PYTHON 3.11

RAW AND UNEDITED

ZERO TO PY

A COMPREHENSIVE GUIDE TO LEARNING THE PYTHON PROGRAMMING LANGUAGE

MICHAEL GREEN, PhD

Zero to Py

A Comprehensive Guide to Learning the Python Programming Language

Michael Green

This book is for sale at http://leanpub.com/zero-to-py

This version was published on 2023-03-11

* * * * *

* * * * *

Table of Contents

Zero to Py: A Comprehensive Guide to Learning the Python Programming Language

Introduction

Python is a powerful and versatile programming language that is widely used across many different industries. From data science and machine learning to web development and automation, Python is a go-to choice for many developers due to its ease of use and its vast developer ecosystem.

This book aims to be a comprehensive guide to learning the Python Programming Language, covering all the essential concepts and topics that a python developer needs to know. This book has been written so to be accessible to readers of all levels, whether you're just starting your journey into programming, or already have some experience with python. Even more advanced users may find some utility from the later chapters.

This book is divided into four parts.

In Part I of this book, we start by introducing you to the basics of Python, building a foundation on the fundamentals as we cover topics such as data types, operators, control flow, functions, and classes.

In Part II of this book, we build upon the foundation established in Part I and dive deeper into more advanced features of the Python programming language. We explore the more advanced features of Python, discussing topics such as generators, the data object model, metaclasses, etc, with the explicit aim to help you write more efficient and elegant code. And finally, we'll look into Python modules and packaging, so we can see how to take your python code and construct libraries which can be shared with the broader python community.

In Part III of this book, we will take a closer look at some of the “batteries included” sections of the Python standard library. These are the modules and packages that are included with every Python installation and provide a wide range of functionality that can be used to solve a variety of problems. We’ll explore packages like functools, itertools, dataclasses, etc.

And finally in Part VI we’ll dig into mechanisms for profiling and debugging python. Furthermore when we identify specific pain points in our implementation, we’ll see how to refactor our projects to use performant C code for a boost in execution speed. We’ll also look into how to use C debuggers to run the python interpreter, so we can debug our C extensions.

Part I: A Whirlwind Tour

Chapter 0: System Setup

If you have a working instance of python installed and are comfortable using it, feel free to skip this chapter.

Python code is executed by a software program called the python interpreter. The python interpreter reads the code from top to bottom, line by line, and performs actions corresponding to the directive of the program. Unlike a compiler, which converts the entire code into machine-readable code before executing it, an interpreter executes the code as it is read.

This book is written against CPython 3.11, and many of the features discussed are version-dependent. If your system comes with a prepackaged version of python that is older than 3.11, you may consider looking into a python version manager. Personally I recommend using `pyenv` to manage your local and global python versions. My typical workflow is to use `pyenv` to switch local versions to a target distribution, and then use that version to create a local virtual environment using `venv`. This however is just one of many opinions, and how you choose to configure your setup is completely up to you.

Installing Python Natively

Installing the python interpreter is the first step to getting started with programming in python. The installation process is different for every operating system. To install the latest version, Python 3.11, you'll want to go to the downloads page of the official website for the Python Software Foundation (PSF), and click the "download" button for the latest release version. This will take you to a page where you can select a version specific for your operating system.

Python Versions

Python is versioned by release number, `<major>.<minor>.<patch>`. The latest version is the one with the greatest major version, followed by the greatest minor version, followed by the greatest patch version. So for example, `3.1.0` is greater that `2.7.18`, and `3.11.2` is greater than `3.10.8` (This is worth noting because in the download page on `python.org`, a patch release for an older version may sometimes be listed above the latest patch release for a newer version).

Windows

The PSF provides several installation options for windows users. The first two options are for 32-bit and 64-bit versions of the python interpreter. Your default choice is likely to be the 64-bit interpreter - the 32 bit interpreter is only necessary for hardware which doesn't support 64-bit, which nowadays is atypical for standard desktop and laptop. The second two options are for installing offline vs. over the web. If you intend to hold onto the original installer for archive, the offline version is what you want. Else, the web installer is what you want.

When you start the installer, a dialogue box will appear. There will be four clickable items in this installer; an "Install Now" button, a "Customize installation" button, a "install launcher for all users" checkbox, and a "Add python to PATH" checkbox. In order to launch python from the terminal, you will need to check the "add python to path" checkbox. With that, click "Install Now" to install Python.

A secondary option to a windows install is to use Windows Subsystem for Linux. WSL is a tool for running a linux kernel along side your windows installation. While the details of this tool are outside the scope of this text, using WSL provides you a linux-like experience without forcing you away from a windows OS. With WSL installed and running in the terminal, you can read the rest of this book as if you were running python on linux. And, given that a vast majority of python projects are web servers, and those web servers run on linux, learning linux alongside python would certainly be beneficial.

macOS

For macOS, a universal installer is provided in the downloads page as a `.pkg` file. Downloading this installer and running through the installation process will install this version of python to `/usr/local/bin/python3`. This install will provide a separate instance of python as compared to the Apple-controlled installation which comes with your mac. This is important because you'll need to take special care that this new installation does not take precedence over your default install, as system utilities, libraries, and programs may rely on the default python install, and overriding the system python preferences may result in instability.

Linux

For linux users, installing the latest version of python can be done by compiling the source code. First, download a tarball from the downloads page and extract into a local directory. Next, install the build dependencies, and run the `./configure` script to configure the workspace. Run `make` to compile the source code and finally use `make altinstall` to install the python version as python<version> (this is so your latest version doesn't conflict with the system installation).

```
root@f26d333a183e:~# apt-get install wget tar make \
> gcc build-essential gdb lcov pkg-config libbz2-dev \
> libffi-dev libgdbm-dev libgdbm-compat-dev liblzma-dev \
> libncurses5-dev libreadline6-dev libsqlite3-dev \
> libssl-dev lzma lzma-dev tk-dev uuid-dev zlib1g-dev
root@f26d333a183e:~# wget
https://www.python.org/ftp/python/3.11.\
2/Python-3.11.2.tar.xz
root@f26d333a183e:~# file="Python-3.11.2.tar.xz"; \
> tar -xvf $file && rm $file
root@f26d333a183e:~# cd Python-3.11.2
root@f26d333a183e:~# ./configure
root@f26d333a183e:~# make
root@f26d333a183e:~# make altinstall
root@f26d333a183e:~# python3.11
Python 3.11.2 (main, Feb 13 2023, 18:44:04) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
informa\
```

```
tion.
>>>
```

What's in a PATH?

Before moving forward, it's worthwhile to talk a bit about the `PATH` variable and what it's used for. When you type a name into the terminal, `python` for example, the computer looks for an executable on your machine that matches this name. But it doesn't look just anywhere; it looks in a specific list of folders, and the `PATH` environment variable is that list.

```
root@2711ea43ad26:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
root@2711ea43ad26:~# which python3.11
/usr/local/bin/python3.11
```

So for example, on this linux machine, the `$PATH` variable is a list of folders separated by a colon (on Windows the separator is a semicolon, but the concept is the same). One of those folders is the `/usr/local/bin` folder, which is where our `python3.11` binary was installed. This means we can simply type `python3.11` into our terminal, and the OS will find this binary and execute it.

If python is not in a folder specified in the `PATH`, or you'd if you'd prefer not to change your `PATH` settings at this time, you can always execute the python executable directly.

```
root@d9687c856f09:/# /usr/local/bin/python
Python 3.11.1 (main, Jan 23 2023, 21:04:06) [GCC 10.2.1 20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more informa\
tion.
>>>
```

Setting up your dev environment

Many developers have many opinions on what the best setup is for writing Python. This debate over the optimal setup is a contentious one, with many developers holding strong opinions on the matter. Some prefer a lightweight text editor like Sublime Text or *Neovim*, while others prefer an IDE like PyCharm or Visual Studio Code. Some developers prefer a minimalist setup with only a terminal and the interpreter itself, while others prefer a more feature-rich environment with additional tools builtin for debugging and testing. Ultimately, the best setup for writing Python will vary depending on the individual developer's needs and preferences. Some developers may find that a certain setup works best for them, while others may find that another setup is more suitable.

With that being said, if you're new to python and writing software, it might be best to keep your setup simple and focus on learning the basics of the language. When it comes down to it, all you really need is an interpreter and a `.py` file. This minimal setup allows you to focus on the core concepts of the programming language without getting bogged down by the plethora of additional tools and features. As you progress and gain more experience, you can explore more advanced tools and setups. But when starting out, keeping things simple will allow you to quickly start writing and running your own code.

Running Python

At this point, we should have a working python installation on our system. This means that somewhere in our file system there is a binary which, if executed, will start the python interpreter.

A program that runs programs

In essence, the python interpreter is a program that runs programs. The standard interpreter, the cpython interpreter, is primarily written in C (though there are other implementations of the interpreter written in different languages, like Jython written in Java and IronPython written in C#). When a Python script is executed, the interpreter reads the code line by line and performs specified actions. This allows developers to write code in a high-level language that is easy to read and understand, while the interpreter takes care of the low-level details of translating the code into machine-readable instructions.

One interpreter, many modes

The python interpreter can be interfaced with in a multitude of ways, and before getting into the main contents of this book, it'll be worth discussing what those ways are, as this book contains many examples which you may want to run locally yourself.

The first mode is called a REPL. REPL is an acronym that stands for "read, evaluate, print, loop". These are the four stages of a cycle which python can use to collect input from a user, execute that input as code, print any particular results, and finally loop back to repeat the cycle.

To enter the python REPL, execute the python binary with no arguments.

```
root@2711ea43ad26:~# python3.11
Python 3.11.2 (main, Feb 14 2023, 05:47:57) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more informa\
tion.
>>>
```

Within the REPL we can write code, submit it, and the python interpreter will execute that code and update its state accordingly. To exit the REPL, simply type `exit()` or press `Ctrl-D`.

Many examples in this textbook are depicted as code which was written in the python REPL. If you see three right-pointing angle brackets >>>, sometimes known as a chevron, this is meant to represent the right pointing angle brackets of the python REPL. Furthermore, any blocks of code which require multiple lines will be continuated with three dots `...`, which is the default configuration of the python REPL.

```
root@2711ea43ad26:~# python3.11
Python 3.11.2 (main, Feb 14 2023, 05:47:57) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
informa\
tion.
```

```
>>> for i in range(2):
...     print(i)
...
0
1
>>>
```

A second mode is where you write python code in a file, and then use the interpreter to execute that file. This is referred to as scripting, where you execute your python script from the terminal using `python ./script.py`, where the only argument is a filepath to your python script, which is a file with a `.py` extension. Many examples in this textbook are depicted as scripts. They all start with the first line as a comment `#` which will depict the relative filepath of the script.

```python
# ./script.py

for i in range(2):
    print(i)
```

```
root@2711ea43ad26:~# python3.11 ./script.py
0
1
```

Occasionally throughout this text you'll encounter examples where a `-c` flag is passed to the python interpreter, followed by a string. This is a special option in the interpreter where instead of passing in a python program as a filename, we're passing it in as a string.

```
root@03987ed73c0e:~# python -c "print('Hello World')"
Hello World
```

There are other ways to interact with the python interpreter and execute python code, and the choice of which one to use will depend on your specific needs and preferences. But for just starting off, these two means of interacting with the interpreter are sufficient for our use cases.

Chapter 1. Fundamental data types

In the Python programming language, the data types are the fundamental building blocks for creating and manipulating data structures. Understanding how to work with different data structures is essential for any programmer, as it allows you to store, process, and manipulate data in a variety of ways. In this chapter, we will cover the basic data types of Python; including integers, floats, strings, booleans, and various container types. We will explore their properties, how to create and manipulate them, and how to perform common operations with them. By the end of this chapter, you will have a solid understanding of the basic data types in Python and will be able to use them effectively in your own programs.

But first, variables

But before we talk about data structures, we should first talk about how we go about referencing a given data structure in Python. This is done by assigning our data structures to variables using the assignment operator, `=`. Variables are references to values, like data structures, and the type of the variable is determined by the type of the value that is assigned to the variable. For example, if you assign an integer value to a variable, that variable will be of the type `int`. If you assign a string value to a variable, that variable will be of the type `str`. You can check the type of a variable by using the built-in `type()` function.

```
>>> x = "Hello!"
>>> x
"Hello!"
>>> type(x)
<class 'str'>
```

Note: this example using `type()` is our first encounter of whats called a "function" in Python. We'll talk more about functions at a later point in this book. For now, just know that to use a function, you "call" it using parentheses, and you pass it arguments. This `type()` function can take one *variable as an argument, and it returns the “type” of the variable it was passed - in this case x is a `str` type, so type(x) returns the `str` type.*

It’s also important to note that, in Python, variables do not have a set data type; they are simply a name that refers to a value. The data type of a variable is determined by the type of the value that is currently assigned to the variable.

Variable names must follow a set of rules in order to be considered valid. These rules include:

- Variable names can only contain letters, numbers, and underscores. They cannot contain spaces or special characters such as `!`, `@`, `#`, `$`, etc.
- Variable names cannot begin with a number. They must begin with a letter or an underscore.
- Variable names are case-sensitive, meaning that `MyVariable` and `myvariable` are considered to be different variables.
- Python has some reserved words that cannot be used as variable names. These include keywords such as `False`, `None`, `True`, `and`, `as`, etc.

```
>>> __import__('keyword').kwlist
['False', 'None', 'True', 'and', 'as', 'assert',
'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

A list of the reserved keywords in python.

Variables are not constant, meaning that their values and types can be changed or reassigned after they are created. Or, once you have created a variable and assigned a value to it, you can later change that value to something else. For example, you might create a variable x and assign it the value of `9`, and later on in your program you can change the value of `x` to `"message"`.

```
>>> x
9
>>> x = "message"
>>> x
"message"
```

The concept of constants does not exist in Python, but it is a common practice to use all uppercase letters for variable names to indicate that the variable is intended to be constant and should not be reassigned.

```
CONSTANT=3.14
```

Primitives and Containers

In Python, data types can be broadly classified into two categories: primitive data types and container data types.

The primitive data types, also referred to as scalar data types, represent a single value and are indivisible. The examples of of these data types include the `bool`, the `int`, the `float`, the `complex`, and to a certain extent, the `str` and `bytes` types. Primitive types are atomic, in that they do not divide into smaller units of more primitive types. With the exception of the string types, since they represent a single value, they cannot be indexed or iterated.

On the other hand, container data types, also known as non-scalar data types, represent multiple values, and are divisible. Some examples of container data types include `list`, `tuple`, `set`, and `dict`. They are used to store collections of values and allow for indexing and iteration (more on that later). These types are built on top of the primitive data types and provide a way to organize and manipulate more primitive data in a structured way.

Python `str` and `bytes` types are somewhat unique in that they have properties of both primitive and container data types. They can be used like primitive data types, as a single, indivisible value; but they also behave like container types in that they are sequences. It can be indexed and iterated, where each letter can be accessed individually. In this sense, these types can be said to be a hybrid that combine the properties of both primitive and container data types.

Mutability vs Immutability

One important concept to understand when working with container data types is the difference between a container being mutable vs immutable. Mutable types can be modified after they are created, while immutable data types cannot be modified. For example, a list is a mutable data type, so you can add, remove or change elements in a list after it is created. On the other hand, a tuple is an immutable data type, so you cannot change its elements after it is created. This difference in behavior can have a significant impact on how you work with data in your programs and it’s important to be aware of it.

```
>>> x = [1, 2, 3]
>>> x[0] = 0
>>> x
[0, 2, 3]

>>> y = (1, 2, 3)
>>> y[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Introduction to Python’s data types

Below is a short introduction to the fundamental data types which are found in Python. We’ll cover each data type more in depth as we progress throughout this book.

Primitive Types

Booleans

The python boolean, represented by the `bool` data type, is a primitive data type that has two possible values: `True` and `False`. It is used to represent logical values and is commonly used to check whether a certain condition is met.

```
>>> x = True
>>> type(x)
<class 'bool'>
```

Integers

The python integer, represented by the `int` data type, is a primitive data type that is used to represent whole numbers. Integers can be positive or negative and are commonly used in mathematical operations such as addition, subtraction, multiplication, and division.

```
>>> x = 5
>>> type(x)
<class 'int'>
```

Integers are by default are coded in base 10. However, Python integers can also be represented in different number systems, such as decimal, hexadecimal, octal, and binary. Hexadecimal representation uses the base 16 number system with digits `0-9` and letters `A-F`. In Python, you can represent a hexadecimal number by prefixing it with `0x`. For example, the decimal number `15` can be represented as `0xF` in hexadecimal. Octal representation uses the base 8 number system with digits `0-7`. In Python, you can represent an octal number by prefixing it with `0o`. For example, the decimal number `8` can be represented as `0o10` in octal. Binary representation uses the base 2 number system with digits `0` and `1`. In Python, you can represent a binary number by prefixing it with `0b`. For example, the decimal number `5` can be represented as `0b101` in binary.

```
>>> 0xF
15
>>> 0o10
8
```

```
>>> 0b101
5
```

Floats

The python floating-point number, represented by the `float` data type, is a primitive data type that is used to represent decimal numbers. Floats are numbers with decimal points, such as `3.14` or `2.718`. There are also a few special float values, such as `float('inf')`, `float('-inf')`, and `float('nan')`, which are representations of infinity, negative infinity, and “Not a Number”, respectively.

```
>>> x = 3.14
>>> type(x)
<class 'float'>
```

Complex

The python complex, represented by the `complex` data type, is a primitive data type that is used to represent complex numbers. Complex numbers are numbers which map to the complex plane using the notation `(x+yj)`, where `x` is the real part and `y` is the imaginary part of the complex number. Complex numbers are often used in mathematical operations such as arithmetic operations, trigonometry, and calculus.

```
>>> x = (1+2j)
>>> type(x)
<class 'complex'>
```

Strings

The python string, represented by the `str` data type, is a primitive data type that is used to represent sequences of characters, such as `"hello"` or `"world"`. Strings are one of the most commonly used data types in Python and are used to store and manipulate text. Strings can be created by enclosing a sequence of characters in `'single'` or `"double"` quotes. They also can be made into multi-line strings using triple quotes (`"""` or `'''`). Strings are immutable, meaning that once they are created, their value cannot be modified.

```
>>> x = "Hello"
>>> type(x)
<class 'str'>
>>> _str = """
... this is a
... multiline string.
... """
>>> print(_str)

this is a
multiline string.

>>>
```

Strings can contain special characters that are represented using backslash `\` as an escape character. These special characters include the newline character `\n`, the tab character `\t`, the backspace character `\b`, the carriage return `\r`, the form feed character `\f`, the single quote `'` the double quote `"` and backslash character itself `\`.

By default, Python interprets these escape sequences in a string, meaning that when an escape character is encountered followed by a special character, python will replace the sequence with the corresponding character. So for example, the string `"hello\nworld"` contains a newline character that will separate `"hello"` and `"world"` on separate lines when printed to the console or written to a file.

Raw strings, represented by the prefix `r` before the string, are used to represent strings that should not have special characters interpreted. For example, a raw string like `r"c:\Windows\System32"` will include the backslashes in the string, whereas a normal string would interpret them as escape characters.

```
>>> "\n"
'\n'
```

```
>>> r"\n"
'\\n'
```

f-strings, also known as “formatted string literals”, allow you to embed expressions inside strings using curly braces `{}`. They are useful for creating strings that include values. To create one, simply prepend the letter `f` to the string you’re trying to create. For example, an f-string like `f"Hello, {name}!"` will include the value of the variable name in the string. These f-strings do not require the provided variables be of type `str`; python will implement an implicit type conversion during formatting.

```
>>> name = "The Primeagen"
>>> slogan = f"My name is {name}!"
>>> slogan
'My name is The Primeagen!'
```

Byte strings, represented by the `bytes` data type, are used to represent strings as a sequence of bytes. They are typically used when working with binary data or when working with encodings that are not Unicode. Byte strings can be created by prefixing a string with a b, for example, `b"Hello"`.

```
>>> type(b"")
<class 'bytes'>
>>>
```

Container Types

Tuples and Lists

The python tuple, represented by the `tuple` data type, is a container data type that is used to store an ordered collection of items. A tuple is immutable; once it is created, its items cannot be modified. Tuples are created by writing a sequence of primitives separated by commas. While technically optional, it’s common to also enclude a set of enclosing parentheses. For example, a tuple of integers can be written as `(1, 2, 3)` and a tuple of strings `("one", "two", "three")`. The primitives contained within a tuple can be of varying types.

```
>>> my_tuple = (1, "two", 3)
>>> type(my_tuple)
<class 'tuple'>
>>> # parentheses are optional (though recommended)
>>> 4, 5, 6
(4, 5, 6)
```

The python list, represented by the list data type, is a container data type that is used to store an ordered collection of items. A list is similar to a tuple, but it is mutable, meaning that its items in the container can be modified after it is created. Lists are created by enclosing a sequence of primitives in square brackets, separated by commas. For example, a list of integers `[1, 2, 3]` or a list of strings `["one", "two", "three"]`. Again, these primitives can be of varying types.

```
>>> my_list = [4, "five", 6]
>>> type(my_list)
<class 'list'>
```

Both tuples and lists in Python can be indexed to access the individual elements within their collection. Indexing is done by using square brackets `[]` and providing the index of the element you want to access. Indexing starts at 0, so the first element in the tuple or list has an index of 0, the second element has an index of 1, and so on.

So for example if you have a tuple `my_tuple = ("Peter", "Theo", "Sarah")`, you can access the first element by using the index `my_tuple[0]`. Similarly, if you have a list `my_list = ["Bob", "Karen", "Steve"]`, you can access the second element by using the index `my_list[1]`.

Both tuples and lists use this convention for getting items from their respective collections. Lists, given their mutability, also use this convention for setting items.

```
>>> my_tuple = ("Peter", "Theo", "Sarah")
>>> my_tuple[0]
"Peter"
>>> my_list = ["Bob", "Karen", "Steve"]
>>> my_list[1]
```

```
"Karen"
>>> my_list[1] = "Sarah"
>>> my_list
["Bob", "Sarah", "Steve"]
```

Negative indexing can also be used to access the elements of the tuple or list in reverse order. For example, given the tuple `my_tuple = ("Peter", "Theo", "Sarah")`, you can access the last element by using the index `my_tuple[-1]`. Similarly, if you have a list `my_list = ["Bob", "Karen", "Steve"]`, you can access the second-to-last element by using the index `my_list[-2]`.

```
>>> my_tuple = ("Peter", "Theo", "Sarah")
>>> my_tuple[-1]
"Sarah"
>>> my_list = ["Bob", "Sarah", "Steve"]
>>> my_list[-3]
"Bob"
```

A third indexing options is to use slicing to access a range of elements from a tuple or list. Slicing is done by creating a slice object using the `:` operator inside the square brackets used for indexing. When providing a slice, it should be known that the interval is half-open, including the first number but excluding the last.

As an example, if you have a list `my_list = ["Jake", "Karen", "Paul", "Tim", "Greg", "Katie"]`, you can access the elements from the second to the fourth by using using `my_list[1:4]`. An optional third value can be provided to indicate a step value.

```
>>> my_list = ["Jake", "Karen", "Paul", "Tim", "Greg", "Katie"]
>>> my_list[1:4]
["Karen", "Paul", "Tim"]
>>> my_list[1:5:2]
["Karen", "Tim"]
```

When you try to access an index that is out of bounds of a list or a tuple, Python raises an exception called `IndexError` (we'll talk more about exceptions later, but for now just consider exceptions as how python indicates that something went wrong). Or in other words, if you try to access an element at an index that does not exist, Python will raise an `IndexError`.

```
>>> my_list = ["Jake", "Karen", "Paul", "Tim", "Greg", "Katie"]
>>> my_list[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Dictionaries

The python dictionary, represented by the `dict` data type, is a container data type that stores key-value pairs. Dictionaries are created by enclosing a sequence of items in curly braces `{}`, with each key-value pair separated by a colon. For example, a dictionary of integers `{1: "one", 2: "two", 3: "three"}` or a dictionary of strings `{"apple": "fruit", "banana": "fruit", "cherry": "fruit"}`. Empty dictionaries can be created simply using `{}`.

To get an item from a dictionary, you can once again use the square bracket notation `[]` and provide a key as the index. For example, if you have a dictionary `my_dict = {"apple": "fruit", "banana": "fruit", "cherry": "fruit"}`, you can get the value associated with the key `"apple"` by using the notation `my_dict["apple"]`, which would return `"fruit"`. Requesting a value from a dictionary using a key which is not indexed causes python to raise a `KeyError` exception.

```
>>> my_dict = {"apple": "fruit", "banana": "fruit", "cherry": "fr\
uit"}
>>> my_dict["apple"]
"fruit"
>>> my_dict["orange"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "orange"
```

Sets

The python set, represented by the set data type, is a container data type that is used to store an unordered collection of unique items. Sets are created by enclosing a sequence of items in curly braces `{}` and separated by commas. For example, a set of integers `{1, 2, 3}` or a set of strings `{"bippity", "boppity", "boop"}`. It is important to note that you can not create an empty set outside of using a constructing function (a constructor), as `{}` by default creates a dictionary. The constructor for creating an empty set is `set()`.

Python also includes a data type called `frozenset` that is similar to the `set` data type, but with one key difference: it is immutable. This means that once a `frozenset` is created, its elements cannot be added, removed, or modified. Frozensets are created by using the `frozenset()` constructor; for example, `frozenset({1, 2, 3})` or `frozenset([4, 5, 6])`.

```
>>> my_set = {"Michael", "Theo", "Michael"}
>>> my_set
{"Theo", "Michael"}
>>> frozenset(my_set)
frozenset({'Theo', 'Michael'})
```

Python Objects

In Python, everything is an object, which means that each element in the language is an instance of a specific class. This includes the built-in data types such as integers, strings, and lists, as well as more complex data structures like functions and modules.

When a variable is assigned a value in Python, that value is actually an object. For example, when you create a variable x and assign the value 5 to it, x is not just a number, it's an instance of the int class, with the value 5.

This is because in Python, the standard implementation of the language is built on C-based infrastructure, where the basic unit of memory that holds the value of a python object is a C structure called a `PyObject*`. This `struct` contains information about the object's type, reference count, and the actual values. `PyObject*` is a pointer to a `PyObject` and it can be used as an opaque handle to the python object, allowing the implementation to abstract the details of these objects and providing python developers a consistent way to interact with any kind of python object, regardless of its type.

In Python, each object has a unique identity that is assigned to it when it is created. This identity is an integer value that is used to distinguish one object from another. The built-in function `id()` can be used to retrieve the identity of an object.

For example, the following code creates two variables, x and y, and assigns the value `257` to both of them. Even though the values of x and y are the same, they are different objects in memory and therefore have different identities. Even though the values are the same, `x` and `y` are given two different id values, because they are different objects.

```
>>> x = 257
>>> y = 257
>>> id(x)
140097007292784
>>> id(y)
140097007292496
```

It is generally the case that variable assignment creates an object that has a unique identity. However, there are some exceptions to this rule.

There are specific values which Python has elected to make as what is called a singleton. Meaning, that there will only ever be a single instance of this object. For example, the integers `-5` through `256` are singletons, because of how common they are in everyday programming. The values `None`, `False`, and `True` are singletons as well.

```
>>> x = 1
>>> y = 1
>>> id(x)
140097008697584
>>> id(y)
140097008697584
```

References to Objects

In Python, variables are references to objects, which means that when you create a variable and assign a value to it, the variable does not store the value directly, it instead stores a reference to the object that contains the value.

For example, consider the following:

```
>>> x = 5
>>> y = x
>>> x = 10
```

In this example, `x` is assigned the value `5`, which means any reference to `x` is fundamentally a reference to the object that is the integer `5`. The variable `y` is then assigned the value of `x`. By the same token then, `y` is fundamentally a reference to the same object which `x` currently references, the `5`. This means that `x` and `y` are now both references to the same object in memory. Now, we can subsequently change the object that `x` references, to say the integer `10`, and this will have no effect on `y`, because `x` and `y` independently referenced the object `5`.

```
x = 5               y = x               x = 5
------              -----               -----
x -- int(5)         x                   x -- int(10)
                     \
                      int(5)
                     /
                    y                   y -- int(5)
```

It’s worth noting that when you work with mutable objects, like lists or dictionaries, you have to be aware that if you assign a variable to another variable, both of the variables fundamentally reference the same object in memory. This means that any modification made to the object from one variable will be visible to the other variable.

```
x = {}              y = x               x["fizz"] = "buzz"
------              -----               -----
x -- {}             x                   x
```

```
                     \                   \
                      {}                  {"fizz": "buzz"}
                     /                   /
                    y                   y
```

If your intention is to have two different objects which can be operated on independently, you must create a copy of the first object, and assign that copy to the second variable. We'll explore how to do this in later chapters.

Chapter 2. Operators

Operators are special symbols in Python that carry out various types of computation. The value that a given operator operates on is called the operand. For example: in the expression `4 + 5 = 9`, `4` and `5` are operands and `+` is the operator that designates addition. Python supports the following types of operators: arithmetic, assignment, comparison, logical, membership, bitwise, and identity. Each of these operators will be covered in depth in the chapter.

Arithmetic

Arithmetic operators are used to perform mathematical operations. The order of operations for these operators follows the same order of operations as in mathematics: Parentheses, Exponentiation, Multiplication and Division, and Addition and Subtraction. That being said it's recommended to use parentheses liberally in order to make the code more readable and avoid confusion about the order of operations.

Here are the most commonly used arithmetic operators in Python:

- `+` - adds two operands
- `-` - subtracts the right operand from the left operand
- `*` - multiplies the operands
- `/` - divides the left operand from the right operand
- `//` - returns a floor division of the left operand from the right operand
- `%` - returns the remainder of a division of the left operand by the right operand
- `**` - raises the left operand to the power of the right operand

```
>>> x = 5
>>> y = 2
>>> x + y
7
>>> x - y
3
```

```
>>> x * y
10
>>> x / y
2.5
>>> x // y
2
>>> x % y
1
>>> x ** y
25
```

Assignment

Assignment operators are used to assign values to variables. The most basic assignment operator in Python is the = operator (of which we've made plenty of use so far).

For example:

```
>>> my_var = "Hello!"
```

This assigns the string `"Hello!"` to the variable `my_var`.

In addition to the basic assignment operator, Python also includes a set of shorthand assignment operators that can be used to perform the same operation, plus a mathematical operation, in a shorter amount of code. These include:

- `+=` - adds the right operand to the left operand and assigns the result to the left operand
- `-=` - subtracts the right operand from the left operand and assigns the result to the left operand
- `*=` - multiplies the left operand by the right operand and assigns the result to the left operand
- `/=` - divides the left operand by the right operand and assigns the result to the left operand
- `%=` - takes the modulus of the left operand by the right operand and assigns the result to the left operand

- `//=` - floor divides the left operand by the right operand and assigns the result to the left operand
- `**=` - raises the left operand to the power of the right operand and assigns the result to the left operand
- `:=` - expresses a value while also assigning it

For example:

```
>>> # () can contain expressions: this assigns 2 to y,
>>> # and expresses the 2 which is added to 3 (more on this later)
>>> x = 3 + (y := 2)
>>> y
2
>>> x
5
>>> x += 5 # is equivalent to x = x + 5
>>> x
10
>>> x *= 2 # is equivalent to x = x * 2
>>> x
20
```

Packing operators

Python allows you to assign the elements of a collection to multiple variables inside a tuple, using a feature called tuple unpacking or tuple assignment. To use tuple unpacking, write a comma-separated tuple with variables on the left side of an assignment statement, and assign to that tuple a sequence with the same number of elements.

```
>>> (a, b) = [1, 2] # note that the parentheses are optional
>>> a
1
>>> b
2
```

Python also defines two unpacking operators which can be used to assign the elements of a sequence to multiple variables in a single expression. These include:

- `*` - iterable unpacking, unpacks a sequence of items
- `**` - dictionary unpacking, unpacks dictionaries into key-value pairs

The `*` operator can be used to unpack a sequence into separate arguments. For example, if you have a tuple `my_tuple = (1, 2)`, you can unpack it within the construction of a separate collection, like a list. The result will be that the items of the tuple are unpacked during the list construction, resulting in a single list of elements.

```
>>> my_tuple = (1, 2)
>>> my_list = [0, *my_tuple, 3, 4, 5]
>>> my_list
[0, 1, 2, 3, 4, 5]
```

These two operations can also be combined to deterministically unpack a collection of values into a list. For example, the previous `my_list` can be unpacked using a tuple assignment into the variables `a`, `b`, and `c`, with variable `b` using the `*` operator as a means to collect any and all elements which aren't explicitly assigned to `a` and `c`.

```
>>> (a, *b, c) = my_list
>>> a
0
>>> b
[1, 2, 3, 4]
>>> c
5
```

The `**` operator can also be used for unpacking operations, though in this case the operator expects operands which are dictionaries. For example, we can merge two smaller dictionaries into one large dictionary by unpacking the two dictionaries during construction of the merged dictionary.

```
>>> dict_one = {1: 2, 3: 4}
>>> dict_two = {5: 6}
>>> {**dict_one, **dict_two}
{1: 2, 3: 4, 5: 6}
```

Comparison

Comparison operators are used to compare two values and return a Boolean value (`True` or `False`) based on the outcome of the comparison. These include:

- `==` - returns `True` if the operands are equal, `False` otherwise
- `!=` - returns `True` if the operands are not equal, `False` otherwise
- `>` - returns `True` if the left operand is greater than the right operand, `False` otherwise
- `<` - returns `True` if the left operand is less than the right operand, `False` otherwise
- `>=` - returns `True` if the left operand is greater than or equal to the right operand, `False` otherwise
- `<=` - returns `True` if the left operand is less than or equal to the right operand, `False` otherwise
- `is` - returns `True` if the left operand is the same instance as the right operand, `False` otherwise
- `is not` - returns `True` if the left operand is not the same instance as the right operand, `False` otherwise

Comparison operators are widely used in decision-making statements. They can be used to compare not only numbers but also strings, characters, lists, and other data types.

```
>>> 1 == 1
True
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 is 1
True
>>> 5 < 2
False
>>> 5 > 2
True
>>> 5 <= 2
False
>>> 2 >= 2
True
>>> x = int(1)
>>> y = int(1)
```

```
>>> x is y # 1 is a singleton in Python
True
>>> x = int(257)
>>> y = int(257)
>>> x is y # 257 is not a singleton
False
>>> x is not y
True
```

Logical

Logical operators are used to combine the results of one or more comparison operations, and can be used to check multiple conditions at the same time. The logical operators include:

- `and` - returns `True` if both operands are truthy, `False` otherwise
- `or` - returns `True` if either of the operands is truthy, `False` otherwise
- `not` - returns `True` if the operand is falsy, `False` if the operand is truthy

```
>>> x = 4
>>> y = 2
>>> z = 8
>>> (x > 2 or y > 5) and z > 6
True
```

This example evaluates to `True` *because a)* `x > 2` *is* `True`*, b)* `y < 5` *is* `False`*, thus* `(True or False)` *is* `True`*, and finally* `True and True` *is* `True`*.*

What is Truthy?

Objects which evaluate to `True` in a boolean context are considered "truthy". In conditional statements, a value does not need to be explicitly `True` or `False`: being "truthy" or "falsy" is sufficient. Most of the time objects are considered "truthy", so it's easier to list the objects which are inherently "falsy":

- `False`
- `None`

- `0`
- `0.0`
- Empty collections (i.e. `[]`, `()`, `{}`, `set()`, etc.)
- Objects which define a `__bool__()` method that returns falsy
- Objects which define a `__len__()` method that returns falsy

Short-Circuits

It's important to note that the `and` and `or` operators are short-circuit operators, which means that they exit their respective operations eagerly. In the case of `and` operator, it only evaluates the second operand if the first operand is `True`, and in the case of `or` operator, it only evaluates the second operand if the first operand is `False`.

Logical Assignments

Logical operators can also be used in assignment expressions. The short-circuit nature of Python operators determines the assignment.

```
>>> x = "" or 3
>>> x
3
```

This example assigns the variable x the value 3 because the empty string is falsy.

```
>>> x = 0 and 6
>>> x
0
```

This example assigns the variable `x` the value `0` because zero is falsy, and since `and` requires the first operand to be truthy, the resulting assignment is `0`.

```
>>> x = 3 and 6
>>> x
6
```

This example assigns the variable `x` the value `6` because `3` is truthy, and since `and` requires the first operand to be truthy, the resulting assignment falls to the second operand.

In instances where we want to explicitly assign a given variable a boolean representation of the expression, we would need some way to evaluate the logical operation before the assignment operation. One way to do this is using an explicit type conversion, by passing the value to the `bool()` constructor:

```
>>> x = bool(3 and 6)
>>> x
True
```

However, we can also do this using only logical operators, and without a function call.

```
>>> x = not not (3 and 6)
>>> x
True
```

In this example the parentheses evaluate `3 and 6` to `6` using the convention above. Next, `not 6` evaluates `False` because `6` is truthy, and finally `not False` evaluates `True`.

Membership

Membership operators are used to test whether a value is a member of a sequence (such as a string, list, or tuple) or a collection (such as a dictionary or set). The membership operators include:

- `in` - returns `True` if the value is found in the sequence, `False` otherwise
- `not in` - returns `True` if the value is not found in the sequence, `False` otherwise

```
>>> collection = {1,2,3}
>>> 5 in collection
False
```

```
>>> 3 in collection
True
>>> "t" in "String"
True
>>> 1 in {1: 3} # Membership of dictionaries in python is
depende\
nt on the key
True
```

Bitwise

Bitwise operators are used to perform operations on the binary representations of integers. These operators work on the individual bits of the integers, rather than on the integers as a whole. The bitwise operators include:

- `&` - performs a bitwise AND operation on the operands
- `|` - performs a bitwise OR operation on the operands
- `^` - performs a bitwise XOR operation on the operands
- `~` - performs a bitwise NOT operation on the operand
- `<<` - shifts the bits of the operand to the left by the number of places specified by the second operand
- `>>` - shifts the bits of the operand to the right by the number of places specified by the second operand

```
>>> x = 0b101
>>> y = 0b011
>>> x & y
1 # 0b001
>>> x | y
7 # 0b111
>>> x ^ y
6 # 0b110
>>> ~x
-6 # -0b110
>>> x << 1
10 # 0b1010
>>> x >> 1
2 # 0b010
```

It's important to note that the bitwise NOT operator (~) inverts the bits of the operand and returns the two's complement of the operand.

It's also worth noting that Python also has support for the analogous bitwise assignment operators, such as `&=`, `|=`, `^=`, <<=, and >>=, which allow you to perform a bitwise operation and assignment in a single statement.

Identity

Finally, The identity operator is used to compare the memory addresses of two objects to determine if they are the same object or not. The identity operators include:

- `is` - returns True if the operands are the same object, False otherwise
- `is not` - returns True if the operands are not the same object, False otherwise

You can use this identity operator to check against singletons, such as `None`, `False`, `True`, etc.

```
# ./script.py

x = None
if x is None:
    print("x is None!")
```

It's important to note that two objects can have the same value but different memory addresses; in this case the == operator can be used to check if they have the same value and the `is` operator can be used to check if they have the same memory address.

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x

>>> z is x
True
>>> x is y
False
```

```
>>> x == y
True
```

This example shows that even though `x` *and* `y` *contain the same elements, they are two different lists, so* `x is not y` *returns* `True`*. The variables* `x` *and* `z` *however point to the same list, so* `x is z` *returns* `True`*.*

Chapter 3. Lexical Structure

Up until now, we've been working with Python in the context of single-line statements. However, in order to proceed to writing full python scripts and complete programs, we need to take a moment to discuss the lexical structure which goes into producing valid Python code.

Lexical Structure refers to the smallest units of code that the language is made up of, and the rules for combining these units into larger structures. In other words, it is the set of rules that define the syntax and grammar of the language, and they determine how programs written by developers will be interpreted.

Line Structure

The python interpreter parses a python program as a sequence of "logical" lines. To the python interpreter, each logical line is represented by a `NEWLINE` token. Statements cannot extend beyond this token, except in cases where the syntax allows for it, such as in compound statements. A logical line is created by combining one or more "physical" lines, i.e. the lines in a file you create by hitting the "enter" key.

A physical line in Python is a sequence of characters that is ended by an end-of-line sequence, corresponding to hitting the "enter" key. This end-of-line sequence either takes the form of the Unix line termination sequence using ASCII LF (linefeed, `b'\n'`), the Windows sequence using ASCII CR LF (carriage return followed by linefeed, `b'\r\n'`), or the Macintosh sequence using ASCII CR (carriage return, `b'\r'`), depending on the platform. The end of a file is also considered an implicit terminator for the final physical line.

```python
# ./script.py

for i in range(2):
    print(i)
```

```
root@a59580cc2448:~# python -c "
> print(open('./script.py', 'br').read())
> "
b'# ./script.py\n\nfor i in range(2):\n    print(i)\n'
root@a59580cc2448:~#
```

In this example, we see how the python script 'script.py' looks in our editor. We can contrast that to how the computer interpret the file, as a sequence of characters separated by newline characters (as this is on a linux machine, the newline character is `\n`).

Comments

A comment in Python begins with the hash character `#` and continues until the end of the physical line. A comment indicates the end of the logical line unless the rules for implicit line joining are applied. Comments are ignored by the syntax and are not executed by the interpreter.

```
>>> # this is a comment, and the next
>>> # line is executable code.
>>> this = True
```

Explicit and Implicit Line Joining

Two or more physical lines in Python can be joined into a single logical line using backslash characters `\`. This is done by placing a backslash at the end of a physical line, which will be followed by the next physical line. The backslash and the end-of-line character are removed, resulting in a single logical line.

```
>>> # you can join multiple physical lines to make a single logic\
al line
>>> x = 1 + 2 + \
... 3 + 4
...
>>> x
10
```

In this example, the logical line "x = 1 + 2 + 3 + 4" is created by combining two physical lines, separated by the backslash.

It should be noted that a line that ends with a backslash cannot contain a comment.

```
>>> x = 1 + 2 \ # comment
  File "<stdin>", line 1
    x = 1 + 2 \ # comment
                 ^
SyntaxError: unexpected character after line continuation charact\
er
```

Expressions in Python that are enclosed in parentheses, square brackets, or curly braces can be spread across multiple physical lines without the need to use backslashes. For example:

```
>>> # These are a valid ways of splitting an expression over
mult\
iple
>>> # physical lines without using backslashes
>>> my_first_variable = (
...     1 + 2 + 3 + 4
... )
...
>>> my_second_variable = [
...     1, 2, 3, 4
... ]
...
>>> my_final_variable = {
...     'a': 1, 'b': 2, 'c': 3, 'd': 4
... }
...
```

In this example, all the expressions are spread across multiple physical lines, but they are still considered as a single logical line, because they are enclosed in parentheses, square brackets, or curly braces.

Indentation

One important aspect of the lexical structure of Python is the use of indentation to define scope. In Python, indentation is used to indicate the scope of a block of code, rather than using curly braces like other programming languages. This means that any code that has the same indentation level, say under an `if` statement or a `for` loop, is considered to be within the scope of that particular block. This feature makes the code more readable and easy to understand, as it is clear at a glance which lines of code are part of a specific block. However, it also means that we need to be very careful about how we indent our code, as improper indentation can lead to syntax errors.

For example, in the following code snippet, the lines of code that are indented under the `if` statement are considered to be within the scope of the `if` statement and will only be executed if the condition `x > 0` is `True`.

```
>>> x = 5
>>> if x > 0:
...     print("x is positive")
...     x = x - 1
... print("x is now", x)
...
x is positive
x is now 4
```

Here, the first print statement `"x is positive"` is only executed if `x` is greater than `0`, and the second print statement `"x is now"` is executed no matter what.

The indentation of the python code is crucial; if it is not correct, it will raise an error, or worse, will do something you did not expect, leading to bugs. Finally, the indentation must be consistent throughout the program, usually 4 spaces or a tab.

Chapter 4. Control Flow

Control flow is a fundamental concept in programming that allows a developer to control the order in which code is executed. In Python, control flow is achieved through the use of conditional statements (such as `if`/`else`) and looping constructs (such as `for` and `while` loops). By using these constructs, developers can create logical conditions for code execution and repeat blocks of code as necessary. Understanding control flow is essential for creating Python programs. In this chapter, we will cover the basics of control flow, focusing on conditional statements, loops, and other related concepts.

`if`, `elif`, and `else`

The `if` statement is a control flow construct that allows a developer to specify that a block of code should only be executed if a certain condition is met. The basic syntax of an `if` statement is as follows:

```
if condition:
    # code to be executed if condition is Truthy
```

In addition to the `if` statement, Python also provides the `elif` (short for "else if") and `else` statements, which can be used to specify additional blocks of code to be executed if the initial condition is not met.

```
if condition:
    # code to be executed if `condition` is truthy
elif other_condition:
    # code to be executed if `condition` is falsy
    # and `other_condition` is truthy
else:
    # code to be executed if both conditions are falsy
```

Question 1:
What values would trigger the `if` statement to execute?

1. `condition = ["", None]` and `other_condition = True`
2. `condition = ""` and `other_condition = True`
3. `condition = ""` and `other_condition = None`

Answer 1: *1. is the correct answer, because `condition` is truthy, even though all of its contents are falsy.*

Question 2:
What values would trigger the `elif` statement to execute?

1. `condition = False` and `other_condition = 0`
2. `condition = "None"` and `other_condition = False`
3. `condition = {}` and `other_condition = [0]`

Answer 2: *3. is the correct answer, because `condition` is falsy, and `other_condition` is truthy*

`while`

The `while` statement is a control flow construct that allows a developer to execute a block of code repeatedly, as long as a certain condition is considered truthy. The basic syntax of a `while` loop is as follows:

```
while condition:
    # code to be executed while `condition` is truthy
```

`condition` is an expression that is either truthy or falsy. If the condition is truthy, the code indented under the `while` statement will be executed. At the end of the block, the program loops back to the `while` statement and the condition will be re-evaluated. If the condition is then falsy, the loop will exit and the program will continue beyond the `while` block.

```
>>> condition = 10
>>> while condition:
...     condition -= 1 # condition = condition - 1
...
>>> condition
0 # 0 is falsy
```

It’s important to note that it’s the programmer’s responsibility to ensure that the condition in the `while` loop will eventually evaluate to `False`, otherwise the loop will run indefinitely. This is known as an infinite loop, and it can cause your program to crash or hang.

`break`

The `break` statement is a control flow construct that can be used to exit loops, including the `while` loop. When the interpreter encounters a `break` statement within a loop, it immediately exits the loop and continues execution at the next line of code following the loop.

```
>>> condition = 0
>>> while condition > 10:
...     if condition == 5:
...         break
...     condition -= 1
...
>>> condition
5
```

`continue`

In instances where you would like to skip an iteration of a loop and move onto the next, you can use a `continue` statement. When the interpreter encounters a `continue` statement within a loop, it immediately skips the rest of the code in the current iteration of the block, and continues on to the next iteration of the loop.

```
>>> i = 0
>>> skipped = {2, 4}
>>> while i <= 5:
...     i += 1
...     if number in skipped:
...         continue
...     print(number)
...
1
3
5
```

`for`

The `for` statement is a control flow construct that allows a developer to iterate over a collection of items, such as a list or a string. The basic syntax of a `for` loop is as follows:

```python
for item in iterable:
    # code to be executed for each item in the sequence of items
```

Here, the variable `item` will take on the value of each item in the sequence in turn, allowing you to operate on each item of the collection one at a time.

What is an Iterable?

An iterable is an object which has the potential to sequentially yields each item of a collection in a predictable order. These items can be any data type, such as numbers, strings, or even other objects. The items in an iterable are accessed in a specific order, usually by their index or in the order in which they were added to a collection.

Examples of built-in iterable objects in Python include lists, tuples, and strings. For example, a list is an ordered collection of items, where each item can be any data type and is accessed by its index:

```python
>>> numbers = [1, 2, 3, 4, 5]
>>> for number in numbers:
...     print(number)
...
1
2
3
4
5
```

In this case, `numbers` is an iterable, where each `item` is a number, and the items can be accessed in order by their index.

A string is also a collection of items, where each item is a letter, and the items can also be accessed by their index.

```
>>> word = "hello"
>>> for letter in word:
...     print(letter)
...
h
e
l
l
o
```

In this case, word is an iterable where each item is a character, and the items can be accessed by their position in the string.

`for/else` and `break`

The `for/else` construct is a special combination of the `for` loop and the `else` statement. It allows you to specify a block of code (the `else` block) that will be executed after the for loop completes, but only if the loop completes without a `break`. This means that if the loop is exited early using a `break` statement, the `else` block will not be executed.

```
>>> numbers = [1, 2, 3, 4, 5]
... for number in numbers:
...     if number == 4:
...         print("Found 4, exiting loop.")
...         break
...     print(number)
... else:
...     print("Loop completed normally.")
...
1
2
3
Found 4, exiting loop.
```

In this example, the `for` loop iterates over the numbers in the list. When it encounters the number `4`, it exits the loop using the `break` statement. Since the loop was exited early, the `else` block is not executed, and the message "Loop completed normally" is not printed.

If the loop completes normally, the `else` block will be executed:

```
>>> numbers = [1, 2, 3, 4, 5]
... for number in numbers:
...     if number == 6:
...         print("Found 4, exiting loop.")
...         break
...     print(number)
... else:
...     print("Loop completed normally.")
...
1
2
3
4
5
Loop completed normally
```

The `for/else` construct can be useful in certain situations where you want to take different actions depending on whether the loop completed normally or was exited early. However, it's not a very common construct, and in most cases, it can be replaced by a simple `if` statement after the loop.

Exception Handling

Exceptions are a way of handling errors and unexpected situations in a program. When an exception occurs, it causes the normal flow of the program to be interrupted, and the control flow is transferred to an exception track.

This is done using the `raise` keyword, followed by an exception object or class. The exception object can be any class that inherits from the built-in `Exception` type. Raising an exception interrupts the normal flow of execution and transfers the control flow to the exception track. Exception classes can generally be instantiated with a single `str` type argument that acts as an exception message.

```
>>> x = None
>>> if x is None:
...     raise ValueError("value 'x' should not be None")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: value 'x' should not be None
```

The exception track can be caught using an exception handler. Exception handlers are blocks of code which allow you to gracefully handle an error or unexpected situation, rather than abruptly terminating the program.

Exception handlers are defined using a `try/except` statement. The `try` block contains the code that may raise an exception, while the `except` block contains code that handles the exception. If an exception is raised in the `try` block, the interpreter jumps to the `except` block, and searches for an exception handler that matches the type of exception that was raised.

```
try:
    # code that may raise an exception
except Exception:
    # code to be executed if the exception occurred
```

Each `try` block can have one or more `except` blocks, so as to handle multiple different types of errors. Each except block can have multiple exceptions to match against, by using a tuple of exceptions. The exception in an `except` block can be assigned to a variable using the `as` keyword.

When an `Exception` is raised in the `try` block, the interpreter checks each `except` block in the order which they were defined to see if a particular exception handler matches the type of exception that was raised. If a match is found, the code in the corresponding `except` block is executed, and the exception is considered handled. If no `except` block matches the exception type, the exception continues to bubble up the call stack until it is either handled elsewhere or the program terminates.

```
>>> try:
...     raise ArithmeticError
... except (KeyError, ValueError):
```

```
...     print("handle Key and Value errors")
... except ArithmeticError as err:
...     print(type(err))
...
<class 'ArithmeticError'>
```

In this example, an `ArithmeticError` is raised in the `try` block. When this error is raised, it is checked against each `except` statement in the exception handler. The first exception handler is a `(KeyError, ValueError)` tuple, and neither the KeyError or the ValueError match the type of the exception raised, so this except block is passed over. The second exception handler is an `ArithmeticError`, which matches the type of the exception, as shown by the print statement, so this `except` block is executed and the error is considered handled.

Most exceptions are derived from the `Exception` class. As such, a general `except Exception` will match most raised exceptions. There are some instances where this may not be the desired effect; it is generally better to be precise with your exception handling, so that unexpected errors aren't caught accidentally. In a similar vein, it's worth noting that not all exceptions need to be handled, and sometimes it may be better to let the program crash if it encounters an error that it cannot recover from.

`raise from`

The `raise from` idiom can be used to re-raise an exception which was caught in an `except` block. This is useful in instance where we want to add more context to an exception, so as to give more meaning to a given exception that was raised.

```
>>> try:
...     x = int("abc")
... except ValueError as e:
...     raise ValueError(f"Variable `x` recieved invalid input:
{\
e}") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

```
ValueError: invalid literal for int() with base 10: 'abc'

The above exception was the direct cause of the following excepti\
on:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Variable `x` recieved invalid input: invalid literal \
for int() with base 10: 'abc'
```

`else` and `finally`

An `else` statement can be added to a `try`/`except` block to handle instances where an exception was not raised. If however an exception is raised in the `try` block, the `else` statement is not executed. Code that should be executed regardless of whether or not an exception was raised can be placed in a `finally` block. This block executes regardless of if the program flow is on the normal track or the exception track.

```
>>> try:
...     pass
... except Exception:
...     print("an exception was raised")
... else:
...     print("no exception was raised")
... finally:
...     print("this is going to be executed regardless")
...
no exception was raised
this is going to be executed regardless
>>> try:
...     raise ValueError
... except ArithmeticError:
...     pass
... finally:
...     print("this is going to be executed regardless")
...
this is going to be executed regardless
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError
```

`match`

A `match` statement allows you to match an expression against a set of patterns, and execute different code depending on which pattern the value matches. It's similar to a series of `if`/`elif`/`else` statements, but instead of relying on the truthiness of a given value, it relies on structural pattern matching.

```
match <expression>:
    case <pattern>:
        <block>
```

Structural pattern matching is a programming paradigm that allows you to match the both the structure of, and the values in, an expression by comparing it against patterns defined in a series of `case` statements. The comparison is done from top to bottom, until an exact match is confirmed. Once a match is confirmed, the action associated with the matching pattern is executed. If an exact match is not found, a wildcard case, defined as `case _:`, if provided, will be used as the matching case with no corresponding variable assignment. If an exact match is not confirmed and a wildcard case does not exist, the entire match block will be a no-op and no action will be taken.

```
>>> coordinate = (0, 1)
>>> match coordinate:
...     case (x, 0):
...         print(f"coordinate is on the y axis, x={x}")
...     case (0, y):
...         print(f"coordinate is on the x axis, y={y}")
...     case _:
...         print("coordinate is on neither axis")
...
coordinate is on the x axis, y=1
```

In this example, a variable `coordinate` is defined as a tuple with the values `(0, 1)`. The match statement is then used to compare the coordinate variable against different patterns in a series of case statements.

In the first `case` statement, the pattern is `(x, 0)`, which matches if the second value of the tuple is 0, and the first value can be any value, represented by the variable `x`. If this pattern is matched, the string “coordinate is on the y axis, x=<value of x>” is printed, where <value of x> is replaced with the value of `x`. This pattern however does not match the coordinate case, so the case block is passed over.

In the second `case`, the pattern is `(0, y)`, which matches if the first value of the tuple is 0, and the second value can be any value, represented by the variable `y`. If this pattern is matched, the string “coordinate is on the x axis, y=<value of y>” is printed, where <value of y> is replaced with the value of y. This pattern is matched to the coordinate case, so the variable `y` is assigned, and the case block executes, printing the formatted string literal.

In the third `case` statement, the pattern is simply `_`, which is the wildcard pattern that matches any value. This case would be executed if no prior cases were to match the expression. Since the second `case` was a match, this block was passed over.

Matches can be composed of any array of objects, from simple primitive types to collections and even user-defined types. Furthermore, collections can make use of unpacking operators to assign values to variables.

```
>>> location = {"city": "Boise", "state": "Idaho", "country": "US\
"}
>>> match location:
...     case {"country": "US", **rest}:
...         print((
...             f"the city {rest['city']}, {rest['state']}"
...             " is in the United States"
...         ))
...
the city Boise, Idaho is in the United States
>>> cities = ["Seattle", "Spokane", "Portland", "Boise"]
>>> match cities:
...     case ["Seattle", "Spokane", *rest]:
...         print(rest)
...
['Portland', 'Boise']
```

type checking

Match statements are capable of performing type checks against a given expression, executing case blocks only if the match expression adheres to a specified type. The mechanism is called a class pattern, as classes are the structure Python uses to define types.

To create a class pattern, the type definition is called with patterns as arguments. If no argument is specified, any expression that matches the type is considered a match.

```
>>> item = 1
>>> match item:
...     case int(0 | -1):
...         print(f"{item} is either 0 or -1")
...     case int():
...         print(f"{item} is an int, not zero or -1")
...
1 is an int, not 0 or -1
```

In this example, we're using a match statement to check both the type and the value of the given `item`. The first case defines a class pattern for matching int types if the item matches the pattern of `0` or `-1`. This pattern fails to match because `item = 1`. The second case defines a class pattern for matching all int types, since a pattern is not provided in the type call. This pattern matches the item, so the case block is executed.

guards

Additional conditions for pattern matching can be included using guard statements. A guard is an `if` statement contained in the case which checks the truthiness of an expression, and if that expression is truthy, the pattern is matched.

```
>>> coordinate = (0, 1)
>>> match coordinate:
...     case (0, y) if y == 1:
...         print((
...             "coordinate is on the x axis,"
```

```
...                 f" and on the unit circle at y={y}"
...             ))
...         case (0, y):
...             print(f"coordinate is on the x axis, y={y}")
...
coordinate is on the x axis, and on the unit circle at y=1
```

In this example, the first case matches because both the pattern of the coordinate matches the case, and the expression `y == 1` is `True`.

Or Patterns

The pipe operator `|` can be used to combine multiple patterns in a single case statement. This defines an “or” relationship between patterns, where the expression can match either the first or second operand. If a value matches either of the pipe’s operands, the case is matched and the case block is executed.

```
>>> coordinate = (0, 1)
>>> match coordinate:
...     case (0, _) | (_, 0):
...         x, y = coordinate
...         print((
...             "coordinate is on an axis,"
...             f" x={x}, y={y}"
...         ))
...
coordinate is on an axis, x=0, y=1
```

If an or pattern is used for variable binding, each pattern should define the same variable names. If different patterns define different variable names, the case will throw a `SyntaxError`.

```
>>> coordinate = (0, 1)
>>> match coordinate:
...     case (0, y) | (x, 0):
...         print((
...             "coordinate is on an axis,"
...             f" x={x}, y={y}"
...         ))
...
```

```
  File "<stdin>", line 2
SyntaxError: alternative patterns bind different names
```

as

The `as` keyword can be used to assign to a variable a pattern that matches a piece of the expression being matched.

```
>>> coordinate = (0, 1)
>>> match coordinate:
...     case (0 as x, y) | (x, 0 as y):
...         print((
...             "coordinate is on an axis,"
...             f" x={x}, y={y}"
...         ))
...
```

In this example, the or pattern is used in the case definition, and each pattern in the case maps the values of the coordinate to the variables `x` and `y`. Both variables are defined in each pattern, so the case does not throw a syntax error. The `as` keyword is used to bind matching values to variables, which can subsequently be used in the case block when the pattern is matched.

Chapter 5. Functions

Functions are some of the most fundamental building blocks in programming. They allow you to encapsulate pieces of code and reuse it multiple times throughout your program. This is beneficial, in that it helps you avoid repeating the same code in multiple places, which can lead to bugs and make your code harder to maintain. In short, functions are a powerful tool that allow you to write better, more efficient, and more maintainable code.

In Python, a function is defined using the `def` keyword, followed by the name of the function, a set of parentheses which contain the arguments of the function, and finally a colon `:`. The code that makes up the function's body is indented under the definition line. Finally, any values that are returned to the function caller are specified by the `return` keyword. If a `return` is not defined, the function implicitly returns `None`.

For example, the following code defines a simple function called greet that takes in a single argument, name, and returns a greeting using that parameter:

```
>>> def greet(name):
...     greeting = f"Hello, {name}!"
...     return greeting
...
>>> greet("Ricky")
'Hello, Ricky!'
```

Functions are not executed immediately when they are defined. Instead, they are executed only when they are *called*. This means that when a script or program is running, the interpreter will read the function definition, but it will not execute the code within the function block until the program calls the function.

Consider the following script:

```
# ./script.py

def my_function():
    print("Function called")

print("Script start")
my_function()
print("Script end")
```

When this script is run, the Python interpreter first reads the function definition. The function is defined, but the function code is not executed until the function is called on the line `my_function()`. As a result, the output will be:

```
root@b854aeada00a:~/code# python script.py
Script start
Function called
Script end
```

Once a function is defined, it can be called multiple times, and each time it will execute the code inside the function.

Function Signatures

A function signature is the combination of the function’s name and its input parameters. In Python, the function signature includes the name of the function after the `def` keyword, followed by the names of the function parameters contained in parentheses.

```
   function name
     |
     |   /--/------- function parameters
def add(a, b):
```

The function name is the identifier that is used to call the function. It should be chosen to be descriptive and meaningful, so other developers can ascertain the function’s purpose.

The parameters of a function are variables that are assigned in the scope of the body of the function when it is called. In Python, the parameters are defined in parentheses following the function name. Each parameter has a name, which is used to reference values within the function. It's also possible to define a function with no parameters by using empty parentheses.

These parameters can be assigned to positional arguments or keyword arguments. A positional argument is a function argument which is assigned a name in accordance to the order in which the names are defined in the function signature. A keyword argument is an argument which is explicitly assigned to a specific name using a `key=value` syntax.

```
>>> def sub(a, b):
...     return a - b
...
>>> sub(1, 2)
-1
>>> sub(2, 1)
1
>>> sub(b=2, a=1)
-1
```

In this example, we define a `sub()` function with two arguments, `a` and `b`. `a` is the first argument in the function signature, and `b` is the second.

When we call this function using positional arguments only, the first argument is assigned to the name `a` and the second argument is assigned to the name `b`. In the first function call, we pass the values `1, 2`. Since `1` is the argument passed first, it is assigned to the name `a`, and subsequently `2` is assigned to `b`. The function returns `a - b`, which in this case is `1 - 2` which results to `-1`.

If we reverse the order of the arguments in the second function call, passing the values `2, 1`, the `2` is now assigned to `a` and the `1` is assigned to `b`. The function still returns the result of `a - b`, but in this case the expression is `2 - 1`, so the function returns `1`.

In the final function call, we explicitly assign values to the argument names using the `key=value` syntax, by passing the values `b=2, a=1`. Keyword arguments take preferential assignment over positional arguments, so even though the `2` is passed first in the function signature, it is explicitly assigned to `b`, where `a` is explicitly assigned to `1`. This again results in returning the expression `1 - 2`, so the function returns `-1`.

Function calls can make use of both positional and keyword arguments, but any positional arguments must be listed first when calling the function. Given this flexibility, it's possible to accidentally call a function with a keyword argument which references a name which was already assigned a value via a positional argument. Yet multiple arguments cannot be assigned to the same name, so doing this will cause the interpreter to raise a `TypeError`.

```
>>> def sub_add(a, b, c):
...     return a - b + c
...
>>> sub_add(1, c=2, b=3)
0
>>> sub_add(1, 2, b=3)
  File "<stdin>", line 1, in <module>
TypeError: sub_add() got multiple values for argument 'b'
```

In this example, we define a `sub_add()` function with three arguments, `a`, `b`, and `c`, which returns the value of the expression `a - b + c`.

In the first function call, we pass `1` as a positional argument, which results in the value `1` being assigned to `a`. The next two arguments are keyword arguments, assigning `c` the value `2` and `b` the value `3`. This results in the function returning the value of the expression `1 - 3 + 2`, which results to `0`.

In the second example, we pass `1, 2` as positional arguments, which results in the value `1` being assigned to `a` and the value `2` being assigned to `b`. However we then specify the keyword argument `b=3`. This results in two different values being assigned to `b`, which is not allowed, so the interpreter raises a `TypeError`.

Explicitly positional/key-value

Optionally, variables can be explicitly made to be positional or keyword arguments. A function signature containing `/` specifies that the arguments to the left in the function signature are positional only. `*` conversely specifies that arguments to the right in the function signature are keyword only. Arguments between the two can be called in either manner.

```
>>> def greet(name, /, title, *, punctuation):
...     return f"Hello {name}, {title}{punctuation}"
>>> greet("Ricky", "Esq", punctuation="!")
"Hello Ricky, Esq?"
>>> greet("Ricky", "Esq", "?")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() takes 2 positional arguments but 3 were given
```

Default Values

Function signatures can define default arguments for parameters by using the assignment operator `=`. If when a function is called, it does not specify a value for a given parameter, the default value is used.

```
>>> def greet(name, punctuation="!"):
...     return f"Hello {name}{punctuation}"
...
>>> greet("Jeff")
Hello Jeff!
```

Mutable Types as Default Values

When assigning a default value to a function, it is generally not recommended to use a mutable type, such as a list or a dictionary, as a default value. When a function is created with a default value, that same object is used in every function call where the default value is not overridden. This effectively creates shared state across all function calls, which can lead to unexpected behavior.

```
>>> def my_function(value, default=[]):
...     default.append(value)
...     return default
...
>>> x = my_function(0)
>>> y = my_function(1)
>>> y
[0, 1]
>>> x
[0, 1]
```

Unless this shared state is desired, it's better to use immutable state for default values, or a unique object to serve as an identity check.

```
>>> def my_function(
...     value,
...     default=(UNDEFINED := object())
... ):
...     if default is UNDEFINED:
...         default = []
...     default.append(value)
...     return default
...
>>> x = my_function(0)
>>> y = my_function(1)
>>> y
[1]
>>> x
[0]
```

In this example, UNDEFINED is a unique object in the enclosing scope. If no value is passed by the caller for `default`, then `default is UNDEFINED` evaluates to True and the variable is replaced with a new list. This guarantees that the returned list is a unique list.

```
>>> def create_definition(name, dtype, value, codes=()):
...     metadata = {"type": dtype}
...     for item in codes:
...         # some validation code operating per-item
...         # ...
...         if "codes" not in metadata:
...             metadata["codes"] = []
```

```
...         metadata["codes"].append(item)
...     obj = {"name": name, "value": value, "metadata": metadata}
...     return obj
...
>>> create_definition("a", "primative", 1)
{'name': 'a',
 'value': 1,
 'metadata': {'type': 'primative'}}
>>> create_definition("b", "collection", ['a', 'b', 'c'], [1, 2, \
3])
{'name': 'b',
 'value': ['a', 'b', 'c'],
 'metadata': {'type': 'collection', 'codes': [1, 2, 3]}}
```

In this example, `codes` defaults to a tuple, which is both immutable and iterable.

Scope

Up until this point, the code we've written has made no sort of distinction between when access to a variable is considered valid. Once a variable has been defined, later code is able to access its value and make use of it.

With the introduction of functions, this assumption of a variable being available at definition is no longer strictly valid. This is because variables defined within a function block are not variables which can be accessed from outside the function block. The function block is said to have it's own *local scope*, also referred to as a *namespace* (or, the space where a given variable name is assigned), and that block scope is separate from the top-level *global scope* of the interpreter.

```
def my_function():
    x = 5
    print(x)
```

In this example, the variable `x` is defined within the local scope of the function `my_function`, and it can only be accessed within the block scope of the function. If you try to access it outside the function, you will get an error.

Nested Scopes

In Python, the block scopes can be nested. A nested scope is a scope that is defined within another scope. Any scope which is nested can read variables from scopes which enclose the nested scope.

When the interpreter searches for a variable referenced in a nested scope, it follows what's referred to as the LEGB rule for searching scopes. LEGB is an acronym for Local, Enclosing, Global, and Built-in. The interpreter first looks in the local scope for a variable. If it is not found, the interpreter then looks in any enclosing scopes for the variable. If it is still not found, the interpreter then looks in the top-level global scope for the variable. If it is still not found, it looks in the built-in scope.

```
>>> greeting = "Hello"
>>> def outer_function():
...     def inner_function():
...         greeting = "Hey"
...         print(greeting)
...     inner_function()
...     print(greeting)
...
>>> outer_function()
Hey
Hello
```

In this example, the variable `greeting` is defined with different values in three different scopes: the global scope, the `outer_function` scope, and the `inner_function` scope. When the inner function is called, the interpreter first looks for the variable `greeting` in the local scope of the inner function, and finds it with the value "Hey", which is then printed to the console. This scope is then exited when the function returns. The next `print(greeting)` call looks for the variable `greeting`. This variable is not defined in the local scope, so it next checks for any enclosing scopes. There is no enclosing scope, as `outer_function` is a top-level function. So the interpreter then checks the global scope for `greeting`, where `greeting` is defined as "Hello", which is then printed to the console.

`nonlocal` and `global`

By default, assignments are assumed to be in local scope. In order to make changes to a variable in either an enclosing scope or the global scope, you need to indicate to the interpreter that you're looking to implement nondefault behavior. Python provides two keywords, `nonlocal` and `global`, for this purpose. `nonlocal` indicates that a variable reference is in an enclosing scope, and `global` indicates that a reference is within the global scope.

```
>>> global_greet = "Hello"
>>> def outer_function():
...     nonlocal_greet = "Hi"
...     def inner_function():
...         nonlocal nonlocal_greet
...         global global_greet
...         nonlocal_greet = "Hi"
...         global_greet = "Hey"
...     inner_function()
...     print(nonlocal_greet, global_greet)
...
>>> outer_function()
Hi Hey
```

In this example, the variables `global_greet` and `nonlocal_greet` are defined with different values within different scopes: `global_greet` is defined in the global scope, and `nonlocal_greet` is defined in the block scope of `outer_function`. From the context of the block scope of the `inner_function`, the scope of the `outer_function` is an enclosing scope, and the block scope of the REPL is a global scope. When `inner_function` is executed, the `nonlocal` keyword is used to specify that the variable `nonlocal_greet` is bound to the enclosing scope, and the `global` keyword is used to specify that the variable `global_greet` is bound to the global scope. When the `inner_function` then executes its assignment operations, it changes the values to which both `global_greet` and `nonlocal_greet` are referencing. This substitution is made evident by the `print()` function, which prints the new values of `global_greet` and `nonlocal_greet` to the console.

Closures

Closures are functions which have access to variables in an enclosing scope, even when the function is invoked outside of that scope. This allows the function to “remember” the values of variables from its enclosing scope, and to continue to access them, even after the enclosing scope has exited. Closures are useful for a variety of tasks, such as creating callbacks, implementing decorators, and encapsulating state.

A closure is created when a nested function references a variable from its enclosing function. For example:

```
>>> def outer_function(message):
...     def inner_function():
...         print(message)
...     return inner_function
...
>>> my_closure = outer_function("Hello")
>>> my_closure()
Hello
```

Here, the inner function `inner_function` references the variable message from its enclosing function `outer_function`. When the outer function is called, a reference to the inner function is returned. The inner function maintains a reference to the variable `message` from its enclosing scope. When the `inner_function` is invoked from the global scope, it can still access the value of the `message` variable, even though the `message` variable itself is not accessible from the global scope.

When evoking an assignment operation in closures, by default Python will create a new local variable with the same name, if one does not exist. Oftentimes however in closures the desired effect is to modify a variable from an enclosing scope. This can be done by specifying that variable using Python’s `nonlocal` keyword.

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count
...         count += 1
...         return count
...     return counter
...
>>> my_counter = make_counter()
>>> my_counter()
1
>>> my_counter()
2
```

In this example, the `make_counter` function defines a `counter` closure that increments a `count` variable and returns the current count. The user can interact with the counter by calling the closure, which increments the `count` value which was defined in the enclosing scope.

Closures are ultimately a mechanism for exposing functionality without (easily) exposing state. By using closures, you can ensure that the state of a given variable is only modified in controlled ways. This mechanism allows you to encapsulate state and control the behavior of the closure, while still providing a simple but limited interface through the returned function.

Anonymous Functions

Anonymous functions are functions that are defined without a name. They are also known as lambda functions. Anonymous functions are defined using the `lambda` keyword, followed by one or more arguments, a colon `:` and finally an expression that defines the function's behavior.

For example, the following code defines an anonymous function that takes one argument and returns its square:

```
>>> square = lambda x: x**2
>>> square(5)
25
```

Lambda functions typically used when you need to define a function that will be used only once, or when you need to pass a small function as an argument to another function.

Decorators

Decorators are a way to modify the behavior of an object by wrapping it within a function. The decorator function takes the original function as an argument and returns a new function that will replace the original function. Decorators can be applied to functions using the `@decorator` syntax, which is placed immediately before the definition of the object that is being decorated.

```
>>> def my_decorator(fn):
...     def wrapper():
...         print("entering the function...")
...         fn()
...         print("exiting the function...")
...     return wrapper
...
>>> @my_decorator
... def my_function():
...     print("inside the function...")
...
>>> my_function()
entering the function...
inside the function...
exiting the function...
```

Decorators are typically 2 or 3 functions deep. Instances of 3-deep functions allow you to configure the context of the decorator in a manner not unlike a closure.

```
>>> def my_closure(value):
...     def my_decorator(fn):
...         def wrapper():
...             print(f"value is: {value}")
...             print("entering the function...")
...             fn()
...             print("exiting the function...")
...         return wrapper
...     return my_decorator
...
```

```
>>> @my_decorator("this")
... def my_function():
...     print("inside the function.")
...
>>> my_function()
value is: this
entering the function...
inside the function.
exiting the function...
```

Decorators can also be stacked, allowing you to continuously layer functionality through the decorator syntax.

```
>>> def decorator_1(fn):
...     def wrapper():
...         print("entering decorator 1...")
...         fn()
...         print("exiting decorator 1...")
...     return wrapper
...
>>> def decorator_2(fn):
...     def wrapper():
...         print("entering decorator 2...")
...         fn()
...         print("exiting decorator 2...")
...     return wrapper
...
>>> def decorator_3(fn):
...     def wrapper():
...         print("entering decorator 3...")
...         fn()
...         print("exiting decorator 3...")
...     return wrapper
...
>>> @decorator_1
... @decorator_2
... @decorator_3
... def my_function():
...     print("inside the function.")
...
>>> my_function()
entering decorator 1...
entering decorator 2...
entering decorator 3...
inside the function.
exiting decorator 3...
exiting decorator 2...
exiting decorator 1...
```

It’s important to note that decorators are a *syntactic sugar*, meaning they make a particular design pattern more elegant, but they do not add any additional functionality to the language.

```
>>> def my_decorator(fn):
...     def wrapper():
...         fn()
...     return wrapper
...
>>> # syntactic sugar, because this...
>>> @my_decorator
... def my_function():
...     pass
...
>>> # is functionally equivalent to this...
>>> my_function = my_decorator(my_function)
...
```

Chapter 6. Classes

Classes are an essential concept in object-oriented programming and are used to define new types of objects. They allow you to abstract the state of an object and encapsulate it within a single entity, making it easy to manage and manipulate. By creating classes, you can define the properties and methods of an object, using them to represent higher-abstraction entities or concepts in your code, all while hiding the implementation details of how those properties and methods work. This allows you to separate the interface of an object, which defines how it can be used, from its implementation, which defines how it works.

In Python, a class is defined using the `class` keyword, followed by the name of the class. The class definition typically starts with an indented block of code, which is known as the class body.

```
>>> class MyClass:
...     pass
```

The pass keyword is used as a placeholder and does not do anything, but it is needed in this case to create an empty class.

Once a class is defined, it can be instantiated by calling the type.

```
>>> my_object = MyClass()
>>> my_object
```

Inside the block scope of a class, you can define methods that are associated with an object or class. Methods are used to define the behavior of an object and to perform operations on the properties of the object.

Methods are defined inside a class using the `def` keyword, followed by the name of the method, a set of parentheses which contain the method’s arguments, and finally a colon `:`. The first parameter of a standard method is always a variable which references the instance of the class. By convention, this is typically given the name `self`.

For example:

```
>>> class MyClass:
...     def my_method(self):
...         print("Hello from my_method!")
...
```

Once a method is defined, it can be called on an instance of the class using a dot notation.

```
>>> my_object = MyClass()
>>> my_object.my_method()
Hello from my_method!
```

Methods can also take additional parameters and return values, similar to functions.

```
>>> class MyClass:
...     def add(self, a, b):
...         return a + b
...
...
>>> my_object = MyClass()
>>> result = my_object.add(2, 3)
>>> result
5
```

In fact, to a certain extent methods *are* functions, bound to what's referred to as the namespace of the class. Functionally there is no difference between `str.join("", ('a', 'b'))` and `"".join(('a', 'b'))`

Finally, classes can subclass other classes, which allows them to derive functionality from a superclass. This concept is generally known as inheritance, which will be discussed in more depth at a later time.

```
>>> class YourClass:
...     def subtrace(self, a, b):
...         return a - b
```

```
...
>>> class MyClass(YourClass):
...     pass
...
>>> my_object = MyClass()
>>> result = my_object.subtract(2, 3)
>>> result
-1
```

data types as classes.

You might have noticed that earlier in the book when we called the `type()` function on an object, the return specified that the `type` was oftentimes a class.

```
>>> x = "Hello!"
>>> x
"Hello!"
>>> type(x)
<class 'str'>
```

In Python, all of the built-in data types, such as integers, strings, lists, and dictionaries, are implemented in a manner similar to classes. And similar to user-defined classes, all of the built-in data types in Python have associated methods that can be used to manipulate their underlying data structures. For example, and as we just saw, the `str` class defines a method called `join()`. This join method takes the instance of the calling object, a string, as `self`, and a collection of other strings which are joined using `self` as the separator.

```
>>> " ".join(('a', 'b', 'c'))
'a b c'
```

Given this, calling the class method is equally valid; in this case we're simply passing the `self` parameter explicitly.

```
>>> str.join(" ", ('a', 'b', 'c'))
'a b c'
```

`__dunder__` methods

"dunder" (short for "double underscore") methods, also known as "magic methods", are special methods that have a specific meaning and are used to define specific behaviors of objects. They are identified by their double underscore prefix and suffix, such as `__init__` or `__len__`.

For now we're only going to focus on one particular method, the `__init__` method, and we'll return to review the others in later chapters.

The `__init__` method

The `__init__` method is a special method that is automatically called after an object is created. It is used to initialize the attributes of the object. The only thing required to create an `__init__` method is to name it as `__init__`.

*Note: `__init__` is **not** a constructor. More on this later.*

Once you have defined the `__init__` method, you can create an instance of the class, and initialize any attributes you wish to assign the instance. This is done by passing in the values as arguments when you create the object by calling it. Those values will be passed into the call of the `__init__` method, where they can be assigned to the instance `self` via an instance attribute.

```
>>> class MyClass:
...     def __init__(self, my_value):
...         self.my_value = my_value
...
>>> my_object = MyClass(5)
>>> my_object.my_value
5
```

Attributes

In Python, attributes are a way to define variables that are associated with an object or class. They are used to store the state of an object and can be accessed or modified using the dot notation.

There are different ways to define attributes in Python, but the most common approach is to use instance variables, which can be defined inside any class method using the `self` keyword.

This was demonstrated in the previous example, where the `__init__` method defines an attribute called `my_value`, which is assigned to the instance `self`.

You can access, assign, or modify attribute values using the dot notation. For example, the following code creates an instance of the `MyClass` class and sets its my_value attribute to 10:

```
>>> my_object = MyClass(5)
>>> my_object.my_value
5
>>> my_object.my_value = 10
>>> my_object.my_value
10
```

Class Attributes

Class attributes are variables that are defined at the class level, rather than the instance level. They are shared amongst all instances of a class and can be accessed using either the class name, or an instance of the class.

```
>>> class MyClass:
...     class_attribute = "I am a class attribute."
...
>>> obj1 = MyClass()
>>> obj2 = MyClass()

>>> obj1.class_attribute
I am a class attribute.
>>> obj2.class_attribute
I am a class attribute.
>>> MyClass.class_attribute
I am a class attribute.
```

It's worth noting that the same precaution about using default function arguments also applies to class attributes. Since class attributes are shared across all instances of the class, mutable state should likewise be avoided.

```
>>> class One:
...     items = []
...
>>> a = One()
>>> b = One()
>>> a.items.append(1)
>>> b.items
[1]
```

A Functional Approach

Python provides the functions `hasattr()`, `getattr()`, `setattr()`, and `delattr()`, which can be used to work with attributes of objects.

The `hasattr()` function is used to check if an object has a particular attribute or not. It takes two arguments: the object, and the name of the attribute as a string. It returns `True` if the object has the attribute, and `False` otherwise.

The `getattr()` function retrieves the value of an attribute of an object. It takes at least two arguments: the object, and the name of the attribute as a string. It returns the value of the attribute if it exists, and raises an `AttributeError` if it does not. You can also provide an optional third parameter to serve as a default value, which will be returned if the attribute does not exist.

The `setattr()` function sets the value of an attribute of an object. It takes three arguments: the object, the name of the attribute as a string, and the value to which to set the attribute. If the attribute does not exist, it will be created.

Finally, the `delattr()` function deletes an attribute of an object. It takes two arguments: the object, and the name of the attribute as a string. If the attribute exists, it will be deleted; if it does not exist the function call will raise an `AttributeError`.

```
>>> class MyClass:
...     def __init__(self, my_value):
...         self.my_value = my_value
...
>>> my_object = MyClass(1)
>>> hasattr(my_object, "my_value")
True
>>> getattr(my_object, "my_value")
1
>>> setattr(my_object, "my_value", 5)
>>> getattr(my_object, "my_value")
5
>>> delattr(my_object, "my_value")
>>> hasattr(my_object, "my_value")
False
>>> getattr(my_object, "my_value", -1)
-1
>>> getattr(my_object, "my_value")
Traceback (most recent call last):
  File "<stdin>", line 13, in <module>
AttributeError: 'object' object has no attribute 'my_value'
```

`@staticmethod`

`@staticmethod` is a decorator that is used to define static methods. A static method is effectively a function which exists in the namespace of the class. It does not have access to the class or instance state, meaning it cannot modify or access any attributes or methods of the class or its instances.

```
>>> class MyClass:
...     @staticmethod
...     def static_method():
...         return "I am a static method."
...
>>> MyClass.static_method()
I am a static method.
>>> obj = MyClass()
>>> obj.static_method()
I am a static method.
```

`@classmethod`

`@classmethod` is a decorator that is used to define class methods. A class method is a method that is bound to the class and not the instance of the object. It can be called on the class itself, as well as on any instance of the class. A class method takes the class as its first argument, typically aliased as `cls`.

```python
>>> class MyClass:
...     def __init__(self, value=None):
...         self.value = value
...
...     @classmethod
...     def class_method(cls, value):
...         return cls(value)
...
>>> MyClass.class_method("Hello World!")
<__main__.MyClass object at 0x7fe83b29c040>
>>> obj = MyClass()
>>> obj.class_method("Hello World!")
<__main__.MyClass at 0x7fe83b39c6d0>
>>> class YourClass(MyClass): pass
>>> YourClass.class_method("Hello World!")
<__main__.YourClass at 0x7fe83bdec3d0>
```

A common use case for class methods is to define alternative constructors for a class. For example, you might want to create an object from a string or from a file. To do this you can define class methods that handle these cases and return instances of the class. The benefit of using a `@classmethod` over a `@staticmethod` in this scenario is that `cls` is bound to the class that calls it, and this includes subclasses.

Part II. A Deeper Dive

Chapter 7. Expressions, Comprehensions, and Generators

In Python an expression is a piece of code that evaluates to a value. They can be used in a variety of contexts, such as in assignment statements, function calls, and control flow statements. Expressions can take many forms, including arithmetic expressions, logical expressions, and function calls. And in certain cases, expressions can be used to create new objects, some of which we'll discuss here.

Generator Expressions

A generator expression is a special type of expression that returns an object, called an iterator, that generates values on-the-fly, one at a time, rather than eagerly creating a data structure in memory. The iterator can be used in a `for` loop or other iteration constructs to iterate over the values in the expression.

The syntax for a generator expression is similar to that of a normal expression, but with a single for clause. The expression is enclosed within parenthesis `()` and followed by one or more `for` clauses and/or `if/else` clauses. The `for` and `if/else` clauses are used to filter or map the data from the iterable and construct the generator.

```
(item for item in iterable if condition)
(item if condition else other for item in iterable)
```

Generator expressions are useful when working with large data sets or when the values in the expression are the result of some expensive computation. Because they generate values on-the-fly, they can be more memory-efficient than creating a list in memory.

They can also be used to build powerful and efficient iterator pipelines. Using iterators in this fashion allows you to perform multiple operations on data without necessitating that each operation hold intermediary values in memory.

```
>>> numbers = range(10)
>>> # generator expression, doesn't
>>> #create intermediary state
>>> even_numbers = (
...     n for n in range(10) if n % 2 == 0
... )
>>> # another generator expression, doesn't
>>> # create intermediary state
>>> squared_numbers = (n**2 for n in even_numbers)
>>> # state only materializes in the final list
>>> list(squared_numbers)
[0, 4, 16, 36, 64]
```

Generator Functions

Generator functions are similar to normal functions, but instead of returning a value, they return a generator object, similar to generator expressions, which can be used to iterate over a sequence of values. A generator function is defined like a normal function, using the `def` keyword, but instead of using `return` to return a value, it uses the `yield` keyword to yield a value per-iteration. The generator function yields multiple values, one at a time, during its execution.

```
>>> def my_generator():
...     yield 1
...     yield 2
...     yield 3
```

When a generator function is called, it returns a generator object, but does not start executing the function. The function is only executed when the generator object is iterated over, which can be done manually using the `next()` function, or in the context of iteration, like a `for` loop. Each time the generator function yields a value, it is returned to the caller and the function’s execution is paused. The next time the generator is iterated over, it resumes executing where it left off, with a saved state.

```
>>> gen = my_generator()
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
```

`yield from`

The `yield from` statement can be used to pass the context of an iteration from one generator to the next. When a generator executes `yield from`, the control of an iteration passes to the called generator or iterator. When that generator is exhausted, control is yielded back to the calling generator for further iteration.

```
>>> def _generator():
...     yield 2
...
>>> def generator():
...     yield 1
...     yield from _generator()
...     yield 3
...
>>> for i in generator():
...     print(i)
1
2
3
```

List Comprehensions

List comprehensions are built-in way to create a list by applying a single expression to each item in an existing iterable. The syntax of a list comprehension is similar to the generator expression, but it is enclosed within square brackets `[]`. The expression is followed by one or more `for` clauses and/or `if/else` clauses, like a generator expression. The `for` clause is used to iterate over the items in the iterable, and the `if/else` clause is used to filter the items.

```
[item for item in iterable if condition]
[item if condition else other for item in iterable]
```

List comprehensions are generally faster than their equivalent python for loop, because the code that generates the list is fully implemented in C.

```
>>> [x**2 for x in range(4)]
[0, 1, 4, 9]
```

Dictionary Comprehensions

Dictionary comprehensions provide a means for creating dictionaries from expressions. The syntax for a dictionary comprehension is similar to that of a list comprehension, but with curly braces `{}` instead of square brackets `[]`.

The dictionary comprehension consists of an expression, and one or more `for` clauses and/or `if`/`else` clauses, which are used to construct the key-value pairs of the dictionary. Oftentimes the iterable is an iterator which yields `len(2)` tuples which can be unpacked into `key` and `value` variables.

```
{key: value for (key, value) in iterable if condition}
```

This however is not strictly necessary; for example the following dictionary comprehension creates a dictionary that maps some letters of the alphabet to their corresponding ASCII values:

```
>>> {chr(i): i for i in range(97, 100)}
{'a': 97, 'b': 98, 'c': 99}
```

Expressions and the Walrus Operator

The walrus operator can be used in a list comprehension to assign a value to a variable in the `for` clause and then use that variable in the expression. This can be useful when you want to use the value of a variable multiple times in the comprehension, or when the value of the variable is the result of an expensive computation.

```
>>> # collects the square of n only if that square is even
>>> [sq for n in range(1, 11) if (sq := n**2) % 2 == 0]
[4, 16, 36, 64, 100]
```

Chapter 8. Python's Built-in Functions

Python provides a wide range of built-in functions that we can directly leverage. These functions are accessible from the built-in namespace, and we can use them without the need to import any additional libraries. These functions are built into the Python interpreter and provide a wide range of functionality, from basic operations such as mathematical calculations and string manipulation, to more advanced functionality such as file I/O and error handling. In this chapter, we will explore some of the most commonly used built-in functions in Python and learn how to use them.

The list of builtin functions include:

- `abs(x)` - Returns the absolute value of a number `x`
- `aiter(i)` - Asynchronous iterator, return an asynchronous iterator from an asynchronous iterable `i`.
- `all(i)` - Returns `True` if all elements in an iterable `i` are true
- `any(i)` - Returns `True` if at least one element in an iterable `i` is true
- `anext(ai)` - Retrieve the next item from an asynchronous iterator `ai`.
- `ascii(o)` - Returns a string containing a printable representation of an object `o`
- `bin(x)` - Converts an integer `x` to a binary string
- `bool(x)` - Converts a value `x` into a Boolean
- `breakpoint()` - Drops the runtime into a python debugging session.
- `bytearray(*args, **kwargs)` - Returns a bytearray object
- `bytes(*args, **kwargs)` - Returns a bytes object
- `callable(o)` - Returns True if the object `o` is callable, False otherwise
- `chr(c)` - Returns a string representing a character `c` whose Unicode code point is the integer
- `classmethod(fn)` - Returns a class method for a function `fn`
- `compile(s, *args, **kwargs)` - Returns a code object from source `s` (string, file, etc.)
- `complex(r, i)` - Returns a complex number where `r` is real and `i` is imaginary

- `delattr(o, n)` - Deletes an attribute of name `n` from an object `o`
- `dict()` - Returns a new dictionary
- `dir(o)` - Returns a list of names in the namespace of object `o`
- `divmod(a, b)` - Takes two numbers `a` and `b` and returns a pair of numbers (a tuple) consisting of their quotient and remainder
- `enumerate(o[, s])` - Returns an enumerate object, which can be used to iterate over an iterable `o` and get the index of each element. Optional start value starts the enumeration at value `s`
- `eval(s, *args)` - Evaluates a string `s` as a Python expression
- `exec(s, *args, **kwargs)` - Used for the dynamic execution of source `s` which is a valid Python programs
- `filter(fn, i)` - Returns an iterator from elements of an iterable `i` for which a function `fn` returns `True`. Uses lazy evaluation.
- `float(x)` - Converts a value `x` to a float
- `format(o)` - Returns a formatted string version of an object `o`
- `frozenset()` - Returns an immutable frozenset object
- `getattr(o, n[, d])` - Returns the value of a named attribute `n` from object `o`. Optional default `d` is returned if attribute does not exist.
- `globals()` - Returns the current global symbol table as a dictionary
- `hasattr(o, n)` - Returns `True` if the object `o` has the given named attribute `n`, `False` otherwise
- `hash(o)` - Returns the hash value of an object `o`
- `help(o)` - Invokes the built-in help system for object `o`
- `hex(i)` - Converts an integer `i` to a hexadecimal string
- `id(o)` - Returns the identity of an object `o`
- `input(p)` - Reads a line from input, after printing the optional prompt `p`
- `int(x)` - Converts a value `x` to an integer
- `isinstance(o, t)` - Returns `True` if the object `o` is an instance of the specified type `t`, `False` otherwise. If `t` is tuple or union, checks against all types in t.
- `issubclass(c, s)` - Returns True if a class `c` is a subclass of a specified superclass `s`, False otherwise
- `iter(o)` - Returns an iterator of object `o`

- `len(o)` - Returns the length of an object `o`
- `list()` - Returns a new list
- `locals()` - Returns an updated dictionary of the current namespace
- `map(fn, i)` - Returns an iterator which applies a function `fn` to all items in an iterable `i`. Uses lazy evaluation.
- `max(*args[, key=fn])` - Returns the largest item of two or more arguments, optional key is a function `fn` to apply to each item
- `memoryview(o)` - Returns a memory view object of the given object `o`
- `min(*args[, key=fn])` - Returns the smallest item of two or more arguments, optional key is a function `fn` to apply to each item
- `next(i)` - Retrieves the next item from an iterator `i`
- `object()` - Returns a new featureless object
- `oct(i)` - Converts an integer `i` to an octal string
- `open(p, m, *args, **kwargs)` - Opens a file and returns a file object
- `ord(c)` - Given a string representing one Unicode character `c`, returns an integer representing the Unicode code point of that character
- `pow(x, y)` - Returns the value of x to the power of y (x**y)
- `print(*args, **kwargs)` - Prints the specified messages `args` to the screen
- `property(fg[, fs, fd, d])` - Gets, sets, or deletes a property of an object
- `range(*args, **kwargs)` - Returns a sequence of numbers
- `repr(o)` - Returns a string containing a printable representation of an object `o`
- `reversed(s)` - Returns a reversed iterator of a sequence `s`
- `round(n, d)` - Rounds a number `n` to the nearest integer, or to the specified number of decimals `d`.
- `set()` - Returns a new set object
- `setattr(o, n, v)` - Sets the value `v` of a named attribute `n` of an object `o`
- `slice(*args, **kwargs)` - Returns a slice object
- `sorted(i[, key=fn, reversed=False])` - Returns a sorted list from the specified iterable `i`, optional key is a function `fn` to apply to each item, and is reversed if keword `reversed` is truthy.

- `staticmethod(fn)` - Returns a static method for a function `fn`
- `str()` - Returns a string object
- `sum(i)` - Sums the items of an iterable `i`
- `super()` - Returns a temporary object of the superclass
- `tuple()` - Returns a new tuple object
- `type(o)` - Returns the type of an object `o`
- `vars(o)` - Returns the dict attribute of an object `o`
- `zip(*args[, strict=False])` - Returns an iterator of tuples, where the first item in each passed iterator is paired together, and then the second item in each passed iterator is paired together, etc.

We'll discuss some of the more common builtin synchronous functions here, and discuss the async functions more in depth later.

Type Conversions

Type conversions are the process of converting one data type to another data type. Python provides the built-in functions `int()`, `float()`, `complex()`, `str()`, `bytes()`, `list()`, `dict()`, `set()`, and `frozenset()` to use for type conversions.

The `int()` function is used to convert a number or a string into an integer. It takes a single object and returns a new object of type `int` if a conversion is possible, else it throws a `ValueError`. An optional `base` keyword can be provided if the string is in a format other than base 10.

```
>>> int('123')
123
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> int('101', base=2)
5
```

The `float()` function is used to convert a number or string a floating-point number. If conversion fails, throws a `ValueError`.

```
>>> float('12.34')
12.34
>>> float('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'a'
```

The `complex()` function is used to create a complex value of the form `(x+yj)`

```
>>> complex(1, 2)
(1+2j)
```

The `str()` function is used to convert create a string representation of a given object. An optional `encoding` option can be provided if the object requires transpiling into utf8.

```
>>> str(123)
'123'
>>> str(b'\xdf', encoding='latin1')
'ß'
```

The `bytes()` function is used to create an immutable bytes object from either a string, an iterable of integers between 0-255, an integer, or another bytes object. If a string is provided, a second `encoding` argument must specify the string encoding for type conversion. If the argument is a single integer, the function returns a null-value bytes object of length equal to the integer.

```
>>> bytes('abc', encoding='utf8')
b'abc'
>>> bytes(2)
b'\x00\x00'
>>> bytes((1, 2, 3))
b'\x01\x02\x03'
```

The `list()` function is used to convert iterables into a list. If no argument is provided, it creates a new, empty list.

```
>>> list('abcde')
['a', 'b', 'c', 'd', 'e']
>>> list()
[]
```

The `dict()` function is used to convert an iterable of paired values into a dictionary. It also can create new dictionaries from `key=value` arguments. If no arguments are provided, it returns a new, empty dictionary.

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

The `set()` function is used to convert an iterable of values into a set. If no iterable is provided, it returns a new, empty set.

```
>>> set([1,2,3,1])
{1, 2, 3}
>>> set()
set()
```

The `frozenset()` function is used to convert an iterable of values into a set that is also immutable. If no iterable is provided, it returns a new, empty frozenset.

```
>>> frozenset([1,2,3,1])
frozenset({1, 2, 3})
>>> frozenset()
frozenset()
```

Mathematical Functions

Python provides several built-in mathematical functions that can be used to perform various mathematical operations, such as `abs()`, `min()`, `max()`, `sum()`, and `pow()`.

The `abs()` function returns the absolute value of a number.

```
>>> abs(-5)
5
>>> abs(5)
5
```

The `min()` function returns the smallest item in an iterable or the smallest of two or more arguments. An optional `key` keyword argument can be provided, which is a function to be applied to each item before comparison. If the item is an iterable, a `default` keyword argument can also be provided, in case the iterable is empty.

```
>>> min([1, 2, 3, 4, 5])
1
>>> min(1, 2, 3, 4, 5)
1
>>> min('abc', key=lambda x: ord(x))
'a'
>>> min((), default=0)
0
```

The `max()` function returns the largest item in an iterable or the largest of two or more arguments. The same optional keyword values from `min` apply.

```
>>> max([1, 2, 3, 4, 5])
5
>>> max(1, 2, 3, 4, 5)
5
>>> max('abc', key=lambda x: ord(x))
'c'
>>> max((), default=0)
0
```

The `sum()` function returns the sum of all items in an iterable. It also accepts an optional second argument which is used as the starting value.

```
>>> sum([1, 2, 3, 4, 5])
15
>>> sum([1, 2, 3, 4, 5], 10)
25
```

The `pow()` function returns the value of x to the power of y (x**y).

```
>>> pow(2, 3)
8
```

`all()` and `any()`

The `all()` and `any()` functions are used to check if all or any of the elements in an iterable are truthy. The `all()` function returns `True` if all elements in an iterable are truthy and `False` otherwise. The `any()` function returns `True` if any of the elements in an iterable are truthy, and `False` otherwise.

```
>>> all([True, True, True])
True
>>> all([True, True, False])
False
>>> all([0, 1, 2])
False
>>> all([])
True

>>> any([True, True, True])
True
>>> any([True, True, False])
True
>>> any([0, 1, 2])
True
>>> any([])
False
```

`all()` and `any()` can also be used with an expression to check if all or any elements in a sequence meet a certain condition.

```
>>> my_list = [1, 2, 3, 4]
>>> all(i > 0 for i in my_list)
True
>>> any(i < 0 for i in my_list)
False
```

`dir()`

The dir() function is used to find out the attributes and methods of an object. When called without an argument, dir() returns a list of names in the current local scope or global scope. When called with an argument, it returns a list of attribute and method names in the namespace of the object.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

`enumerate()`

The `enumerate()` function is used to iterate over an iterable. It returns an enumerate object, which can be used to access the index and value of each element in the iterable. It takes an iterable as an argument, and an optional `start` value which specifies from which value to start counting.

```
>>> fruits = ['apple', 'banana', 'orange']
>>> for i, fruit in enumerate(fruits, start=1):
...     print(i, fruit)
...
1 apple
2 banana
3 orange
```

`eval()` and `exec()`

The `eval()` and `exec()` functions are built-in Python functions that are used to evaluate and execute code, respectively.

The `eval()` function takes a least one argument, which is a string containing a valid Python expression, and evaluates it, returning the result of the expression. It also takes optional `globals` and `locals` arguments which act as global and local namespaces. If these values aren't provided, the global and local namespaces of the current scope are used.

```
>>> x = 1
>>> y = 2
>>> eval("x + y")
3
>>> eval("z + a", {}, {"z": 2, "a": 3})
5
```

Similarly, `exec()` function takes a single argument, which is a string containing valid Python code, and executes it. It also takes optional `globals` and `locals` arguments.

```
>>> x = 1
>>> y = 2
>>> exec("result = x + y")
>>> result
3
```

It's important to note that `eval()` and `exec()` can execute any code that could be written in a Python script. If the strings passed to these functions are not properly sanitized, a hacker could use them to execute arbitrary code with the permissions of the user running the script.

Both `eval()` and `exec()` have the potential to introduce security vulnerabilities in your code, so it's important to be extremely careful when using them, and to avoid them if possible.

`map()`

The map() function applies a given function to all items of an input iterable. It returns a map object as a lazy iterable, meaning that the values of the mapping are only produced when the map object is consumed.

```
>>> numbers = [1, 2, 3, 4, 5]
>>> # the squares are not calculated
>>> # when the map() is created
>>> squared_numbers = map(lambda x: x**2, numbers)
>>> # the squares are only calculated once
>>> # the squared_numbers iterable is consumed,
>>> # in this case during the construction of a list.
>>> list(squared_numbers)
[1, 4, 9, 16, 25]
```

`filter()`

The filter() function takes a function `f` and an iterable, and returns a filter object. This filter object is a lazy iterable, only yielding values as it is consumed. The values it yields are all items `i` of the iterable where the application of the function `f(i)` returns a truthy value.

```
>>> numbers = [1, 2, 3, 4, 5]
>>> even_numbers = filter(lambda x: x%2==0, numbers)
>>> list(even_numbers)
[2, 4]
>>> numbers = (-1, 0, 1)
>>> # 0 is falsy, so filter will drop it
>>> # and keep both -1 and 1
>>> list(filter(lambda x: x, numbers))
[-1, 1]
```

`input()` and `print()`

The `input()` and `print()` functions are are used to read input from the user and print output to the console, respectively.

The `input()` function reads a line of text from the standard input (usually the keyboard) and returns it as a string. A prompt can be passed as an optional argument that will be displayed to the end user.

```
>>> name = input("What is your name? ")
What is your name? TJ
>>> name
'TJ'
```

Conversely, the `print()` function writes a string to the standard output (usually the console). The `print()` function takes one or more objects to be printed, separated by commas, with an optional separator between objects, which defaults to a space. The function also takes an optional `end` parameter, which is appended after the last object, which defaults to a newline character, an an optional `file` argument, which is the file to write to, by default sys.stdout, and a `flush` argument, which defaults to False, which specifies whether to flush the buffer or not on printing to the file.

```
print(
    *objects,
    sep=' ',
    end='\n',
    file=sys.stdout,
    flush=False
)
```

open()

The `open()` function takes the name of the file or a file-like object as an argument and opens it in a specified mode. Different modes can be selected by passing a second `mode` argument, or the `mode` keyword. Modes include `r` for read mode (default), `w` for write mode, `a` for append mode, `x` for exclusive creation mode, `b` for binary mode, `t` for text mode (which is the default), and `+` for both reading and writing. Modes which don't conflict can be used in concert.

```
>>> file = open("./file.txt", mode="wb")
>>> file.write(b"Hello!\n")
>>> file.close()
```

The `open()` function talk a number of other optional arguments.

- `buffering` - the buffering policy for the file. The default is to use the system default buffering policy (`-1`). A value of `1` configures the file object to buffer per-line in text mode. A value greater than `1` configures a buffer size, in bytes, for the file object to use. Finally, a value of `0` switches buffering off (though this option is only available with binary mode).

- `encoding` - the encoding to be used for the file. The default is `None`, which sets the file object to use the default encoding for the platform. Can be any string which python recognizes as a valid codec.
- `errors` - the error handling policy for the file. The default is `None`, which means that errors will be handled in the default way for the platform.
- `newline` - configures the file object to anticipate a specific newline character. The default is `None`, which means that universal newlines mode is disabled.
- `closefd` - whether the file descriptor should be closed when the file is closed. The default is True.
- `opener` - a custom opener for opening the file. The default is None, which means that the built-in opener will be used.

range()

The `range()` function is used to generate an iterable for yielding a sequence of numbers. The function takes up to three arguments: If only one value is provided, it yields a sequence from `[0, n)`, where `n` is the stop value passed to `range`. If two arguments are provided, the range is `[a, b)`, where `a` is the start value and `b` is the stop value. If a third argument is provided, it acts as a step value, incrementing the count per-iteration by the value of the third `step` argument.

```
>>> tuple(range(2))
(0, 1)
>>> list(range(1, 3))
[1, 2]
>>> for i in range(start=2, stop=8, step=2):
>>>     print(i)
2
4
6
```

sorted()

The `sorted()` function is used to produce a sorted list of elements. It takes an argument for the sequence of elements to be sorted, an optional argument for a function used to extract a key from each element for sorting and an optional argument, a boolean indicating whether the elements should be sorted in descending order, which defaults to False. It returns a new list, instead of mutating the original.

```
>>> numbers = ["3", "2", "1", "4", "5"]
>>> sorted_numbers = sorted(
...     numbers,
...     key=lambda x: int(x),
...     reverse=True
... )
>>> print(sorted_numbers)
['5', '4', '3', '2', '1']
>>> numbers
['3', '2', '1', '4', '5']
```

`reversed()`

The `reversed()` function is used to create an iterator which iterates over a collection of items in reverse order.

```
>>> numbers = [1, 2, 3, 4, 5]
>>> reversed_numbers = list(reversed(numbers))
>>> print(reversed_numbers)
[5, 4, 3, 2, 1]
```

`zip()`

The `zip()` function creates an iterator which consumes iterables. The first value yielded by the iterator is a tuple containing the first item yielded by each iterable provided. The second item yielded is a tuple containing the second item yielded by each iterable. This process of yielding tuples continues until one of the iterables provided to the iterator is exhausted. An optional `strict` keyword can be provided to `zip()` which will cause the iterator to raise a `ValueError` should one iterable be exhausted before the others.

```python
>>> a, b = (1,2,3), (4,5,6)
>>> for items in zip(a, b):
>>>     print(items)
(1, 4)
(2, 5)
(3, 6)
>>> list(zip([1, 2], [3, 4, 5]))
[(1, 3), (2, 4)]
>>> list(zip([1, 2], [3, 4, 5], strict=True))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is longer than argument 1
```

Chapter 9. The Python Data Model

Underlying infrastructure that defines how Python objects interact with one another is referred to as the Python Data Model. It is the set of rules and conventions that govern how Python objects can be created, manipulated, and used.

The functionality of the data model is implemented through the use of special methods, also known as “dunder methods” or “magic methods”, which can be used to hook into specific behaviors provided to objects by the python interpreter. These methods have a set syntax, where they both start and end with double underscores, such as `__init__` and `__str__`. They are called automatically by the Python interpreter whenever specific conditions are met, such as when an object is created or when it is used in a specific context.

By leveraging the Python data model, we can create rich interactions between our Python objects. For example, we can define custom behavior for mathematical operations such as addition and subtraction, or we can define how our objects should be represented as a string. This functionality allows us to create objects which behave in a natural and intuitive way.

Object Creation Using `__new__` and `__init__`

`__new__` and `__init__` are two special methods in Python that are used in the process of creating and initializing new objects.

`__new__` is a method that is called when a new object is to be constructed. It is responsible for creating a new instance of the class. It takes the class as its first argument, and any additional arguments passed to the class constructor are passed into the `__new__` method. The new method is then responsible for creating an instance of a class, and returning that object to the interpreter.

Once an instance of an object has been created by `__new__`, it is then passed to the `__init__` method, along with the provided initializing values as positional and keyword arguments. `__init__` is then responsible for initializing the state of the new object, and to perform any other necessary setup.

In most cases, the `__new__` method is not necessary to be implemented by the developer. Most of the time it is sufficient to use the base implementation, which creates a new instance of the class, and then calls `__init__` on the instance, passing the instance as `self` and further passing in all the provided arguments.

Singletons

To demonstrate the usage of `__new__` and `__init__`, let's consider the singleton pattern. A singleton is a design pattern that ensures that a class can only have one instance, while providing a global access point to that instance. This pattern can be implemented in Python by using the `__new__` method.

The basic idea is to override the `__new__` method in the singleton class so that it only creates a new instance if one does not already exist. If an instance already exists, the `__new__` method simply returns that instance, instead of creating a new one.

```
>>> class Singleton:
...     _instance = None
...
...     def __new__(cls, *args, **kwargs):
...         if cls._instance is None:
...             cls._instance = object.__new__(cls)
...         return cls._instance
...
...     def __init__(self, my_value):
...         self.my_value = my_value
...
>>> my_object = Singleton(1)
>>> my_object.my_value
1
other_object = Singleton(2)
```

```
other_object.my_value
2
my_object.my_value
2
```

The function signature of `__new__` uses a common `*args, **kwargs` motif, which leverages tuple and dictionary packing to package the function arguments and keyword arguments into a tuple/dictionary pair without explicitly naming which arguments and keyword arguments a function or method expects.

In this example, the `__new__` method first checks if an instance of the class already exists on the class attribute `cls._instance`. If it does not exist, the method creates a new instance using the `__new__` method on `object` as its constructor, and it assigns the new instance to the `_instance` class variable. If an instance already exists on `cls._instance`, the `__new__` method simply returns the existing instance. This ensures that the class can only ever have one instance.

Rich Comparisons

In Python, rich comparison methods are special methods that allow you to define custom behavior for comparison operators, such as <, >, ==, `!=`, `<=`, and >=.

The rich comparison methods are:

- `def __lt__(self, other):` - Implements the < operator
- `def __le__(self, other):` - Implements the <= operator
- `def __eq__(self, other):` - Implements the == operator
- `def __ne__(self, other):` - Implements the `!=` operator
- `def __gt__(self, other):` - Implements the > operator
- `def __ge__(self, other):` - Implements the >= operator

Let’s take a look at an example implementation:

```python
class Money:
    def __init__(self, amount: int, currency: str):
        self.amount = amount
        self.currency = currency

    def __eq__(self, other):
        return (
            self.amount == other.amount
            and self.currency == other.currency
        )

    def __lt__(self, other):
        if self.currency != other.currency:
            raise ValueError(
                "Can't compare money of differing currencies."
            )
        return self.amount < other.amount

    def __le__(self, other):
        if self.currency != other.currency:
            raise ValueError(
                "Can't compare money of differing currencies."
            )
        return self.amount <= other.amount
```

In this example, the `Money` class has two attributes: amount, which is the monetary value, and currency, which is the currency type. The `__eq__` method compares the amount and currency of two Money objects, and returns True if they are the same, and False otherwise. The `__lt__` and `__le__` methods compare the `amount` value between two `Money` objects with the same currency, and raises an error if they have different currencies.

You are typically only required to define half of the rich comparison methods of any given object, as Python can infer the inverse value if a requested operation is not defined.

With these methods defined, a Money object can be used in comparison operations such as <, >, ==, `!=`, <=, and >= in a natural and intuitive way. For example, `Money(10, "USD") < Money(20, "USD")` will return `True`, and `Money(10, "USD") == Money(10, "EUR")` will return False.

Operator Overloading

Operator overloading refers to the ability to define custom behavior for operators such as `+`, `-`, `*`, `/`, etc. when they are used with objects of a certain class. This is achieved by specific special methods, shown below.

- `def __add__(self, other):` - Implements the `+` operator.
- `def __sub__(self, other):` - Implements the `-` operator.
- `def __mul__(self, other):` - Implements the `*` operator.
- `def __truediv__(self, other):` - Implements the `/` operator.
- `def __floordiv__(self, other):` - Implements the `//` operator.
- `def __mod__(self, other):` - Implements the `%` operator.
- `def __pow__(self, other):` - Implements the `**` operator.
- `def __and__(self, other):` - Implements the `&` operator.
- `def __or__(self, other):` - Implements the `|` operator.
- `def __xor__(self, other):` - Implements the `^` operator.
- `def __lshift__(self, other):` - Implements the << operator.
- `def __rshift__(self, other):` - Implements the >> operator.

By using operator overloading, we can create classes that have natural and intuitive behavior when used as operands.

```
>>> class Money:
...     def __init__(self, amount: int, currency: str):
...         self.amount = amount
...         self.currency = currency
...
...     def __add__(self, other):
...         if self.currency != other.currency:
...             raise ValueError(
...                 "Can't add money of differing currencies"
...             )
...         return Money(
...             self.amount + other.amount,
...             self.currency
...         )
...
...     def __sub__(self, other):
...         if self.currency != other.currency:
...             raise ValueError(
...                 "Can't subtract money of differing currencies"
...             )
...         return Money(
```

```
...             self.amount - other.amount,
...             self.currency
...         )
...
...     def __mul__(self, other):
...         if not isinstance(other, int):
...             raise ValueError(
...                 "Can't multiply by non-int value"
...             )
...         return Money(self.amount * other, self.currency)
...
...     def __truediv__(self, other):
...         if not isinstance(other, int):
...             raise ValueError(
...                 "Can't divide by non-int value"
...             )
...         # divmod() is a builtin which does
...         # // and % at the same time
...         quotient, remainder = divmod(self.amount, other)
...         return (
...             Money(quotient, self.currency),
...             Money(remainder, self.currency)
...         )
...
```

In this example, the `Money` class has two attributes: `amount`, which is the monetary value, and `currency`, which is the currency type.

The `__add__` method overloads the `+` operator, allowing you to add two `Money` objects. It also check to make sure that the operands are of the same currency, otherwise it raises a `ValueError`.

The `__sub__` method overloads the `-` operator, allowing to subtract two `Money` objects. It also check to make sure that the operands are of the same currency, otherwise it raises a `ValueError`.

The `__mul__` method overloads the `*` operator, allowing to multiply a Money object by an int. If the value is not an `int`, it raises a `ValueError`.

The `__truediv__` method overloads the `/` operator, allowing to divide a `Money` object by an int. It returns the quotient and the remainder in the form of new `Money` objects in the same currency.

```
>>> my_money = Money(100, "USD")
>>> my_money += Money(50, "USD")
>>> my_money.amount
150
>>> q, r = my_money / 4
>>> q.amount
37
>>> r.amount
2
```

String Representations

Humans communicate using text. As such, utilizing the special methods which render Python objects in string format such as `__str__` and `__repr__`, makes it much easier for humans to understand the value of an object and its purpose. These methods allow us to define a user-friendly and unambiguous string representation of the object, making it more intuitive to interact with.

```
>>> class Money:
...     def __init__(self, amount: int, currency: str):
...         self.amount = amount
...         self.currency = currency
...
...     def __repr__(self):
...         return f"Money({self.currency} {str(self)})"
...
...     def __str__(self):
...         return f"${round(self.amount/100, 2)}"
...
...
>>> Money(1200, "USD")
Money(USD $12.05)
>>> str(Money(1200, "USD"))
'$12.0'
```

Emulating Containers

Container types such as lists, tuples, and dictionaries have built-in behavior for certain operations, such as the ability to iterate over elements, check if an element is in their collection, and retrieve the length of the container. We can emulate this behavior using special methods.

The [] operator can be overloaded using the following methods:

- `def __getitem__(self, key):` - This method is called when the `[]` operator is used to retrieve an item from the container. The key parameter represents the index or key of the item being retrieved. This method should return the item at the specified key or raise an `IndexError` if the key is not found.
- `def __setitem__(self, key, value):` - This method is called when the `[]` operator is used to set an item in the container. The `key` parameter represents the index or key of the item being set, and the value parameter represents the new value of the item. This method should set the item at the specified key to the specified value.
- `def __delitem__(self, key):` - This method is called when the `del` statement is used to delete an item from the container. The key parameter represents the index or key of the item being deleted. This method should remove the item at the specified key.

In addition, the `in` operator can be overloaded using the following:

- `def __contains__(self, item):` - This method allows a class to define its own way of checking if an item is contained in the container. This method should return a Boolean indicating whether the item is contained in the container.

Finally, the `len()` function can be overloaded using the following:

- `def __len__(self):` - This method is called when the function `len()` is called on a container. It should return an integer representing the length of the object.

By using these methods, we can create classes that have similar behavior to built-in container types, making our code more effecient and Pythonic.

```
>>> class MutableString:
...     def __init__(self, text: str):
...         self._text = list(text)
...
...     def __getitem__(self, idx):
```

```
...         return self._text[idx]
...
...     def __setitem__(self, idx, value):
...         self._text[idx] = value
...
...     def __delitem__(self, idx):
...         del self._text[idx]
...
...     def __str__(self):
...         return "".join(self._text)
...
...     def __len__(self):
...         return len(self._text)
...
>>> my_str = MutableString("fizzbuzz")
>>> my_str[0] = "F"
>>> str(my_str)
Fizzbuzz
>>> len(my_str)
8
```

In this example, the `MutableString` class has a single attribute `_text`, which is a list of characters representing the string. With the `__getitem__`, `__setitem__` and `__delitem__` methods defined, a `MutableString` object can be used to get, set, and delete characters in the string, allowing for string manipulation in a similar way to a list of characters. The `__str__` method then allows a user to compile the `MutableString` object into a python string.

Emulating Functions

A class can emulate a function by defining a `__call__` method. The `__call__` method is itself called when an instance of the class is called using parentheses (i.e., `obj()`).

```
>>> class MyFunction:
...     def __call__(self, x, y):
...         return x + y
...
>>> my_function = MyFunction()
>>> my_function(1, 2)
3
```

Using Slots

`__slots__` is a special class attribute that you can define to reserve a fixed amount of memory for each instance of the class. It is used to optimize memory usage for classes that have a large number of instances, and that do not need to add new attributes dynamically.

When you define `__slots__` in a class, it creates a fixed-size array for each instance to store the attributes defined in `__slots__`. This array is much smaller than the dictionary that is used by default to store instance attributes. The result is that `__slots__` objects save a considerable amount of memory, particularly if you have many instances of the class. Slots also have the benefit of faster attribute access.

```
>>> class SlotsClass:
...     __slots__ = ['x', 'y']
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> obj = SlotsClass(1, 2)
>>> print(obj.x, obj.y)
1 2
```

Customizing Attribute Access

In Python, we can customize attribute interactions using special methods such as `__getattribute__`, `__getattr__`, `__setattr__`, and `__delattr__`. These methods allow us to control how an object's attributes are accessed, set, and deleted.

There are two special methods for accessing the attributes of a given object. The `__getattribute__` method is the first attribute accessor to be called when an attribute is accessed using the dot notation (e.g., `obj.attribute`) or when using the `getattr()` built-in function. It takes the attribute name as its parameter and should return the value of the attribute. The `__getattr__` method is called only when an attribute is not found by `__getattribute__`, i.e. when `__getattribute__` raises an `AttributeError`. It takes the attribute name as its parameter and should return the value of the attribute.

The benefit to this dual implementation is that you can get other known attributes on the instance inside `__getattr__` using dot notation, without running the risk of recursion errors, while still hooking into the accessor protocol before an object attribute is returned.

```
>>> class ADTRecursion:
...     def __init__(self, **kwargs):
...         self._data = kwargs
...
...     def __getattribute__(self, key):
...         return self._data.get(key)
...
>>> class AbstractDataType:
...     def __init__(self, **kwargs):
...         self._data = kwargs
...
...     def __getattr__(self, key):
...         return self._data.get(key)
...
>>> this = ADTRecursion(my_value=1)
>>> this.my_value
    File "<stdin>", line 12, in __getattribute__
    File "<stdin>", line 12, in __getattribute__
    File "<stdin>", line 12, in __getattribute__
    [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
>>> that = AbstractDataType(my_value=1)
>>> that.my_value
1
```

If it is necessary to override the default `__getattribute__` method on a class, you can use the `__getattribute__` method of the `object` class in order to avoid the issue of infinite recursion.

```
>>> class ObjectGetAttr:
...     def __init__(self, **kwargs):
...         self._data = kwargs
...
...     def __getattribute__(self, key):
...         _data = object.__getattribute__(self, "_data")
...         return _data.get(key)
...
```

```
>>> this = ObjectGetAttr(my_value=1)
>>> this.my_value
1
```

The `__setattr__` method is called when an attribute is set using the dot notation (e.g., `obj.attribute = value`) or when using the `setattr()` built-in function. It takes the attribute name and value as its parameters and should set the attribute to the specified value.

The `__delattr__` method is called when an attribute is deleted using the `del` statement (e.g., `del obj.attribute`) or when using the `delattr()` built-in function. It takes the attribute name as its parameter and should delete the attribute.

Iterators

The iterator protocol is a set of methods that allow Python objects to define their own iteration behavior. The protocol consists of two methods: `__iter__` and `__next__`.

The `__iter__` method is used to create an iterator object. It is called when the `iter()` built-in function is used on an object or when a for loop is used to iterate over an object. It should return an iterator object that defines a `__next__` method.

The `__next__` method is used to retrieve the next item from the iterator. It is called automatically by the for loop or when the `next()` built-in function is used on the iterator. It should return the next item or raise a `StopIteration` exception when there are no more items.

```
>>> class CountByTwos:
...     def __init__(self, start, stop):
...         self.internal_value = start
...         self.stop = stop
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
```

```
...         snapshot = self.internal_value
...         if snapshot >= self.stop:
...             raise StopIteration
...         self.internal_value += 2
...         return snapshot
...
>>> _iter = CountByTwos(start=5, stop=13)
>>> next(_iter)
5
>>> next(_iter)
7
>>> for i in _iter: # takes the partially consumed iterator and e\
xhausts it.
...     print(i)
9
11
```

In this example, the `CountByTwos` class has an `__iter__` method that returns `self`, and a `__next__` method that lazily generates the next number in the sequence. Calling the `iter()` function on our `_iter` value simply returns `self`, as the instance already conforms to the iterator protocol. When `next()` is called, either manually or in the context of a `for` loop, the iterator yields the next number in the sequence. The class stops iterating when the internal value reaches the stop value, as at this point the `StopIteration` exception is raised.

Lazy Evaluation

Since an iterator only generates the next value when it is requested, it avoids the need to generate and store all the values at once. This is particularly useful when working with large datasets that do not fit in memory, as it allows the program to process the data one piece at a time without having to load the entire dataset into memory. This sort of lazy evaluation also allows the developer to avoid unnecessary computation. For example, if the developer is looking for a specific value in an iterator, the program can stop generating new values as soon as the value is found, rather than generating all the remaining values.

A particular example where this might be useful is in the context of infinite sequences. For example, lets consider a modified version of `CountByTwos` which has no internal stopping mechanism.

```
>>> class CountByTwos:
...     def __init__(self, start):
...         self.internal_value = start
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         snapshot = self.internal_value
...         self.internal_value += 2
...         return snapshot
...
```

This iterator produces an infinite sequence of values, starting at the `start` value. If this were greedily consumed, the program would hang. But since iterators evaluate lazily, we can use this structure in contexts where iteration can be stopped, by either a `break` or a `return`.

```
>>> for i in CountByTwos(5):
...     if i >= 9:
...         break
...     print(i)
...
5
7
>>> i
9
>>> _iter = CountByTwos(0)
>>> while True:
...     val = next(_iter)
...     if val >= 6:
...         break
...     print(val)
...
0
2
4
```

Context Managers

Context Managers provide a clean and convenient method for managing resources that need to be acquired and released. The `with` statement ensures that resources are acquired before the `with` block is executed, and released after the block of code is exited, even if an exception is raised.

To hook into the context manager protocol, an object should define the special methods `__enter__` and `__exit__`. The `__enter__` method is called when the context is entered, and can be used to acquire the resources needed by the block of code. It can return an object that will be used as the context variable in the `as` clause of the `with` statement. The `__exit__` method is called when the context is exited, and it can be used to release the resources acquired by the `__enter__` method. It takes three arguments: an exception type, an exception value, and a traceback object. The `__exit__` method can use these arguments to perform cleanup actions, suppress exceptions, or log errors. If you don't plan on using the exception values in the object, they can be ignored.

```
>>> class ContextManager(object):
...     def __enter__(self):
...         print('entering!')
...         return self
...
...     def __exit__(self, *args, **kwargs):
...         print('exiting!')
...
...     def print(self):
...         print('in context')
...
>>> with ContextManager() as cm:
...     cm.print()
...
entering!
in context
exiting!

>>> with ContextManager():
...     raise ValueError
entering!
exiting!
Traceback (most recent call last):
    File "<stdin>", line 7, in <module>
    raise ValueError
ValueError
```

In this example, when the with statement is executed, an instance of `ContextManager()` is created, and the `__enter__` method is called, printing “entering!”. The `__enter__` object returns the instance `self` which is assigned to the variable `cm`. The block of code inside the `with` statement is then executed, where the `print()` method of the class is called and it prints “in context”. Finally, the `__exit__` method is called, printing “exiting!”.

Next, The `__enter__` method of the `ContextManager()` class is called, printing the “entering!” message to indicate that the context has been entered. Then, the block of code inside the `with` statement is executed, where a `ValueError` exception is raised. Since an exception is raised within the `with` block, the interpreter leaves the block, and the `__exit__` method is called, passing the `ValueError` exception, its value and its traceback as arguments. The `__exit__` method simply prints “exiting!” and the `ValueError` exception propagates up to the next level of the call stack, where it can be handled by an enclosing exception handler.

A single `with` statement can execute multiple context managers in concert:

```
>>> class ContextManager(object):
...     def __init__(self, val):
...         self.val = val
...
...     def __enter__(self):
...         print(f'entering {self.val}!')
...         return self
...
...     def __exit__(self, *args, **kwargs):
...         print(f'exiting! {self.val}')
...
...     def print(self):
...         print(f'in context of {self.val}')

>>> with (
...     ContextManager(1) as cm1,
...     ContextManager(2) as cm2,
... ):
...     cm1.print()
...     cm2.print()
...
entering 1!
entering 2!
in context of 1
```

```
in context of 2
exiting! 2
exiting! 1
```

Descriptors

Descriptors are objects that define one or more of the special methods `__get__`, `__set__`, and `__delete__`. These methods are used to customize the behavior of attribute access, such as getting, setting and deleting attributes respectively.

A descriptor can be a class or an instance of a class that defines one or more of these special methods. A class or an instance that defines a descriptor is called a descriptor class or descriptor object. Descriptors are then assigned as class attributes, and act on objects per-instance.

The `__get__(self, instance, owner)` method is called when an attribute is accessed using the dot notation (e.g., `instance.attribute`) or using the built-in `getattr()` function. It takes two arguments: the instance that the descriptor is an attribute of, and the owner class which defines the instance. The `__get__` method should return the value of the attribute.

The `__set__(self, instance, value)` method is called when an attribute is set using the dot notation (e.g., `instance.attribute = value`) or using the built-in `setattr()` function. It takes two arguments: the instance that the descriptor is an attribute of and the value to set. The `__set__` method should set the attribute to the specified value.

The `__delete__(self, instance)` method is called when an attribute is deleted using the `del` statement (e.g., `del instance.attribute`) or using the built-in `delattr()` function. It takes one argument: the instance that the descriptor is an attribute of. The `__delete__` method should delete the attribute.

Descriptors can be used to define attributes that have custom behavior, such as computed properties, read-only properties, or properties that enforce constraints. For example, we can write a descriptor which requires attributes to be non-falsy.

```
>>> class MyDescriptor:
...     def __get__(self, instance, owner):
...         return getattr(instance, "_my_attr", None)
...
...     def __set__(self, instance, value):
...         if not value:
...             raise AttributeError(
...                 "attribute must not be falsy"
...             )
...         setattr(instance, "_my_attr", value)
...
...     def __delete__(self, instance):
...         delattr(instance, "_my_attr")
...
>>> class MyClass:
...     desc = MyDescriptor()

>>> my_object = MyClass()
>>> my_object.desc
>>> my_object.desc = 1
>>> my_object.desc
1
>>> my_object.desc = 0
Traceback (most recent call last):
  File "<stdin>", line 14, in <module>
AttributeError: attribute must not be falsy
del my_object.desc
```

Python provides a builtin descriptor called `property`. A property descriptor is defined using the `property()` built-in constructor and it can take several arguments, including fget, fset, and fdel, as functions.

```
>>> class MyClass:
...     my_value = property(
...         fget=lambda self: getattr(self, "_value", None),
...         fset=lambda self, val: setattr(self, "_value", val),
...         fdel=lambda self: delattr(self, "_value")
...     )
...
>>> my_object = MyClass()
>>> my_object.my_value = 5
>>> my_object.my_value
5
```

While vaild, the isn't the typical use case of the property descriptor. You will most commonly see the `property` descriptor evoked as a decorator (we'll talk more about decorators later).

```
>>> class MyClass:
...     @property
...     def my_value(self):
...         return getattr(self, "_my_value", None)
...
...     @my_value.setter
...     def my_value(self, val):
...         self._my_value = val
...
...     @my_value.deleter
...     def my_value(self):
...         del self._my_value
...
>>> my_object = MyClass()
>>> my_object.my_value = 5
>>> my_object.my_value
5
```

Chapter 10. Concepts in Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that organizes code into objects. Through features such as polymorphism, encapsulation, and inheritance, we can create objects which have consistent behaviors, hide away implementation details, and can be easily reused and extended. The ultimate goal of object-oriented programming is the abstraction of state, and Python provides developers with the tools necessary to do so.

Inheritance

Inheritance is a feature of object-oriented programming that allows a new class to inherit the properties and methods of an existing class. The new class is called the derived class or sub class, and the existing class is called the base class or super class.

In Python, a class can inherit from another class using parentheses.

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

    def print_name(self):
        print(self.name)


class SubClass(SuperClass):
    pass
```

Here, the `SubClass` inherits from the `SuperClass`, and automatically has access to all the properties and methods defined in the `SuperClass`.

A derived class can override or extend the methods of the super class by redefining them in the derived class.

```python
>>> class SuperClass:
...     def __init__(self, name):
...         self.name = name
...
...     def print_name(self):
...         print(self.name)
...
... class SubClass(SuperClass):
...     def print_name(self):
...         print(self.name.upper())
...
```

Here, the `SubClass` overrides the `print_name` method of the `SuperClass`, to print the name in uppercase.

Calling the Super Class using `super()`

The `super()` function is a built-in function in Python that allows a derived class to call methods from its super class. It is often used when a derived class wants to extend or override the functionality of a method defined in its super class.

For example, let's say we have a class `SuperClass` with a method `print_name` and a class `SubClass` that inherits from `SuperClass` and wants to extend the functionality of the `print_name` method.

```python
>>> class SuperClass:
...     def __init__(self, name):
...         self.name = name
...
...     def print_name(self):
...         print(self.name)
...
>>> class SubClass(SuperClass):
...     def print_name(self):
...         # call the print_name() fn
...         # of the parent class
...         super().print_name()
...         print("child class")
...
```

Here, the `SubClass` overrides the `print_name` method of the `SuperClass`, but it also wants to call the `print_name` method of the parent class to keep its original functionality. The `super().print_name()` call in the `SubClass` will call the `print_name` method of the `SuperClass`.

Multiple Inheritance and Method Resolution Order

In Python, a class can inherit from multiple classes by listing them in the class definition, separated by commas. For example:

```
>>> class SubClass(SuperClass1, SuperClass2):
...     pass
```

Here, the `SubClass` inherits from both `SuperClass1` and `SuperClass2`.

When a class inherits from multiple classes, it can potentially have multiple versions of the same method or property. This is known as the diamond problem, and can lead to ambiguity about which version of the method or property to use.

To resolve this ambiguity, Python uses a method resolution order (MRO) to determine the order in which the classes are searched for a method or property. The MRO used in Python is the C3 linearization algorithm, which creates a linearization of the class hierarchy that is guaranteed to be consistent and predictable.

C3 guarentees the following:

- subclasses take precedent over superclasses
- order of preference for multiple superclasses is left to right
- a class only appears once in MRO

```
>>> class A:
...     def method(self):
...         print("A")
...
... class B(A):
...     pass
```

```
...
... class C(A):
...     def method(self):
...         print("C")
...
... class D(B, C):
...     pass
...
```

Here, class `D` inherits from both `B` and `C` which both inherit from `A`. If we create an instance of `D` and call the method method, `C` is printed, because `C` is a direct superclass, where as `A` is a superclass once removed and `B` does not implement the method.

```
>>> D().method()
C
>>> D.__mro__
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

Encapsulation

Encapsulation is a feature of object-oriented programming that allows for the hiding of implementation details within an object. It is the practice of keeping the internal state of an object private, and providing a public interface for interacting with the object - separating the interaction with the implementation.

In Python, encapsulation is achieved through the use of "private" properties and methods, which are denoted by a single or double underscore prefix. It should be noted that this is only a convention, and data is never truly private in Python.

For example, a method or attribute with a single underscore prefix like `_attribute` is considered a private method or attribute, and should only be accessed by the class and its subclasses internally. An attribute or method with double underscore prefix like __attribute is also considered a private method or attribute, and should be accessed only within the class.

For attributes prefixed with double underscores, Python will “mangle” the attribute name so it is harder to intuit from outside the class. This causes the attribute name to be prefixed with “classname_”.

```
>>> class MyClass:
...     def __init__(self, value):
...         self.__mangled = value
...         self._private = value
...
>>> my_object = MyClass('string')
>>> dir(my_object)
['_MyClass__mangled',
 '__class__',
    ...
 '_private']
```

It should be noted that, unlike many other languages, Python doesn’t truly keep methods and attributes private. The underscore convention is merely a convention. Any user of the class can access these methods with impunity.

Polymorphism

Polymorphism is a feature of object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. This means that objects of different classes can be used interchangeably, as long as they implement the same methods or properties. This feature allows for the creation of flexible, reusable and extensible code.

Python is a dynamically-typed language, which allows for a more flexible approach to polymorphism, known as “duck typing”, which is based on the idea that “if it quacks like a duck, it’s a duck”.

In other words, the type of an object is irrelevant to the execution of your program, so long as you can operate on an object through an expected interface, your code will execute.

```
>>> class Mouse:
...     def speak(self):
...         return "Squeak"
...
... class Cat:
```

```
...     def speak(self):
...         return "Meow"
...
... class Dog:
...     def speak(self):
...         return "Woof"
...
>>> for animal in (Mouse(), Cat(), Dog()):
>>>     animal.speak()
Squeak
Meow
Woof
```

Chapter 11. Metaclasses

A metaclass in Python is a class that defines the behavior of other classes. Or, it is a class that is used to create other classes. All classes are themselves objects, and those objects are instances of `type`, the base metaclass. The `type` object is itself a class, which can be subclassed to create custom metaclasses.

When a class is defined, Python automatically creates an instance of the type class, and assigns it as the metaclass of the new class. This instance is used to create the class object, and to define its behavior. By default, the behavior of a class is defined by the methods of the type class, but a custom metaclass can be used to change this behavior.

The syntax for creating a custom metaclass is to define a new class that inherits from the type class. From there, we can overload the default methods of the type class to create custom hooks that run when a class is defined. This is typically done using the `__new__` method to define the behavior of class creation. The `__init__` method is also commonly overridden to define the behavior of the class initialization.

For example, the following code defines a custom metaclass that adds a greeting attribute to the class definition, and prints to the console when this is done:

```
>>> class MyMetaclass(type):
...     def __new__(cls, name, bases, attrs):
...         attrs["greeting"] = "Hello, World!"
...         print(f"creating the class: {name}")
...         return type.__new__(cls, name, bases, attrs)
...
>>> class MyClass(metaclass=MyMetaclass):
...     pass
...
creating the class: MyClass
>>> MyClass.greeting
Hello, World!
```

As we can see, the `print()` function is executed at the moment which MyClass is defined. We can also see that the class has a class attribute `greeting` which is the string “Hello World!”

Metaclasses allow you to hook into user code from library code by providing a way to customize the behavior of class creation. By defining a custom metaclass and setting it as the metaclass of a user-defined class, you can change the way that class is created and initialized, as well as add new attributes and methods to the class.

Lets consider a library that provides a custom metaclass for creating singletons. The metaclass could override the `__call__` method to ensure that only one instance of the class is ever created, and return that instance whenever the class is called.

```
>>> class SingletonMetaclass(type):
...     def __init__(cls, name, bases, attrs):
...         super().__init__(name, bases, attrs)
...         cls._instance = None
...
...     def __call__(cls, *args, **kwargs):
...         if cls._instance is None:
...             cls._instance = super().__call__(*args, **kwargs)
...         return cls._instance
...
>>> class BaseSingleton(metaclass=SingletonMetaclass):
...     pass
...
>>> class MySingleton(BaseSingleton):
...     pass
...
>>> id(MySingleton())
140486001221456
>>> id(MySingleton())
140486001221456
```

Metaclasses can also be used to enforce from the library constraints or expectations on user code. Consider another example where a library expects users to define certain methods on a derived classes. You can use a metaclass to catch type errors *at the class instantiation*, instead of later during runtime.

```
>>> class LibraryMetaclass(type):
...     def __new__(cls, name, bases, attrs):
...         if "my_method" not in attrs:
...             raise AttributeError(
...                 "derived class must define method my_method()"
...             )
...         return type.__new__(cls, name, bases, attrs)
...
>>> class BaseClass(metaclass=LibraryMetaclass):
...     my_method = None
...
>>> class DerivedClass(BaseClass):
...     def my_method(self):
...         pass
...
>>> class BadClass(BaseClass):
...     pass
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __new__
AttributeError: derived class must define method my_method()
```

Chapter 12. The Data Types, Revisited.

In this chapter, we're going to circle back and look more in depth as some of python's built-in data types. As previously discussed, the built-in data types are all classes which implement a series of attributes and methods.

In order to inspect these attributes and methods, we can use the built-in function `dir()` to see a list of all the names of attributes and methods that the object has. This can be a useful tool for exploring and understanding the functionality of a particular object or module in code. The `help()` function can also be used in order to show the official documentation for any method not covered here.

```
>>> help(str.count)
Help on method_descriptor:

count(...)
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences
    of substring sub in string S[start:end].  Optional
    arguments start and end are interpreted as in slice
    notation.
```

Numbers

The three major numeric types in Python are `int`, `float`, and `complex`. While each data type is uniquely distinct, they share some attributes so to make interoperability easier. For example, the `.conjugate()` method returns the complex conjugate of a complex number - for integers and floats, this is just the number itself. Furthermore, the `.real` and `.imag` attributes define the real and imaginary part of a given number. For floats and ints, the `.real` part is the number itself, and the `.imag` part is always zero.

Integers

The `int` type represents integers, or whole numbers. Integers can be positive, negative, or zero and have no decimal point. Python supports arbitrarily large integers, so there is no limit on the size of an integer value, unlike some other programming languages. Python only guarantees singletons for the integers -5 through 256, so when doing integer comparisons outside this range, be sure to use == instead of `is`.

Bits and Bytes

The Python `int` has several methods for convenience when it comes to bit and byte representations. The `int.bit_length()` method returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros. `int.bit_count()` returns the number of bits set to 1 in the binary representation of an integer. `int.from_bytes()` and `int.to_bytes()` convert integer values to and from a bytestring representation.

```
>>> int(5).bit_count()
2
>>> int(5).bit_length()
3
>>> int(5).to_bytes()
b'\x05'
>>> int.from_bytes(b'\x05')
5
```

Floats

The python floating point value, represented by the `float` data type, is a primitive data type that is used to represent decimal numbers. Floats are numbers with decimal points, such as `3.14` or `2.718`. There are also a few special float values, such as `float('inf')`, `float('-inf')`, and `float('nan')`, which are representations of infinity, negative infinity, and “Not a Number”, respectively.

Float Methods

The `float()` type defines a method `.is_integer()` which returns `True` if the float is directly convertible to an integer value, and `False` otherwise.

```
>>> float(5).is_integer()
True
```

`float.as_integer_ratio()` returns a numerator and denominator integer value whose ratio is exactly equal to the original float. The denominator is guaranteed to be positive. If the float is infinite, this operation raises an `OverflowError`; if it is NaN, this operation raises a `ValueError`.

```
>>> float(1.5).as_integer_ratio()
(3, 2)
```

Hex Values

The `float.fromhex()` classmethod and `float.hex()` method are built-in methods that are used to work with floats in a hexadecimal format.

```
>>> hex = float(3.4).hex()
>>> hex
'0x1.b333333333333p+1'
>>> float.fromhex(hex)
3.4
```

Complex Numbers

The complex number is a value represented by a real portion and an imaginary, in the form of `(x+yj)`. For complex numbers, the `complex.real` attribute returns the real portion of the number `x`, and the `complex.imag` attribute returns the imaginary portion of the number `yj`. The `conjugate()` method returns the complex conjugate of the complex number.

```
>>> (3-4j).conjugate()
(3+4j)
```

Strings

As stated previously, strings are a built-in data type used to represent sequences of characters. They are enclosed in either single quotes `'` or double quotes `"`, and can contain letters, numbers, and symbols. Strings are immutable, which means that once they are created, their values cannot be changed.

When operating on a method of the `str` class, the result of the method call returns a new string. This is important to remember in cases where you have multiple references to the initial string, and when you're doing comparisons between strings; be sure to use == instead of `is`.

Split and Join

The `str.split()` and `str.join()` methods are methods of the Python string class that convert strings to and from iterables. `str.split()` takes one optional argument, which is a delimiter, and returns a list of substrings that are separated by the delimiter. If no delimiter is provided, the method will split the string using whitespace. `str.join()` takes an iterable of strings as an argument and returns a new string that concatenates the elements of the iterable using the string on which the method is called as a separator.

```
>>> _str = "a b c d e f g"
>>> _list = _str.split()
>>> _list
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> " ".join(_list)
'a b c d e f g'
```

Search and Replace

The `str.index()`, `str.find()`, `str.rfind()`, and `str.replace()` methods allow you to search and replace substrings from a given string. The `str.index()`, `str.find()`, `str.rfind()` methods each take one argument, the substring, and they each return the index of the first occurrence of the string. The differences are that `str.index()` raises a `ValueError` if the string is not found, but the `-find()` methods simply return `-1`. Furthermore, `rfind()` searches in reverse order compared to the `find()` method.

`str.replace()` takes two required arguments, the first is the substring to search for, and the second is the substring to replace with. It returns a new string where all the occurrences of the first substring are replaced with the second substring. A third, optional `count` argument may be provided, which specifies the number of found instances of the substring which should be replaced.

```
>>> _str = "Hello World!"
>>> _str.index("World")
6
>>> _str.index("Hey!")
  File "<stdin>", line 3, in <module>
ValueError: substring not found
>>> _str.find("Hey!")
-1
>>> "Hello, World. World. World.".rfind("World")
21
>>> "Hello, World!".replace("World", "Universe")
Hello Universe!
```

Paddings

The `str.ljust()`, `str.lstrip()`, `str.rjust()`, `str.rstrip()`, `str.zfill()`, and `str.center()` methods allow you to add and remove padding from the lefthand and righthand sides of a Python string. The `-just()` and `center()` methods take two arguments, the first of which is the length of the final string, and the second is the fill character which is to be used to add characters to the resulting string. The `-strip()` methods do the opposite, instead of padding strings they strip the provided characters from the lefthand and righthand sides of the string. Finally, the `zfill()` method leftpads the string with zeros to a specified width.

```
>>> _str = "Hello World"
>>> _str.ljust(15, "-")
Hello World----
>>> _str.rjust(15, "-")
----Hello World
```

```
>>> _str = _str.center(19, "-")
>>> _str
----Hello World----
>>> _str.lstrip("-")
Hello World----
>>> _str.rstrip("-")
----Hello World
>>> "1".zfill(4)
'0001'
```

Formatting

The `str.format()` and `str.format_map()` methods are used to insert values into a string using placeholders. These methods are used to create formatted strings, which are useful for displaying data in a specific way or for creating string templates that can be reused.

`str.format()` takes any number of arguments, which are used to replace placeholders in the string. Placeholders are indicated by curly braces {} in the string, and the index of the argument determines which placeholder is replaced.

`str.format_map()` on the other hand takes a single argument, which is a dictionary. The dictionary contains keys that match the placeholders in the string, and values that are used to replace the placeholders. In this case the placeholders are curly braces `{}` which contain the name of the corresponding dictionary key.

```
>>> "My name is {} and I am {} years old".format("Jessica", 27)
'My name is Jessica and I am 31 years old'
>>> data = {"name": "Jessica", "age": 31}
>>> "My name is {name} and I am {age} years old".format_map(data)
'My name is Jessica and I am 31 years old'
```

Translating

The `str.translate()` and `str.maketrans()` methods are used to manipulate strings by replacing specific characters or groups of characters. `str.maketrans()` method can take a dictionary which is a mapping of individual characters (or their ordinal values) to a corresponding replacement string. It returns a translation table that can be used as an argument for `str.translate()` method.

```
>>> _str = "Hello World!"
>>> tbl = str.maketrans({"!": "?"})
>>> str.translate(_str, tbl)
Hello World?
```

Partitioning

The `str.partition()` and `str.rpartition()` methods are used to split a string into three parts: a part before a specified delimiter, the delimiter itself, and a part after the delimiter. `str.partition()` method takes one required argument, which is the delimiter to search for. It returns a tuple containing the part of the string before the delimiter, the delimiter itself, and the part of the string after the delimiter. If the delimiter is not found, the tuple contains the original string, followed by two empty strings. The `str.rpartition()` method is similar to `str.partition()` method but it returns the last occurrence of the delimiter in the string, instead of the first. If the delimiter is not found, the tuple contains two empty strings, followed by the original string.

```
>>> "Hello World!".partition(' ')
('Hello', ' ', 'World!')
>>> "Hello World World!".rpartition(' ')
('Hello World', ' ', 'World!')
```

Prefixes and Suffixes

The `str.endswith()` and `str.startswith()` methods are used to check if a string starts or ends with a specific substring. `str.endswith()` takes one required argument, which is the suffix to be checked, and returns `True` if the string ends with the specified suffix, and `False` otherwise. `str.startswith()` also takes one required argument, which is the prefix to be checked, and returns `True` if the string starts with the specified prefix, and `False` otherwise.

```
>>> _str = "Hello World!"
>>> _str.startswith("Hello!")
True
>>> _str.endswith("world!")
False
```

The `str.removeprefix()` and `str.removesuffix()` are used to remove a specific substring from the start or end of a string. `str.removeprefix()` takes one required argument, which is the prefix to be removed, and removes the prefix from the string if the string starts with the specified prefix, and returns the modified string. `str.removesuffix()` also takes one required argument, which is the suffix to be removed, and removes the suffix from the string if the string ends with the specified suffix and returns the modified string. They are non-failing, so they return a resultant string regardless of whether or not the substring was present.

```
>>> _str = "Hello, World!"
>>> _str.removeprefix("Hello, ")
"World!"
>>> _str.removesuffix("World!")
"Hello, "
```

Boolean Checks

A number of methods on the string class are checks to return `True` or `False` depending on if a certain condition is true. Those methods are as follows:

- `str.isalnum()` - Returns `True` if all characters in the string are alphanumeric, `False` otherwise.
- `str.isalpha()` - Returns `True` if all characters in the string are alphabetical, `False` otherwise.
- `str.isascii()` - Returns `True` if all characters in the string are ASCII, `False` otherwise.
- `str.isdecimal()` - Returns `True` if all characters in the string are decimal digits, `False` otherwise.
- `str.isdigit()` - Returns `True` if all characters in the string are digits, `False` otherwise.

- `str.isidentifier()` - Returns `True` if the string is a valid Python identifier, `False` otherwise.
- `str.islower()` - Returns `True` if all characters in the string are lowercase, `False` otherwise.
- `str.isnumeric()` - Returns `True` if all characters in the string are numeric, `False` otherwise.
- `str.isprintable()` - Returns `True` if all characters in the string are printable, `False` otherwise.
- `str.isspace()` - Returns `True` if all characters in the string are whitespace, `False` otherwise.
- `str.istitle()` - Returns `True` if the string is in title case, `False` otherwise.
- `str.isupper()` - Returns `True` if all characters in the string are uppercase, `False` otherwise.

Case Methods

A number of methods on the string class are methods for formatting strings based on the cases of individual characters. Those methods are as follows:

- `str.capitalize()` - Returns a copy of the string with its first character capitalized and the rest in lowercase.
- `str.casefold()` - Returns a version of the string suitable for caseless comparisons.
- `str.lower()` - Returns a copy of the string with all uppercase characters converted to lowercase.
- `str.swapcase()` - Returns a copy of the string with uppercase characters converted to lowercase and vice versa.
- `str.title()` - Returns a copy of the string in title case, where the first letter of each word is capitalized and the rest are lowercase.
- `str.upper()` - Returns a copy of the string with all lowercase characters converted to uppercase.

It should be noted that when sorting strings in a case-insensitive manner, it is recommended to use the `str.casefold()` method instead of using the `str.lower()` method. The `str.casefold()` method is a more powerful version of the `str.lower()` method that is specifically designed for case-insensitive string comparisons.

The `str.lower()` method simply converts all uppercase characters in a string to lowercase. However, this method may not work as intended for certain characters and character encodings. For example, the `str.lower()` method may not properly handle the conversion of uppercase characters in non-Latin scripts. On the other hand, the `str.casefold()` method performs a more aggressive case-folding that is suitable for case-insensitive string comparisons.

```
>>> sorted(["Massive", "Maß"])
['Massive', 'Maß']
>>> sorted(["Massive", "Maß"], key=lambda x: x.lower())
['Massive', 'Maß']
>>> sorted(["Massive", "Maß"], key=lambda x: x.casefold())
['Maß', 'Massive']
>>> 'Maß'.lower()
'Maß'
>>> 'Maß'.upper()
'MASS'
>>> 'Maß'.casefold()
'mass'
```

Encodings

The `str.encode()` method is used to convert a string to bytes using a specific encoding. Encoding is a process of converting a string of characters into a sequence of bytes, and it's necessary when working with text data that needs to be stored or transmitted in a specific format.

The `str.encode()` method takes one argument, which is the name of the encoding to be used, by default utf-8. Some of the other encodings are utf-16, ascii, and latin1. It returns a bytes object that contains the encoded string.

```
>>> "Hello, World!".encode("utf8")
b'Hello, World!'
```

Bytes

So far in this text we've been referring to bytes as related to strings. And this is with some justification; many of the methods for strings are available for bytestrings, and the syntax is largely the same, save for the `b''` prefix.

However, `str` and `bytes` are two distinct data types. A `str` is a sequence of Unicode characters; `bytes` however are fundamentally a sequence of integers, each of which is between 0 and 255. They represent binary data, and it's useful when working with data that needs to be stored or transmitted in a specific format. If the value of an individual byte is below 127, it may be represented using an ASCII character.

```
>>> "Maß".encode('utf8')
b'Ma\xc3\x9f'
```

There are a few distinct methods which are unique to the `bytes` type, and we'll discuss those here.

```
>>> [m for m in dir(bytes)
... if not any((m.startswith("__"), m in dir(str)))]
['decode', 'fromhex', 'hex']
```

Decoding

The `bytes.decode()` method is used to convert a bytestring to a string. The method takes one required argument, which is the character encoding of the bytestring.

```
>>> b = b'Hello World'
>>> b.decode('utf-8')
'Hello World'
```

Hex Values

The `bytes.fromhex()` and `bytes.hex()` methods are used to work with bytes in a hexadecimal format.

```
>>> b = b"Hello World"
>>> hex = b.hex()
>>> hex
'48656c6c6f20576f726c64'
>>> bytes.fromhex(hex)
b"Hello World"
```

Tuples

The Python tuple data type has limited functionality, given its nature as an immutable data type. The two methods it does define however are `tuple.count()` and `tuple.index()`.

The `.count()` method is used to count the number of occurrences of a specific element in a tuple. The method takes one required argument, which is the element to count, and returns the number of occurrences of that element in the tuple.

```
>>> t = (1, 2, 3, 2, 1)
>>> t.count(2)
2
```

The `.index()` method is used to find the index of the first occurrence of a specific element in a tuple. The method takes one required argument, which is the element to find, and returns the index of the first occurrence of that element in the tuple. If the element is not found in the tuple, the method raises a `ValueError`.

```
>>> t = (1, 2, 3, 2, 1)
>>> t.index(2)
1
```

Lists

The Python `list` type defines a suite a methods which are used to inspect and mutate the data structure.

Counting and Indexing

The `.count()` method is used to count the number of occurrences of a specific element in a list. The method takes one required argument, which is the element to count, and returns the number of occurrences of that element in the list.

```
>>> l = [1, 2, 3, 2, 1]
>>> l.count(2)
2
```

The `.index()` method is used to find the index of the first occurrence of a specific element in a list. The method takes one required argument, which is the element to find, and returns the index of the first occurrence of that element in the list. If the element is not found in the list, the method raises a `ValueError`.

```
>>> l = [1, 2, 3, 2, 1]
>>> l.index(2)
1
```

Copying

The copy method is used to create a shallow copy of a list. It does not take any arguments and it returns a new list that is a copy of the original list.

```
>>> l1 = [object()]
>>> l2 = l1.copy()
>>> l2
[<object at 0x7f910fec4630>]
>>> l1[0] is l2[0]
True
```

Mutations

The `.append()` method is used to add an element to the end of a list. The method takes one required argument, which is the element to add, and adds it to the end of the list.

```
>>> l = [0]
>>> l.append(1)
```

```
>>> l
[0, 1]
```

The `.extend()` method is similar to append, but it is used to add multiple elements to a list. The method takes one required argument, which is an iterable, and adds each element of the iterable to the list.

```
>>> l = [1, 2, 3]
>>> l.extend((4, 5, 6))
>>> l
[1, 2, 3, 4, 5, 6]
```

The `.insert()` method is used to insert an element at a specific position in a list. The method takes two required arguments: the first is the index where the element should be inserted, and the second is the element to insert. An insert operation does not replace the element at a given index, it simply shifts the list accordingly.

```
>>> l = [1, 2, 3, 4]
>>> l.insert(2, 5)
>>> l
[1, 2, 5, 3, 4]
```

The `.remove()` method is used to remove the first occurrence of a specific element from a list. The method takes one required argument, which is the element to remove, and removes the first occurrence of that element in the list. If the element is not found in the list, the method raises a `ValueError`.

```
>>> l = [1, 2, 3, 2, 1]
>>> l.remove(2)
>>> l
[1, 3, 2, 1]
```

The `.pop()` method is used to remove an element from a list and return it. The method takes one optional argument, which is the index of the element to remove. If no index is provided, the method removes and returns the last element of the list.

```
>>> l = [1, 2, 3, 4]
>>> l.pop(2)
3
>>> l
[1, 2, 4]
```

Finally, the `.clear()` method is used to remove all elements from a list. It does not take any arguments and removes all elements from the list.

```
>>> l = [1, 2, 3, 4]
>>> l.clear()
>>> l
[]
```

Orderings

The `.reverse()` method is used to reverse the order of elements in a list. The method is an in-place operation and returns `None`.

```
>>> l = [1, 2, 3, 4]
>>> l.reverse()
>>> l
[4, 3, 2, 1]
```

Finally, the `.sort()` method is used to sort the elements in a list. The method is an in-place operation and returns `None`. The method does not take any arguments but accepts a few optional arguments, such as `key` and `reverse`. The `key` argument specifies a function that is used to extract a comparison value from each element in the list. The `reverse` argument is a Boolean value that specifies whether the list should be sorted in a ascending or descending order. The `sort()` method sorts the elements in ascending order by default.

```
>>> l = [3, 2, 4, 1]
>>> l.sort()
>>> l
[1, 2, 3, 4]
```

Dictionaries

The Python `dict` type defines a suite a methods which are used to inspect and mutate the data structure.

Iter Methods

The `.keys()`, `.values()`, and `.items()` methods are the primary mechanism for iterating over a dictionary. `.keys()` returns an iterator which yields each of the keys contained in the dictionary. `.values()` returns an iterator which yields each of the values matched to a key in the dictionary. Finally, `.items()` yields each of the key-value pairs as a tuple.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> list(d.keys())
['a', 'b', 'c']
>>> list(d.values())
[1, 2, 3]
>>> list(d.items())
[('a', 1), ('b', 2), ('c', 3)]
```

Getter/Setter Methods

The `.get()` method is used to retrieve the value of a specific key in a dictionary. The method takes one required argument, which is the key to retrieve, and returns the value of that key. If the key is not found in the dictionary, the method returns `None`, or an optional default value passed as a second argument.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.get('b')
2
>>> d.get('d', -1)
-1
```

The `.setdefault()` method is used to insert a key-value pairs into a dictionary if the key is not already present. The method takes two required arguments, the first is the key to insert, and the second is the value to set for that key should the key not be present. The method then returns the value for the key.

```
>>> d = {}
>>> d.setdefault("a", []).append(1)
>>> d
{'a': [1]}
```

Mutations

The `.pop()` method is used to remove a specific key-value pair from a dictionary and return its value. The method takes one required argument, which is the key to remove, removes that key-value pair from the dictionary, and returns the value to the caller. If the key is not found in the dictionary, the method raises a `KeyError`.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.pop('b')
2
>>> d
{'a': 1, 'c': 3}
```

The `.popitem()` method is used to remove an arbitrary key-value pair from a dictionary and return it as a tuple. The method does not take any arguments.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.popitem()
('c', 3)
>>> d
{'a': 1, 'b': 2}
```

The `.clear()` method is used to remove all key-value pairs from a dictionary. It It does not take any arguments and removes all key-value pairs from the dictionary.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.clear()
>>> d
{}
```

The `.update()` method is used to add multiple key-value pairs to a dictionary. The method takes one required argument, which is a dictionary, and it adds each key-value pair of that dictionary to the original dictionary. Any naming conflicts favors the new value.

```
>>> d = {'a': 1, 'b': 2}
>>> d.update({'c': 3, 'd': 4})
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> d = dict(a=1, b=2)
>>> d.update({'a': 2})
>>> d
{'a': 2, 'b': 2}
```

Creating new Dictionaries

The `.copy()` method is used to create a shallow copy of a dictionary. It does not take any arguments and it returns a new dictionary that is a copy of the original dictionary.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d2 = d.copy()
>>> d2
{'a': 1, 'b': 2, 'c': 3}
```

Finally, the `.fromkeys()` method is used to create a new dictionary. The method takes one or two arguments; the first is an iterable of keys and the second is an optional value to set as the default value for each key. If a default value is not specified, it defaults to `None`.

```
>>> keys = ('a', 'b', 'c')
>>> d = dict.fromkeys(keys, 0)
>>> d
{'a': 0, 'b': 0, 'c': 0}
```

Sets

Sets are used to store unique elements. They are useful for operations such as membership testing, removing duplicates from a sequence and mathematical operations. They are mutable and have a variety of built-in methods to perform various set operations.

Mutations

The `.add()` method is used to add an element to a set. The method takes one required argument, which is the element to add, and adds it to the set.

```
>>> s = {1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

The `.remove()` method is used to remove a specific element from a set. The method takes one required argument, which is the element to remove. If the element is not present in the set, it raises a `KeyError`.

```
>>> s = {1, 2, 3}
>>> s.remove(2)
>>> s
{1, 3}
```

The `.discard()` method is used to remove an element from a set if it is present. The method takes one required argument, which is the element to remove. This method is similar to the `.remove()` method, but it does not raise an error if the element is not present in the set.

```
>>> s = {1, 2, 3}
>>> s.discard(2)
>>> s
{1, 3}
>>> s.discard(4)
>>> s
{1, 3}
```

The `.pop()` method is used to remove and return an arbitrary element from a set. The method does not take any arguments and removes and returns an arbitrary element from the set. If the set is empty, it raises a `KeyError`.

```
>>> s = {1, 2, 3}
>>> s.pop()
1
>>> s
{2, 3}
```

The `.update()` method is used to add multiple elements to a set. The method takes one or more iterable arguments and adds each element from those iterables to the set.

```
>>> s = {1, 2}
>>> s.update([2, 3], (4, 5))
>>> s
{1, 2, 3, 4, 5}
```

The `.clear()` method is used to remove all elements from a set. It does not take any arguments and removes all elements from the set.

```
>>> s = {1, 2, 3}
>>> s.clear()
>>> s
set()
```

Set Theory

Set theory is a branch of mathematics which models the relationship between objects, focusing on the different ways they can be organized into a collection. Python’s builtin `set` type provides a multitude of methods which allow you to compute relationships between sets.

The `.union()` method returns a new set that contains all elements from both the set and an iterable.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.union(s2)
{1, 2, 3, 4}
```

The `.intersection()` and `.intersection_update()` methods compute a new set that contains only the elements that are common to both the set and the iterable. Calling `.intersection()` returns a new set, and `.intersection_update()` does the operation in-place.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.intersection(s2)
{2, 3}
>>> s1.intersection_update(s2)
>>> s1
{2, 3}
```

The `.difference()` and `.difference_update()` methods compute a new set that contains all elements that are in the set but not in an iterable. Calling `.difference()` returns a new set, and `.difference_update()` does the operation in-place.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.difference(s2)
{1}
>>> s1.difference_update(s2)
>>> s1
```

The `.symmetric_difference()` and `.symmetric_difference_update()` methods compute a new set that contains elements that are in either the set or an iterable, but not in both. Calling `.symmetric_difference()` returns a new set, and `.symmetric_difference_update()` does the operation in-place.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.symmetric_difference(s2)
{1, 4}
>>> s1.symmetric_difference_update(s2)
>>> s1
{1, 4}
```

Boolean Checks

Several methods available for set objects allow you to check the relationship between a pair of sets.

The `.isdisjoint()` method returns `True` if two sets have no common elements.

```
>>> s1 = {1, 2, 3}
>>> s2 = {4, 5, 6}
>>> s1.isdisjoint(s2)
True
```

The `.issubset()` method returns `True` if all elements of a set are present in another set.

```
>>> s1 = {1, 2, 3}
>>> s2 = {1, 2, 3, 4, 5, 6}
>>> s1.issubset(s2)
True
```

The `.issuperset()` method returns `True` if a set contains all elements of another set.

```
>>> s1 = {1, 2, 3, 4, 5, 6}
>>> s2 = {1, 2, 3}
>>> s1.issuperset(s2)
True
```

Copying

Finally, the `.copy()` method allows you to create a shallow copy of a given set. It does not take any arguments and returns a new set that is a copy of the original set. The references in each set are the same.

```
>>> s1 = {"Hello World",}
>>> s2 = s1.copy()
>>> s1.pop() is s2.pop()
True
```

Chapter 13. Type Hints

Type hints, also known as type annotations, are a recent addition to Python which allow developers to add static type information for function arguments, return values, and variables. The philosophy behind type hints is to provide a way to improve code readability and catch certain types of errors earlier in the development cycle.

Type hints make it easier for other developers to understand the expected inputs and outputs of functions, without having to manually inspect the code or drop into a debugger during test. They also allow static type checkers, such as `mypy` or Language Server Protocols (LSP), to analyze code and detect type-related issues before the code is executed.

It's important to note that type hints are optional, and Python's dynamic typing is still maintained. Type hints do not affect runtime behavior and are only used for static analysis and documentation purposes. It should also be explicitly noted that, absent a mechanism for type checking, type hints are not guaranteed to be accurate by the interpreter. They are *hints* in the very sense of the word.

Incorporating Type Hints

One of the best places to start adding type hints is in existing functions. Functions typically have explicit inputs and outputs, so adding type definitions to I/O values is relatively straightforward. Once you have identified a function that can be type annotated, determine the types of inputs and outputs for that function. This should be done by inspecting the code and looking at the context in which each variable of the function is used. For example, if a function takes two integers as inputs and returns a single integer as output, you would know that the inputs should have type hints `int` and the return type hint should be `int` as well.

Type hints for functions go in the function signature. The syntax for these hints is `name: type[=default]` for function arguments, and `-> type:` for the return value.

```
def multiply(a: int, b: int) -> int:
    return a * b
```

In this example, the `multiply` function is given type annotations for the `a` and `b` variables. In this case, those types are specified as integers. Since the resulting type of multiplying two integers is always an integer, the return type of this function is also `int`.

Its worth reiterating that this `multiply` function will not fail at runtime if it is passed values which aren't integers. However, static checkers like mypy will fail if the function is called elsewhere in the code base with non-int values, and text editors with LSP support will indicate that you are using the function in error.

```
# ./script.py

def multiply(a: int, b: int) -> int:
    return a * b

print(multiply(1, 2.0))
```

```
root@77a3d1e5fa49:/code/hints# python script.py
2.0
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
script.py:6: error: Argument 2 to "multiply" has incompatible typ\
e "float"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
root@77a3d1e5fa49:/code/hints#
```

Union types

You can specify multiple possible types for a single argument or return value by using a union type. This is done either by using the `Union` type from the typing module, enclosing the possible types in square brackets, or by using the bitwise or operator `|` as of python 3.10.

```
# ./script.py

```

```
def multiply(a: int|float, b: int|float) -> int|float:
    return a * b

print(multiply(1, 2.0))
```

In this example, we've modified our `multiply()` function to type check against both integer types, as well as floats. Running a static type checker against this new function signature throws no errors.

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
Success: no issues found in 1 source file
root@77a3d1e5fa49:/code/hints#
```

Optional

The `Optional` type is a type hint which is used to indicate that an argument to a function or method is optional, meaning that it does not have to be provided in order for the function or method to be called. It is typically paired with a default value which is used when the function caller does not pass an explicit argument.

```
# ./script.py

from typing import Optional

def greet(name: str, title: Optional[str] = None) -> str:
    if title:
        return f"Hello, {title} {name}"
    return f"Hello, {name}"

print(greet("Justin", "Dr."))
print(greet("Cory"))
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
Success: no issues found in 1 source file
root@77a3d1e5fa49:/code/hints# python script.py
Hello, Dr. Justin
Hello, Cory
root@77a3d1e5fa49:/code/hints#
```

`type|None`

An optional argument is by definition a union between some type `t` and `None`. In addition to using `Optional` as an explicit type definition, it is possible to achieve the same type definition using the `|` operator as `t|None`. Both methods of type annotation serve the same purpose, but the choice of which to use often comes down to personal preference and the specific use case.

```python
# ./script.py

def greet(name: str, title: str|None = None) -> str:
    if title:
        return f"Hello, {title} {name}"
    return f"Hello, {name}"

print(greet("Justin", "Dr."))
print(greet("Cory"))
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
Success: no issues found in 1 source file
root@77a3d1e5fa49:/code/hints# python script.py
Hello, Dr. Justin
Hello, Cory
root@77a3d1e5fa49:/code/hints#
```

Literal

The `Literal` type allows you to restrict the values that an argument or variable can take to a specific set of literal values. This can be used for ensuring that a function or method is only called with a specific correct argument.

```python
# ./script.py

from typing import Literal

def color_picker(
    color: Literal["red", "green", "blue"]
) -> tuple[int, int, int]:
    match color:
        case "red":
            return (255, 0, 0)
        case "green":
            return (0, 255, 0)
```

```
        case "blue":
            return (0, 0, 255)

print(color_picker("pink"))
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
script.py:14: error: Argument 1 to "color_picker" has incompatibl\
e type "Literal['pink']"; expected "Literal['red', 'green', 'blue\
']"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
root@77a3d1e5fa49:/code/hints#
```

Final

In instances where a value is expected to be constant, its type can be specified as `Final`. `Final` types allow you to provide stronger guarantees about the behavior of a variable. This also applies to variables being overridden in subclasses.

```
# ./script.py

from typing import Final

API_KEY: Final = "77d75da2e4a24afb85480c3c61f2eb09"
API_KEY = "c25e77071f5a4733bcd453c037adeb3f"

class User:
    MAXSIZE: Final = 32

class NewUser(User):
    MAXSIZE: Final = 64
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
script.py:6: error: Cannot assign to final name "API_KEY"  [misc]
script.py:12: error: Cannot override final attribute "MAXSIZE" (p\
reviously declared in base class "User")  [misc]
Found 2 errors in 1 file (checked 1 source file)
root@77a3d1e5fa49:/code/hints#
```

`TypeAlias`

In instances where inlining type hints becomes too verbose, you can use a `TypeAlias` type to create an alias for a definition based on existing types.

```
# ./script.py

from typing import Literal, TypeAlias

ColorName = Literal["red", "green", "blue"]
Color: TypeAlias = tuple[int, int, int]

def color_picker(color: ColorName) -> Color:
    match color:
        case "red":
            return (255, 0, 0)
        case "green":
            return (0, 255, 0)
        case "blue":
            return (0, 0, 255)

print(color_picker("green"))
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
Success: no issues found in 1 source file
root@77a3d1e5fa49:/code/hints# python script.py
(0, 255, 0)
root@77a3d1e5fa49:/code/hints#
```

NewType

In contrast to a `TypeAlias`, the `NewType` type allows you to define new types based on existing types. This type is treated as it's own distinct type, separate from whatever original type it was defined with. Creating a variable defined as the new type is done by calling the type with values matching the type's type definition.

```
# ./script.py

from typing import Literal, NewType

ColorName = Literal["red", "green", "blue"]
Color = NewType("Color", tuple[int, int, int])

def color_picker(color: ColorName) -> Color:
    match color:
```

```
        case "red":
            return Color((255, 0, 0))
        case "green":
            return Color((0, 255, 0))
        case "blue":
            return Color((0, 0, 255))

print(color_picker("green"))
```

```
root@77a3d1e5fa49:/code/hints# python -m mypy script.py
Success: no issues found in 1 source file
root@77a3d1e5fa49:/code/hints# python script.py
(0, 255, 0)
root@77a3d1e5fa49:/code/hints#
```

TypeVar

Generic Types can be created as a placeholders for specific types that are to be determined at a later time. They still provide type safety by enforcing type consistency, but they do not assert what type is to be expected. Generic types can be constructed using the `TypeVar` constructor.

```
# ./script.py

from typing import TypeVar

T = TypeVar('T')

def identity(item: T) -> T:
    return item

identity(1)
```

`TypeVar`'s can also be either constrained or bound. A bound `TypeVar` can only be instantiated with a specified type or its subtype. A constrained `TypeVar` can only be of a given type or types.

```
# ./script.py

from typing import TypeVar

class Missing(int):
    _instance = None
```

```
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, -1)
        return cls._instance

# True, False, and Missing all satisfy this type
# because they are subclasses of int. However,
# int types also satisfy.
BoundTrinary = TypeVar("Trinary", bound=int)

def is_falsy(a: BoundTrinary) -> bool:
    return a is not True

print(
    is_falsy(True),
    is_falsy(False),
    is_falsy(Missing()),
    is_falsy(-1)
)

# True, False, and Missing all satisfy this type
# because the type is constrained. In this case
# however int does not satisfy, and throws a type
# error.
ConstrainedTrinary = TypeVar("Trinary", bool, Missing)

def is_truthy(a: ConstrainedTrinary) -> bool:
    return a is True

print(
    is_truthy(True),
    is_truthy(False),
    is_truthy(Missing()),
    is_truthy(-1)
)
```

```
root@a22f8a0c68fb:/code/hints# python -m mypy ./script.py
script.py:37: error: Value of type weariable "ConstrainedTrinary"\
 of "is_truthy" cannot be "int"  [type-var]
Found 1 error in 1 file (checked 1 source file)
root@a22f8a0c68fb:/code/hints#
```

In this example we're adding a `Missing()` singleton for representing a "Missing" state for trinary logic (in comparison to boolean logic which has 1 and 0, trinary logic has three states, 1, 0, and -1). Since this new missing state, as well as the `True` and `False` singletons, all are subclasses of `int`, we can create a `TypeVar` named `BoundTrinary` which is bound to the `int` type, and this type definition will satisfy the three states in the trinary. However this type will also match any variable of `int` type; if this is undesirable we can instead constrain the type definition to exclusively the `bool`, and `Missing` types as demonstrated in the `ConstrainedTrinary` type definition. Static type checkers will now throw a type error if an `int` type is passed where a `ConstrainedTrinary` is expected.

Protocols

Instead of checking against a specific type, sometimes we only want to check that an object implements some sort of expected interface. We can do this using the `Protocol` type. A protocol describes the minimal interface that an object must implement in order to be considered a particular type.

```
# ./script.py

from typing import Protocol

class Incrementable(Protocol):
    def increment(self) -> None: ...

class Counter:
    def __init__(self):
        self.value = 0

class CountByOnes(Counter):
    def increment(self):
        self.value += 1

class CountByTwos(Counter):
    def increment(self):
        self.value += 2

def increment_n(counter: Incrementable, n: int) -> None:
    for _ in range(n):
        counter.increment()

for c in (CountByOnes(), CountByTwos()):
    increment_n(c, 10)
    print(c.value)
```

```
<b>root@a22f8a0c68fb:/code/hints# python -m mypy script.py</b>
Success: no issues found in 1 source file
<b>root@a22f8a0c68fb:/code/hints# python script.py</b>
10
20
```

In this example, the `Incrementable` Protocol specifies an interface which contains a specific method `increment()` as a method that takes no arguments and returns `None`. Any object which defines an `increment()` method satisfies the type definition of the protocol.

The `increment_n` function takes an object of type `Incrementable` and an integer `n` and calls the `.increment()` method `n` times. The function does not need to know the specific type of the counter object, as long as it implements the `Incrementable` protocol. Since both `CountByOnes` and `CountByTwos` implement this protocol, we can safely pass both objects to the `increment_n` function.

Runtime type checking

Protocols by default are not type checkable at runtime. However, protocols can hook into Python’s runtime type check mechanisms by including a `typing.runtime_checkable` decorator, which adds runtime infrastructure without interfering with static analysis.

```python
# ./script.py

from typing import Protocol, runtime_checkable

@runtime_checkable
class Incrementable(Protocol):
    def increment(self) -> None: ...

def increment_n(counter: Incrementable, n: int) -> None:
    if not isinstance(counter, Incrementable):
        raise TypeError
```

Generics

Generics are used to create type-safe data abstractions while being type agnostic. To do this, we can create classes which subclasses the `Generic` type from the `typing` module. When the class is later instantiated, a type definition can then be defined at the point of object construction using square brackets.

```python
# ./script.py

from typing import Generic, TypeVar

T = TypeVar("T")

class Queue(Generic[T]):
    def __init__(self) -> None:
        self._data: list[T] = []

    def push(self, item: T) -> None:
        self._data.append(item)

    def pop(self) -> T:
        return self._data.pop(0)

int_queue = Queue[int]()
str_queue = Queue[str]()

# fails type check, pushing a string to an `int` queue
int_queue.push('a')
```

```
root@a22f8a0c68fb:/code/hints# python -m mypy ./script.py
script.py:21: error: Argument 1 to "push" of "Queue" has incompat\
ible type "str"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
root@a22f8a0c68fb:/code/hints#
```

In this example, `Queue` is a generic data abstraction which defines a type-safe implementation for pushing and popping items from a collection. The type is defined at instantiation; when `int_queue` and `str_queue` are created, a type is specified during construction of the queue using square bracket notation. This makes it so code fails static type checking if items of incorrect type are pushed and popped from either of the queues.

TypedDict

The `TypedDict` class is used to create typesafe dictionaries. With `TypedDict`, you can ensure that dictionaries have the right keys and values, and those values are type safe at instantiation.

```python
# ./script.py

from typing import TypedDict

class Person(TypedDict):
    name: str
    age: int

person = Person(name="Chloe", age=27)
```

Classes which subclass `TypedDict` can use class variables to specify keys in a dictionary key-value store. The included type annotation asserts during static type check that the value associated with a given key is of a specified type.

At instantiation, `TypedDict` objects require that all variables are passed to the constructor as keyword arguments. Failing to pass an expected variable will cause static type checking to fail. This default behavior can be changed however, by passing a `total=False` flag in the class definition. Setting this flag will make all keys optional by default. We can further specify a specific key as either `Required` or `NotRequired`, so to designate if a specified key in the collection requires a value at instantiation.

```python
# ./script.py

from typing import TypedDict, NotRequired, Required

class FirstClass(TypedDict, total=False):
    a: str
    b: str

class SecondClass(TypedDict):
    a: str
    b: NotRequired[str]

class ThirdClass(TypedDict, total=False):
    a: str
    b: Required[str]

```

```
first = FirstClass() # passes type check, nothing is required
second = SecondClass() # fails type check, missing 'a'
third = ThirdClass() # fails type check, missing 'b'
```

```
root@a22f8a0c68fb:/code/hints# python -m mypy ./script.py
script.py:18: error: Missing key "a" for TypedDict "SecondClass" \
 [typeddict-item]
script.py:19: error: Missing key "b" for TypedDict "ThirdClass"  \
[typeddict-item]
Found 2 errors in 1 file (checked 1 source file)
root@a22f8a0c68fb:/code/hints#
```

Finally, `TypedDict` objects can subclass the `Generic` type. This allows dictionary values to be collections of types which are specified at instantiation.

```
# ./script.py

from typing import TypedDict, Generic, TypeVar

T = TypeVar("T")

class Collection(TypedDict, Generic[T]):
    name: str
    items: list[T]

coll = Collection[int](name="uid", items=[1,2,3])
```

Chapter 14. Modules, Packages, and Namespaces

Up until now, we have been utilizing the functionality that is built directly into the Python runtime. We have been using literals, containers, functions, and classes that are available by default. However, this is only a small portion of the tools and functionality that are available to Python developers. To access this additional functionality, we must delve into the intricacies of Python's module system and its packaging system.

Modules

Python's module system allows developers to organize and reuse code in a structured way. A module is simply a file containing Python definitions and statements, and its file name serves as the module name with a .py file extension. For example, a file named `module.py` would be a module containing Python code that can be imported and used in other parts of your program.

Consider the following directory structure with two files. The first file is a module file, and the second file is a script.

```
root@b9ba278f248d:/code# find . -type f
./module.py
./script.py
```

In the module file, we define the functions `add()` and `subtract()`. These functions now exist in what is called the "namespace" of the module, i.e. the name `module` is recognized to contain the functions `add` and `subtract`. Everything defined in the top level scope of the `module` file is defined under the `module` namespace.

```python
# ./module.py

def add(a, b):
    return a + b
```

```

def subtract(a, b):
    return a - b
```

In the script file, we use the `import` statement to import our module. The import statement will import the entire module and make all of its definitions and statements available in the current namespace of the script, under the name `module`. We can access all of the defined functionality of the module using the dot syntax, similar to accessing the attributes of a class.

```python
# ./script.py

import module

print(module.add(1, 2))
print(module.subtract(1, 2))
```

With this, we can run `python ./script.py` and the output to the console is `3`, and then `-1`.

```
root@b9ba278f248d:/# cd code/
root@b9ba278f248d:/code# ls
module.py  script.py
root@b9ba278f248d:/code# python script.py
3
-1
root@b9ba278f248d:/code#
```

If we want to instead import our module under a given alias, we can use the `as` keyword in our import statement to rename the module in the scope of the current namespace.

```python
# ./script.py

import module as m

print(m.add(1, 2))
print(m.subtract(1, 2))
```

If we are only interested in importing specific objects from the module namespace, we can use the `from` keyword to import individual objects.

Multiple objects can be imported via a mechanism similar to tuple unpacking.

```python
# ./script.py

from module import (
    add,
    subtract,
)

print(add(1, 2))
print(subtract(1, 2))
```

In instances where our modules are collected into folders within the root directory, we can use dot syntax to specify the path to a module.

```
root@b9ba278f248d:/code# find . -type f
./modules/module.py
./script.py
```

```python
# ./script.py
import modules.module as m

print(m.add(1, 2))
```

Module Attributes

There are several attributes that are automatically created for every module in Python. Some of the most commonly used module attributes are:

- `__name__` - This attribute contains the name of the module as a string. If the module is being run as the main program, this attribute will be set to `__main__`.
- `__doc__` - This attribute contains the module's documentation string, which is specified at the top of the module using triple quotes `"""`.
- `__file__` - This attribute contains the file path of the module as a string.
- `__package__` - This attribute contains the package name of the module as a string, or `None` if the module is not part of a package.

`if __name__ == "__main__":`

The process of importing a module causes the python interpreter to run the module. In order to distinguish between a module being run as the main process as compared to it being run as an import, the `if __name__ == "__main__"` idiom is used. `__name__` will only be equal to `__main__` when the file is being executed as the main program. `__name__` will set to the module name in all other instances.

```python
# ./script.py

import module

def main():
    print(module.add(1, 2))

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

Packages

In addition to modules, a collection of files and folders can be organized into packages. A package in Python is a way to group together related modules and the package name serves as a prefix for the modules it contains. For example, a package named `mypackage` could contain modules `mypackage.module1`, `mypackage.module2` and so on.

To create a package, you need to create a directory with the package name, and within that directory, you can have one or more modules. The directory should contain an `__init__.py` file, which is a special file that tells Python that this directory should be treated as a package.

Outside the directory you should have a setup file. Typically these come in the form of either a `setup.py` file or a `pyproject.toml` file. The `setup.py` file is a setup script that the `setuptools` package uses to configure the build of a python package. A typical `setup.py` file will define a function called `setup()` that is used to specify build parameters and metadata. A `pyproject.toml` file accomplishes much of the same functionality, but uses a declarative configuration scheme to define the project's metadata, dependencies, and build settings. For now, we'll use just use the `pyproject.toml` toolchain, and introduce `setup.py` later when necessary.

Let's consider the following package structure:

```
my_package/
  logging/
    log.py
  math/
    addition.py
    subtraction.py
  __init__.py
  __main__.py
pyproject.toml
```

The `pyproject.toml` file should, at minimum, define which build system should be used to install a package. The default buildsystems for python are `setuptools` and `wheels`.

```
# ./pyproject.toml

[build-system]
requires = ["setuptools", "wheel"]
```

Next, inside the `my_package` directory there should be an `__init__.py` file. For this example it is blank, but you can place any number of imports in this file for objects you wish to be included in the top-level package namespace. We can also optionally add a `__main__.py` file, which is executed when the package is executed from the terminal using the syntax `python -m my_package`.

```
# ./my_package/__main__.py

if __name__ == "__main__":
    print("hello from main!")
```

Inside the `my_package` directory is two folders. The first is a folder called `logging`, and inside that folder there is a `log.py` file. There is only one function in this file, named `fn`. This function simply calls the `print()` function.

```
# ./my_package/logging/log.py

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

Inside the second folder `math` is two files. One file, `addition.py`, contains a function which adds two numbers. This file also calls the `fn` from logging.log to print out the two numbers whenever it is called. The second file, `subtraction.py` does the same, except it returns the result of subtracting the two numbers, instead of adding them.

```
# ./my_package/math/addition.py

from ..logging.log import fn

def add(a, b):
    fn(a, b)
    return a + b
```

```
# ./my_package/math/subtraction.py

from my_package.logging.log import fn

def subtract(a, b):
    fn(a, b)
    return a - b
```

Imports within packages

Packages allow for both relative and absolute imports across the package. Relative import allow you to specify the package and module names relative to the current namespace using dot notation. For example, in `addition.py` the log file's `fn` is imported relatively. The logging module is once removed up the folder tree, so `..logging` specifies that the `logging` module is once removed from the current directory using a second `.`. As another example, from the `addition.py` file you could import the `subtract` function by specifying a relative import `from .subtraction import` `subtract`. Absolute imports on the other hand specify the full package and module name, as seen in `subtraction.py`. This is more explicit and straightforward, but in deeply nested package directories it can get verbose.

Installation

In order to make use of this package, we need to install it. Python's default package installer is `pip`, which stands for "pip installs packages". Pip also allows you to easily search for, download, and install packages from PyPI, the Python Package Index, which is an online repository of Python packages. Packages are installed from the package index by running `python -m pip install <package_name>` in your terminal. For local packages, you can install from the package directory using `python -m pip install ./`. The `-e` flag can be optionally passed so that the package is installed as `--editable`. This links the installed python package to the local developer instance, so that any changes made to the local codebase are automatically reflected in the imported module.

```
root@e08d854dfbfe:/code# ls
my_package  pyproject.toml

root@e08d854dfbfe:/code# python -m pip install -e .
Obtaining file:///code
  Installing build dependencies ... done
  Checking if build backend supports build_editable ... done
  Getting requirements to build editable ... done
  Preparing editable metadata (pyproject.toml) ... done
Building wheels for collected packages: my-package
  Building editable for my-package (pyproject.toml) ... done
  Created wheel for my-package: filename=my_package-0.0.0-
0.edita\
ble-py3-none-any.whl size=2321
sha256=2137df7f84a48acdcb77e9d1ab3\
33bf838693910338918fe16870419cb351979
  Stored in directory: /tmp/pip-ephem-wheel-cache-
qqft00bw/wheels\
/71/fd/a9/eb23a522d4ed2deb67e9d98937897b0b77b5bf9c1ac50a2378
Successfully built my-package
Installing collected packages: my-package
Successfully installed my-package-0.0.0
WARNING: Running pip as the 'root' user can result in broken
perm\
```

```
issions and conflicting behaviour with the system package
manager\
. It is recommended to use a virtual environment instead:
https:/\
/pip.pypa.io/warnings/venv

root@e08d854dfbfe:/code# python
Python 3.11.1 (main, Jan 17 2023, 23:30:27) [GCC 10.2.1
20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more
informa\
tion.
>>> from my_package.math import addition
>>> addition.add(1, 2)
1 2
3
>>>
>>> from my_package.math import subtraction
>>> subtraction.subtract(3, 4)
3 4
-1
>>>
root@71c53592c77f:~# python -m my_package
hello from main!
```

Part III. The Python Standard Library

The Python Standard Library is a collection of modules and functions that are included with every Python installation. It provides a range of functionality that can be used to accomplish a wide variety of tasks, such as string and text processing, file and directory manipulation, networking, mathematics and statistics, multithreading and multiprocessing, date and time operations, compression and archiving, and logging and debugging. The inclusion of such an expansive standard library by default means that developers can focus on writing their application code, rather than worrying about recreating standard implementations from scratch. It is a big reason as to why Python is often referred to as a “batteries included” language.

Chapter 15. Virtual Environments

As we start to create packages and start integrating other packages into projects, we're going to quickly run into the issue where we want to keep project dependencies separate. This is where Python virtual environments come in handy. Virtual environments allow us to create isolated environments with their own independent set of dependencies, which can be managed separately from other projects and the global Python environment.

Without virtual environments, different projects might require different versions of the same package or even completely different packages. Installing or upgrading a package in one project could affect the behavior of other projects, potentially causing errors or breaking functionality.

Using virtual environments ensures that each project has access to the specific versions of packages that it needs, without worrying about conflicts or breaking other projects. It also allows for easier sharing and portability of projects, as the environment and its dependencies can be packaged and distributed together with the project.

To create a virtual environment, start by navigating in your terminal to a project directory. Next, run the command `python -m venv $name`, where `$name` is the name of your virtual environment (typically just `venv` or `.venv`). This will create a directory in the current working directory with the folder name `$name`. To note, if you're using a version control tool like `git`, be sure to add this folder to the `.gitignore` file so packages aren't added to the project VC history.

```
root@aa41f93b0e59:~# python -m venv venv
root@aa41f93b0e59:~# ls
venv
```

Inside the newly created `venv` directory, we'll see a few folders. First is the `./venv/bin` folder (on Windows this will be the `./venv/Scripts` folder) which is where executables will be placed. The two files within this folder that are worth knowing about are the `python` file, which is a symlink to the binary executable of the global python version, and the `activate` script, which for POSIX systems (linux, mac) is generally `./venv/bin/activate` (or an analogous `activate.fish` or `activate.csh` if using either the `fish` shell or `csh`/`tcsh` shells, respectively), and for Windows is `./venv/Scripts/Activate.ps1` if using PowerShell. The `activate` script acts differently depending on which platform you are using, but in general its responsibility is to add the executables folder to your shell's `PATH`. Doing this makes such that when it goes to execute `python`, it first finds the virtual environment executable instead of finding the global install. It also adds environment variables such as `VIRTUAL_ENV` so that the interpreter knows that it is running in a virtual environment mode, and finally it updates the shell prompt to reflect that the virtual environment is active. Finally, it adds a `deactivate` command to restore the shell to it's original state once we are done with the virtual environment.

```
root@cd9ed409ec97:~# which python
/usr/local/bin/python
root@cd9ed409ec97:~# echo $PATH
/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:\
/sbin:/bin
root@cd9ed409ec97:~# . venv/bin/activate
(venv) root@cd9ed409ec97:~# which python
/root/venv/bin/python
(venv) root@cd9ed409ec97:~# echo $PATH
/root/venv/bin:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr\
/sbin:/usr/bin:/sbin:/bin
(venv) root@cd9ed409ec97:~#
```

Apart from the `./venv/bin` folder, the `venv` package also creates a `./venv/lib` folder. Inside this folder nested under a python version folder is a folder called `site-packages`. This `site-packages` directory inside the `./venv/lib/` directory of the virtual environment is where third-party packages are installed when the virtual environment is active.

```
(venv) root@959fd3c7f2e4:~# python -m pip install ciso8601
Collecting ciso8601
  Downloading ciso8601-2.3.0-cp311-cp311-
```

```
manylinux_2_5_x86_64.man\
ylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(39\
 kB)
Installing collected packages: ciso8601
Successfully installed ciso8601-2.3.0

[notice] A new release of pip available: 22.3.1 -> 23.0.1
[notice] To update, run: pip install --upgrade pip
(venv) root@13be81b026f6:~# ls -1a ./venv/lib/python3.11/site-
pac\
kages/
.
..
_distutils_hack
ciso8601
ciso8601-2.3.0.dist-info
ciso8601.cpython-311-x86_64-linux-gnu.so
distutils-precedence.pth
pip
pip-22.3.1.dist-info
pkg_resources
setuptools
setuptools-65.5.0.dist-info
```

In this example, using `python -m pip` to install a `ciso8601` library, a `ciso8601` folder was created in the virtual environment's `site-packages` folder. This library install can now be imported into projects which run in the context of this virtual environment. This library is separate from the global installation of python; attempting to import it outside of the context of the virtual environment will cause an `ImportError`.

Chapter 16. Copy

In previous sections, we made reference to the idea that, when you make an assignment to a variable `a` from a differing variable `b`, the assignment operation is merely creating a new reference to the value that `a` is currently referencing. In cases where this value is a mutable collection, changes to `b` will be reflected in `a`, and vice versa.

If instead we want to make a separate and unique object, where state is not bound between the two variables, we need to instead make a copy of that value, and then assign that copy, instead of assigning the original variable. Python provides a `copy` module in the standard library for just this purpose.

The `copy` module defines two functions, `copy` and `deepcopy`. The function `copy.copy(item)` creates a shallow copy of a given item (meaning, only make a copy of the object itself). Comparatively, `copy.deepcopy(item)` creates a deeply nested copy (meaning, each mutable object which is referenced by the object is also copied).

Lets consider an example. Given an object `my_list` which is a list of lists, we can call `copy.copy` on this list, which returns a value we assign to `shallow_copy`. The shallow_copy is a copy of the list `my_list`, so calling `id()` on `shallow_copy` and `my_list` yields different identifiers, as they are different lists. However, calling `id()` on any of the items of the two lists, such as the item at the zeroth index, yields the same identifier, as the contents of both lists reference the same items.

```
>>> my_list = [["a", "b", "c"], [1, 2, 3]]

>>> import copy
>>> shallow_copy = copy.copy(my_list)

>>> id(my_list), id(shallow_copy)
(140097007507264, 140097007504960) # id's are different

>>> id(my_list[0]), id(shallow_copy[0])
(140097007512368, 140097007512368) # id's are the same
```

This can be contrasted against the action of `deepcopy`. Given the same object `my_list`, we can call `copy.deepcopy` on this list, which returns a variable we assign to `deep_copy`. The `deep_copy` is a copy of the list `my_list`, so calling `id()` on `deep_copy` and `my_list` yields different identifiers, as they are different lists. The difference however is that now if we call `id()` on any of the items of the two lists, such as the item at the zeroth index, the call yields different values. This is because `copy.deep_copy` recursively traverses collections, making copies of each item, until the new copy is completely independent of the original item.

```
>>> deep_copy = copy.deepcopy(my_list)
>>> id(my_list), id(deep_copy)
(140097007554944, 140097007504960) # id's are different

>>> id(my_list[0]), id(deep_copy[0])
(140097007345472, 140097007551616) # id's are also different
```

Chapter 17. Itertools

The itertools library in Python is a collection of tools for working with iterators. The main philosophy behind it is to provide a set of efficient, reusable, and easy-to-use functions for common tasks related to iterators.

One of the main reasons for its existence is to make it easier to work with large datasets or sequences of data without having to write complex and repetitive code. The library also aims to promote a more functional programming style, as many of its functions are designed to be used in combination with other functions to build up more complex operations. The result is that itertools allows for a more expressive and concise codebase.

Chaining Iterables

The `itertools.chain()` function is used to take multiple iterators and "chain" them together into a single iterator. This can be useful when you have multiple lists, tuples, or other iterable objects that you want to treat as one sequence. `itertools.chain()` is variadic, so you can pass any number of iterables to the function call, and it'll return a single iterable.

```
>>> import itertools
>>> a = [1, 2]
>>> b = [3]
>>> list(itertools.chain(a, b))
[1, 2, 3]
```

If the iterables are already in an iterable collection, the `itertools.chain.from_iterable()` classmethod can be used instead as a way to iterate over an iterable of iterables.

```
>>> import itertools
>>> lists = [(1, 2), (3, 4)]
>>> list(itertools.chain.from_iterable(lists))
[1, 2, 3, 4]
```

Filtering Iterables

The `itertools.compress()` function is used to filter an iterator based on a second masking iterator. It returns an iterator that only includes the elements of the first iterator for which the corresponding element in the masking iterator is truthy. The two iterators passed to `itertools.compress()` should be of the same length, otherwise, the function will raise an exception.

```
>>> import itertools
>>> data = [1, 2, 3, 4, 5, 6]
>>> mask = [True, False, True, False, True, False]
>>> result = itertools.compress(data, mask)
>>> list(result)
[1, 3, 5]
```

The `itertools.filterfalse()` function is used to create an iterator that excludes elements from the input iterator which satisfy the callback. It has the opposite action of the built-in `filter()` function.

```
>>> import itertools
>>> data = range(1, 10)
>>> list(itertools.filterfalse(lambda x: x%2 or x%3, data))
[6]
```

Finally, if you only wish to filter out the elements of an iterable while the callback is satisfied, the `itertools.dropwhile()` function can be used. This function creates an iterator that drops elements from the input iterator as long as a given callback is truthy for those elements. Once the callback returns a falsy value for an element, the remaining elements are included in the output iterator. There's also an analogous `itertools.takewhile()` function, which instead of dropping values, it yields values until the callback returns falsy.

```
>>> import itertools
>>> data = range(1, 10)
>>> list(itertools.dropwhile(lambda x: x%2 or x%3, data))
[6, 7, 8, 9]
>>> list(itertools.takewhile(lambda x: x < 5, data))
[0, 1, 2, 3, 4]
```

Cycling Through Iterables

The `itertools.cycle()` function is used to create an iterator that repeatedly iterates over the elements of a given iterable. It creates an infinite iterator that cycles through the elements of the input iterable.

```
>>> import itertools
>>> data = [1, 2, 3]
>>> result = itertools.cycle(data)
>>> next(result)
1
>>> next(result)
2
>>> next(result)
3
>>> next(result)
1
>>> next(result)
2
```

Creating Groups

The `itertools.groupby()` function is used to group elements from an iterator into groups based on a given callback. The callback takes one argument, which is an element from the input iterator, and returns a value that is used to determine which group the element belongs to. The `groupby()` function returns an iterator that produces pairs of the form `(key, group)`, where `key` is the result of the callback on a given element, and `group` is an iterator which continues to yield items from the iterable so long as `callback(item)` produces the same key.

```
>>> import itertools
>>> data = [1, 2, 4, 3, 5, 6]
>>> groups = itertools.groupby(data, lambda x: x % 2)
>>> for k, g in groups:
...     print(k, list(g))
...
1 [1]
0 [2, 4]
1 [3, 5]
0 [6]
```

```
>>> data = [1, 2, 3, 4, 5, 6]
>>> groups = itertools.groupby(data, lambda x: x % 2)
>>> c = {}
>>> for k, g in groups:
...     c.setdefault(k, []).extend(g)
...
>>> c
{1: [1, 3, 5], 0: [2, 4, 6]}
```

Slicing Iterables

The `itertools.islice()` function is used to create an iterator that returns selected elements from the input iterator. It takes three arguments: the input iterator, a start index, and a stop index. The function returns an iterator that starts at the start index and continues until the stop index, or the end of the input iterator, whichever comes first. An optional step argument can be provided to skip over elements.

```
>>> import itertools
>>> data = [1, 2, 3, 4, 5, 6]
>>> list(itertools.islice(data, 2, 5))
[3, 4, 5]
```

Zip Longest

The `itertools.zip_longest()` function allows for iterators of different lengths. It combines elements from multiple iterators into a single iterator of tuples. When one of the input iterators is exhausted, it fills in the remaining values with a specified fillvalue.

```
>>> import itertools
>>> data = [(1, 2, 3), (4, 5, 6, 7)]
>>> list(itertools.zip_longest(*data, fillvalue=-1))
[(1, 4), (2, 5), (3, 6), (-1, 7)]
```

Chapter 18. Functools

The functools library in Python is a collection of tools for working with functions and functional programming. One of the main reasons for its existence is to make it easier to write efficient, expressive, and maintainable code by providing a set of higher-order functions designed to work with other functions. These functions are highly composable and reusable, making them well-suited for use in functional programming.

Partials

The `functools.partial()` function is used to create a new function with some of the arguments pre-filled. It takes a function and any number of parameters, and returns a new function that, when called, will call the original function with the pre-filled arguments and any additional arguments passed to the new function.

```
>>> import functools
>>> fn = functools.partial(lambda x, y: x + y, 1)
>>> fn(2)
3
```

An analgous `functools.partialmethod()` is also available, which allows you to emulate this behavior on object methods.

Reduce

The `functools.reduce()` function is used to apply a binary function to the elements of an iterable in a cumulative way. The function takes two arguments: a function and an iterable, and returns a single value. The function is applied cumulatively to the elements of the iterable, from left to right, so as to reduce the iterable to a single value. An optional third value can be passed as the initial value, instead of relying on the first element of the iterable.

```
>>> import functools
>>> data = range(10)
>>> functools.reduce(lambda x, y: x+y, data)
45
```

Pipes

Pipes are a concept in functional programming which allow developers to chain function calls. It's used to pass the output of one function as the input to another function, creating a pipeline through which data is passed.

We can emulate this functionality in python using `functools.reduce()`, where instead of having an iterable of data, we have an iterable of functions.

```
>>> import functools
>>> fns = (lambda x: x + 2, lambda x: x ** 2, lambda x: x // 3)
>>> functools.reduce(lambda x, f: f(x), fns, 7)
27
```

Caching

`functools.cache()` is a function that is used to cache the results of a function call, so that the function does not need to be computed again for the same input. It stores the result of the function call for a specific input in a cache. If the function is called again with the same input, the cached result is returned instead of recomputing the result. This can be useful for reducing the amount of computation required for a function, and can also be useful for improving the performance of a function that is called multiple times with the same input.

```
>>> import functools
>>> def fibonacci(n):
...     if n == 2 or n==1:
...         return 1
...     return fibonacci(n-1) + fibonacci(n-2)
...
>>> fibonacci(40) # this take ~5 seconds on my Ryzen 5800x desktop
>>> fibonacci = functools.cache(fibonacci)
>>> fibonacci(40) # this operation is almost immediate
```

In some instances it may be necessary to limit the cache size. In this case, you can use `functools.lru_cache()` to set an optional maxsize for the cache, where the “least recently used” values that exceed the cache size are dropped. By default this value is 128 items. The cache can be invalidated using the `.cache_clear()` method on the cache.

```
>>> import functools
>>> def fibonacci(n):
...     if n == 2 or n==1:
...         return 1
...     return fibonacci(n-1) + fibonacci(n-2)
...
>>> fibonacci = functools.lru_cache(maxsize=10)(fibonacci)
>>> fibonacci(40) # this operation is almost immediate
>>> fibonacci.cache_clear()
```

Finally, `functools.cached_property()` is a function that is used to cache the results of a method call on an object, so that the method does not need to be computed again for the same object. It stores the result of the method call for a specific object in a cache, and if the method is called again with the same object, the cached result is returned instead of recomputing the value.

```
>>> import functools
>>> class MyClass:
...     @functools.cached_property
...     def value(self):
...         print("called")
...         return True
...
>>> my_object = MyClass()
>>> my_object.value
called
True
>>> my_object.value
True
```

It should be noted that `cached_property()` is thread-safe, but it’s underlying implementation is slow in multithreaded workloads. There is some discussion about reimplementing it at a later date.

Dispatching

The `functools.singledispatch()` function is a function that is used to create a single-dispatch generic function, which can be used to define multiple implementations of a function for different types of input. The function takes a single argument, the type of the input, and returns the appropriate implementation for that type. The analogous method implementation is `functools.singledispatchmethod()`. It supports nesting with other decorators so long as it is the outer most decorator (i.e. it must be called first).

```
>>> fn = functools.singledispatch(
...     lambda x: print("unknown type:", x)
... )
>>> fn.register(int)(lambda x: print(f"type int: {x}"))
>>> fn.register(float)(lambda x: print(f"type float: {x}"))
>>> fn("a")
unknown type: a
>>> fn(1)
type int: 1
>>> fn(1.0)
type float: 1.0
```

Chapter 19. Enums, NamedTuples, and Dataclasses

As programs become more complex and interconnected, it becomes important to limit the variability of state wherever you can. By defining a set of named values and limiting the possible options, you can reduce the chance of introducing bugs and errors due to unexpected states. It also makes your code more readable and understandable, as the meaning and purpose of specific values are clear and consistent throughout the program.

In this chapter we're going to look at three separate methods for enumerating values in your code - Enums, NamedTuples, and Dataclasses.

Enums

Enums (enumerated types) in Python are used to define a set of named values, which can be used to represent specific states in a program. They are a way to give more structure and meaning to a set of values. Enums are defined using the `Enum` class. The `Enum` class is used to define the enumerated type, and each member of the `Enum` is defined as a class variable. Each member is given a name and a value, and the value can be any valid Python object, but it's common to use integers or strings as values. If the value is of no consequence to you, you can use any object with a unique identifier, such as `object()`.

An `Enum` can also be created using a factory function. In this case, the first argument is the name of the Enum, and the second argument is an iterable containing each class variable. The values are automatically assigned as integers incrementing from 1.

```
from enum import Enum

class Weekends(Enum):
    Saturday = object()
    Sunday = object()

```

```
Weekdays = Enum(
    "Weekdays",
    (
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
    ),
)
```

Once an `Enum` is defined, you can use the enumerated values as if they were variables. `Enums` in python are immutable, meaning that their values cannot be changed after they are created, and they are also comparable, meaning that you can use them in comparison operations.

```
match current_day:
    case Weekdays.Monday:
        print("Today is Monday")
    case Weekdays.Tuesday:
        print("Today is Tuesday")
    case Weekdays.Wednesday:
        print("Today is Wednesday")
    case Weekdays.Thursday:
        print("Today is Thursday")
    case Weekdays.Friday:
        print("Today is Friday")
    case Weekends.Saturday | Weekends.Sunday:
        print("Weekend!")
```

NamedTuples

NamedTuples are a special type of tuple in Python that allows you to give names to the elements of the tuple. They are designed to be interoperable with regular tuples, but they also allow you to access elements by name, in addition to by index.

The `NamedTuple` class from the `typing` module can be used to create a NamedTuple. The derived class defines attributes via type annotations. Instance variables are defined at instantiation. A `NamedTuple` can be instantiated using variadic arguments or keyword arguments; in the case of using variadic args, the order of attributes defined in the class determine which indexed value is assigned to the attribute.

A `NamedTuple` can also be created using the factory function `namedtuple` in the `collections` module. In this case, the first argument is the name of the NamedTuple, and the second argument is an iterable containing the name of each attribute.

```
>>> from typing import NamedTuple
>>> class Point(NamedTuple):
...     x: int | float
...     y: int | float
...
>>> point = Point(1, 2)
>>> point
Point(x=1, y=2)
>>> point.x
1
>>> from collections import namedtuple
>>> Coordinate = namedtuple("Coordinate", ('x', 'y'))
>>> coordinate = Coordinate(1, 2)
>>> coordinate
Coordinate(x=1, y=2)
>>> coordinate.y
2
>>> list(coordinate)
[1, 2]
```

Dataclasses

Dataclasses are the most feature-rich implementation of data as objects. They allow you to define classes that are less verbose than traditional Python classes, while still providing useful functionality such as automatic generation of default special methods.

The `dataclasses.dataclass()` decorator automatically generates a default implementation of special methods like `__init__`, `__repr__`, `__eq__`, `__lt__` etc. so you don't have to write them yourself. This allows you to focus on the data that the class is holding, rather than the mechanics of the class.

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Person:
...     name: str
...     age: int
...     country: str = "USA"
...
>>> Person("Reid", 28)
Person(name='Reid', age=28, country='USA')
>>> Person("Sai", 27, "India")
Person(name='Sai', age=27, country='India')
```

The dataclass decorator takes multiple optional arguments. They are as follows:

- `init` - a boolean indicating whether or not to generate an `__init__` method.
- `repr` - a boolean indicating whether or not to generate a `__repr__` method.
- `eq` - a boolean indicating whether or not to generate `__eq__` method.
- `order` - a boolean indicating whether or not to generate `__lt__`, `__le__`, `__gt__`, and `__ge__` methods.
- `frozen` - a boolean indicating whether or not to make the class immutable.
- `unsafe_hash` - a boolean indicating whether or not to force create a hash function. If `True`, all mutable attributes need to be passed over by including `hash=False` in the corresponding field factory function.
- `slots` - a boolean indicating whether or not to prefer `__slots__` architecture to store attributes over a dictionary

Mutable default values can be provided using the `fields` function. `fields` takes an optional `default_factory` argument that must be a callable of zero arguments.

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class Group:
...     people: list[str] = field(default_factory=list)
...
```

```
>>> Group(["Jessie", "Ryan"])
Group(people=['Jessie', 'Ryan'])
```

Chapter 20. Multithreading and Multiprocessing

Parallelism refers to the concept of running multiple tasks simultaneously, in order to improve performance and speed up the overall execution of a program. In Python, parallelism can be achieved through the use of multithreading and multiprocessing.

Multithreading involves the use of multiple threads, which are small units of a process that can run independently of each other. The threading module can be used to create and manage threads. Multithreading can be useful for tasks that involve a lot of waiting, such as waiting for input/output operations to complete or waiting for a response from a network resource.

Multiprocessing, on the other hand, involves the use of multiple processes, which are separate instances of a program that run concurrently. The multiprocessing module can be used to create and manage processes. Multiprocessing can be useful for tasks that involve a lot of computation, such as data processing or machine learning tasks.

It's important to note that while both multithreading and multiprocessing can be used to achieve parallelism in Python, they have different use cases and tradeoffs. Multithreading is typically used for IO-bound and high-level structured network code, while multiprocessing is used for CPU-bound and low-level code.

Multithreading

In Python, the threading module provides means through which to create and manage threads. A thread is a separate execution flow that runs in parallel with the main program. Threads share the same memory space as the main program, which allows them to access and share data.

Threading uses a preemptive task switching model, where the interpreter schedules and switches between threads. The interpreter can interrupt a running thread at any time, in order to give another thread a chance to run. Python uses a Global Interpreter Lock (GIL) to achieve this: only the thread which has acquired the GIL may execute, and there is only one lock, the global interpreter lock, which any thread can acquire. The GIL is released when a thread makes a blocking I/O call, such as reading from a file or waiting for a network response. When the call returns, the GIL must be acquired again before the thread can continue to execute. This means that if one thread is blocked on I/O, another thread can take over and execute Python code. However, if all threads are performing CPU-bound tasks, they will be effectively running sequentially and not concurrently.

The threading module in Python provides a way to create and manage threads. To create a new thread, you can use the Thread class from the threading module. The Thread class takes a target function as an argument, which is a function that will be executed by the thread. The function `args` and `kwargs` can be passed through using keyword arguments of the same name.

```
>>> import threading
>>> def my_function(*args, **kwargs):
...     print(f"Hello from the thread! {args} {kwargs}")
...
>>> thread = threading.Thread(target=my_function, args=(1,), kwar\
gs={"a": 1})
>>> thread.start()
Hello from the thread! (1,) {'a': 1}
```

The `threading` module also includes a `Thread` class which can be subclassed. It has several methods and attributes that can be used to control and manipulate threads. Some of the most commonly used methods are as follows:

- `Thread.start()` - starts the execution of the thread. This method creates a new native thread and calls the run method on it.
- `Thread.run()` - entry point of the thread. This method is where the code that will be executed by the thread should be placed.
- `Thread.join(timeout=None)` - waits for the thread to complete. If the optional timeout argument is provided, the method will wait for at

most time seconds.

- `Thread.is_alive()` - returns True if the thread is executing, and False otherwise.

If you want to subclass the `Thread` object, you need to be sure to explicitly call the parent class's `__init__` method inside the child class's `__init__` method, or use a `@classmethod` for instantiation. Furthermore, you should not call the `.run()` method directly. Rather, use the `.start()` method to start the thread. The `.start()` method from `threading.Thread` will create a new native thread, and *it* will call the `.run()` method.

```
>>> import threading, time
>>> class MyThread(threading.Thread):
...     @classmethod
...     def create(cls, value):
...         self = cls()
...         self.value = value
...         self.start()
...         return self
...     def run(self):
...         time.sleep(self.value)
...         print(f"Thread slept {self.value} seconds")
...
>>> threads = [MyThread.create(i) for i in range(5)]
>>> # .join() returns None, so any() will exhaust the expression
>>> any(t.join() for t in threads)
Thread slept 0 seconds
Thread slept 1 seconds
Thread slept 2 seconds
Thread slept 3 seconds
Thread slept 4 seconds
False
```

Thread Locks

Threads have the ability to operate on shared memory. However, access to that memory must be "thread safe", i.e. controlled in such a manner that no two threads are attempting to change shared mutable state at the same time. Failure to keep shared memory thread safe can lead to undefined behavior.

```
>>> import threading, time
>>> value = 0
```

```
>>> def increment():
...     global value
...     tmp = value
...     time.sleep(0.1)
...     value = tmp + 1
...
>>> threads = (
...     threading.Thread(target=increment)
...     for _ in range(2)
... )
...
>>> for t in threads:
...     t.start()
...
>>> for t in threads:
...     t.join()
...
>>> value # should be 2
1
```

The `threading.Lock()` class provides a way to synchronize access to shared resources between multiple threads. A lock is a synchronization object that can be in one of two states: “locked” or “unlocked”. When a thread acquires a lock, it changes the state of the lock to “locked” and prevents other threads from acquiring the lock. When the thread releases the lock, it changes the state of the lock to “unlocked” and allows other threads to acquire the lock.

The `Lock()` class defines two methods: `.acquire()` and `.release()` that can be used to acquire and release the lock. The `.acquire()` method can also take an optional blocking parameter, which defaults to `True`. When blocking is `True`, the `.acquire()` method blocks the execution of a thread until the lock can be acquired; when blocking is `False`, the `.acquire()` method returns immediately with a boolean indicating whether the lock was acquired or not. Additionally, the `.acquire()` method takes an optional timeout parameter, which will release the lock after a certain amount of time, in case of deadlock.

A lock object can also be used as a context manager. The arguments of the `__enter__` method are equivalent to the `.acquire()` method.

```
>>> import threading, time
>>> value = 0
>>> lock = threading.Lock()
>>> def increment():
...     with lock:
...         global value
...         tmp = value
...         time.sleep(0.1)
...         value = tmp + 1
...

>>> threads = (
...     threading.Thread(target=increment)
...     for _ in range(2)
... )
...
>>> for t in threads:
...     t.start()
...
>>> for t in threads:
...     t.join()
...
>>> value # should be 2
1
```

Multiprocessing

The multiprocessing module in Python provides a way to write concurrent code using multiple processes, instead of threads. The multiprocessing module provides several classes and functions for creating and managing processes, including `Process`, `Queue`, `Lock`, `Pool`, and `Pipe`.

Process and Pool

The `Process()` class can be used to create and manage new processes. A process is a separate execution environment, with its own memory space and Python interpreter. This allows you to take advantage of multiple cores on a machine, and to work around the Global Interpreter Lock (GIL) that prevents multiple threads from executing Python code simultaneously. The `multiprocessing.Process` class has the same API as the `threading.Thread` class.

```
>>> def f(name):
...     print(f"Hello {name}")

>>> multiprocessing.Process(target=f, args=("Peter",)).start()

Hello Peter

>>> import multiprocessing, time
>>> class MyProcess(multiprocessing.Process):
...     @classmethod
...     def create(cls, value):
...         self = cls()
...         self.value = value
...         self.start()
...         return self
...     def run(self):
...         time.sleep(self.value)
...         print(f"Process slept {self.value} seconds")
...
>>> ps = [MyProcess.create(i) for i in range(5)]
>>> any(p.join() for p in ps)
Process slept 0 seconds
Process slept 1 seconds
Process slept 2 seconds
Process slept 3 seconds
Process slept 4 seconds
False
```

The multiprocessing library also provies a `Pool` class that can be used to orchestrate multiple tasks in parallel. A pool of worker processes is a group of processes that can be reused to execute multiple tasks.

```
>>> import multiprocessing
>>> def double(n):
...    return n * 2
...
>>> with multiprocessing.Pool(processes=4) as pool:
...     results = pool.map(double, range(8))
...     print(results)
...
[0, 2, 4, 6, 8, 10, 12, 14]
```

In this example, a `Pool()` object is created with four processes and assigned to the variable `pool`. The pool's `.map()` method is used to apply `my_function()` on multiple inputs in parallel. The `.map()` method returns a list of the results in the order of the input.

The `Pool()` class also provides several other methods for submitting tasks, such as `.imap()`, `.imap_unordered()` and `.apply()`. The `.imap()` and `.imap_unordered()` methods are similar to `.map()`, except these methods return iterators which yields results lazily. The distinction between the two is that `.imap()` will yield results in order, and `.imap_unordered()` will yield results arbitrarily. Finally, the `.apply()` method is used to submit a single task for execution, and it blocks the main process until the task is completed.

Process Locks

Similar to the `threading.Lock()` class, the `multiprocessing.Lock()` class can be used to synchronize access to shared resources across multiple processes. When a process acquires a lock, it changes the state of the lock to “locked” and prevents other processes from acquiring the lock. When the process releases the lock, it changes the state of the lock to “unlocked” and allows other processes to acquire the lock.

The `multiprocessing.Lock()` class has the same API as the `threading.Lock()` class. It has similar `.acquire()` and `.release()` methods, and can be used as a context manager.

```
>>> import multiprocessing, time
>>> lock = multiprocessing.Lock()
>>> def increment(n):
...     with lock:
...         time.sleep(1-n)
...         print(n)
...
>>> ps = (
...     multiprocessing.Process(target=increment, args=(i,))
...     for i in range(2)
... )
...
>>> for p in ps:
...     p.start()
...
```

```
>>> for p in ps:
...     p.join()
...
0
1
```

Pipes and Queues

Since processes run in isolated execution environments, it's difficult to share data between them. You can't just pass a reference to an object in the main process and expect separate processes to interact with it. In order to establish interprocess communication, we need to use IPC mechanisms to send and receive data. The `multiprocessing` module provides both pipes and queues for this purpose.

`multiprocessing.Pipe()` creates a two-ended connection between two processes, one end of the connection is for sending messages, and the other end is for receiving messages. It returns a pair of connection objects, one for each end of the pipe. You can pass an optional `duplex=` parameter, which if `True` makes the connections bidirectional. There's also a `multiprocessing.Queue()` object which creates a queue that is shared between multiple processes. It allows processes to put and get messages in a thread-safe and process-safe way.

```
>>> import multiprocessing
>>> def sender(conn, item):
...     conn.send(item)
...     conn.close()
...
>>> downstream, upstream  = Pipe()
>>> process = multiprocessing.Process(
...     target=sender,
...     args=(upstream, "Hello!")
... )
>>> process.start()
>>> process.join()
>>> downstream.recv()
'Hello!'
```

In this example, a pipe is created, and then a second process is created. The second process sends a message through the pipe and the main process receives it.

```
>>> import multiprocessing
>>> def sender(queue, value):
...     queue.put(value)
...
>>> queue = multiprocessing.Queue()
>>> process = multiprocessing.Process(target=sender, args=(queue,\
 "Hello!"))
>>> process.start()
>>> process.join()
>>> queue.get()
'Hello!'
```

In this example, a queue is created, and then a second process is created. The second process places a message in the queue for the main process to get.

`concurrent.futures`

The `concurrent.futures` module in Python provides a high-level API for asynchronously executing callables using threading or multiprocessing. This module provides several classes and functions that abstract away the details of thread and process management, and provide a consistent interface for executing tasks in parallel.

The `ThreadPoolExecutor` class is used for asynchronously executing callables using a pool of worker threads. A thread pool is a group of worker threads that can be reused to execute multiple tasks. The `ThreadPoolExecutor` class allows you to submit tasks to be executed by the worker threads, and to retrieve the results of the tasks when they are completed.

The `ProcessPoolExecutor` class is used for asynchronously executing callables using a pool of worker processes. Similar to a thread pool, a process pool is a group of worker processes that can be reused to execute multiple tasks. It provides an API that is the same as `ThreadPoolExecutor`, but it is important to note that due to the nature of processes, tasks will not share the same memory space and you will therefore have to use interprocess communication mechanisms like the `multiprocessing.Queue()` in order to communicate between the various processes.

When you submit a task to be executed by a `ThreadPoolExecutor` or `ProcessPoolExecutor`, the `.submit()` method returns a `Future` object. You can use this `Future` object to check the status of the task, retrieve the result of the task when it is completed, or wait for the task to complete.

The `concurrent.futures` module in Python provides two functions for working with a collection of `Future` objects: `wait()` and `as_completed()`.

The `concurrent.futures.wait()` function takes an iterable of Future objects and blocks until all of the futures are completed or the optional timeout is reached. A `return_when` parameter controls when the function returns; it can be set to FIRST_COMPLETED, FIRST_EXCEPTION, or ALL_COMPLETED (the default value). If FIRST_COMPLETED is used, the function will return as soon as any future completes. If FIRST_EXCEPTION is used, the function will return as soon as any future raises an exception. If ALL_COMPLETED is used, the function will return only when all futures have completed, regardless of whether any raised an exception. The function returns a named tuple of sets, containing the completed, not completed, and done futures.

The `concurrent.futures.as_completed()` function takes an iterable of `Future` objects and returns an iterator that yields `Future` objects as they are completed. The timeout parameter can be used to specify a maximum time to wait for the futures to complete before stopping the iteration.

It’s common to utilize `as_completed` when you want to process the results of the tasks as soon as they are completed, regardless of the order they were submitted. On the other hand, wait is useful when you want to block until all the tasks are completed and retrieve the results in the order they were submitted.

```
>>> import concurrent.futures
>>> def double(n):
...     return n * 2
...
>>> with concurrent.futures.ThreadPoolExecutor() as exec:
...     results = [exec.submit(double, i) for i in range(10)]
...     values = [
...         future.result() for future in
...         concurrent.futures.as_completed(results)
...     ]
...
>>> values
[12, 18, 10, 8, 2, 16, 0, 4, 6, 14]
>>> with concurrent.futures.ProcessPoolExecutor() as exec:
...     jobs = [exec.submit(double, i) for i in range(10)]
...     results = concurrent.futures.wait(jobs)
...     values = list(future.result() for future in results.done)
...
>>> values
[0, 18, 6, 14, 2, 12, 16, 4, 8, 10]
```

Chapter 21. Asyncio

The `asyncio` library is a redesign of how you use threading in python to handle I/O and network related tasks. Previously, in order to implement threading, you would use the `threading` module, which relies on a preemptive task switching mechanism for coordination between threads. In preemptive multitasking, the scheduler is responsible for interrupting the execution of a task and switching to another task. This means that the interpreter can interrupt the execution of a thread and switch to another thread at any point. This can happen when a thread exceeds its time slice or when a higher-priority thread becomes ready to run.

In contrast, `asyncio` relies on cooperative task switching. In this paradigm tasks voluntarily yield control to the scheduler, in this case an event loop. Each task is individually responsible for releasing control back to the scheduler, allowing other tasks to execute. In the `asyncio` library, tasks are represented by coroutines, and they use an `await` keyword to yield control to the event loop. The event loop schedules the execution of coroutines and switches between them based on their current states and the availability of resources.

In general, cooperative task switching is more efficient and less error-prone than preemptive task switching, because it allows tasks to run for as long as they need without interruption. However, it also requires more explicit coordination between tasks, and it can lead to issues such as deadlocks and livelocks if not implemented correctly. On the other hand, preemptive task switching can be more complex to implement and it can lead to race conditions and other synchronization issues, but it can also prevent tasks from monopolizing resources.

`asyncio` is an extensive topic, and not everything will be covered in this chapter. Our focus instead will be how developers can use `asyncio` in the context of backend systems - framework developers are encouraged to read the full spec in python’s official documentation.

Coroutines

In asyncio, tasks are represented by coroutines, which are special types of generator functions that can be paused and resumed. Coroutines are defined using the `async` def keyword, and it can use the await keyword to suspend its execution and allow other tasks to run.

```
>>> import asyncio
>>> async def my_coroutine():
...     print("Hello ", end="", flush=True)
...     await asyncio.sleep(1)
...     print("World!")
...
>>> asyncio.run(my_coroutine())
Hello World!
```

This example demonstrates a simple example of an asynchronous coroutine in Python using the asyncio library. It immediately prints "Hello" to the console. Then the coroutine uses the await keyword to wait for 1 second using the `asyncio.sleep(1)` function. Finally, the coroutine prints "World!" after the delay of 1 second.

It's important to note that when using coroutines, you should avoid blocking operations, such as waiting for file I/O or performing computations, as it will block the event loop and prevent other tasks from running. Instead, you should use non-blocking operations, such as `asyncio.sleep()` or `async with` statements.

Tasks and Task Groups

Multiple coroutines can be aggregated so to run concurrently. To do this, you can use the `asyncio.create_task()` function to aggregate multiple coroutines either within a single coroutine, or using an aggregator such as `asyncio.gather()`, and then use `asyncio.run()` to run the aggregate.

```
>>> import asyncio
>>> async def first_coroutine():
...     await asyncio.sleep(0.25)
...     await asyncio.sleep(0.25)
```

```
...     print("World!")
...
>>> async def second_coroutine():
...     await asyncio.sleep(0.25)
...     print("Hello ", end="", flush=True)
...
>>> async def main():
...     first_task = asyncio.create_task(first_coroutine())
...     second_task = asyncio.create_task(second_coroutine())
...     await first_task
...     await second_task
...
>>> asyncio.run(main())
Hello World!
>>> async def main():
...     await asyncio.gather(first_coroutine(),
second_coroutine(\
))
...
>>> asyncio.run(main())
Hello World!
```

It should be noted that `asyncio.create_task()` uses a `weakref` to the task internally, so if the task returned by the function is deleted, or goes out of scope, it will decrease in reference count to 0 and be garbage collected. In order to avoid this, they should be collected into a collection, and discarded only when the task is done.

In python 3.11, an asynchronous context manager was added for managing groups of asyncio tasks. The `asyncio.TaskGroup()` class defines a `create_task()` method with a signature to match `asyncio.create_task()`. The context manager on exit awaits all the registered tasks.

```
>>> import asyncio
>>> async def first_coroutine():
...     await asyncio.sleep(0.25)
...     await asyncio.sleep(0.25)
...     print("World!")
...
>>> async def second_coroutine():
...     await asyncio.sleep(0.25)
...     print("Hello ", end="", flush=True)
...
>>> async def main():
...     async with asyncio.TaskGroup() as tg:
```

```
...         first_task = tg.create_task(first_coroutine())
...         second_task = tg.create_task(second_coroutine())
...
>>> asyncio.run(main())
Hello World!
```

`ExceptionGroup` and Exception unpacking

In instances where multiple coroutines of a task group raise an exception, the task group bundles the exceptions into an `ExceptionGroup` and raises the group. Exceptions from the group can be selectively handled using an `except*` syntax, which is akin to unpacking exceptions from the group to handle individually.

```
>>> import asyncio
>>> async def first_coroutine(n):
...     await asyncio.sleep(n)
...     print(f"coroutine {n} reporting...")
...
>>> async def second_coroutine():
...     raise AssertionError("raised AssertionError")
...
>>> async def third_coroutine():
...     raise ValueError("raise ValueError")
...
>>> async def main():
...     try:
...         async with asyncio.TaskGroup() as tg:
...             for i in range(3):
...                 tg.create_task(first_coroutine(i))
...             tg.create_task(second_coroutine())
...             tg.create_task(third_coroutine())
...     except* AssertionError as err:
...         aerr, = err.exceptions
...         print(f"AssertionError caught: {aerr}")
...     except* ValueError as err:
...         verr, = err.exceptions
...         print(f"ValueError caught: {verr}")
...
>>> asyncio.run(main())
coroutine 0 reporting...
AssertionError caught: raised AssertionError
ValueError caught: raise ValueError
```

In this example, the `try` block catches both the `AssertionError` and the `ValueError` within a single iteration of the event loop. Both exceptions are grouped together into a single `ExceptionGroup` by the `TaskGroup`, and the `ExceptionGroup` is raised. Tasks which aren't finished by the time the `ExceptionGroup` is raised are cancelled.

The `except*` handler unpacks the exceptions of the exception group and it allow you to handle each exception in isolation from other exceptions caught by the group exception handler. In this case, the `AssertionError` and the `ValueError` are print to the console, and the main block exits.

Part VI. The Underbelly of the Snake

As Python developers, it’s important to have the skills and tools to find and fix bugs, and optimize the performance of our code. As the codebase grows and becomes more complex, it becomes increasingly difficult to identify and fix issues. Additionally, as our respected platforms start to gain users, or start deployed in more demanding environments, the importance of ensuring that it is performing well becomes even more critical.

In this section, we’ll cover the various techniques and tools available for debugging and profiling Python, as well as how to optimize bottlenecks with C extensions.

Chapter 22. Debugging

Debugging Python can be a bit different experience, as compared to debugging code in a statically typed language. In a dynamically typed language like Python, the type of a variable is determined at run-time, rather than at compile-time. This can make it more difficult to catch certain types of errors, such as type errors, before they occur.

However, Python also provides a number of tools and features that can make debugging easier. By taking advantage of these tools and familiarizing yourself with them, you can become more effective at finding and fixing bugs.

pdb

The Python Debugger, also known as `pdb`, is a built-in module that allows you to debug your code by stepping through it line by line and inspecting the state of the program at each step. It is a command-line interface that provides a set of commands for controlling the execution of the program and for inspecting the program's state.

When using `pdb`, you can set breakpoints in your code, which will cause the interpreter to pause execution at that point, and drop you into a debugger REPL. You can then use various commands to inspect the state of the program, such as viewing the values of variables, inspecting the call stack, and seeing the source code. You can also step through your code, line by line, to see how each command mutates state.

```python
# ./closure.py

def my_closure(value):
    def my_decorator(fn):
        def wrapper(*args, **kwargs):
            _enclosed = (fn, value)
            breakpoint()
        return wrapper
    return my_decorator

```

```python
@my_closure("this")
def my_function(*args, **kwargs):
    pass

my_function(None, kwarg=1)
```

In this example we're setting a breakpoint inside the wrapper function of a decorator. The breakpoint will drop us into the `pdb` repl when the interpreter hits this line in execution.

From here, we have a set of commands which we can use to inspect the state of our program. Here are some of the most common:

- `help` - Display a list of available commands or get help on a specific command.
- `list` or `l` - List source code for the current file.
- `where` or `w` - Display the stack trace and line number of the current line.
- `next` or `n` - Execute the current line and move to the next line of code.
- `step` or `s` - Step into the function call at the current line.
- `return` or `r` - Continue execution until the current function returns.
- `break` or `b` - Set a breakpoint at the current line or at a specific line.
- `clear` - Clear all breakpoints, or a breakpoint at a specific line.
- `watch` or `watch expression` - Set a watchpoint on a variable, so that the execution will pause when the value of the variable changes.
- `args` or `a` - Display the argument list of the current function.
- `print` or `p` - Print the value of an expression.
- `up` or `u` - Move up the stack trace.
- `down` or `d` - Move down the stack trace.
- `quit` or `q` - Quit the debugger and exit.

```
root@e08d854dfbfe:~# ls
script.py
root@e08d854dfbfe:~# python ./script.py
--Return--
> /root/script.py(5)wrapper()->None
-> breakpoint()
(Pdb) help

Documented commands (type help <topic>):
```

```
========================================
EOF    c          d        h          list      q        rv
\
undisplay
a      cl         debug    help       ll        quit     s
\
unt
alias  clear      disable  ignore     longlist  r        source
\
until
args   commands   display  interact   n         restart  step
\
up
b      condition  down     j          next      return   tbreak
w
break  cont       enable   jump       p         retval   u
\
whatis
bt     continue   exit     l          pp        run      unalias
\
where

Miscellaneous help topics:
==========================
exec  pdb
(Pdb) help whatis
whatis arg
        Print the type of the argument.
```

In addition, the pdb repl is able to run executable python. This includes builtin functions, comprehensions, etc.

```
(Pdb) list
  1     def my_closure(value):
  2         def my_decorator(fn):
  3             def wrapper(*args, **kwargs):
  4                 _enclosed = (fn, value)
  5  ->             breakpoint()
  6             return wrapper
  7         return my_decorator
  8
  9     @my_closure("this")
 10     def my_function(*args, **kwargs):
 11         pass

(pdb) where
  /root/script.py(13)<module>()
-> my_function(0, kwarg=1)
```

```
> /root/script.py(5)wrapper()->None
-> breakpoint()

(pdb) p args
(0, )

(pdb) p value
'this'

(Pdb) dir(kwargs)
['__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__g\
e__',
'__getattribute__', '__getitem__', '__getstate__', '__gt__', '__h\
ash__',
'__init__', '__init_subclass__', '__ior__', '__iter__', '__le__',\
 '__len__',
'__lt__', '__ne__', '__new__', '__or__', '__reduce__', '__reduce_\
ex__',
'__repr__', '__reversed__', '__ror__', '__setattr__', '__setitem_\
_',
'__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fr\
omkeys',
'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',\
 'values']

(pdb) [x for x in dir(kwargs) if not x.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'pop\
item',
'setdefault', 'update', 'values']

(Pdb) n
--Return--
> /root/script.py(13)<module>()->None
-> my_function(0, kwarg=1)

(Pdb) l
  8
  9     @my_closure("this")
 10     def my_function(*args, **kwargs):
 11         pass
 12
```

```
13  -> my_function(0, kwarg=1)
14
```

Python programs can be run within the context of the python debugger. This allows the debugger to catch unexpected exceptions, dropping you into the debugger REPL at the point where an `Exception` is raised, as opposed to python just printing the stack trace on exit. To do this, run your program via the `pdb` module by executing `python -m pdb ./script.py`.

```python
# ./script.py

def my_closure(value):
    def my_decorator(fn):
        def wrapper(*args, **kwargs):
            _enclosed = (fn, value)
            raise ValueError
        return wrapper
    return my_decorator

@my_closure("this")
def my_function(*args, **kwargs):
    pass

my_function(0, kwarg=1)
```

```
root@e08d854dfbfe:~# python script.py
Traceback (most recent call last):
  File "/root/script.py", line 13, in <module>
    my_function(0, kwarg=1)
  File "/root/script.py", line 5, in wrapper
    raise ValueError
ValueError

root@e08d854dfbfe:~# python -m pdb script.py
> /root/script.py(1)<module>()
-> def my_closure(value):

(Pdb) c
Traceback (most recent call last):
  File "/usr/local/lib/python3.11/pdb.py", line 1774, in main
    pdb._run(target)
  File "/usr/local/lib/python3.11/pdb.py", line 1652, in _run
    self.run(target.code)
  File "/usr/local/lib/python3.11/bdb.py", line 597, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
```

```
  File "/root/script.py", line 13, in <module>
    my_function(0, kwarg=1)
  File "/root/script.py", line 5, in wrapper
    raise ValueError
ValueError
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /root/script.py(5)wrapper()
-> raise ValueError

(Pdb) ll
  3             def wrapper(*args, **kwargs):
  4                 _enclosed = (fn, value)
  5  ->             raise ValueError
```

Other Debuggers

There are several third-party python debuggers in the python ecosystem that are also worth discussing. `Pdb++` (pdbpp) is a drop-in replacement for the built-in `pdb`, so you can use it in the same way you would use `pdb`. pdbpp implements feature enhancements such as syntax highlighting, tab completion, and smart command parsing. As of this writing though its main branch is only compatible with python 3.10.

```
root@e08d854dfbfe:~# python -m pip install pdbpp
Collecting pdbpp
  Downloading pdbpp-0.10.3-py2.py3-none-any.whl (23 kB)
  ...

root@ab2771e522c8:~# python -m pdb script.py
[2] > /root/script.py(3)<module>()
-> def my_closure(value):
(Pdb++)
```

A second optional debugger is the `ipdb` debugger. `ipdb` is similar to `pdb++`, but it is built as a standalone library, rather than a drop-in `pdb` replacement. The main benefit of `ipdb` is that it uses the IPython shell, which is a powerful interactive shell for Python that provides features such as syntax highlighting, tab completion, and better support for multi-lined input. To use `ipdb`, instead of placing a `breakpoint()` in your code, you need to toggle the debugger directly using an import statement like `__import__('ipdb').set_trace()`. Also, to use it as a package, from the terminal you execute its main script via `python -m ipdb ./script.py`.

```
root@6b5606b62e20:~# python -m pip install ipdb
Collecting ipdb
  Using cached ipdb-0.13.11-py3-none-any.whl (12 kB)
  ...

root@6b5606b62e20:~# python -m ipdb script.py
/usr/local/lib/python3.10/runpy.py:126: RuntimeWarning:
'ipdb.__m\
ain__' found in sys.modules after import of package 'ipdb', but
p\
rior to execution of 'ipdb.__main__'; this may result in
unpredic\
table behaviour
  warn(RuntimeWarning(msg))
> /root/script.py(1)<module>()
----> 1 def my_closure(value):
      2     def my_decorator(fn):
      3         def wrapper(*args, **kwargs):

ipdb> c
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/site-
packages/ipdb/__main__.py"\
, line 322, in main
    pdb._runscript(mainpyfile)
  File "/usr/local/lib/python3.10/pdb.py", line 1592, in
_runscri\
pt
    self.run(statement)
  File "/usr/local/lib/python3.10/bdb.py", line 597, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "/root/script.py", line 13, in <module>
    my_function(0, kwarg=1)
  File "/root/script.py", line 5, in wrapper
    raise ValueError
ValueError
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /root/script.py(5)wrapper()
      4             _enclosed = (fn, value)
----> 5             raise ValueError
      6         return wrapper

ipdb> ll
      3         def wrapper(*args, **kwargs):
```

```
      4             _enclosed = (fn, value)
----> 5             raise ValueError
      6         return wrapper

ipdb> l
      7     return my_decorator
      8
      9 @my_closure("this")
     10 def my_function(*args, **kwargs):
     11     pass
     12
     13 my_function(0, kwarg=1)
     14

ipdb>
```

Chapter 23. Profiling

Profiling is the process of measuring the performance of your code and identifying bottlenecks. This allows you to optimize your code and improve its performance. Python provides several built-in libraries and tools for profiling your code. They are designed to hook into audit events such as `call` or `return`, and report how much time elapsed between the auditable events of a given set of instructions. There are also third party libraries available which can analyze other aspects of the runtime, like memory usage.

cProfile

One of the most commonly used tools for profiling Python code is the `cProfile` module. It is a built-in library that generates statistics on the number of calls and the time spent in each function. This information can be used to identify which parts of the code are taking the most time to execute, and make adjustments accordingly.

To use `cProfile`, you can run your script with the command `python -m cProfile ./script.py` and it will output the statistics of the script's execution. You can pass an optional `-s` argument so as to control how the output is sorted; by default the output is sorted by the call count, but can be set to `cumulative` to sort by cumulative time, `ncalls` to sort by the call count, etc. You can also pass `-o ./file.prof` to dump the results to a file, though `-s` and `-o` are mutually exclusive.

```python
import time


def slow_mult(a, b):
    time.sleep(1.1)
    return a * b


def fast_mult(a, b):
    time.sleep(0.1)
    return a * b
```

```
def run_mult(a, b):
    x = slow_mult(a, b)
    y = fast_mult(a, b)
    _abs = abs(x - y)
    return _abs < 0.001


def main():
    a, b = 1, 2
    run_mult(a, b)


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

```
root@3668136f44b5:/code# python -m cProfile -s cumulative
./scrip\
t.py

root@3668136f44b5:/code# python -m cProfile -s cumulative
./scrip\
t.py
         10 function calls in 1.200 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(fun\
ction)
        1    0.000    0.000    1.200    1.200 {built-in method
bu\
iltins.exec}
        1    0.000    0.000    1.200    1.200
script.py:1(<module\
>)
        1    0.000    0.000    1.200    1.200 script.py:19(main)
        1    0.000    0.000    1.200    1.200
script.py:12(run_mu\
lt)
        2    1.200    0.600    1.200    0.600 {built-in method
ti\
me.sleep}
        1    0.000    0.000    1.100    1.100
script.py:3(slow_mu\
lt)
        1    0.000    0.000    0.100    0.100
script.py:7(fast_mu\
```

```
lt)
          1    0.000    0.000    0.000    0.000 {method 'disable'
o\
f '_lsprof.Profiler' objects}
          1    0.000    0.000    0.000    0.000 {built-in method
bu\
iltins.abs}
```

flameprof

The data dump of a `cProfile` run can be used to generate what's called a flame graph. Flame graphs are visual representations of how much time is spent within the scope of a given function call. Each bar in a flame graph represents a function and its subfunctions, with the width of the bar representing the amount of time spent in that function. Functions that take up more time are represented by wider bars, and functions that take up less time are represented by narrower bars. The functions are stacked vertically, with the main function at the bottom and the subfunctions at the top.

The python library `flameprof` can be used to generate flame graphs from the output of `cProfile`. To generate one, first run `cProfile` with the `-o` argument to dump results to a file. Next, use `flameprof` to ingest the dump file. `flameprof` will use that profile to generate an `svg` file. You can open this file in a web browser to see the results.

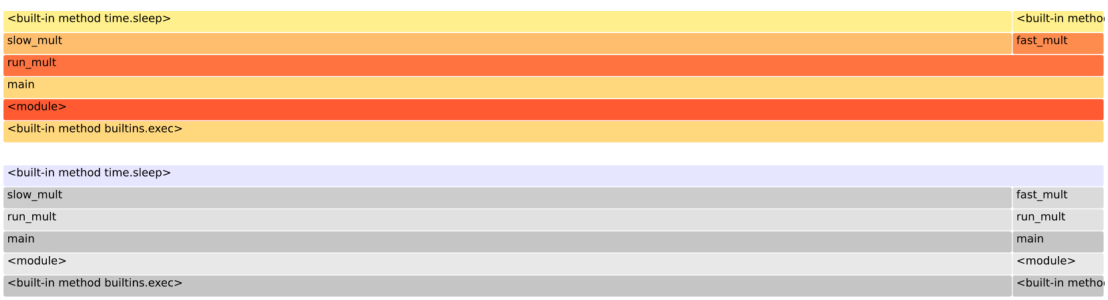
```
root@3668136f44b5:/code# python -m cProfile -o ./script.prof
./sc\
ript.py

root@3668136f44b5:/code# python -m pip install flameprof
Collecting flameprof
  Downloading flameprof-0.4.tar.gz (7.9 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: flameprof
  Building wheel for flameprof (setup.py) ... done
  Created wheel for flameprof: filename=flameprof-0.4-py3-none-
an\
y.whl size=8009
sha256=487bbd51bcf377fa2f9e0795db03b46896d1b41adf\
719272ea8e187abbd85bb5
  Stored in directory:
/root/.cache/pip/wheels/18/93/7e/afc52a495\
```

```
a87307d7b93f5e03ee364585b0edf120fb98eff99
Successfully built flameprof
Installing collected packages: flameprof
Successfully installed flameprof-0.4
WARNING: Running pip as the 'root' user can result in broken
perm\
issions and conflicting behaviour with the system package
manager\
. It is recommended to use a virtual environment instead:
https:/\
/pip.pypa.io/warnings/venv

root@3668136f44b5:/code# python -m flameprof ./script.prof >
scri\
pt.svg

root@3668136f44b5:/code# ls
script.prof  script.py  script.svg
```



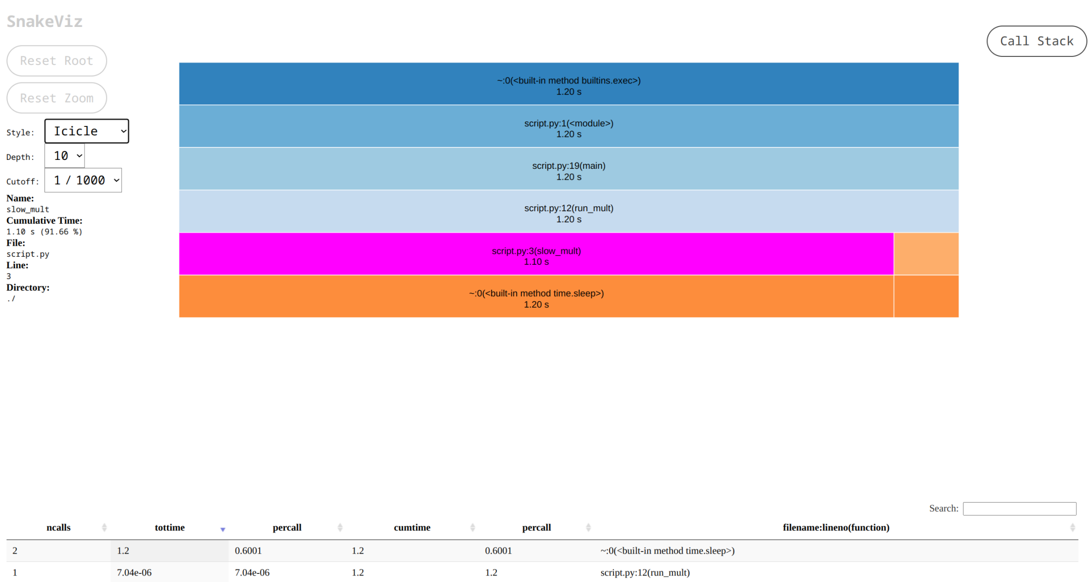
flamegraph

snakeviz

In addition to `flameprof`, the `snakeviz` library can be used to visualize the output of `cProfile`. `snakeviz` is a browser-based tool that takes the profile dump as input and runs a lightweight web server for interactive analysis. In this webpage you can filter data by module, function, file, and sort it by different criteria such as the number of calls or cumulative time spent in a function.

```
root@3668136f44b5:/code# python -m pip install snakeviz
Collecting snakeviz
  Downloading snakeviz-2.1.1-py2.py3-none-any.whl (282 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 282.1/282.1 kB 4.5
```

```
\
MB/s eta 0:00:00
Collecting tornado>=2.0
  Downloading tornado-6.2-cp37-abi3-
manylinux_2_5_x86_64.manylinu\
x1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (423 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 424.0/424.0 kB
17.4\
 MB/s eta 0:00:00
Installing collected packages: tornado, snakeviz
Successfully installed snakeviz-2.1.1 tornado-6.2
WARNING: Running pip as the 'root' user can result in broken
perm\
issions and conflicting behaviour with the system package
manager\
. It is recommended to use a virtual environment instead:
https:/\
/pip.pypa.io/warnings/venv
root@3668136f44b5:/code# python -m snakeviz script.prof
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to
ex\
it
http://127.0.0.1:8080/snakeviz/%2Fhome%2Fmichael%2FProjects%2Ftex\
tbook%2Fprofiling%2Fscript.prof
Opening in existing browser session.
```



snakeviz

`memory_profiler`

`memory_profiler` is a Python library that helps you profile the memory usage of your Python code. To use memory_profiler, you need to install the library, import it into your code, and add the `@profile` decorator to the functions you want to profile. Then, when you run your code, memory_profiler will collect and display memory usage data for each line of the decorated functions.

```
# ./script.py

@__import__("memory_profiler").profile
def main():
    my_list = [257] * (10**6)
    return my_list


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

```
root@3668136f44b5:/code# python ./script.py
Filename: /code/script.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    24     21.1 MiB     21.1 MiB           1
@__import__("memor\
y_profiler").profile
    25                                         def main():
    26     28.5 MiB      7.5 MiB           1       my_list =
[257\
] * (10**6)
    27     28.5 MiB      0.0 MiB           1       return my_list


root@3668136f44b5:/code#
```

Chapter 24. C extensions

C extensions are a powerful tool that allow you to write performance-critical parts of your Python code in C. By writing these performance-critical parts in C, you can greatly improve the speed and efficiency of your Python code, which can be especially useful for large or complex applications.

One of the benefits of using C extensions is that C is much faster than Python, especially for tasks that are computationally intensive or involve heavy manipulation of data. With C extensions, you can take advantage of the performance benefits of C while still being able to use Python for the higher-level logic and user interface parts of your code.

Additionally, C extensions can also be used to interface with existing C libraries and APIs. This allows you to leverage existing libraries and tools that are already available in C, making it easier to build complex systems and applications.

It is important to note that the use of C extensions in Python requires a good understanding of programming in C. If you are unfamiliar with C or are not comfortable with its syntax and concepts, it is recommended that you first learn C before attempting to use C extensions in Python. Writing and using C extensions can be complex and requires a strong understanding of both Python and C, so it is important to have a solid foundation in both languages before diving into this aspect of Python development.

Hello World

To start, we're going to create a new python package with the following folder structure:

```
my_package/
  __init__.py
  hello_world.c
setup.py
```

The `__init__.py` file is an empty file designating `my_package` as a python package. The `hello_world.c` file contains our extension code. The `setup.py` file contains build instructions and metadata for our package. Finally, the `hello_world.c` file defines a set of static constructs which abstract C functions into objects with which the python interpreter knows how to interact.

`hello_world.c`

The following is an example file for the `hello_world.c` extension:

```c
// ./my_package/hello_world.c

#include <stdio.h>

#include <Python.h>

static PyObject* hello_world() {
    puts("Hello World!");
    Py_RETURN_NONE;
}

static char HelloWorldFunctionDocs[] =
    "prints 'Hello World!' to the screen, from C.";

static PyMethodDef MethodTable[] = {
    {
        "hello_world",
        (PyCFunction) hello_world,
        METH_NOARGS,
        HelloWorldFunctionDocs
    },
    {NULL, }
};

static char HelloWorldModuleDocs[] =
    "module documentation for 'Hello World'";

static struct PyModuleDef HelloWorld = {
    PyModuleDef_HEAD_INIT,
    "hello_world",
    HelloWorldModuleDocs,
    -1,
    MethodTable
};
```

```
PyMODINIT_FUNC PyInit_hello_world() {
    return PyModule_Create(&HelloWorld);
}
```

Starting from the bottom of this file; the PyMODINIT_FUNC macro initializes the module, and it returns the module definition to the interpreter. This initialization function should be the only item in your extension file that's not defined as `static`.

`PyModuleDef` is a structure defined in the C/Python API that defines information about a Python module. It's used to specify the module's name, methods, documentation, and other details about the module. The `PyModuleDef` structure is used in the definition of the module's init function, which is called when the module is imported. This structure includes the `PyModuleDef_HEAD_INIT` macro, the name of the module (`"hello_world"`), the module's documentation (`HelloWorldModuleDocs`), the size of the per-interpreter state of the module (if the module does not need to support subinterpreters, this can be set to `-1`), and the `MethodTable` array which indicates the methods that the module provides.

The `PyMethodDef` structure is used in the definition of the module's `PyModuleDef` structure to specify the methods that the module provides. It is a NULL-terminated array of arrays, where each array of the table defines the function name (`"hello_world"`), a function pointer (`hello_world`), a function signature (`METH_NOARGS`), and a docstring. In this case, our `hello_world` C function takes no arguments, so the function signature is `METH_NOARGS`. We'll explore other options for function signatures in a later section.

The C function `hello_world` has a function definition of `static PyObject*`. Even though this is a no-op function, from the perspective of the interpreter functions which return nothing still return the singleton `None`. The macro `Py_RETURN_NONE` handles incrementing the reference count of the `None` object that is ultimately returned. The function itself simply uses the `puts()` C function to print `"Hello World!"` to the console.

Finally, the C extension needs to source the C/Python API. This is done by including the python header file via `#include <Python.h>`. Since it also makes use of the `puts()` function, we include the standard io header file via `#include <stdio.h>`

`setup.py`

Now that we have our C extension written to a file, we need to tell python how to compile this extension into a dynamically linked shared object library. The most straightforward way to do this is to use `setuptools` to handle compilation. This is done using the `setup.py` file, in which we create a collection of `Extension` objects.

Each `Extension` object takes, at minimum, a `name` which python uses for import purposes, and a list of C source files to be compiled into the extension. Since python package imports are relative to the root of the package, it's typically convenient to have the name mimic the import path of the extension file, i.e. `./my_package/hello_world.c` maps to the `my_package.hello_world` import statement. While not strictly necessary, I find this makes it easier for compilation purposes. The filepaths to these C files are relative to the `setup.py` file. The collection is then passed to the `setup()` function under the keyword argument `ext_modules`.

```python
# ./setup.py

import os.path
from setuptools import setup, Extension

extensions = [
    Extension(
        'my_package.hello_world',
        [os.path.join('my_package', 'hello_world.c')]
    )
]

setup(
    name="my_package",
    ext_modules=extensions
)
```

Finally, to use our extension, install the package and call the module function
in Python.

```
root@edc7d7fa9220:/code# ls
my_package  setup.py

root@edc7d7fa9220:/code# python -m pip install -e .
Obtaining file:///code
  Preparing metadata (setup.py) ... done
Installing collected packages: my-package
  Running setup.py develop for my-package
Successfully installed my-package-0.0.0

[notice] A new release of pip available: 22.3.1 -> 23.0
[notice] To update, run: pip install --upgrade pip

root@edc7d7fa9220:/code# python
Python 3.11.1 (main, Jan 23 2023, 21:04:06) [GCC 10.2.1
20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more
informa\
tion.
>>> from my_package import hello_world
>>> hello_world.__doc__
"module documentation for 'Hello World'"
>>> hello_world.hello_world.__doc__
"prints 'Hello World!' to the screen, from C."
>>> hello_world.hello_world()
Hello World!
>>>
```

If you need to make changes to a C extension, you need to recompile. This can be done using the `setup.py` script argument `build_ext`. By default, building a C extension places the shared object file into a `./build` directory. You can pass the argument `--inplace` to have the setup script copy the shared object library from the `./build` directory into your module automatically, and place it next to the location of the `.c` file.

```
// ./my_package/hello_world.c

#include <stddef.h>
#include <stdio.h>

```

```c
#include <Python.h>

static PyObject* hello_world() {
    puts("Hello World! \n- With <3 from C");
    Py_RETURN_NONE;
}

static char HelloWorldFunctionDocs[] =
    "prints 'Hello World!' to the screen, from C.";

static PyMethodDef MethodTable[] = {
    {
        "hello_world",
        (PyCFunction) hello_world,
        METH_NOARGS,
        HelloWorldFunctionDocs
    },
    {NULL, }
};

static char HelloWorldModuleDocs[] =
    "module documentation for 'Hello World'";

static struct PyModuleDef HelloWorld = {
    PyModuleDef_HEAD_INIT,
    "hello_world",
    HelloWorldModuleDocs,
    -1,
    MethodTable
};

PyMODINIT_FUNC PyInit_hello_world() {
    return PyModule_Create(&HelloWorld);
}
```

```
root@edc7d7fa9220:/code# python setup.py build_ext --inplace
running build_ext
building 'my_package.hello_world' extension
creating build
creating build/temp.linux-x86_64-cpython-311
creating build/temp.linux-x86_64-cpython-311/my_package
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/hello_world.c -o bui\
ld/temp.linux-x86_64-cpython-311/my_package/hello_world.o
creating build/lib.linux-x86_64-cpython-311
creating build/lib.linux-x86_64-cpython-311/my_package
gcc -pthread -shared build/temp.linux-x86_64-cpython-
```

```
311/my_packa\
ge/hello_world.o -L/usr/local/lib -o build/lib.linux-x86_64-
cpyth\
on-311/my_package/hello_world.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-
311/my_package/hello_world\
.cpython-311-x86_64-linux-gnu.so -> my_package

root@edc7d7fa9220:/code# python
Python 3.11.1 (main, Jan 23 2023, 21:04:06) [GCC 10.2.1
20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more
informa\
tion.
>>> from my_package import hello_world
>>> hello_world.hello_world()
Hello World!
- With <3 from C
>>>
```

Passing data in and out of Python

In order to pass python objects into C functions, we need to set a flag in the function signature to one of either `METH_O` (indicating a single PyObject), `METH_VARARGS` (including positional arguments only), or `METH_VARARGS|METH_KEYWORDS` (including both positional and keyword arguments).

All objects in Python are PyObject C structs. In order to make use of our python objects in C, the data from python needs to be unwrapped from their PyObject shells and then cast into a corresponding C data type. The primitive python data types have corresponding `Py<Type>_<As/From><Type>` functions for doing this cast operation.

```c
#include <Python.h>

static PyObject* sum(PyObject* self, PyObject* args) {
    PyObject* iter = PyObject_GetIter(args);
    PyObject* item;

    long res_i = 0;
    double res_f = 0;

```

```
    while ((item = PyIter_Next(iter))) {
        if (PyLong_Check(item)) {
            long val_i = PyLong_AsLong(item);
            res_i += val_i;
        }
        else if (PyFloat_Check(item)) {
            double val_f = PyFloat_AsDouble(item);
            res_f += val_f;
        }
        Py_DECREF(item);
    }
    Py_DECREF(iter);

    if (res_f) {
        double result = res_f + res_i;
        return PyFloat_FromDouble(result);
    }
    return PyLong_FromLong(res_i);
}

static PyMethodDef MethodTable[] = {
    {
        "sum",
        (PyCFunction) sum,
        METH_VARARGS,
        "returns the sum of a series of numeric types"
    },
    {NULL, }
};


static struct PyModuleDef MyMathModule = {
    PyModuleDef_HEAD_INIT,
    "math",
    "my math module",
    -1,
    MethodTable,
};

PyMODINIT_FUNC PyInit_math() {
    return PyModule_Create(&MyMathModule);
}
```

In this example, we're creating a `sum()` function which takes variadic arguments of the function call and calculates the sum of the numeric types. We're parsing the arguments lazily, by creating an iterable to loop over. Each item of the iterable is cast to either a `long` or a `double` depending on if its corresponding python type is an `int` or a `float`. Those values are then added to a resultant.

The function's return "type" is dependent on the arguments of the function call. If no `float` types are passed to the sum function, then the returned value is of Python type `int`. If however there were `float` types provided, the entire result is cast to a `double` and then returned as a `float` type.

```
root@edc7d7fa9220:/code# python setup.py build_ext --inplace --fo\
rce
running build_ext
building 'my_package.hello_world' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/hello_world.c -o bui\
ld/temp.linux-x86_64-cpython-311/my_package/hello_world.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/hello_world.o -L/usr/local/lib -o build/lib.linux-x86_64-cpyth\
on-311/my_package/hello_world.cpython-311-x86_64-linux-gnu.so
building 'my_package.math' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/math.c -o build/temp\
.linux-x86_64-cpython-311/my_package/math.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/math.o -L/usr/local/lib -o build/lib.linux-x86_64-cpython-311/\
my_package/math.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-311/my_package/hello_world\
.cpython-311-x86_64-linux-gnu.so -> my_package
copying build/lib.linux-x86_64-cpython-311/my_package/math.cpytho\
n-311-x86_64-linux-gnu.so -> my_package

root@edc7d7fa9220:/code# python
Python 3.11.1 (main, Jan 23 2023, 21:04:06) [GCC 10.2.1 20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more informa\
```

```
tion.

>>> from my_package import math
>>> (val := math.sum(1, 2, 3))
6
>>> type(val)
class 'int'
>>> (val := math.sum(1, 2, 3, 4.0))
10.0
>>> type(val)
class 'float'
>>>
```

Memory Management

Reference counting is one of the core concepts of memory management in Python. It is used to keep track of the number of references to an object in the program and automatically reclaim memory when an object is no longer in use, or when the reference count drops to zero. The rationale behind this approach is to simplify memory management for the interpreter and reduce the risk of memory leaks and other issues associated with manual memory management.

In Python, the reference count of an object is increased whenever a reference to the object is created, and is decremented whenever a reference to the object is deleted. When the reference count of an object reaches zero, the object is slated for deallocation so its memory is freed. In a C extension, it is possible to interact with the reference counting mechanism of Python objects by using the `Py_INCREF` and `Py_DECREF` functions. `Py_INCREF` is used to increment the reference count of an object, indicating that the object is being used in multiple places. `Py_DECREF` is used to decrement the reference count of an object, indicating that the object is no longer needed and can be garbage collected if its reference count reaches zero.

References in C can be either owned or borrowed. If a function call returns an owned reference, you are responsible for one of the counts in the reference count of the object. Failing to properly decrement the reference count will result in a memory leak. In the previous example, `PyObject_GetIter()` and `PyIter_Next()` both returned owned references.

Thus, we call `Py_DECREF()` on these objects when we were done with them, so that they are properly garbage collected. If however a function call returns a borrowed reference, you are equally responsible for not decrementing the reference count of the object. Calling `Py_DECREF()` on a borrowed reference will cause an object to be garbage collected prematurely, leading to dangling pointers.

Parsing Arguments

Positional arguments are passed down to C extension functions in the form of a tuple, and keyword arguments are passed as a dictionary. As such, any of the tuple and dictionary functions in the C/Python API can be used to individually extract `PyObject*` items from their respected collections. However, there are several functions designed to batch-parse these PyObjects into local variables, mainly `PyArg_ParseTuple()` and `PyArg_ParseTupleAndKeywords()`. These functions return *borrowed* references. Meaning, you do not need to increase or decrease the reference count if you only plan to use these references in the context of a specific C function.

Parsing Tuple Arguments

`PyArg_ParseTuple()` will parse a tuple out to local variables. It takes a tuple, in this case the `args` PyObject pointer, a format string that specifies the expected types of the arguments, and variables to store the parsed values. The format string uses codes to indicate the type of each argument. For example, `i` for an integer, `s` for a string, etc. The function returns non-zero if the arguments are successfully parsed and stored in the specified variables, and zero otherwise. If the arguments are invalid, it raises an exception in Python to indicate the error, and the C function should immediately return `NULL`.

```
static PyObject* print_int(PyObject* self, PyObject* args) {
    int val;
    if (!PyArg_ParseTuple(args, "i", &val))
        return NULL;

```

```
    printf("%d\n", val);
    Py_RETURN_NONE;
}
```

```
root@cb4fbbf71628:/code# python -c "from my_package import math; \
math.print_int(1)"
1
root@cb4fbbf71628:/code# python -c "from my_package import math; \
math.print_int(1.0)"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

Parsing Keyword Arguments

`PyArg_ParseTupleAndKeywords()` will parse a tuple/dictionary pair of `args, kwargs` pointers out to local variables. It takes a tuple, a dictionary, a format string for positional parsing, a null-terminated array of keywords to parse from the function call, and finally the variables from where to store the parsed values. If the arguments are invalid, it raises an exception in Python to indicate the error, and the function should immediately return `NULL`. It should be noted that the `kwlist` should include names for all of the arguments, whether or not they are strictly positional vs keyword.

```
static PyObject* print_ints(PyObject* self, PyObject* args, PyObj\
ect* kwargs) {
    int pos;
    int kwd;
    char *kwlist[] = {"pos", "keyword", NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "l$l", kwlist,\
 &pos, &kwd))
        return NULL;

    printf("postional: %d\nkeyword: %d\n", pos, kwd);
    Py_RETURN_NONE;
}
```

```
root@cb4fbbf71628:/code# python setup.py build_ext --inplace --fo\
rce
running build_ext
```

```
building 'my_package.hello_world' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/hello_world.c -o bui\
ld/temp.linux-x86_64-cpython-311/my_package/hello_world.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/hello_world.o -L/usr/local/lib -o build/lib.linux-x86_64-cpyth\
on-311/my_package/hello_world.cpython-311-x86_64-linux-gnu.so
building 'my_package.math' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/math.c -o build/temp\
.linux-x86_64-cpython-311/my_package/math.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/math.o -L/usr/local/lib -o build/lib.linux-x86_64-cpython-311/\
my_package/math.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-311/my_package/hello_world\
.cpython-311-x86_64-linux-gnu.so -> my_package
copying build/lib.linux-x86_64-cpython-311/my_package/math.cpytho\
n-311-x86_64-linux-gnu.so -> my_package

root@cb4fbbf71628:/code# python -c "from my_package import math; \
math.print_ints(pos=1, keyword=2)"
postional: 1
keyword: 2
root@cb4fbbf71628:/code#
```

Creating PyObjects

In the previous section we looked at how we can take PyObjects and parse them into C types for computation. We also need to know how to do the inverse; given a collection of C types, create PyObjects which we can pass back to the interpreter. As mentioned previously, primitive data types can be packaged into PyObject's using their corresponding `Py<Type>_<As/From><Type>` functions. These functions create owned references, which can be returned directly from our C functions.

Primitive data types can also be packaged into collections. Each collection type has a corresponding constructor, `Py<Type>_New()`, which can create a new object of a specified type. Each datatype also has corresponding functions to operate on their respected PyObjects; these functions mostly correspond to the methods of a given type available at the python level - `PyList_Append()` vs. `list.append()`, `PySet_Add()` vs. `set.add`, etc.

```c
static PyObject* new_collections() {
    PyObject* _int = PyLong_FromLong(1);
    PyObject* _float = PyFloat_FromDouble(1.0);

    PyObject* _set = PySet_New(NULL);
    PySet_Add(_set, _int);
    PySet_Add(_set, _float);

    PyObject* _list = PyList_New(0);
    PyList_Append(_list, _set);

    PyObject* _str = PyUnicode_FromString("data");
    PyObject* _dict = PyDict_New();

    PyDict_SetItem(_dict, _str, _list);

    PyObject* _tuple = PyTuple_New(1);
    PyTuple_SetItem(_tuple, 0, _dict);

    return _tuple;
}
```

```
root@cb4fbbf71628:/code# python setup.py build_ext --inplace --fo\
rce
running build_ext
building 'my_package.hello_world' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/hello_world.c -o bui\
ld/temp.linux-x86_64-cpython-311/my_package/hello_world.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/hello_world.o -L/usr/local/lib -o build/lib.linux-x86_64-cpyth\
on-311/my_package/hello_world.cpython-311-x86_64-linux-gnu.so
building 'my_package.math' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
```

```
I/usr/local/include/python3.11 -c my_package/math.c -o
build/temp\
.linux-x86_64-cpython-311/my_package/math.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-
311/my_packa\
ge/math.o -L/usr/local/lib -o build/lib.linux-x86_64-cpython-
311/\
my_package/math.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-
311/my_package/hello_world\
.cpython-311-x86_64-linux-gnu.so -> my_package
copying build/lib.linux-x86_64-cpython-
311/my_package/math.cpytho\
n-311-x86_64-linux-gnu.so -> my_package
root@cb4fbbf71628:/code# python -c "from my_package import
hello_\
world; print(hello_world.new_collections())"
({'data': [{1}]},)
root@cb4fbbf71628:/code#
```

It should be noted that in this example each data type is only being assigned once, if you wish to assign the same object to multiple collections, for example appending the `_float` to the `_list`, as well as adding it to the `_set`, you'll need to increase the reference count.

Importing Modules

In a C extension, you can import other Python modules using the `PyImport_ImportModule()` function from the C/Python API. This function takes a single argument, a string representing the name of the module you wish to import. For example, to import the "math" module, you can assign the module to a variable using `PyObject *math_module = PyImport_ImportModule("math")`. Once the module is locally assigned, accessing objects from the module namespace can be done using `PyObject_GetAttrString()`.

```
root@cb4fbbf71628:/code# python setup.py build_ext --inplace --
fo\
rce
running build_ext
building 'my_package.hello_world' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -
\
```

```
I/usr/local/include/python3.11 -c my_package/hello_world.c -o bui\
ld/temp.linux-x86_64-cpython-311/my_package/hello_world.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/hello_world.o -L/usr/local/lib -o build/lib.linux-x86_64-cpyth\
on-311/my_package/hello_world.cpython-311-x86_64-linux-gnu.so
building 'my_package.math' extension
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -\
I/usr/local/include/python3.11 -c my_package/math.c -o build/temp\
.linux-x86_64-cpython-311/my_package/math.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-311/my_packa\
ge/math.o -L/usr/local/lib -o build/lib.linux-x86_64-cpython-311/\
my_package/math.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-311/my_package/hello_world\
.cpython-311-x86_64-linux-gnu.so -> my_package
copying build/lib.linux-x86_64-cpython-311/my_package/math.cpytho\
n-311-x86_64-linux-gnu.so -> my_package

root@cb4fbbf71628:/code# python -c "from my_package import math; \
print(math.my_sqrt(4))"
2.0
```

Defining New Types

In addition to creating functions, we can create entirely new Python objects using the C/Python API. These objects can define attributes and methods which are written in C, and can be evoked from python code directly. It should be noted that creating Python objects in C is an exhaustive process and we will undoubtedly be unable to cover all the nuances. That being said for the sake of completeness we'll introduce the fundamental concepts, and we suggest to consult the official python documentation for the more esoteric aspects of the craft.

To create a new type, you first need to define a `struct` that represents the type and initialize it using the `PyTypeObject` struct. The `PyTypeObject` struct contains fields such as; `tp_name`, which is the name of the type; `tp_basicsize`, which is the size of the type in bytes; `tp_new`, `tp_init`, and `tp_dealloc`, which are functions that are called when an instance of the type is being allocated, initialized, and destroyed respectively; and both Member and Method tables for defining the attributes and methods which are exposed by our object. Once you have defined the struct, you can create the new type by calling the `PyType_Ready()` function and passing it a reference to the struct. This function initializes the type and makes it available to in the interpreter. Finally, to export the type from the module, you define a module initialization function using the `PyMODINIT_FUNC` macro and call the `PyModule_Create()` function to create a new module. On this object you use the `PyModule_AddObject()` function to insert the custom `PyTypeObject` into the module.

Below is a full working example of a `Person` class, with `first_name` and `last_name` attributes, and a `full_name` method for creating a string that includes both attributes.

```
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first_name;
    PyObject *last_name;
} Person;

static void Person_Destruct(Person* self) {
    Py_XDECREF(self->first_name);
    Py_XDECREF(self->last_name);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static int Person_Init(Person *self, PyObject *args, PyObject
*kw\
args) {

    PyObject *first=NULL, *last=NULL;
    static char *kwlist[] = {"first_name", "last_name", NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "OO", kwlist,
\
&first, &last))
        return -1;
```

```

    PyObject* _first = self->first_name;
    Py_INCREF(first);
    self->first_name = first;
    Py_XDECREF(_first);

    PyObject* _last = self->last_name;
    Py_INCREF(last);
    self->last_name = last;
    Py_XDECREF(_last);

    return 0;
}

static PyObject* Person_FullName(Person* self) {
    if (self->first_name == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first_name is not \
defined");
        return NULL;
    }

    if (self->last_name == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last_name is not d\
efined");
        return NULL;
    }

    return PyUnicode_FromFormat("%S %S", self->first_name, self->\
last_name);
}

static PyMemberDef PersonMembers[] = {
    {
        "first_name",
        T_OBJECT_EX,
        offsetof(Person, first_name),
        0,
        "first name"
    },
    {
        "last_name",
        T_OBJECT_EX,
        offsetof(Person, last_name),
        0,
        "last name"
    },
    {NULL, }
```

```
};

static PyMethodDef PersonMethods[] = {
    {
        "full_name",
        (PyCFunction)Person_FullName,
        METH_NOARGS,
        "return the full name of the Person"
    },
    {NULL, }
};

static PyTypeObject PersonType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "person.Person",                /* tp_name */
    sizeof(Person),                 /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)Person_Destruct,    /* tp_dealloc */
    0,                              /* tp_print */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_reserved */
    0,                              /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash  */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT
     | Py_TPFLAGS_BASETYPE,         /* tp_flags */
    "Person objects",               /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    PersonMethods,                  /* tp_methods */
    PersonMembers,                  /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    (initproc)Person_Init,          /* tp_init */
```

```
    0,                              /* tp_alloc */
    PyType_GenericNew,              /* tp_new */
};

static PyModuleDef PersonModule = {
    PyModuleDef_HEAD_INIT,
    "person",
    "Example module for creating Python types in C",
    -1,
    NULL
};

PyMODINIT_FUNC PyInit_person() {
    if (PyType_Ready(&PersonType) < 0)
        return NULL;

    PyObject* module = PyModule_Create(&PersonModule);
    if (module == NULL)
        return NULL;

    Py_INCREF(&PersonType);
    PyModule_AddObject(module, "Person", (PyObject*)&PersonType);
    return module;
}
```

Now, this is a lot to take in at once, so lets go over each section individually, this time from top to bottom.

```
#include <Python.h>
#include "structmember.h"
```

To start, we include the standard `Python.h` header file. We also need to include the `structmember.h` header file, which includes declarations needed for creating `PyMemberDef` and `PyMethodDef` structs.

```
typedef struct {
    PyObject_HEAD
    PyObject *first_name;
    PyObject *last_name;
} Person;
```

Next, we define a `Person` struct. This struct represents the new type in Python that can be used to model a person. The struct is initialized using the

`PyObject_HEAD` macro for including the standard prefixes for all Python objects in C, and two custom fields, `first_name` and `last_name`, which are pointers to the Python Objects which will represent the first and last name of our person.

```
static void Person_Destruct(Person* self) {
    Py_XDECREF(self->first_name);
    Py_XDECREF(self->last_name);
    Py_TYPE(self)->tp_free((PyObject*)self);
}
```

Next, we define the function that is responsible for freeing up resources of an object when its reference count hits zero. The function takes a single argument, self, which is a pointer to an instance of the Person type. `Py_XDECREF()` decrements the count of the `first_name` and `last_name` PyObjects. It's similar to `Py_DECREF()` but does not error if the function is called on `NULL`. Lastly, we call the `tp_free` function, which is a member of the `ob_type` attribute that is the type struct. This is pointed to by the `Person` instance at `self->ob_base->ob_type`, but the `Py_TYPE()` function can also be used as a shorthand. The `tp_free()` function takes `self` cast as a `PyObject*` and it frees the memory of the object.

```
static int Person_Init(Person *self, PyObject *args, PyObject *kw\
args) {

    PyObject *first, *last;
    static char *kwlist[] = {"first_name", "last_name", NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "OO", kwlist, \
&first, &last))
        return -1;

    PyObject* _first = self->first_name;
    Py_INCREF(first);
    self->first_name = first;
    Py_XDECREF(_first);

    PyObject* _last = self->last_name;
    Py_INCREF(last);
    self->last_name = last;
    Py_XDECREF(_last);

```

```
    return 0;
}
```

Next, we define the initializer, which is used to initialize instances of the `Person` type. The function takes a pointer `self` to the instance of a `Person` type, and PyObject pointers for an `args` tuple and a `kwargs` dictionary. The `args` and `kwargs` are parsed out to the PyObject pointers `first` and `last`, using the `PyArg_ParseTupleAndKeywords()` function. Finally, the `first` and `last` PyObjects are assigned to the `Person` instance. This is done in a specific order; we first take the reference to whatever value `self->first_name` was originally pointing to and assign it to a `temp` value; we then increase the reference count of the new `first` attribute; next we assign the new attribute to the `self->first_name` pointer; and finally, we `Py_XDECREF()` the `temp` value, which could possibly be `NULL`. This ordering is important, as garbage collection on the attribute value should only be allowed after the attribute reference is reassigned to the new value.

```
static PyObject* Person_FullName(Person* self) {
    if (self->first_name == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first_name is not \
defined");
        return NULL;
    }

    if (self->last_name == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last_name is not d\
efined");
        return NULL;
    }

    return PyUnicode_FromFormat("%S %S", self->first_name, self->\
last_name);
}
```

Next, we define a C function for creating a unicode string representing the full name of our person. The function checks if both first_name and last_name attributes of the Person instance have been set, and if not it sets an error using the `PyErr_SetString` function with a message of “first_name is not defined” or “last_name is not defined”. If both attributes are set, it creates a Python Unicode string using the `PyUnicode_FromFormat()` function, which takes a template string and variadic PyObjects.

```
static PyMemberDef PersonMembers[] = {
    {
        "first_name",
        T_OBJECT_EX,
        offsetof(Person, first_name),
        0,
        "first name"
    },
    {
        "last_name",
        T_OBJECT_EX,
        offsetof(Person, last_name),
        0,
        "last name"
    },
    {NULL, }
};

static PyMethodDef PersonMethods[] = {
    {
        "full_name",
        (PyCFunction)Person_FullName,
        METH_NOARGS,
        "return the full name of the Person"
    },
    {NULL, }
};
```

Next, we define two structs which hold definitions of the attributes and methods of the Person type. `PersonMembers` is an array of `PyMemberDef` structs, where each struct of the array defines an attribute of the `Person` type, including their name, type, offset, read/write access, and description. `PersonMethods` is an array of `PyMethodDef` structs, which defines a methods name, function pointer, flag, and description. The `Person_FullName` function is included in this array as a method named “full_name”. Both structs are `NULL` terminated.

```
static PyTypeObject PersonType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "person.Person",                /* tp_name */
```

```c
    sizeof(Person),                 /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)Person_Destruct,    /* tp_dealloc */
    0,                              /* tp_print */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_reserved */
    0,                              /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash  */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT
     | Py_TPFLAGS_BASETYPE,         /* tp_flags */
    "Person objects",               /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    PersonMethods,                  /* tp_methods */
    PersonMembers,                  /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    (initproc)Person_Init,          /* tp_init */
    0,                              /* tp_alloc */
    PyType_GenericNew,              /* tp_new */
};
```

Each of these constituent elements goes into defining a custom `PyTypeObject` named `PersonType`, which defines a python type called `person.Person`. This struct specifies the methods, members and other properties of the type. The type has a `tp_basicsize` of `sizeof(Person)`, indicating the size of the type in bytes, `tp_dealloc` as the destructor method `Person_Destruct`, and the `tp_methods` and `tp_members` as defined in the `PersonMethods` and `PersonMembers` arrays, respectively. The

`PyType_GenericNew` function is used to instantiate new objects of the type - we could provide a custom `tp_new` function to hook into the constructor protocol of this type, similar to how you would overload the `__new__` method in Python, but for this example the generic `PyType_GenericNew` implementation is sufficient.

```c
static PyModuleDef PersonModule = {
    PyModuleDef_HEAD_INIT,
    "person",
    "Example module for creating Python types in C",
    -1,
    NULL
};

PyMODINIT_FUNC PyInit_person() {

    PyObject* module = PyModule_Create(&PersonModule);
    if (module == NULL)
        return NULL;

    if (PyType_Ready(&PersonType) < 0)
        return NULL;

    Py_INCREF(&PersonType);
    PyModule_AddObject(module, "Person", (PyObject*)&PersonType);
    return module;
}
```

Finally, we create a module definition `PersonModule` and instantiate it using `PyModule_Create()` in the module initialization function. Once the module is created, we call the function `PyType_Ready()` for final initialization of our new type, increase it’s reference count, and add it as an object to the module under the name “Person” using `PyModule_AddObject()`. Once all of this is completed, we return the module.

After writing this to a `.c` extension and adding it to our `setup.py` file, we can compile the extension and run the interpreter.

```
root@e94757ea01f2:/code/extensions# python setup.py build_ext --i\
nplace
running build_ext
building 'my_package.person' extension
```

```
gcc -pthread -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -fPIC -
\
I/usr/local/include/python3.11 -c my_package/person.c -o
build/te\
mp.linux-x86_64-cpython-311/my_package/person.o
gcc -pthread -shared build/temp.linux-x86_64-cpython-
311/my_packa\
ge/person.o -L/usr/local/lib -o build/lib.linux-x86_64-cpython-
31\
1/my_package/person.cpython-311-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-
311/my_package/person.cpyt\
hon-311-x86_64-linux-gnu.so -> my_package
root@e94757ea01f2:/code/extensions# python
Python 3.11.1 (main, Jan 23 2023, 21:04:06) [GCC 10.2.1
20210110]\
 on linux
Type "help", "copyright", "credits" or "license" for more
informa\
tion.
>>> from my_package import person
>>> my_person = person.Person(first_name="John",
last_name="Smith\
")
>>> my_person.full_name()
'John Smith'
>>>
```

`Stack` type

Let’s look at another example. Here we’re going to be implementing a basic `stack` type with the methods `push()` and `pop()`. These methods will operate on an internal memory allocation of `PyObject*`’s, adding and removing items as necessary. We’ll also define a `length` attribute to show the current size of the data structure.

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    size_t length;
    PyObject* pop;
    PyObject* push;
    PyObject** _data;
} Stack;
```

```

static PyObject* Stack_Push(Stack* self, PyObject* item) {
    size_t len = self->length + 1;
    self->_data = realloc(self->_data, len*sizeof(PyObject*));
    Py_INCREF(item);
    self->_data[self->length] = item;
    self->length = len;
    Py_RETURN_NONE;
}

static PyObject* Stack_Pop(Stack* self) {
    if (self->length == 0)
        Py_RETURN_NONE;
    long len = self->length - 1;
    PyObject* item = self->_data[len];
    self->_data = realloc(self->_data, len*sizeof(PyObject*));
    self->length = len;
    return item;
}

static PyObject* Stack_New(PyTypeObject* type, PyObject* args, Py\
Object* kwargs) {
    Stack* self = (Stack*)type->tp_alloc(type, 0);
    if (!self)
        return NULL;
    self->length = 0;
    self->_data = malloc(0);
    return (PyObject*)self;
}

static void Stack_Destruct(Stack* self) {
    for (size_t i = 0; i < self->length; i++)
        Py_DECREF(self->_data[i]);
    free(self->_data);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyMemberDef StackMembers[] = {
    {"length", T_LONG, offsetof(Stack, length), READONLY, "stack \
length"},
    {NULL, }
};

static PyMethodDef StackMethods[] = {
    {"push", (PyCFunction)Stack_Push, METH_O, "push item to the s\
tack"},
    {"pop", (PyCFunction)Stack_Pop, METH_NOARGS, "pop an item to
```

```
\
the stack"},
    {NULL, }
};

static PyTypeObject StackType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "stack.Stack",                 /* tp_name */
    sizeof(Stack),                 /* tp_basicsize */
    0,                             /* tp_itemsize */
    (destructor)Stack_Destruct,    /* tp_dealloc */
    0,                             /* tp_print */
    0,                             /* tp_getattr */
    0,                             /* tp_setattr */
    0,                             /* tp_reserved */
    0,                             /* tp_repr */
    0,                             /* tp_as_number */
    0,                             /* tp_as_sequence */
    0,                             /* tp_as_mapping */
    0,                             /* tp_hash  */
    0,                             /* tp_call */
    0,                             /* tp_str */
    0,                             /* tp_getattro */
    0,                             /* tp_setattro */
    0,                             /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT
      | Py_TPFLAGS_BASETYPE,       /* tp_flags */
    "Stack objects",               /* tp_doc */
    0,                             /* tp_traverse */
    0,                             /* tp_clear */
    0,                             /* tp_richcompare */
    0,                             /* tp_weaklistoffset */
    0,                             /* tp_iter */
    0,                             /* tp_iternext */
    StackMethods,                  /* tp_methods */
    StackMembers,                  /* tp_members */
    0,                             /* tp_getset */
    0,                             /* tp_base */
    0,                             /* tp_dict */
    0,                             /* tp_descr_get */
    0,                             /* tp_descr_set */
    0,                             /* tp_dictoffset */
    0,                             /* tp_init */
    0,                             /* tp_alloc */
    Stack_New,                     /* tp_new */
};

static PyModuleDef StackModule = {
    PyModuleDef_HEAD_INIT,
    "stack",
```

```
    "module for custom stack object",
    -1,
    NULL
};

PyMODINIT_FUNC PyInit_stack() {
    if (PyType_Ready(&StackType) < 0)
        return NULL;

    PyObject* module = PyModule_Create(&StackModule);
    if (!module)
        return NULL;

    Py_INCREF(&StackType);
    PyModule_AddObject(module, "Stack", (PyObject*)&StackType);
    return module;
}
```

It should be noted that this example reallocates the `_data` structure on each item write; this is not optimized for purposes of brevity.

In this example, we define a `Stack_New()` function instead of using the standard `PyType_GenericNew`. This allows us to customize the instantiation of our Stack struct, setting the default values for the `_data` and `length` fields. Since `Stack()` won't take any initialization arguments, we set `tp_init` to 0. We also use `T_LONG` as the implementation function in the `PyMemberDef` struct to automatically cast the field value to a Python `int` when the struct field is requested from the interpreter. We also set this attribute as `READONLY` so as to prevent its value from being reassigned.

Since this object is responsible for its own data collection, it's worth talking a moment to look at it's implementation in finer detail. Specifically, the `push()` and `pop()` methods of the struct, as well as the `Stack_Destruct` implementation that is called at garbage collection.

When an item is passed to the `Stack` during a push operation, this PyObject* is being passed as a *borrowed* reference. Since we are writing a copy of this reference to `_data`, we need to explicitly inform the interpreter that the `Stack` owns a reference, and until `Stack` releases this reference, the item should not be garbage collected. We do this by increasing the reference count.

When an item is popped from the `Stack`, we can simply return it without decreasing the reference count, because we are returning an *owned* reference to the caller.

Finally, when the `Stack` is deallocated, any owned reference in the collection of PyObject*'s must be dereferenced; if this is not done then the reference count of each item in the stack will never hit zero, resulting in memory leaks. This is done by iterating over the length of the `_data` array and calling `Py_DECREF` on each of the items. Only after this is done can you call `tp_free` to delete the `Stack` object.

Debugging C extensions

Similar to `pdb`, it's possible to stop the execution of the python interpreter inside c extensions using a C debugger like `gdb`. `gdb` allows you to find and fix errors in C code by providing you with information about the state of the extension while it is running.

To best use this, it's important to first compile extensions without optimizations. By default, python compiles extensions with an `-O3` optimization flag; this is good for production but can result in objects being optimized out of the extension. To compile without optimizations, set the `CFLAGS` environment variable to `-O0` during the extension build step.

```
root@be45641b03f3:/code/extensions# CFLAGS="-O0" python setup.py \
build_ext --inplace --force
running build_ext
...
```

Once the C extension is compiled, use gdb to start the python interpreter. It should be noted that `gdb` requires a binary executable, so modules (for example `pytest`) and scripts need to be invoked from the `python` executable as either a module or a script.

```
root@be45641b03f3:/code/extensions# gdb --args python script.py
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licens\
es/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from python...
(gdb)
```

Once in the `gdb` shell, we can do things such as set breakpoints, inspect the call stack, observe variables, and run our program.

```
(gdb) b Stack_Push
Function "Stack_Push" not defined.
Make breakpoint pending on future shared library load? (y or [n])\
 y
Breakpoint 1 (Stack_Push) pending.
(gdb) run
Starting program: /usr/local/bin/python script.py
warning: Error disabling address space randomization: Operation n\
ot permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_\
db.so.1".

Breakpoint 1, Stack_Push (self=0x7f0d84d2bde0, item=0) at my_pack\
age/stack.c:13
13          size_t len = self->length + 1;
(gdb) b 17
Breakpoint 2 at 0x7f0d84adb25c: file my_package/stack.c, line 17.
(gdb) c
```

```
Continuing.

Breakpoint 2, Stack_Push (self=0x7f0d84d2bde0, item=0) at
my_pack\
age/stack.c:17
17          self->length = len;
(gdb) p len
$1 = 1
(gdb) l
12      static PyObject* Stack_Push(Stack* self, PyObject* item)
{
13          size_t len = self->length + 1;
14          self->_data = realloc(self->_data,
len*sizeof(PyObjec\
t*));
15          Py_INCREF(item);
16          self->_data[self->length] = item;
17          self->length = len;
18          Py_RETURN_NONE;
19      }
20
21      static PyObject* Stack_Pop(Stack* self) {
(gdb) p *self
$2 = {ob_base = {ob_refcnt = 2, ob_type = 0x7f0d84ade140
<StackTy\
pe>}, length = 0,
  _data = 0x563ee09d7000, push = 0x0, pop = 0x0}
(gdb) p *item
$3 = {ob_refcnt = 1000000155, ob_type = 0x7f0d8564b760
<PyLong_Ty\
pe>}
<b>(gdb) p (long)PyLong_AsLong(item)</b>
$4 = 0
(gdb)
```