0>, B from __repr__(), C
from __repr__()]
```

① The internal implementations of lists, and other collection types, use the `__repr__` method of its elements for their string representations. Since the type `A` does not implement this method, it uses the default string representation of `a` (which includes its type and memory location, etc.).

We can, and we should in many cases, override some of these dunder methods to customize the behavior of the custom classes. For example, methods like `__eq__` (for `==`), `__ne__` (for `!=`), `__ge__` (for `>=`), `__gt__` (for `>`), `__le__` (for `<=`), and `__lt__` (for `<`) are often overwritten to customize the equality and comparison-related behaviors of the user-defined types.

For instance,

```
>>> from functools import total_ordering
>>> import random
>>>
>>> @total_ordering                              ①
>>> class Pt(object):
...     def __init__(self, x, y):                ②
...         self.x, self.y = x, y
...     def __eq__(self, other):                 ③
...         return (self.x == other.x) and (self.y == other.y)
...     def __lt__(self, other):                 ④
...         return (self.x + self.y) < (other.x + other.y)
...     def __repr__(self):                      ⑤
...         return f"({self.x}, {self.y})"
...
>>> points = [Pt(-2, 0), Pt(-1, 0), Pt(0, 1), Pt(0, 2)]
>>>
>>> random.shuffle(points)                       ⑥
>>> print("Shuffled:", points)                    ⑦
Shuffled: [(-2, 0), (0, 2), (0, 1), (-1, 0)]
>>>
>>> result = sorted(points)                      ⑧
>>> print("Sorted:  ", result)
Sorted:   [(-2, 0), (-1, 0), (0, 1), (0, 2)]
```

① By using the `@total_ordering` decorator, we need to implement `__eq__` and only one of `__ge__`, `__gt__`, `__le__`, and `__lt__`. The rest of the related dunder methods are automatically generated by this decorator.

② This class represent a two-dimensional point.

③ A typical implementation for the `__eq__` method. It uses member-wise comparison.

④ We give an (arbitrary) ordering to the points in 2-D space.

⑤ We implement a `__repr__` method, but not `__str__` method for this class.

⑥ The `random.shuffle` function shuffles a sequence in place.

⑦ When a list of points is used in the string context, e.g., as an argument to a `print` function call, it calls the `__repr__` method of its element type, `Pt`.

⑧ The builtin `sorted` function relies on the ordering of the items in a sequence.

14.3.3. Inheritance

Another salient feature of the OOP is the type inheritance. To define a class that inherits from a subtype of `object`, we specify the base class (or, the parent class or super class) in the parentheses following the class name. For example,

```
>>> class Animal(object): pass
...
>>> class Pet(Animal):                          ①
...     def __init__(self, name = ""):          ②
...         self.name = name
...     def __repr__(self):                     ③
...         return self.name
...     def bite(self):                         ④
...         print(f"{self.__class__.__name__}s do not bite.")
...
```

① `Animal` is a (direct) base class of `Pet`.

② We override the `__init__` method, which is defined in `object` (since `Animal` does not have its own `__init__` method) to add an instance variable `self.name`.

③ A simple implementation of the `__repr__` method, primarily for debugging purposes.

④ The `Pet` class includes another instance method called `bite`. The builtin `object.__class__` attribute returns the type of the given

object, and `type.__name__` returns the name of the type.

Let's try using this `Pet` class:

```
>>> dog = Pet("puppy")                          ①
>>> dog.name                                    ②
'puppy'
>>> dog.bite()                                  ③
Pets do not bite.
>>> print(dog)                                  ④
puppy
```

① Calling `Pet` returns an instance. (The user-defined types are mutable, and hence `Pet()` creates and returns a *new* object every time it is called.)

② The `dog` has `name` (an instance variable).

③ And, it can do `bite` (an instance method). They work as expected.

④ In the string context, the "correct" method `Pet.__repr__` is called.

As we will see shortly, this `class` definition for `Pet(Animal)` is *exactly* the same as the following using the multiple inheritance syntax:

```
class Pet(Animal, object): ...
```

Note that the class inheritance hierarchy, if you will, goes from left to right. That is, `Pet → Animal → object`. Therefore, `class Pet(object, Animal)` defines a different, and in fact invalid, class because `object` does not inherit from `Animal`.

Here's another class `Python`, which is a subclass of `Pet`:

```
>>> class Python(Pet):                          ①
...     def __init__(self, name, length):       ②
```

```
...         super().__init__(name)              ③
...         self.length = length                ④
...     def __str__(self):                      ⑤
...         return f"Hi, I'm {self.name}. I'm {self.length}
feet long."
...     def bite(self):                         ⑥
...         super().bite()                      ⑦
...         print("But we swallow our prey~~")  ⑧
...
>>> python = Python("Monty Python", 10)        ⑨
>>> python.name                                 ⑩
'Monty Python'
>>> python.bite()                               ⑪
Pythons do not bite.
But we swallow our prey~~
>>> print(python)                               ⑫
Hi, I'm Monty Python. I'm 10 feet long.
```

① This `class` declaration is the same as `Python(Pet, Animal, object)`, for instance, using the multiple inheritance syntax. The `Python` class includes everything that `Pet` has, as well as those which `Animal` and `object` have, e.g., by "inheritance".

② We overwrite `Python`'s initializer.

③ In the `__init__` implementation in the derived class, we (almost always) call the (direct) base class's initializer method. `super()` refers to the closest base class (`e.g., `Pet`) in the class inheritance hierarchy. In this particular example, the `name` instance variable is initialized in `Pet`'s initializer, and hence we need to call it from `Python`'s `__init__`. Note that, in case of `Pet` and `Animal`, their base classes, including `object`, have empty initializers.

④ The instance variable `length` is defined in `Python`, but not in `Pet`.

⑤ We overwrite the `__str__` method in `object`. `Pet` and `Animal` do not implement this method. We can refer to instance variables defined in one of its base classes (e.g., `self.name`) just like they are its own.

⑥ We also overwrite `Pet.bite`.

⑦ In this particular example, we call the super method. Again, `super()` refers to the direct parent base class. If `Pet` did not have this `bite` method, then Python would have searched for it through `Pet`'s base class hierarchy, starting from `Animal`.

⑧ In the string context, `__str__` will be called although `Python` has `__repr__`, inherited from `Pet`. If we had a list of `Pythons` instead and tried to print the list, `Pet.__repr__ would have been used.

⑨ Python constrictor. ☺

⑩ Since `Python` does not have the attribute `name`, it is found in one of its base classes, starting from `Pet`.

⑪ Since `Python` has its own `bite` method, it is called.

⑫ In the string context, `Python.__str__` is called.

14.3.4. Multiple inheritance

When there are multiple direct base classes for a given class, these base classes as well as their base classes, and their bases classes, etc., up to `object`, are "linearized", or ordered, for the purposes of looking up any inherited attributes.

This is called the MRO, or method resolution order, in Python, and it is done through a method called the "C3 Linearization algorithm". It essentially tries to find an order that is consistent with the left-to-right ordering of the base classes of each class involved. This method yields, if successful, a unique ordering among all (direct or indirect) base classes, including `object`, of a given class. This is not always achievable, and in such cases, a compile error is raised. An obvious example is attempting to inherit from two classes whose base classes have incompatible orders.

```
class A: pass
```

```
class B: pass
class C(A, B): pass
class D(B, A): pass
class E(C, D): pass
```

In this example, there is no way to consistently order all base classes of `E` (`A`, `B`, `C`, `D`, and `object`) because `C` requires an `A → B` ordering, whereas `D` requires a `B → A` ordering. This statement will throw a `TypeError` exception.

One thing to note is that, without `E`, these are all valid statements although it *appears* that `C` and `D` are in conflict. The linearization of the base classes are done with respect to each class. Depending on what methods are implemented in `A` and `B`, etc., the behaviors of `C` and `D` might turn out rather different. But, this is still a valid Python program (without `E`).

Here's an example of valid multiple inheritance. This example includes what is often referred to as the "diamond inheritance pattern".

```
class A:
    def w(self): print("A.w")
    def x(self): print("A.x")
    def y(self): print("A.y")
    def z(self): print("A.z")
class B(A):
    def x(self): print("B.x")
class C(B):
    def w(self): print("C.w")
class D(A):
    def x(self): print("D.x")
    def y(self): print("D.y")
class E(C, D):
    pass
```

In this example, class `E`, for instance, has two (non-`object`) direct parent classes, `C` and `D`.

```
E --> C --> B --> A --> object
  --> D --------> A -->
```

`E` inherits indirectly from `A` through two different "paths", `E → C → B → A` and `E → D → A`. Hence it is a diamond inheritance pattern.

Before we continue, let's try running the following program. What would be the output?

```
e = E()
print("e.w()"); e.w(); print("==")
print("e.x()"); e.x(); print("==")
print("e.y()"); e.y(); print("==")
print("z.y()"); e.z(); print("==")
```

If you try running this program, you will realize that

- `e.w()` ends up calling `C.w`,
- `e.x()` ends up calling `B.x`,
- `e.y()` ends up calling `D.y`, and
- `e.z()` ends up calling `A.z`.

One of the "strangest" behavior is that `e.x()` calls the method `B.x`, and not `D.x`, although both have the `x` method defined and `D` is one of the *direct* base classes of `E`, as specified in `E(C, D)`'s class definition. This is because Python's "multiple inheritance" uses the aforementioned linearization among all base classes (whether they are explicitly specified in the class definition or not).

In fact, Python has a builtin method `type.mro` to display this

information. For example,

```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '
__main__.B'>, <class '__main__.D'>, <class '__main__.A'>,
<class 'object'>]
```

The order is,

```
E --> C --> B --> D --> A --> object
```

Therefore, the fact that `e.x()` ends up calling `B.x`, and not `D.x` (which is further up the chain), makes sense.

14.3.5. `super()`

In many programming languages that support OOP, the terms like super, base, and parent have certain related meanings. In Python, the `super` method, which roughly refers to a "base class", has more specific semantics, especially in the context of multiple inheritance.

For example, when `super().x` is referenced in a class, for data or method attribute, Python goes through the linearized/ordered base class hierarchy (MRO) of the given class, from left to right (or, from bottom to top, depending on how you see the inheritance tree), to find the attribute, starting from the class itself. Once it is found during the traversal, that implementation of the attribute is used and all others (e.g., in their base classes upstream) are ignored.

Note that the MRO is defined per class. For instance, using the above example, `B`'s MRO is not a partial segment of `D`'s MRO. It can be completely different.

```
>>> B.mro()
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

The order is `B → A → object`.

14.3.6. Duck typing

Python is fundamentally an object-based programming language, unlike other class-based OOP languages like C++ or Java, or C#. These traditional OOP languages use classes and inheritance to support the runtime *polymorphism*, among other things. A variable declared as one type at compile time can be assigned a different type at run time and it can behave differently.

Python does not need this kind of polymorphism. Python is a dynamically typed programming language. Although Python has adopted a lot of features from the OOP languages over the years, the primary purpose of a `class` (and, its supporting features like inheritance) is to use it as a template, or a prototype, for objects, that is, to create more than one objects that are "structurally equivalent". A lot of dynamic programming languages that support some kind of "classes" like JavaScript, Perl, Lua, etc. all belong to this same category.

To demonstrate the "pseudo-polymorphism" in Python, let's create a few new types:

```
class Frog:
    def __repr__(self):
        return "Frog"
    def jump(self):
        print("Big jump")

class Bullfrog(Frog):
    def __repr__(self):
        return "Bull"
```

```
    def jump(self):
        print("Huge jump")

class Flea:
    def __repr__(self):
        return "Flea"
```

The two types, `Frog` and `Flea`, have little to do with each other, other than the fact that both inherit from `object` just like every type in Python. In particular, the `Frog` type has a method `jump` whereas `Flea` does not. On the other hand, `Bullfrog` is a subclass of `Frog`, and it overwrites the `__repr__` method as well as `jump`.

Now, let's assume that we have a list of objects and we need to call the `jump` method on each of them, say, in a `for` loop. In a traditional OOP language, all objects in the list need to be a type of `Frog` or its subclass. For example, this is generally how it works:

```
>>> frog = Frog()
>>> bull = Bullfrog()
>>>
>>> for jumper in (frog, bull):
...     print(jumper, end=": ")
...     jumper.jump()
...
Frog: Big jump
Bull: Huge jump
```

In an example like this, the `jumper` variable needs to be the type `Frog`. (We are ignoring value vs reference, etc.) Even if we add the same `jump` method to `Flea`, fleas cannot be used in this `for` loop, in the strongly typed OOP languages. On the other hand, in Python, the type does not matter. Any object which has a `jump` method will work in this `for` loop regardless of their specific types.

For instance, let's try this:

```
>>> flea = Flea()
>>> flea.jump = lambda:print("Small jump")
>>>
>>> for jumper in (frog, flea):
...     print(jumper, end=": ")
...     jumper.jump()
...
Frog: Big jump
Flea: Small jump
```

The type of `flea` is `Flea`, and yet we can mix `Flea` and `Frog` in the `for` loop. The type of `jumper` is not important. What's important is the fact that every `jumper` in the iterations has a `jump` method defined.

As mentioned, the type systems like this are generally called the "duck typing" (as in "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck"). Python's support for types, e.g., the `object.__class__` attribute, the `issubclass` and `isinstance` functions, and the classes and class inheritance, etc., provide (a lot of) convenience, but they are not essential. Ultimately, Python uses duck typing at run time.

14.4. Data Classes

A `class` can be used for purely data organization and access, and not for behavior. This kind of class is often called a "record" or "struct type". Python provides a helper module, `dataclasses`, in the standard library for creating a record-like class, called the data class.

One can use the `@dataclasses.dataclass` decorator to create a data class, without having to manually implement a number of essential methods. For example,

```
>>> from dataclasses import dataclass
>>>
>>> @dataclass                                  ①
... class Point:
...       x: int = 0                           ②
...       y: int = 0
...
>>> p1 = Point(1, 2)                           ③
>>> p2, p3 = Point(x=1, y=2), Point(2)
>>> p1, p2, p3                                 ④
(Point(x=1, y=2), Point(x=1, y=2), Point(x=2, y=0))
>>> p1 is p2, p1 == p2                         ⑤
(False, True)
>>> p1 is p3, p1 == p3                         ⑥
(False, False)
```

① By default, the `@dataclass` decorator automatically adds the implementations for `__init__`, `__repr__`, and `__eq__` methods to the decorated class, among other things.

② You specify the data attributes, or "fields", with the type annotations, e.g., `x: int` and `y: int`, in this example. Optional default values can be added, e.g., `= 0`.

③ The constructor with the specified fields is automatically created. Because both fields include the default values, both constructor function parameters are optional as well, in this example.

④ The auto-generated `__repr__` implementation uses the names/values of the class and their fields.

⑤ The auto-generated `__eq__` method ensures the value equality semantics for data classes. That is, two different objects with the same values are considered equal.

⑥ Likewise, two objects with different values cannot be the same object.

In general, the `@dataclass` decorator accepts the following arguments (all optional):

```
@dataclass(
     init=True,              ①
     repr=True,              ②
     eq=True,                ③
     order=False,            ④
     unsafe_hash=False,      ⑤
     frozen=False,           ⑥
     match_args=True,        ⑦
     kw_only=False,          ⑧
     slots=False             ⑨
)
class DataClass:
    pass
```

① The `__init__` method will not be auto-generated if `init=False` or if the method already exists in the decorated class.

② The `__repr__` method will not be auto-generated if `repr=False` or if the method already exists.

③ The `__eq__` method will not be auto-generated if `eq=False` or if the method is already implemented.

④ The `__lt__`, `__le__`, `__gt__`, and `__ge__` methods will be auto-generated if `order=True` and none of these methods exist.

⑤ By default, `@dataclass` will add the `__hash__` method only if it is "safe" to do so. (Consult the official reference for more information.) Setting `unsafe_hash=True` will always auto-generate `__hash__` regardless of whether it is "safe" or not (as long as it is not already implemented).

⑥ If `frozen=True` and if the decorated class does not have `__setattr__` and `__delattr__` methods, the class is made to be "immutable". Attempting to assign to fields of an immutable data

class will raise an exception.

⑦ If `match_args=False`, or if the `__match_args__` attribute is already included in the decorated class, then it will not auto-generate this attribute. The `__match_args__` attribute (a tuple type) is used to support the structural pattern matching for the user-defined classes.

⑧ If `kw_only=True`, all fields will be made keyword-only in the auto-generated `__init__` method.

⑨ If `slots=True`, then the `__slots__` attribute will be generated, and a new class will be returned instead of the original one.

14.5. Enums

An `enum` is a collection of names bound to (related) constant value objects. These objects are called the members of the enum, and they must be unique within the given `enum`. Each member of an `enum` has a name and a value. The members of an `enum` type can be iterated over just like an object of a sequence type. The values of the enum members can be of any type, but immutable types such as `int` or `str` are typically used.

You create a new `enum` class by inheriting from the `enum.Enum` type, or one of its subtypes such as `enum.IntEnum`. Here's an example:

```
>>> from enum import Enum
>>>
>>> class Color(Enum):
...     RED = "red"
...     GREEN = "green"
...     BLUE = "blue"
...
```

The type of an enum member (e.g., `Color.RED`) is not the type of its

value (e.g., `"red"`). Its type is the same `enum` type, e.g., `Color` in this example.

```
>>> isinstance(Color.RED, Color)
True
>>> issubclass(Color, Enum)
True
```

Note that you cannot use an enum type as a base class for other type, including another enum type. If the type inheritance is important, then you will need to use the normal `class` to create new types.

We can iterate over the enum `__members__`, e.g., using the `for - in` statement. (That is, a collection of all `enum` members is an `iterable`.) For instance,

```
>>> for c in Color:
...     print(f"{c}:\t{c.value}")
...
Color.RED:      red
Color.GREEN:    green
Color.BLUE:     blue
```

We can also add a `__str__` method to the `Color` enum class to customize its string representation. For example,

```
>>> class Color(Enum):
...     RED = "red"
...     GREEN = "green"
...     BLUE = "blue"
...     def __str__(self):
...         match self.value:
...             case "red":
...                 return "Red"
```

```
...             case "green":
...                 return "Green"
...             case "blue":
...                 return "Blue"
...             case _:
...                 raise ValueError
...
>>> for c in Color:
...     print(c)
...
Red
Green
Blue
```

14.6. Class Decorators

Classes can also be decorated in the same way functions are decorated.

The evaluation rules for the decorator expressions are the same as those for the function decorators. The result of the decorator expression, a `class`, is then bound back to the target class name.

For example,

```
@decorator_function
class ClassX:
    pass
```

This is semantically equivalent to the following:

```
class ClassX:
    pass

ClassX = decorator_function(ClassX)
```

Chapter 15. Coroutines & Asynchronous Programming

A coroutine is essentially a generalization of a function, or subroutine. A function has a single entry point and, once done executing the function's statements according to its logic, it returns control to the caller once and for all. In contrast, coroutines are first entered, and they can be paused and resumed multiple times, before they are ultimately terminated, or closed.

Most modern programming languages support some kind of coroutines. Python has seen a few different incarnations of coroutines in the past several years. In Python, coroutines were initially designed, and implemented, as a generalization of generators. Now, the newer `async def` based coroutines (and, `tasks` and `futures`) are the ones that should be used moving forward. The generator-based coroutine syntax is deprecated at this point (although the internal implementations are still based on generators).

15.1. Generators

As indicated, Python includes a number of builtin types. Python also has quite a few "duck types" defined in the standard modules such as `collections.abc`, which are primarily used to add some (ad-hoc) structure to the otherwise dynamically typed Python programming language. We have seen some examples throughout this reference, including `iterable` and `callable`, etc.

The `typing` module is another example of an ongoing effort to add more "structures" to the Python's dynamic type system. The type hints can be used to aid static type checking during the Python software development process.

In this and the next couple of sections, we will briefly take a look at `iterators` and `generators` before we move on to the main topic of coroutines for the rest of the chapter.

15.1.1. Iterators

Collection objects like lists, tuples, and dictionaries that can be used with the `for` loop, for instance, are "iterables". A type that implements the dunder method `__iter__` is an `iterable`. Objects of `iterable` types are *iterable,* as we have seen throughout this reference.

When the builtin function `iter()` is called with an `iterable` object (with an `__iter__` method), it returns an object of an `iterator` type. An `iterator` implements `__iter__` and `__next__` methods. Hence, an `iterator` is also an `iterable`. (Internally, an `iterable` object and the object's `iterator` object usually share the same (stream of) data.)

Repeatedly calling the builtin `next()` function with an `iterator` object (which calls the object's `__next__` method) returns the successive items in the object. When no more item is available, e.g., when it reaches the end of the data stream, a `StopIteration` exception is raised.

Here's a simple class that is an `iterator`:

```
class ABC:
    def __init__(self):
        self.pointer = 0
        self.list = ('A', 'B', 'C')

    def __iter__(self):                ①
        return self

    def __next__(self):                ②
        if self.pointer >= len(self.list):
            raise StopIteration
```

```
        val = self.list[self.pointer]
        self.pointer += 1
        return val
```

① It has an `__iter__` method defined. Hence, it is also an `iterable`.

② The type `ABC` implements both `__iter__` and `__next__` methods, and therefore it is an `iterator`.

An object of this class can be used anywhere `iterator`, or `iterable`, is expected. For example,

```
>>> abc = ABC()                    ①
>>> for i in abc:                  ②
...     print(i)
...
A
B
C
```

① `abc` is an `iterable` object with an `__iter__` method, which returns an `iterator` (the same object, in this example).

② The `for` statement uses the iterator's `__next__` method to iterate over the data in `abc`. This is a very general pattern (e.g., as in "design pattern") that many of the modern programming languages use, not just Python. Once the `for` statement catches a `StopIteration`, it exits the loop. (This is hidden in the `for` statement implementation.)

15.1.2. Generator functions

Generator functions (synchronous or asynchronous) and coroutines are rather similar to each other. They can have more than one entry point, they can yield multiple times, and their execution can be suspended and resumed.

- When a *generator function* is called, it returns a *generator object* of

an `iterator` type, which then controls the execution of the generator function.

- When an asynchronous generator function is called, it returns an asynchronous iterator known as an *async generator object*, which then controls the execution of the asynchronous generator function.

15.2. `yield` Expressions

A `yield` expression is used when defining a *generator function* or *asynchronous generator function*.

Python also has a `yield` statement, which is semantically equivalent to a (newer) `yield` expression. There is little difference between the `yield` simple statement and the `yield` expression statement.

A function that includes a `yield` expression/statement in the function body is, *by definition*, a generator function. Likewise, an `async def` function that includes a `yield` expression/statement is an asynchronous generator function.

For example,

```
>>> def a():                    ①
...     yield 1                 ②
...     yield 2                 ③
...
>>> for i in a():               ④
...     print(i)
...
1
2
```

① `a` is (automatically) a generator function since it includes `yield` expressions (or, statements).

② An `yield` expression/statement behaves like a `return` statement. But it does not terminate the function execution. Instead, it temporarily yields the control to the caller, and when it is called again, it resumes the execution from *after the last executed* `yield`. Note that the `a` function need not explicitly provide the implementations of, for example, the `__iter__` method.

③ This is the last `yield` statement in the generator function `a`, and since there is no more statements after this statement, `a`, or its generator object, will throw a `StopIteration` when it is resumed again.

④ *When called,* `a()` returns a *generator object*, which is an `iterator`. Note that the returned generator object implicitly implements `__iter__` and `__next__` methods. In other words, a generator function hides the complexity of having to implement an iterator directly (e.g., as we illustrated in the previous section).

Here's an example of an `async` generator function:

```
>>> async def b():              ①
...     yield 'a'
...     yield 'b'
...
>>> async def c():              ②
...     async for i in b():     ③
...         print(i)
...
>>> import asyncio              ④
>>> asyncio.run(c())            ⑤
a
b
```

① Python creates `b` as an `async` generator function (since it is an

`async def` function that includes `yield` expressions/statements).

② As we discuss later in this chapter, `async` statements, like the `async for` in the first line in the function body, can only be used in `async` functions. We declare `c` as an `async` function.

③ *Calling b()* returns an `async` generator object, which can be used in the `async for` loop.

④ Although `async` and `await` are Python keywords, much of the asynchronous programming support is included in the standard library `asyncio` module.

⑤ An async function can be run using the `asyncio.run` function.

15.3. Generator Expressions

A generator expression is rather similar to a comprehension, not only in syntax but also in spirit. A generator *expression* yields a new generator object, using the comprehension syntax, except that it uses parentheses. For example,

```
>>> double = (                          ①
...     x * 2                           ②
...     for x in range(3)               ③
... )
>>> for x in double:                    ④
...     print(x)
...
0
2
4
```

① A generator expression is enclosed in parentheses, and hence it can be written over multiple lines. The expression is evaluated and assigned to a variable `double`, for illustration. It is more common to define and use the generator expressions where they are needed like

Lambda functions.

② The "next value" expression, similar to the comprehension syntax.

③ The `for` clause.

④ A generator expression returns a generator object (the one bound to `double`, in this example), which is an iterator.

15.3.1. Asynchronous generator expressions

If a generator expression contains `async for` clauses or `await` expressions, then it returns a new asynchronous generator object, which can be asynchronously iterated over.

```
>>> async def a():                 ①
...     yield 0; yield 1; yield 2
...
>>> adouble = (                    ②
...     x * 2                      ③
...     async for x in a()         ④
... )
>>> async def c():                 ⑤
...     async for x in adouble:    ⑥
...         print(x)
...
>>> import asyncio
>>> asyncio.run(c())               ⑦
0
2
4
```

① An `async` generator function.

② An `async` generator expression is also enclosed in parentheses. This is an `async` generator expression because it includes an `async for` clause.

③ The "next value" expression, similar to the comprehension syntax.

④ The `async for` clause. Note that `a()` returns an `async` generator object.

⑤ We define an `async` function here, for illustration, because `async` statements can only be included in `async` functions.

⑥ `adouble` references an `async` generator object, which is an `async` iterator. Again, the result of the `async` generator expression need not have been assigned to a separate variable. We could have just used it here "in place".

⑦ We can run the `c()` function with the help of the `asyncio.run` function.

An iterator type implements `__iter__` and `__next__`. Likewise, an asynchronous iterator type implements `__aiter__` and `__anext__` methods.

When they run out of items, an iterator raises a `StopIteration` exception, whereas an `async` iterator raises a `StopAsyncIteration` exception. These details are all hidden in the implementations of (synchronous or asynchronous) generator functions and generator expressions.

15.4. Coroutine Objects

15.4.1. Awaitable objects

A type that implements a special `__await__` method is an `awaitable`. An `await` expression can be used with an `awaitable`.

`object.__await__(self)`

It returns an `iterator`. It is used to define an `awaitable` type. For example, `asyncio.Future` implements this method, and hence a `Future` object can be used as an operand of an `await` expression.

15.4.2. Coroutine objects

Coroutine objects are `awaitable` objects. The `async def` functions, for example, return coroutine objects.

A coroutine works in a similar way that an iterable works. Calling a coroutine's `__await__` method returns an `iterator`, and the coroutine is executed by iterating over this `iterator` object. When the coroutine has finished executing and returns, the iterator raises a `StopIteration` exception.

In contrast to an iterable, which can be iterated multiple times, however, a coroutine object cannot be `await`ed more than once. Attempting to do so will raise a `RuntimeError` exception.

Furthermore, coroutine objects do not directly use the iterator methods to support iteration. Instead, they include the following methods:

`coroutine.send(value)`

It starts or resumes the execution of a coroutine. If the argument is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If value is not `None`, then this method delegates to the `generator.send` method of the iterator.

`coroutine.throw(type, value, traceback)`

It raises an exception of the specified type, at the suspension point of the coroutine. If the iterator has a `generator.throw` method, it delegates to this method. If the exception is not caught in the coroutine, it propagates back to the caller.

`coroutine.close()`

It causes the coroutine to exit, after running clean-up code, if any. If the iterator has a `generator.close` method, then it optionally delegates to that method first. Then it raises a `GeneratorExit` exception at the suspension point, causing the coroutine to

immediately clean itself up. Finally, it is marked as having finished executing, ending the lifecycle of the coroutine.

The standard library `asyncio` module provides a number of convenience types and methods, such as `Task` and `Future`, for managing coroutines.

15.5. Coroutine Functions

The `async def` statement defines a coroutine function in a similar way that the `def` statement defines a (normal) function. A coroutine defined with `async def` *can*, but is not required to, include `await` expressions and `async` statements such as `async for` and `async with`.

Calling a coroutine function object returns an `awaitable` object, more specifically, a coroutine object. Execution of coroutines can be suspended and resumed, as explained in the previous section.

```
>>> async def a():           ①
...     print("a")
...
>>> async def b():           ②
...     await a()             ③
...
>>> import asyncio
>>> asyncio.run(b())         ④
a
```

① A coroutine function definition without any `async`/`await` expressions/statements.

② Another coroutine function definition which includes an `await` expression.

③ Calling `a()` will return a coroutine object, which can be `await`ed.

④ A coroutine object, which is returned by calling `b()`, can be run using the `asyncio.run` function. Calling a (normal) function executes the statements defined in the function object. On the contrary, calling a coroutine function object merely returns a coroutine object.

15.6. Await Expressions

An `await` expression suspends the execution of coroutine on an *awaitable* object. It can only be used inside a coroutine function such as an `async def` function.

For example,

```
>>> import asyncio                          ①
>>> async def a():                          ②
...     print("a() called")
...     await asyncio.sleep(5.0)             ③
...     print("Leaving a() after sleeping 5 seconds")
...
>>> asyncio.run(a())                        ④
a() called
Leaving a() after sleeping 5 seconds
```

① Much of the async/await runtime support is included in the `asyncio` module.

② The `async def`/coroutine function, when called, returns a coroutine object (or, just a coroutine, for short).

③ The `asyncio.sleep` function returns a coroutine that completes after a given time (in seconds). The `await` expression can be used in a coroutine function with an `awaitable` object.

④ The coroutine can be run using `asyncio.run`.

15.7. Other `async` Statements

15.7.1. The `async for` statement

An "asynchronous iterable" can call asynchronous code in its `iter` implementation, and "asynchronous iterator" can call asynchronous code in its `next` method. The `async for` statement allows convenient iteration over asynchronous iterators.

The `async for in else` compound statement has more or less the same syntax as the `for in else` statement. But, it iterates of an `async iterator` instead of a synchronous `iterator`.

Here's an example:

```
>>> async def words():                  ①
...     yield "hello"
...     yield "parallel"
...     yield "universe"
...
>>> async def greet():                  ②
...     async for word in words():      ③
...         print(word, end=' ')
...     else:                           ④
...         print()
...
>>> import asyncio
>>> asyncio.run(greet())
hello parallel universe                 ⑤
>>>
```

① An `async def` function with `yield` statements (implicitly) returns an async iterator object (more specifically, a coroutine).

② The `async` statements can only be included in a coroutine function.

③ The `async for` clause. The type of `words()` is an async iterator.

④ An optional `else` clause. This clause is executed when the `async for` exits after normally iterating over the `async` iterator.

⑤ The trailing newline is printed from the `else` clause of the `async for` statement in the body of `greet`.

15.7.2. The `async with` statement

An asynchronous context manager is a context manager that is able to suspend execution in its enter and exit methods. The `async with` compound statement works much the same way as the `with` statement. But, instead of using a context manager, it uses an asynchronous context manager, which supports `__aenter__` and `__aexit__` methods (corresponding to `__enter__` and `__exit__` of the context manager, respectively).

Here's the syntax:

```
async with Expression as Target:
    Suite
```

- The `Expression` must evaluate to an asynchronous context manager type.
- The `as` clause is optional as with the `with` statement.
- The *Suite* can include `await` expressions and other `async` statements.

15.8. Producer Consumer Problem

As a final example, and an exercise, here's a simple implementation of the classic (single) producer - (single) consumer problem using Python's coroutines. (For more information, refer to the Wikipedia doc

[https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem].)

This code sample is provided as is without *any* comments or annotations. ☺ The readers are encouraged to go through the code, or try implementing a similar asynchronous program on their own.

The program consists of three files, *producer.py* and *consumer.py* under the *pc* subfolder, and *main.py* in the project folder. You can try different values for the four constants defined in the `producer` and `consumer` modules, and see how they affect the coordination between the producer and the consumer. (Note: This sample program uses some APIs from the `asyncio` module, which are not explicitly covered in this book.)

pc/producer.py

```
import asyncio

_PRODUCE_ITEMS = 3
_PRODUCE_IVAL = 0.5

async def produce(queue: asyncio.Queue,
                  items: int = _PRODUCE_ITEMS,
                  interval: float = _PRODUCE_IVAL):
    for i in range(items):
        print("Producing:", i)
        await asyncio.sleep(interval)
        await queue.put(i)
        print("Produced:", i)

    print("Producer done!")
```

pc/consumer.py

```
import asyncio

_CONSUME_ITEMS = 5
```

```
_CONSUME_IVAL = 1.0

async def consume(queue: asyncio.Queue,
                  items: int = _CONSUME_ITEMS,
                  interval: float = _CONSUME_IVAL):
    for _ in range(items):
        item = await queue.get()
        print("Got item:", item)
        await asyncio.sleep(interval)
        queue.task_done()
        print("Consumed", item)

    print("Consumer done!")
```

main.py

```
import asyncio
from pc.consumer import consume
from pc.producer import produce

async def _main():
    try:
        queue = asyncio.Queue()
        p = asyncio.create_task(produce(queue))
        c = asyncio.create_task(consume(queue))

        await asyncio.gather(p)
        await queue.join()
        c.cancel()
        await asyncio.gather(c)
    except asyncio.CancelledError:
        return
    except RuntimeError as err:
        print(err)

if __name__ == "__main__":
    asyncio.run(_main())
```

A. How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That's quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs.

Although it's a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer*. As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That's the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don't understand and move on, or you can flip through the book to go back and forth. It's up to you. One thing you don't want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

The best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

Other Programming Books by the Author

- The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate
- The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate
- Python for Serious Beginners: A Practical Introduction to Modern Python with Simple Hands-on Projects

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages.* We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

> Already published, or to be published, throughout 2023

- Go Mini Reference
- Modern {cs} Mini Reference
- Python Mini Reference
- Typescript Mini Reference
- Rust Mini Reference
- C++20 Mini Reference
- Modern Java Mini Reference
- Julia Mini Reference
- Javascript Mini Reference
- Haskell Mini Reference
- Scala 3 Mini Reference
- Lua Mini Reference

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating and publishing quality books takes a great deal of time and effort, and we really appreciate the readers' feedback.

Revision 1.1.1, 2023-05-14