Python 1o1 tutorial

Release 0.1

Jan 07, 2022

Contents

Our world is changing.
We’re not alone anymore.
We work with a new intelligence.
We have the technology.

Machines work faster than we do.
They don’t sleep.
They don’t make mistakes.

I can talk to machines.
I tell them what to do.
I speak their language.
I code.

CHAPTER 1

Before anything

This is a collection of bite-sized lessons for learning python programming even if you don't have any programming experience.

If you're a newcommer into the world of programming languages, you are either:

- *lucky* that you chose the easiest way in
- very well *documented* before making the right choice
- or, you have *really good friends* that care for you

In any case you just need to tackle these small lessons step by step so you can enjoy the warm pleasant feeling of python programming.

CHAPTER 2

The plan

2.1 About Python

Python is one of those rare languages which can claim to be both simple and powerful. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

I will discuss most of these features in more detail in the next section.

2.1.1 The story behind the name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

2.1.2 Features of Python

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

Easy to Learn

As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.

Free and Open Source

Python is an example of a FLOSS (Free/Libré and Open Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like Kivy to create games for your computer and for iPhone, iPad, and Android.

Interpreted

A program written in a **compiled** language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just run the program directly from the source code. Internally, Python converts the source code into an intermediate form called *bytecodes* and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc.

This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In procedure-oriented languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In object-oriented languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use it from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give scripting capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions,documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the *Batteries Included* philosophy of Python.

Besides the standard library, there are various other high-quality libraries which you can find at the Python Package Index.

2.1.3 Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

Python 3 versus 2

You can ignore this section if you're not interested in the difference between "Python version 2" and "Python version 3". But please do be aware of which version you are using. This tutorial is written for Python version 3.

Remember that once you have properly understood and learn to use one version, you can easily learn the differences and use the other one. The hard part is learning programming and understanding the basics of Python language itself. That is our goal in this book, and once you have achieved that goal, you can easily use Python 2 or Python 3 depending on your situation.

2.2 First Steps

It has become a programmer tradition that when you learn a new programming language, you start with a simple program that only says "hello world".

A way to show the world that we are civil.

We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to *write* and then *run* Python programs.

There are two ways of using Python to execute your program:

- using the interactive interpreter prompt, or
- using a source file.

We will now see how to use both of these methods.

2.2.1 Using The Interpreter Prompt

First we need to verify if python is installed on our computer by opening the `terminal` application (for Windows users, type Win+R and then *cmd*).

Type in *python3* and see what happens

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 07:35:52)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

If you see a version number and a >>> prompt, you're good to go. But if you see an error, you need to download and install python. I recommend to visit the python comunity website and download the latest python3 version.

Let's get more familiar with the python interpreter. The easiest way you can interact with the python interpreter is through this console. Here you can *talk* directly to the core of python and send your request.

Once you have started Python, you should see >>> where you can start typing stuff. This is called the Python interpreter prompt.

At the Python interpreter prompt, type:

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

followed by the [enter] key. You should see the words `Hello World` printed on the screen.

Here is an example of what you should be seeing, when using a Mac OS X computer. The details about the Python software will differ based on your computer, but the part from the prompt (i.e. from >>> onwards) should look the same regardless of the operating system you are using.

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 07:35:52)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

Notice that Python gives you the output of the line immediately! What you just entered is a single Python statement. We use print to (unsurprisingly) print any value that you supply to it. Notice, that we are supplying the text "Hello World" and this is promptly printed to the screen.

How to Quit the Interpreter Prompt

If you are using a GNU/Linux or OS X shell, you can exit the interpreter prompt by pressing [ctrl + d] or entering exit() (note: remember to include the parentheses, ()) followed by the [enter] key.

If you are using the Windows command prompt, press [ctrl + z] followed by the [enter] key.

Python as an interactive calculator

To get your feet wet with Python you can use the Python interpreter as a calculator. You have the usual mathematical operators at your disposal, like

- + addition,
- − subtraction,

- `*` multiplication,
- `/` division,
- `**` exponent,
- `//` integer division, and
- `%` modulus.

If you are not familiar with one of them just give it a try in the Python interpreter – python does not limit you to integer numbers, feel free to try also floating point numbers or even strings.

You can also use brackets as you would use them in mathematical expressions.

Can you find out whether Python uses the proper mathematical rules with regards to the order of execution of the operators.

For example:

Listing 1: why don't you try some numbers or an expression, or some text, be creative

```
Python 3.7.2 (default, Dec 27 2018, 07:35:52)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> 10
10
>>> 10 + 10
20
>>> 10 + 2 * 3
16
>>> 'hello'
'hello'
>>> 1/2
0.5
>>> 1 + 2)
  File "<stdin>", line 1
    1 + 2)
         ^
SyntaxError: invalid syntax
>>> _
```

Observe that python tries to *understand* what we type, be it numbers, arithmetic operations or strings, computes the result and prints it back for us. However, if we accidentally type something that doesn't make sense, *Python* will do its best to point out the problem, but it will let us decide how and when we'll fix it.

The `print()` function

When you are typing expresions in the python interpreter (remember the >>> prompt?) they are evaluated and the result is being printed on the screen for us. But when you are running the python program (see below) you need to be explicit to python about when to display an actual value or an expression.

The function you used in your first program, the `print(...)` function, behaves almost the same as the interpreter, it looks at our input, be it numbers, text, expression or even other functions, tries to understand it, evaluates tem and then shows the result on the screen.

2.2.2 Using a source file

Besides the interactive Python interpreter you can also write Python programs (sometimes called scripts). A python program is a file that contains a list of python *expressions* or *statements* that can be executed from the command line. A script can be really simple e.g. searching a text inside a file or it could be as complex as a car crash simulation.

Now we'll say *hello* using a python program instead of the interactive console, so exit python by typing `quit()`

```
>>> quit()
macbook$ _
```

Create a folder somewhere on your computer `python_lessons`, open your favourite text editor and type in the same command we used earlier `print("Hello world!")` then save your file and call it `01_hello_world.py`

All python programs are files whose names end in **.py**.

The **01_** at the beginning is so all our files created in this tutorial are nicely sorted in a single folder.

```
macbook ~ $ mkdir python_lessons
macbook ~ $ cd python_lessons
macbook python_lessons $
```

And finally, execute this program by typing in **python3** followed by the name of our file **01_hello_world.py**.

If you did everything correctly then your console should look like this:

```
macbook python_lessons $ python 01_hello_world.py
Hello world!
macbook python_lessons $ _
```

You got lucky, you just wrote your *first python program*.

2.2.3 Choosing and editor

Now, that you've entered the world of python programming you may want to take it to the next level by using a text editor that was designed for editing python programs.

Even though there are many options available, I recommend you these two:

---

sublime text

is easier to use, it has all the features you may want and you don't event know them yet; it is light, fast and has a beautiful color theme. Long story short: it is *sublime*.

You can get it from the sublimetext website

And if you need any help have a look a this video tutorial.

---

pycharm

is just a bloated version of *sublime text* with the advantage that you could execute your programs directly in the editor or just line-by-line (read as *debug your code*) which could be useful when learning python to understand how complex structures are being executed.

You can get it from jetbrains website.

It also has an educational version.

2.2.4 Exercise

Write a program that given the radius of a circle it computes the circumference and the area. You will need to use the `pi`, so please define it as:

```
pi = 3.141592653589793
```

The End

2.3 Numbers

Among other many other things, Python understands numbers. Let's open the python interpretor console and try them out.

```
Python 3.7.2 (default, Dec 27 2018, 07:35:52)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10
10
>>> 10 + 5
15
>>> 1.5 * 2
3.0
>>>
```

In Python 3 there are three types of numbers, in Python 2 there are four types of number. And if you're using C++ then you will be glad that you're learning Python.

You can inspect the type of any number, variable, string or even expressions by using the function `type()`.

2.3.1 int

The first type of numbers in python is `int`, which is short for *integer* or numbers without a decimal point.

```
Python 3.7.2 (default, Dec 27 2018, 07:35:52)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 42 # is an integer
42
>>> type(42)
<class 'int'>
>>> perfect_number = 28     # let's assign the value 28 to a variable named perfect_
↪number
>>> type(perfect_number)
<class 'int'>
>>> perfect_number
28
>>> # or, if you love typing, you can print it explicitly
... print(perfect_number)
28
>>> _
```

Tip: The hash-tag (**#**) is how your write comments in Python. Everything after a hashtag `#` is ignored by the interpreter. This can be extremely useful to explain something in your code which may not be obvious for your colleagues.

In Python 3 you don't have to worry about the size of an integer. You can create integers of any size provided your computer is large enough store it.

No size limits, no overflows, no worries.

2.3.2 float

The second type of numbers in Python 3 is *float*. This is how decimal values are stored. To make a float, just type in a number that has a decimal point. We'll use the famous number *e* which is approximately 2.718281828.

To confirm that this is a float, look at its type:

```
>>> e = 2.718281828
>>> type(e)
<class 'float'>
>>> e
2.718281828
>>>
```

2.3.3 complex

The third type of numbers in python is *complex numbers*.

Complex numbers are an extension of the familiar real number system in which all numbers are expressed as a sum of a *real* part and an *imaginary* part.

Imaginary numbers are real multiples of the *imaginary unit* (the square root of -1), often written *i* in mathematics or *j* in engineering.

Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with the letter *j* as suffix, e.g. `3+1j`.

Again, you can confirm that this is a complex number by checking its type:

```
>>> z = 2 + 3j
>>> z
(2+3j)
>>> type(z)
<class 'complex'>
>>> _
```

You can also display the real and imaginary parts separetely. To access the real part you can use the `.real` property (notice the dot) and you can access the imaginary part through the `.imag` property.

```
>>> z = 2 + 3j
>>> z.real
2.0
>>> z.imag
3.0
>>> _
```

Did you noticed that even that the real part is a float while we typed it as an integer. This is because Python stores the real and imaginary part of complex numbers as floats.

Now that you know about `int`, `float` and `complex` numbers you are ready to tackle arithmetic.

Just remember:

- the mathematical *i* is called *j*,
- complex numbers are made of *floats*
- the `type()` function can tell you what type of numbers you really have

2.3.4 Type conversion

Whenever you write an arithmetic expression Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to convert a number explicitly from one type to another:

- type `int(x)` to convert x to a plain integer
- type `float(x)` to convert x to a floating-point number
- type `complex(x)` to convert x to a complex number with real part x and imaginary part zero
- type `complex(x, y)` to convert x and y to a complex number with real part **x** and imaginary part **y**, where **x** and **y** can be numeric expressions

2.4 Strings

When programming, text is represented by strings. In Python there are many ways to create a string, after all there are many different types of text. Some have apostrophes, some have quotation marks and some long strings have new lines.

Your goal today: make strings in three different ways.

2.4.1 Create strings

First, we create a string a store it in a variable:

```
message = "Meet me tonight."
```

This stores the string *Meet me tonight.* in a variable called *message*. You do not have a type a semicolon at the end like you do in many other languages like Java or C++. You can see the string stored in *message* by printing it:

```
>>> message = "Meet me tonight."
>>> print(message)
Meet me tonight.
>>> _
```

We've created our first string using quotation marks, but you can also create strings using single quotes:

```
>>> another_message = 'The clock strikes at midnight.'
```

Let's verify the text stored in this new variable by printing it:

```
>>> print(another_message)
The clock strikes at midnight.
```

So you can create strings using single quotes (') or double quotes ("), we did not had to specify that *message* or *another_message* were strings, we just assign them strings and Python was smart enough to know that the variables should hold strings. But what's the point of this, to have more than one way to make a string?

Well, suppose you want to make a string and one of the words had an apostrophe, if you try to make the string using single quotes you'll get a syntax error

```
>>> message2 = 'I'm looking for someone to share an adventure.'
  File "<stdin>", line 1
    message = 'I'm looking for someone to share an adventure.'
                 ^
SyntaxError: invalid syntax
>>>
```

Do you see the problem ? When Python encountered the apostrophe in the word *I'm* it thought that it was the end of the string so the remaining text caused the error. There are two ways to fix this.

One way is to escape the apostrophe, by putting a backslash () in front of it.

```
>>> message2 = 'I\'m looking for someone to share an adventure.'
>>>
```

This tells Python that the single quote is to be treated as a single character and not the end of the string, but the savvy way is to create the string using quotation marks:

```
>>> message3 = "I'm looking for someone to share an adventure."
>>>
```

No errors, no escape characters.

If you make a string using double quotes but your text contains a quotation mark you get another error:

```
>>> message4 = "The phrase "Beam me up, Scotty!" was never said on Star Trek."
  File "<stdin>", line 1
    message4 = "The phrase "Beam me up, Scotty!" was never said on Star Trek."
                               ^
SyntaxError: invalid syntax
>>>
```

This is because Python interprets the quotation mark before the word *Beam* as the end of the string. We can avoid this error by using single quotes to make the string.

```
>>> message4 = 'The phrase "Beam me up, Scotty!" was never said on Star Trek.'
>>>
```

But how do you make more complicated strings, which may contain apostrophes and quotation marks? For this case you cand begin and end the string using triple quotes. You can use three double quotes or three single quotes. We'll use double quotes:

```
>>> movie_quote = """One of my favourite lines from The Godfather is:
... "I'm going to make him an offer he can't refuse."
... Do you know who said this?"""
>>>
```

This text has single quotes, double quotes and even new lines.

Did you notice?

The triple dots which appeared while typing this? That's how Python tells you the command you're typing is taking more than one line.

So you can create strings in Python using single quotes, double quotes or triple quotes. This makes it easy to store all kinds of texts without having to resort to trickery and you can quote me on that.

2.4.2 Operators

You have already seen the operators + and * applied to numbers. These two operators can be applied on strings as well.

The + operator

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
>>> s = 'foo'
>>> t = 'bar'
>>> u = 'baz'
>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'
>>> print('Go team' + '!!!')
Go team!!!
```

The * operator

The * operator creates multiple copies of a string. If s is a string and n is an integer, either of the following expressions returns a string consisting of n concatenated copies of s:

```
>>> s = 'foo.'
>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

The multiplier operand n must be an integer. You'd think it would be required to be a positive integer, but amusingly, it can be zero or negative, in which case the result is an empty string:

```
>>> 'foo' * -8
''
```

If you were to create a string variable and initialize it to the empty string by assigning it the value 'foo' * -8, anyone would rightly think you were a bit daft. But it would work.

The in Operator

Python also provides a membership operator that can be used with strings. The in operator returns True if the first operand is contained within the second, and False otherwise:

```
>>> s = 'foo'
>>> s in "That's food for thought."
True
>>> s in "That's good for now."
False
```

There is also a not in operator, which does the opposite:

```
>>> 'z' not in 'abc'
True
>>> 'z' not in 'xyz'
False
```

2.4.3 Built-in functions

Python provides many functions that are built-in to the interpreter and always available. Here are just a few that work for strings:

- `len()` returns the length of a string
- `str()` returns a string representation of an object
- `lower()` converts alphabetic characters to lowercase
- `upper()` converts alphabetic characters to uppercase

len()

Returns the length of a string.

With **len()**, you can check Python string length. **len(s)** returns the number of characters in s:

```
>>> s = 'I am a programmer.'
>>> len(s)
18
```

str()

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string. **str(x)** returns the string representation of variable or expression **x**:

```
>>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3.21 + 29)
'32.21'
>>> str('to the moon and back')
'to the moon and back'
```

lower()

Given a variable named s holding a string, by typing `s.lower()` you will get a copy of s with all alphabetic characters converted to lowercase:

```
>>> s = "NYSE News: What happened to Google stocks price?"
>>> s.lower()
'nyse news: what happened to google stocks price?'
```

upper()

`s.upper()` returns a copy of **s** with all alphabetic characters converted to uppercase:

```
>>> s = "I want an expresso."
>>> s.upper()
'I WANT AN EXPRESSO.'
```

A line of text in all caps looks like someone is yelling.

2.4.4 f-Strings

Also called *formatted string literals*, f-strings are string literals that have an **f** at the beginning and *curly braces* inside containing expressions that will be replaced with their values.

Here are some of the ways f-strings can make your life easier.

```
>>> name = "Eric"
>>> age = 24
>>> f"Hello, {name}. You are {age} years old or {age * 12} months old."
'Hello, Eric. You are 24 years old or 288 months old.'
```

Look how easy it is to read or predict how it will look.

2.5 Logical operations

Computers are funny creatures. They think in terms of *1*s and *0*s, *True* and *False*. While Python has several numeric types, there is only one logical type: *boolean*. A boolean can only take two values: **True** or **False**. And this is all you need, **if** you are logical…

2.5.1 Bool type

Booleans are a built-in data type in Python. Take care to note that *True* and *False* are both capitalized.

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

If you type it incorrectly, you will receive a name error.

Booleans are commonly encountered when comparing two objects. For example, to compare two numbers use the *double equals* (==) operator:

```
>>> a = 3
>>> b = 5
>>> a == b
False
>>>
```

We get *False* since **a** and **b** are different integers. Notice that you use one equal sign to assign numbers to a variable, and a *double equal* sign to compare them. In addition to testing if two numbers are the same you can test if they are different usint the *not equal* operator.

```
>>> a != b
True
>>>
```

This comparison returns *True* because it is true that *a does not equal b*. By the way, the exclamation mark is commonly used as a logical **not** symbol in programming languages. So this symbol literally reads as **not equal**.

And finally, in addition to comparing two numbers for equality or inequality, you can test to see if one is larger than the other.

```
>>> a > b
False
>>> a < b
True

>>>
```

Is *a* greater than *b* ? No. This is a false statement.

Is *a* less than *b* ? Yes. This is a true statement.

Falsy values

If you inspect the type of *True* and *False*:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>>
```

You see the type is **bool**. This suggests another way to create booleans: by passing values to the boolean constructor. For example, let's convert some numbers to booleans:

```
>>> bool(28)
True
>>> bool(-2.71828)
True
>>> bool(0)
False
>>>
```

In Python, 0 is converted to *False*, while every other number is converted to *True*.

We can also convert strings to booleans. For example,

```
>>> bool("Turing")
True
>>> bool(" ")
True
>>> bool("")
False
>>>
```

In Python, the empty string is converted to *False*, while every other string is converted to *True*.

This is a general principle in Python. When converting a value to a boolean, trivial values are converted to *False*, while non-trivial values are converted to *True*.

Just as you can convert objects to booleans you can convert booleans to other types of objects:

```
>>> str(True)
'True'
>>> str(False)
'False'
```

If you convert *True* to a string, it returns **"True"**, but notice this is surrounded by quotes, so it is a string. The boolean value does not have quotes.

You can also convert booleans to numbers. If you convert *True* to an integer, you get **1** and if you convert *False* to an integer you get **0**.

```
>>> int(True)
1
>>> int(False)
0
>>> 5 + True
6
>>>
```

Look what happens if you add a number and a boolean: Python recognizez that you are trying to add *True* to an integer, so it first converts it to an integer then adds. What do you think **10 * False** will be ?

```
>>> 10 * False
0
>>>
```

Like before, Python recognizez you are trying to perform an arithmetic operation so it converts *False* to the number 0 then multiplies. Is this something you will use ? Probabily not. But it does highlight that Python treats 1 as True and 0 as False, and vice-versa. In computer science, this is a fundamental fact.

2.5.2 If statement

When coding in Python you will frequently encounter a fork in the road, depending on the values of certain data you may want to go in one direction or the other, there may be even more than two directions for you to choose, the if-then-else statements help you navigate these situations:

> **If** you want to learn more:
> **then** continue reading this lesson
> **else** try the next lesson.

if-then

In this example we will collect a string then test its length to see if it has at least 8 characters. This is something you may need to do when validating new passwords.

To begin, create a file called *if_then.py* and type in the following.

```
password = input("Please enter a test string: ")

if len(password) < 8:
        print("Your string is too short.")
        print("Please enter a string with at least 8 characters.")
```

The first line uses the `input()` function which will prompt the user to enter a string and then store the value in the variable `password`.

Next we use the **if-then** command to see if the length of the provided string is less than 8 characters long. If *true* the following indented lines are executed, if *not* these lines are skipped.

Note: Did you noticed that the **if** line ends with a colon (**:**) and the following lines are indented. This is how you identify *a code-block* in Python. This is a big difference from other languages such as Java or C++.

In those languages indentation does not matter and you group code with braces, in Python you start a new code-block with a colon and group the commands with indentation. You can use any size indentation as long as the commands line up.

Now save this file and run the program from the command line and enter the word *magic*.

```
$ python3 if_then.py
Please enter a test string: magic
Your string is too short.
Please enter a string with at least 8 characters.
$ _
```

Because this word has fewer than 8 characters the **if** statement is true (*the length is less than 8*) so the following code-block is executed and the two lines are printed.

Run the program again and enter a longer string like *fantastic*.

```
$ python3 if_then.py
Please enter a test string: fantastic
$ _
```

Because *fantastic* has more than 8 characters the **if** statement is false so the following code-block is skipped.

if-then-else

Let's see another example. Create a file called *if_then2.py*. This time we will prompt the user to enter a number and we will test to see if it is even or odd.

First, the `input()` function returns a string, so we want to convert it to an integer. We will do this using the `int()` constructor. If the user does not type an integer this will cause an error. We will learn how to handle errors in a future lesson.

```python
value = input("Please enter an integer: ")
number = int(value)

if number % 2 == 0:
        print("Your number is even.")
else:
        print("Your number is odd.")
```

Next, if the **number** you entered is a multiple of two then print *Your number is even.* else print *Your number is odd.*.

The `%` operator returns the remainder when you divide the first number by the second one. In this case, we are computing the remainder when you divide the number by 2. If the remainder is 0 then the number is even, otherwise it is odd.

We're ready to test this code, save the program and execute it in console.

```
$ python3 if_then2.py
Please enter an integer: 17
Your number is odd.
$ _
```

Everything worked, the if-then statement correctly identified 17 as odd. Run the program again and enter 50.

```
$ python3 if_then2.py
Please enter an integer: 50
Your number is even.
$ _
```

Correct again.

if-elif-else

For our final example we will create an if-then statement that handles more than two cases. Create a file called *if_then3.py*. We will prompt the user to enter the lengths of the sides of a triangle and we will determine if it is scalene, isosceles or equilateral.

A scalene triangle is one where all three sides are different lengths, an isosceles triangle has two sides of the same length and an equilateral triangle is one where all the sides are equal.

First, we prompt the user to enter the lengths of the three sides. Like before, we need to convert the strings to integers, we will do this in one line this time.

```python
# scalene triangle: all three sides are different
# isosceles triangle: two sides of the same length
# equilateral triangle: all the sides are equal

a = int(input("The length of side a: "))
b = int(input("The length of side b: "))
c = int(input("The length of side c: "))

if a == b and b == c:
        print("This is an equilateral triangle.")
elif a != b and b != c and a != c:
        print("This is a scalene triangle.")
else:
        print("This is an isosceles triangle.")
```

Next, we compare the sides to determine the type of the triangle:

- if **a** equals **b** and **b** equals **c** then all three sides are identical, therefore it's an equilateral triangle
- if **a** does not equal **b** and **b** does not equal **c** and **a** does not equal **c** then all three sides are different, it's a scalene triangle
- if it is neither scalene nor equilateral then it must be an isosceles triangle.

Note: This example illustrates how to handle more than two cases. Once again, if and else lines end in colons. The code-blocks that follow end in indentation. What's different is the use of `elif` which is short for `else if`. This allows you to try another test. There is no limit on how many else ifs you can use. And finally the `else` statement is a catch-all, if all of the ifs above fail then this block is executed.

Let's test our code. Save the file and run it in console. Enter the sides: 3, 4 and 5:

```
$ python3 if_then3.py
The length of side a: 3
The length of side b: 4
The length of side c: 5
This is a scalene triangle.
$ _
```

Our program is correct, a triangle with these sides is scalene. Run the program again and enter: 5, 5 and 7:

```
$ python3 if_then3.py
The length of side a: 5
The length of side b: 5
The length of side c: 7
This is an isosceles triangle.
$ _
```

Excellent, a triangle with these sides is definitely isosceles. One more, run the program and enter: 4, 4, 4:

```
$ python3 if_then3.py
The length of side a: 4
The length of side b: 4
The length of side c: 4
This is an equilateral triangle.
$ _
```

Perfect, these are the sides of an equilateral triangle. By the way, we did not test the three numbers to make sure that they make a valid triangle. For example you could enter negative integers and the program will still run.

Here's a problem for you to think about: how do you test three numbers to see if they form a triangle?

The `if`, `elif` and `else` statements allow you to handle any number of cases in your code, they let you control the flow of your code.

> *If* you are serious about programming in Python
> *then* you should master these statements
> or *else* ...

2.5.3 Conditional expressions

In Python, they are more commonly known as *ternary operators*. These operators evaluate something based on a condition being true or not. Here you can see a blueprint of using these conditional expressions:

```
value_for_true if condition else value_for_false
```

This allows you to shorten a multi-line if statement to a single line, making your code compact but still readable. For example, let's determine whether the water is solid (frozen) or liquid based on a temperature reading from an external sensor and compare the :

```
temperature = -12  # external sensor reading

# Using an if statement
if temperature < 0:
    water_state = "solid"
else:
    water_state = "liquid"

# Using the ternary operator
water_state = "solid" if temperature < 0 else "liquid"
```

Short-hand ternary

In Python, there is also an even shorter version of the normal ternary operator you have seen above. Its blueprint looks like this *value* **or** *alternative value*, and this can allow you to easily provide a default value.

```
message = input() or "No data provided."
print(message)
```

This is helpful in case you quickly want to check for the output of a function or the input provided by the user and give a useful message if the value is missing (actually if **bool(value) is False**).

2.6 Lists

Order matters. And python lists make it easy to work with ordered data. Lists are a builtin data structure for storing and accessing objects which belong in a specific sequence. We will now learn how to create and use lists, and we will do so in a linear and orderly fashion.

2.6.1 Create

There are two ways to create a list. One way is to use the `list()` *constructor*. But a simpler and more common way is to use *brackets*.

```
samples = list()
# or ...
samples = []
```

When creating a list, you can also pre-populate it with values. For example, let's create a list with the first few prime numbers:

```
prime_numbers = [2, 3, 5, 7, 11, 13]
```

If you feel that these numbers are not enough, you can always add values later by using the `append()` method which allows you to add new values *to the end* of the list. Let's append the next two prime numbers: 17 and 19.

```
prime_numbers.append(17)
prime_numbers.append(19)

print(prime_numbers)
```

If you display the list, you will see it contains the new values. Notice how lists preserve the order of the data, in lists order is everything.

Lists can contain more than prime numbers. They can contain integers, booleans, strings, floats, and even other lists.

```
examples = [128, True, "Alphabet", 3.14, [32, 64, False]]
print(examples)
```

Many languages require lists to contain values of the same type, but not Python. With Python you are free to insert multiple data types in the same list. Lists can also contain duplicate values. Here is another way lists are different from sets. For example, suppose you want to record the numbers you roll on a pair of dice. Pretend you roll a 4, 7, 2, 7, 12, 4 and 7.

```
>>> rolls = [4, 7, 2, 7, 12, 4, 7]
>>> rolls
[4, 7, 2, 7, 12, 4, 7]
```

If you look at the list, all the values are there, even the repeated rolls.

2.6.2 Concatenate

You can also combine lists. To see how, create two separate lists: a list of numbers and a list of letters… To combine these two lists into a single list use the plus sign.

```
>>> numbers = [1, 2, 3]
>>> letters = ["a", "b", "c"]
>>> numbers + letters
[1, 2, 3, 'a', 'b', 'c']
```

But order matters, if you reverse this and compute `letters + numbers` you get `'a', 'b', 'c', 1, 2, 3`. Combining lists is called concatenation. Observe. The list of numbers and the list of letters are unchanged.

```
>>> letters + numbers
['a', 'b', 'c', 1, 2, 3]
>>> numbers
[1, 2, 3]
>>> letters
['a', 'b', 'c']
```

2.6.3 Access

You do not have to view the entire list. If you want to see a specific value, you can access it by its index.

Note: In computer science, we start counting indexes with 0, not 1. So in our list *prime numbers* are indexed 0, 1, 2, 3…

```
[ 2,  3,  5,  7, 11, 13, 17, 19 ]
  ^
  0   1   2   3   4   5   6   7
```

To view the first item, you type the name of the list and the index in brackets. The first item is 2. The second item has index 1 and the second item is 3. And so on.

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[0]
2
>>> prime_numbers[1]
3
>>> prime_numbers[2]
5
```

Notice how the indexes increase by one as you go from left to right. And they decrease by one as you go from right to left. When you get to the beginning the index is 0. If you decrease the index once more, you get -1. Here, Python wraps back around to the end of the list. So the last item has the index -1, the next to last -2, and so on.

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[-1]
19
>>> prime_numbers[-2]
17
>>> prime_numbers[-8]
2
```

This is convenient when you want to look at the values at the end of a list. The last item is 19, the next to last prime is 17. And so on, until we reach the beginning of the list with index -8. Be careful, you can only wrap around once. If you try to find the value of index -9, you get an index error.

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[-9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

2.6.4 Slicing

Another way to access values in a list is by slicing. This let's you retrieve a range of values from your list. We will continue to use our lists of primes. To slice this list, type the name of the list, bracket, *a starting index*, a colon, *a stopping index*, then a closing bracket.

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[2:5]
[5, 7, 11]
```

The result is a sublist that starts at index 2, and continues until it reaches index 5. Be careful, slicing includes the value at the starting index, but excludes the stopping index. The beginning value is included, the ending value is not.

One more slice. . .

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[0:6]
[2, 3, 5, 7, 11, 13]
```

This will start at the beginning, which is index 0, and continue to index 6, which is 17. It will not include the final number, so this slice includes the primes from 2 through 13, in other words: the first 6 values.

Notice, that if you start from the beginning, you can ommit the 0 completely and the slice will assume that you want to start from index 0. Similarly, if you omit the stopping index it will assume that you want to go the end of the list.

```
>>> prime_numbers
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prime_numbers[:6]
[2, 3, 5, 7, 11, 13]
>>> prime_numbers[6:]
[17, 19]
```

There are many other methods for working with lists. To see them all, please read the official Python list documentation.

2.6.5 Lists comprehension

When coding you spend a lot of time making lists, in many languages this can be tedious: create an empty list, set up a for loop, then add the items to the list one by one. Python cares about your sanity and gives you a tool to simplify this process: *list comprehension*. In most cases let you construct a new list in a single line of code. It's now time for Python to shine and save time with a single line.

We will cover many examples of lists comprehensions, but first let's talk about them generally. In Python lists are a collection of data surounded by brackets and the elements are separated by commas. A list comprehension is also surounded by brackets but instead of a list of data inside you enter an expression followed by for loops and if clauses. Here is the most basic form for a list comprehension:

[*expr* for *value* in *collection*]

The first *expression* generates the elements in the list and you follow this with a for loop over some *collection* of data. This will evaluate the expression for every item in the collection. If you want to include the expression for certain pieces of data you can add on an if clause after the for loop. The expression will be added to the list only if clause its true.

[*expr* for *value* in *collection* if *condition*]

You can even have more than one if clause and the expression will be added to the list only if all the clauses are true.

[*expr* for *value* in *collection* if *condition1* and *condition2*]

And you can even loop over more than one collection.

[*expr* for *val1* in *collection1* for *val2* in *collection2*]

Let's now see some examples. For our first example, let's create a list of the squares of the first 10 pozitive integers. Let's first do this without list comprehensions.

To begin you might create an empty list called `squares`, next you would loop over the first 10 positive integers. You would then append the square of each to the list of squares.

```
squares = []
for i in range(1, 11):
    squares.append(i**2)
```

(continues on next page)

(continued from previous page)

```
print(squares)

# this is the output
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Notice that an exponent in Python is represented by double asterisks. To see that this works print list *squares*.

Let's do this once more using list comprehensions:

```
squares2 = [i**2 for i in range(1,11)]
```

If you print this, you get the exact same list, but we only needed one line of code instead of three. Let's now look at a slightly more complex example. We'll create a list of remainders when you divide the first 10 squares by 5.

> To find the remainder when you divide by 5 use the `%` operator.

```
remainders = [(x ** 2) % 5 for x in range(1,11)]
print(remainders)

# this is the output
[1, 4, 4, 1, 0, 1, 4, 4, 1, 0]
```

If you print the list, you'll see that there are only three perfect squares mod 5: 0, 1 and 4. This example shows you that the expressions in the list comprehensions can be complex. By the way, if you look at the remainders when you divide by a prime number *p* you'll notice an interesting pattern: the number of remainders is (p+1)/2. The problem of finding which number appear in the list is a complex puzzle from number theory known as *quadratic reciprocity* and was first proved by Gauss.

Next, let's create a list comprehension that has an if clause. Suppose we have a list of movies and we want to find those movies that start with the letter G. Let's see how to do this with and without lists comprehensions.

If you're not using list comprehensions you'd start by making an empty list, next loop over the list of movies. We can use the `startswith()` method to see if the title starts with the letter G. If it does, then append it to out list.

```
movies = [
    "Star Wars", "Ghandi", "Casablanca", "Shawshank Redemption",
    "Toy Story", "Gone with the wind", "Citizen Kane", "It's a wonderful life",
    "The Wizard of Oz", "Gattaca", "Rear Window", "Ghostbusters",
    "To Kill a Mockingbird", "Good Will Hunting", "2001: A Space Odissey",
    "Riders of the Lost Ark", "Groundhog Day",
    "Close Encounters of the Third King", "Scent of a Woman",
]

g_movies = []
for title in movies:
    if title.startswith("G"):
        g_movies.append(title)
```

Print the list to make sure that it worked. But this four line routine can be done in a single line with a list comprehension. The expression we want to appear in our list is simply the title, next loop over the movies, but also check that the title starts with the letter G.

```
g_movies = [title for title in movies if title.startswith("G")]
```

Print and observe: we get the same answer with a single line of code.

> `["Ghandi", "Gone with the wind", "Gattaca", "Ghostbusters", "Good Will Hunting", "Groundhog Day"]`

Let's complicate this example a bit more. Suppose our list of movies is a list of pairs containing both the title of the movie and the year it was released. What if we want a list of titles of all movies that were released before the year 2000. How would you do this using lists comprehensions.

As before we want our list to only contain the titles, but this time when we write the *for-loop* each element is a tuple. Next we select the movies released before 2000 using an *if* clause on the year.

```
movies = [
    ("Citizen Kane", 1941), ("Spirited Away", 2001),
    ("It's a wonderful life", 1946), ("Gattaca", 1997),
    ("No Country for Old Men", 2007), ("Rear Window", 1954),
    ("The Lord of the Rings: The Fellowship of the Ring", 2001),
    ("Groundhog Day", 1993), ("Close Encounters of the Third King", 1977),
    ("The Aviator", 2004), ("Riders of the Lost Ark", 1981),
]

pre2k = [title for title, year in movies if year < 2000]
```

If you print the list, you can see that it worked. In this example the if clause used the *year* but the *year* was not included in the list, only the title is included.

Let's see a mathematical example, suppose you use a list to represent a vector, how would you perform scalar multiplication on this vector?

```
v = [2, -3, 1]
```

That is what if we want to multiply each number by 4. You might be tempted to try `4 * v` but look what happens, this is unusual:

```
>>> v = [2, -3, 1]
>>> 4 * v
[2, -3, 1, 2, -3, 1, 2, -3, 1, 2, -3, 1]
```

What happened here is **4** times **v** is the same as **v + v + v + v** and in Python if you add two lists it concatenates them rather than adding them component wise. For example if you add **[2, 4, 6]** and **[1, 3]** you get the list **[2, 4, 6, 1, 3]**, so **4 * v** is just a list containing 4 copies of **v**. This is not what we want. We can achieve scalar multiplication with a list comprehension where we multiply each component by 4.

```
v = [2, -3, 1]
result = [4 * x for x in v]
```

If you print this vector you can see we get the desired result.

For our final example let's use list comprehensions to compute the cartesian product of sets. The cartesian product is named after the French scholar Rene Descartes. Recall that if you have two sets A and B is the set of pairs where the first component is in A and the second component is in B.

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

For example

$$\begin{aligned} A &= \{1, 3\} \\ B &= \{x, y\} \\ A \times B &= \{(1, x), (1, y), (3, x), (3, y)\} \end{aligned}$$

Now let's compute the cartesian product of two sets in Python using lists comprehensions.

```
A = [1, 3, 5, 7]
B = [2, 4, 6, 8]

cartesian_product = [(a, b) for a in A for b in B]
```

If you print the product, you can see the list contains all 16 possible pairs. Using this technique you can even compute the cartesian product of three or more sets.

Note: Lists start at 0 and they end precisely when you are finished. You can slice them, you can concatenate them, you can reverse them, you can sort them, *comprehend* them. You can even clear them.

If I were to make a list of all uses of lists, I would have a very, VERY long list.

2.6.6 Exercises

1. Given a Python list you should be able to display Python list in the following order

   ```
   given = [100, 200, 300, 400, 500]
   expected = [500, 400, 300, 200, 100]
   ```

2. Remove empty strings from the list of strings

   ```
   given = ["Mike", "", "Emma", "Kelly", "", "Brad"]
   expected = ["Mike", "Emma", "Kelly", "Brad"]
   ```

3. Given a Python list, find value 20 in the list, and if it is present, replace it with 200. Only update the first occurrence of the value.

   ```
   given = [5, 10, 15, 20, 25, 50, 20]
   expected = [5, 10, 15, 200, 25, 50, 20]
   ```

4. Given a Python list, remove all occurrence of 20 from the list.

   ```
   given = [5, 20, 15, 20, 25, 50, 20]
   expected = [5, 15, 25, 50]
   ```

5. Concatenate two lists index-wise

   ```
   list1 = ["M", "na", "i", "Ri"]
   list2 = ["y", "me", "s", "ck"]
   expected = ["My", "name", "is", "Rick"]
   ```

2.7 Sets

Python comes equipped with different objects to help organize your data. These data structures include lists, tuples, sets and dictionaries. But right now, we'll focus on sets.

Sets are useful when you are working with data and the order or frequency of the values does not matter. Get ready to become an element of the set of people who understand sets.

2.7.1 Create

We will begin by creating an empty set.

```
example = set()
```

Now we want to add things to our set, and the `add()` function does exactly what we expect. Duplicates are not stored: if you try to add the same item twice, the set will store it the first time and ignore it the second time. We will use this method to add several objects to this set.

```
example.add(42)
example.add(False)
example.add(3.141592653589793)
example.add("Thorium")
```

Notice that you can add data of different types to the same set. If you print the example set we just prepared, Python will show you the items inside the set. Each item inside a set is called an "element".

```
print(example)
# Output
{False, 42, 3.141592653589793, 'Thorium'}
```

When you try this, the elements may appear in a different order for you than what is displayed here. Do not panic. For sets, the order does not matter. This is different for lists and tuples where the order does matter.

Now look what happens when you try to add the number **42** to the set a second time.

```
example.add(42)
print(example)
# Output
{False, 42, 3.141592653589793, 'Thorium'}
```

The set still contains just one copy of the number **42**. Sets do not contain duplicate elements. To see the number of elements in a set, use the length function, which is shortened to : **len(example)**.

There is a second way to create a set, which can be faster in some instances. When creating the set you can pre-populate the set with a collection of elements.

```
example2 = set([False, 42, 3.141592653589793, 'Thorium'])
print(example2)
# Output
{False, 42, 3.141592653589793, 'Thorium'}
```

2.7.2 Remove

To remove an element from this set, use the **remove** method. Beware, if you attempt to remove an element that is not in the set you will get an error. To test this method, let's remove the number **42**.

```
example.remove(42)
print(len(example))
print(example)
# Output
3
{False, 3.141592653589793, 'Thorium'}
```

We can check that it worked either by looking at the number of elements or displaying all the elements inside the set. Look what happens if we try to remove the number **50** which is not in the set:

```
example.remove(50)
# Output
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 50
```

To avoid the possibility of an error, there is a second way to remove an element: the **discard** method. With the *discard* method, if you try to remove an element which is not in the set, the method does nothing – it quietly returns without making a change. Watch what happens when we discard the integer **50**, which is not in the set.

```
>>> example.discard(50)
>>>
```

Nothing… Peace and quiet. The choice is yours; if you want to be alerted when your code tries to remove an element not in the set, use **remove()**. Otherwise, discard provides a convenient alternative.

There is also a faster way to remove elements. To empty out the set and remove all elements, use the **clear()** method.

```
>>> print(example)
{False, 3.141592653589793, 'Thorium'}
>>> example.clear()
>>> len(example)
0
```

This set now contains no elements – it has become the empty set. We can move along; there is nothing to see here.

2.7.3 Union and intersection

Now that we know how to create and modity a set, let's learn how to evaluate the union and intersection of two sets. If you have two sets **A** and **B**, then the union is the combination of all elements from the two sets and in math is denoted as $A \bigcup B$. The intersection is the set of elements inside both A and B, and is denoted as $A \bigcap B$.

To see these in action, let's look at the integers from 1 through 10.

```
odds = set([1, 3, 5, 7, 9])
evens = set([2, 4, 6, 8, 10])
primes = set([2, 3, 5, 7])
composites = set([4, 6, 8, 9, 10])
```

The union of the odd and even integers are all numbers from 1 to 10. You get the same answer if you reverse everything.

```
>>> odds.union(evens)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> evens.union(odds)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

>>> odds
{1, 3, 5, 7, 9}
>>> evens
{2, 4, 6, 8, 10}
```

Notice how the set of *odds* and the set of *evens* are unchanged. We can find the set of odd prime numbers by computing the intersection of the sets of **odds** and **primes**.

```
>>> odds.intersection(primes)
{3, 5, 7}
```

(continues on next page)

(continued from previous page)

```
>>> primes.intersection(evens)
{2}
```

And there is only one even prime number: **2**.

Which integers are both *even* and *odd* ?

```
>>> evens.intersection(odds)
set()
```

There are none. The intersection of these two sets is the empty set.

```
>>> primes.union(composites)
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The union of the prime numbers and composite numbers are the integers from **2** through **10**. Notice 1 is missing – this is becausse 1 is neither prime nor composite.

2.7.4 Membership

Another common opreation is testing to see if one element is inside a set. To do this in Python use the **in** operator. Is 2 in the set of prime numbers ?

```
>>> 2 in primes
True
```

Yes. This is a true statement.

Is 6 an odd integer?

```
>>> 6 in odds
False
```

No. This is a false statement. You can also test to see if an element is NOT in a set.

```
>>> 9 not in evens
True
```

9 is *not* an even integer, so this is a true statement.

There are many more methods and operations you can perform with sets. Take a moment to explore these methods. You will not regret it.

Sets are a built-in data type in Python. They come equipped with all the luxury features: unions, intersections, adding elements, removing elements, and much more. Everything you will ever need for your data hungry code… Provided your sets are *finite*.

2.7.5 Exercises

1. Add a list of elements to a given set.

   ```
   given = {"Yellow", "Orange", "Black"}
   some_items = ["Blue", "Green", "Red"]
   expected = {"Green", "Yellow", "Black", "Orange", "Red", "Blue"}
   ```

2. Create a new set with all items from both sets by removing duplicates

   ```
   given_a = {10, 20, 30, 40, 50}
   given_b = {30, 40, 50, 60, 70}
   expected = {70, 40, 10, 50, 20, 60, 30}
   ```

3. Remove 10, 20, 30 elements from the following set

   ```
   given_a = {10, 20, 30, 40, 50}
   expected = {40, 50}
   ```

4. Determine wether or not the following two sets have any elements in common

   ```
   given_a = {10, 20, 30, 40, 50}
   given_b = {60, 70, 80, 90, 10}
   # Expected output:
   The two sets have items in common.
   {10}
   ```

5. Count the number of vowels in a given string

   ```
   given = "Understanding sets is easy."
   # Expected output
   The given string has 7 vowels.
   ```

2.8 Tuples

Not all data can be stored in a pile, oftentime data must be ordered in a sequence to be useful. Python offers several ways to store and work with ordered data. *Lists* are the most common tool, but there is a smaller, faster alternative the *tuple*.

2.8.1 Comparison to lists

What is the difference between **lists** and **tuples**?

The quick explanation is that a list contains a sequence of data surrounded by brackets, while the tuple contains a sequence of data surrounded by parentheses.

```
# List example
prime_numbers = [2, 3, 5, 7, 11, 13, 17]

# Tuple example
perfect_squares = (1, 4, 9, 16, 25, 36)

# Display lengths
print("# Primes:  ", len(prime_numbers))
print("# Squares: ", len(perfect_squares))

# Iterate over both sequences
for p in prime_numbers:
    print("Prime: ", p)
for n in perfect_squares:
    print("Square: ", n)
```

Other than notation there seems to be little difference between these two. In both cases you can use the **len()** function to display the number of elements in the sequence, or you can iterate over the sequences and get identical behavior.

So how are lists and tuples different from one another? To see the difference let's have a look at the methods available to the lists. Then, let's look at the methods available to the tuples. Notice that lists have more methods available than tuples. This extra functionality comes at a price: lists occupy more memory than tuples. When you are working with big data sets this can be significant.

Another difference between lists and tuples is that you can add, remove or change data in lists. Tuples cannot be changed. We say they are **immutable**. Once you make a tuple, it is set in stone. Knowing that tuples cannot change, enables Python to make significant optimizations.

For example, tuples can be made more quickly than lists.

2.8.2 Create

Tuples use parentheses. To make an empty tuple you simply type a pair of parentheses. Let's quickly make a few test tuples.

```
empty_tuple = ()
test1 = ("a")
test2 = ("a", "b")
test3 = ("a", "b", "c")
```

Next, let's print these four tuples:

```
print(empty_tuple)
print(test1)
print(test2)
print(test3)

# Output
()
a         # <-- wait, what's this ?
('a', 'b')
('a', 'b', 'c')
```

Notice that the tuple containing a single element looks different from the others. It looks like a string and not a tuple. To make a tuple with just one element you need to type a comma at the end. Let's make a change and re-run.

```
empty_tuple = ()
test1 = ("a",)
test2 = ("a", "b")
test3 = ("a", "b", "c")

print(empty_tuple)
print(test1)
print(test2)
print(test3)

# Output
()
('a',)
('a', 'b')
('a', 'b', 'c')
```

Everything is now a tuple.

Before we explain the mistery behind the tuple with one element, let's see another way to make a tuple. If you want, you can leave out the parentheses altogether. Like before, to make a tuple with one element you need to end with a comma.

```
test1 = "a",
test2 = "a", "b"
test3 = "a", "b", "c"

print(test1)
print(test2)
print(test3)

print(type(test1))
print(type(test2))
print(type(test3))
```

If we print each item and its type we see that all three tests are in fact tuples.

2.8.3 Tuple unpacking

Let's now examine the eccentric behavior of tuples with one element. The reason for this is a feature called *tuple unpacking*.

Suppose you are working with a large dataset containing three pieces of data about each person. Their **age**, **country** and wheter or not they **know_python**. Perhaps this data was collected in a survey to study the popularity of Python. We will store the results for each person in a tuple. Here is the information from a single person from the survey.

```
# (age, country, knows_python)
survey = (27, "Romania", True)
```

To access the data it is tempting to extract each piece of data individually, as with lists, you can access elements by index. We will print the values to make sure this method is successful.

```
age = survey[0]
country = survey[1]
knows_python = survey[2]

print("Age: ", age)
print("Country: ", country)
print("Knows Python: ", knows_python)
```

This works, but tuples provide a faster alternative. Consider a second survey. You can assign all elements in a tuple to different variables in a single line.

```
survey2 = (31, "Switzerland", False)
age, country, knows_python = survey2
```

This will assign the first element to *age*, second to *country*, and third to *knows_python*. Python unpacks all the values and assigns them for you. Please print each value to confirm this works.

Tuple unpacking explains the need for a trailing comma when making tuples with a single element.

```
country = ("Greece")
```

According to the rules of tuple unpacking this would assign the string "Greece" to the variable **country**. By adding an extra comma at the end you are telling Python that here you do in fact want **country** to be a tuple and you do not want to unpack the values into the variable.

Please make sure that the number of variables matches the number of elements in the tuple. We will look at two cases.

```
a, b, c = (1, 2, 3, 4)
```

Here we do not have enough variables to hold all the values in the tuple. Running this causes a *value error*. Similarly if you have more variables than elements in the tuple Python will raise a *value error*.

```
x, y, z = (1, 2)
```

There is no room for sloppy behavior when we're working with tuples.

2.8.4 Exercises

1. Check wether an element exists within a tuple.
2. Sum up all the number elements within a tuple.
3. Find the repeated items of a tuple.
4. Print out all pair combinations of two tuples.

2.9 Functions

When programming, you will often encounter calculations and logical operations you need to perform repeatedly. One way to handle this is to write the same code over, and over, and over… A better solution is to write a *function*.

Functions enable you to reuse logic an infinite number of times without repeating yourself. This is especially helpful to avoid moments of deja vu.

A better solution is to write a *function*. Functions enable you to reuse logic an infinite number of times without repeating yourself. This is especially helpful… to avoid… moments of deja vu.

2.9.1 Define

To define a function, you write **def** followed by the name of the function. We will call this function **f**. Next, you write parentheses. Inside the parentheses you list the inputs to the function. Another name for *inputs* is *arguments*.

```
def f():
    pass
```

This function has 0 arguments. For our first function, we will keep things simple. We will *pass*. The word *pass* is how you tell Python to skip this line and do nothing.

Tip: The colon is how you start a new code block in Python. Notice that we indented the code inside the function. Python requires you to group code blocks by indentation.

There we have created our first function. It is short, it is simple. But does it work ?

To *run* the function (in programming we say *call the function*), type the name and parentheses…

```
>>> f()
>>>
```

The function **f** did nothing, just like we told it to. By the way, look what happens if you forget to type parentheses.

```
>>> f
<function f at 0x101241a60>
>>>
```

Without parentheses, Python displays that **f** is a function and gives the memory address. Interesting, but not very helpful. You need to include parentheses to actually call the function.

2.9.2 Return

Now, we will write a function that actually does something. We'll name this function **ping**. This function will have 0 arguments. Functions in Python can return values. To return a value, type the word **return** and then the object. This function will return the string **"Ping"**, and to show our enthusiasm, we will add an exclamation point.

```
def ping():
    return "Ping!"
```

Return values are optional, you are under no obligation to return something. Now call the function.

```
>>> ping()
'Ping!'
>>>
```

This function return the string **"Ping!"**. Since we did not assign this to a variable, Python printed it to the terminal. But we can also store the return value to a variable. To see that this worked, print the value of **x**.

```
>>> x = ping()
>>> print(x)
Ping!
```

2.9.3 Arguments

For our next example, recall that the volume of a sphere is

$$V = \frac{4}{3}\pi r^3$$

Where *r* is the radius of the sphere. We will write a function which will return the volume of a sphere when given the radius. To do this, we will need to use the number *pi* (π). This is available in Python, but first you must import the *math* module.

We will call this function *volume*. This function will have a single argument: the radius **r**. Next, we will write a brief comment describing this function. This is called a **docstring** and provides documentation on what the function does and how to use it.

```
import math

def volume(r):
    """Returns the volume of a sphere with radius r."""
    v = (4/3) * math.pi * r**3
    return v
```

Notice that we used a double asterisk (`**`) for exponents. Let's test this function. Compute the volume of a sphere with radius 2.

```
>>> volume(2)
33.510321638291124
>>>
```

Because we used an argument when defining the function, you must provide an input when calling it. **r** is a required argument. Look what happens if you call *volume* without an argument.

```
>>> volume()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: volume() missing 1 required positional argument: 'r'
>>>
```

You get an error and a reminder to use an argument. We can use the help function to see how to use the volume function.

```
>>> help(volume)
Help on function volume in module __main__:

volume(r)
    Returns the volume of a sphere with radius r.
```

We have created functions with no arguments and with one argument. We will now throw caution into the vacuum of space and write a function with two arguments. Our function will compute the area of a triangle. Recall that the area of a triangle is

$$A\triangle = \frac{1}{2} base \times height$$

Our previous function was named *volume*, which was somewhat vague. The name does not tell you what shape is being considered. This time we will be more explicit and name the function **triangle_area**. To compute the area, we need two arguments: the base **b** and the height **h**. Write a *docstring* giving a brief description of this function and finally return the area of the triangle.

```
def triangle_area(b, h):
    """Returns the area of a triangle with base b and height h."""
    return 0.5 * b * h
```

We are ready to test the function.

```
>>> triangle_area(3, 6)
9.0
>>>
```

There is no limit to how many arguments you can use in your function, but if the number of inputs is too large, you will alienate other and receive accusing glares...

2.9.4 Keyword arguments

Functions in Python can accept another kind of argument called *keyword arguments*. To show how to use these kinds of arguments, we will write a function which convers a person's height from imperial units, feet and inches to centimeters.

Recall that $1 inch = 2.54 cm$ and $1 foot = 12 inches$.

We'll name this function **cm**, for centimeters. This function will accept two arguments: **feet** and **inches**. Next, add a *docstring* describing the function.

```
def cm(feet = 0, inches = 0):
    """Converts a length from feet and inches to centimeters."""
    inches_to_cm = inches * 2.54
    feet_to_cm = feet * 12 * 2.54
    return inches_to_cm + feet_to_cm
```

Notice that the arguments have equal signs after them. It looks as we are assigning values to these arguments. In fact, we are. We are assigning a default value of 0 to each argument. For this reason, Python also calls keyword arguments *default arguments*.

We could have combined all computations on one line, but it is better to write clean code which is easy to read, as opposed to compact code which impresses no one.

Here is how you call a function with keyword arguments:

```
>>> cm(feet = 5)
152.4
>>> cm(inches = 70)
177.8
>>> cm(feet = 5, inches = 8)
172.72
>>>
```

We can also perform this last calculation by specifying inches first:

```
>>> cm(inches = 8, feet = 5)
172.72
>>>
```

Keyword arguments help you to write flexible functions and clean code.

There are two kinds of arguments you can use when writing a function. A keyword argument (or default argument), which has an equal sign, and a required argument which does not have an equal sign. When writing a function, you can use both kinds of arguments. But if you do this, the keyword arguments must come last.

For example, if you define a function **g** with a keyword argument first, you get a syntax error.

```
>>> def g(x = 0, y):
...     return x + y
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
```

Notice that Python calls these *default argument*. This is another name for a *keyword argument*. To fix this, you have to list non-default arguments first. These are also called *required arguments* since they are required.

```
>>> def g(y, x = 0):
...     return x + y
...
>>> g(7)
7
>>>
```

To call this function, you must pass in a value for the required argument **y**. The keyword argument **x** is optional. If you do not provide a value for x, the default value is used.

If you want to pass in a value for the keyword argument, then you must specify it by its name. Required arguments are not given a name. They are determined by their position.

```
>>> g(7, x=3)
10
>>>
```

Functions in Python are flexible contraptions. Required arguments, keyword arguments, docstrings, return values – together they empower you to write some amazing, reusable code. And if your function does not require an input, you will not get an argument from me…

2.9.5 Exercises

1. Calculate the factorial of a number (positive integer).
2. Write a function that flattens a list. (Remember that a list may contain an element which is in fact another list)
3. Generate a list with Fibonacci sequence up to a number.
4. Generate a list with all prime numbers less than a number.

2.10 Dictionaries

A standard data structure in computer science is the *associative array*, which is also called a *map*. In Python, this structure is called a *dictionary*. Dictionaries are used when you have *key-value* pairs of data – an input which is mapped to an output, with the requirement that the keys are unique (within the same dictionary of course).

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type (strings, numbers, booleans, tuples). You can't use lists as keys, since lists can be modified in place.

2.10.1 Create

Suppose we are creating a new social network called "FriendSpace" where any life form or intelligent agent is allowed to join. Let us make our first post to *FriendSpace*, we will use this post to show you how to create a dictionary.

To begin, we will collect several pieces of data for each post:

```
>>> # FriendSpace post
>>> # user_id = 209
>>> # message = "I am learning about dictionaries. Yay!"
>>> # language = "English"
>>> # created = "2021-03-02T10:11:06.278326"
>>> # location = (46.766667, 23.583333)
>>>
```

Using dictionaries we can store all of this data in a single object. A pair of braces create an empty dictionary:

```
>>> empty_dictionary = {}
>>>
```

Placing a comma-separated list of key-value pairs withing the braces will add initial key-value pairs to the dictionary. This is also the way dictionaries are printed on console.

```
>>> post = {"user_id": 209, "message": "I am learning about dictionaries. Yay!",
↪"language": "English", "created": "2021-03-02T10:11:06.278326", "location": (46.
↪766667, 23.583333)}
>>>
```

We have created a dictionary called **post** with 5 pieces of data. If you think of a dictionary as a map, there are 5 inputs and 5 outputs. In Python, these inputs are called *keys*, and the outputs are called *values*.

Table 1: post dictionary

Keys	Values
user_id	209
message	“I am learning about dictionaries. Yay!”
language	“English”
created	“2021-03-02T10:11:06.278326”
location	(46.766667, 23.583333)

Notice how the values have a variety of data types in this dictionary: an integer, three strings and a tuple of floats. Python is flexible: you can use all kinds of data types for both keys and values. You are free to mix and match data to suit your needs.

```
>>> type(post)
<class 'dict'>
>>>
```

If you use the *type* function, you will see *post* is an instance of the *dict* class. This suggests we can use the **dict** constructor to create a dictionary. Let’s create another *FriendSpace* post using the dict constructor. This time, we will only provide a few pieces of data. . .

```
>>> post2 = dict(message="There is always another way.", language="English")
>>> print(post2)
{'message': 'There is always another way.', 'language': 'English'}
>>>
```

You can see this made a dictionary. We add additional pieces of data by using brackets. The key name goes inside the brackets, and you use the equals sign to assign the value.

```
>>> post2["user_id"] = 209
>>> post2["created"] = "2021-03-02T10:11:06.278326"
>>> print(post2)
{'message': 'There is always another way.', 'language': 'English', 'user_id': 209,
↪'created': '2021-03-02T10:11:06.278326'}
>>>
```

Notice that in the constructor, you did not have to put quotes around the keyname, but when you add new data using brackets, you do use quotes. If you print the dictionary, you will see all the data is there inside braces.

2.10.2 Access data

Just as you use brackets to store new data in a dictionary, you use brackets to access data as well. For example, to see the message from the first post, use the key name “message” inside brackets.

```
>>> print(post["message"])
I am learning about dictionaries. Yay!
>>>
```

But do not be careless. Let’s see what happens if you try to access data that is not in the dictionary.

```
>>> print(post2["location"])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
KeyError: 'location'
>>>
```

When we created **post2**, we did not assign a location. If we try to access this value, we get a *KeyError*. How do we avoid such catastrophe? Simple. One way is to use the **in** operator to first check if a key is in the dictionary.

```
>>> if "location" in post2:
...     print(post2["location"])
... else:
...     print("The post does not have a location")
...
The post does not have a location
>>>
```

Another way to access data in a dictionary and handle the possibility it does not have a certain key is the *get* method. This lets you try and get the value for a specific key. If the dictionary does not contain data for that key, you can specify a default value. For our case let's return an empty tuple.

```
>>> loc = post2.get("location", tuple())
>>> print(loc)
()
>>>
```

If you print the value you see the *get* method did returned an empty tuple.

Now, turn your attention back to the original post.

```
>>> print(post)
{'user_id': 209, 'message': 'I am learning about dictionaries. Yay!', 'language':
↪'English', 'created': '2021-03-02T10:11:06.278326', 'location': (46.766667, 23.
↪583333)}
>>>
```

A common task is to iterate over all the key value pairs in a dictionary. A straightforward way to do this is to loop over all the keys, then get the value:

```
>>> for key in post:
...     value = post[key]
...     print(key, "=", value)
...
user_id = 209
message = I am learning about dictionaries. Yay!
language = English
created = 2021-03-02T10:11:06.278326
location = (46.766667, 23.583333)
>>>
```

When you simply loop over a dictionary, this will give you all the keys for that dictionary. The order of the data may be different for you. Do no panic. Dictionaries are not ordered data. As long as you see all the data, everything is under control.

Another way to iterate over all the key value pairs in a dictionary is to use the **items** method. This will give you both the key and the value in each step of the iteration.

```
>>> for key, value in post.items():
...     print(key, "=", value)
```

(continues on next page)

(continued from previous page)

```
...
user_id = 209
message = I am learning about dictionaries. Yay!
language = English
created = 2021-03-02T10:11:06.278326
location = (46.766667, 23.583333)
>>>
```

There are a variety of methods for working with a dictionary. The **pop** and **popitem** methods allow you to remove a single item from a dictionary, while the **clear** method will remove all data. To complete this lesson please take a moment to explore and experiment with these methods.

2.10.3 Exercises

1. Create a dictionary out of the lists below:

```
keys = ["name", "phone", "email"]
values = ["Mike", "+40791882123", "michael@yahoo.com"]

# Expected output
{'name': 'Mike', 'phone': '+40791882123', 'email': 'michael@yahoo.com'}
```

2. Given a dictionary describing a car attributes, print the model and the year of the car or the string **N/A** if the value is not present.

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

3. Print all the key value pairs of any dictionary, but make sure they are shown in the same order every time (order the keys alphabetically).

4. Count the frequency of all words in the given text:

```
text = "He ordered his regular breakfast. Two eggs sunnyside up, hash␣
↪browns, and two strips of bacon. He continued to look at the menu␣
↪wondering if this would be the day he added something new. This was␣
↪also part of the routine. A few seconds of hesitation to see if␣
↪something else would be added to the order before demuring and saying␣
↪that would be all. It was the same exact meal that he had ordered every␣
↪day for the past two years."

# expected output
{'the': 6, 'he': 4, 'two': 3, 'of': 3, 'to': 3, 'would': 3, 'be': 3,
↪'ordered': 2, 'and': 2, 'if': 2, 'this': 2, 'day': 2, 'added': 2,
↪'something': 2, 'was': 2, 'that': 2, 'his': 1, 'regular': 1, 'breakfast
↪': 1, 'eggs': 1, 'sunnyside': 1, 'up': 1, 'hash': 1, 'browns': 1,
↪'strips': 1, 'bacon': 1, 'continued': 1, 'look': 1, 'at': 1, 'menu': 1,
↪'wondering': 1, 'new': 1, 'also': 1, 'part': 1, 'routine': 1, 'a': 1,
↪'few': 1, 'seconds': 1, 'hesitation': 1, 'see': 1, 'else': 1, 'order':␣
↪1, 'before': 1, 'demuring': 1, 'saying': 1, 'all': 1, 'it': 1, 'same':␣
↪1, 'exact': 1, 'meal': 1, 'had': 1, 'every': 1, 'for': 1, 'past': 1,
↪'years': 1}
```

2.11 Text files

Files are everywhere in the digital universe. When you write a Python program, you save your code in a file. Starting an app with your VR eye implants will launch a cloud service that is also contained in a file. Even your operating system consists of a mountain of files. Working with files is an essential part of life as a software engineer. So today we will learn how to read and write text files. And by the way, the lesson you are now reading is stored in a file.

2.11.1 Read

There are two kinds of files: text and binary. Text files are generally human readable data, like plain text, XML, JSON or even source code. Binary files are used for storing compiled code, app data, and media files, like images, audio and video files.

To open a file in Python, you use the built-in **open** function. This is a powerful and versatile function. There are pages of documentation on all ways you can use this function. Today we will focus on the *file* and the *mode*.

Begin by making a sample text file using your favourite editor. What better example than a brief biography of Guido van Rossum, the creator of Python. This text file will be named **guido_bio.txt**. To save time, I am pasting the text from Guido's personal web site:

```
Guido van Rossum is the creator of the Python programming language. He grew
up in the Netherlands and studied at the University of Amsterdam, where he
graduated with a Master's Degree in Mathematics and Computer Science. His
first job after college was as a programmer at CWI, where he worked on the
ABC language, the Amoeba distributed operating system, and a variety of
multimedia projects. During this time he created Python as side project. He
then moved to the United States to take a job at a non-profit research lab
in Virginia, married a Texan, worked for several other startups, and moved
to California. In 2005 he joined Google, where he obtained the rank of Senior
Staff Engineer, and in 2013 he started working for Dropbox as a Principal
Engineer. In October 2019 he retired. Until 2018 he was Python's BDFL
(Benevolent Dictator For Life), and he is still deeply involved in the
Python community. Guido, his wife and their teenager live in Silicon Valley,
where they love hiking, biking and birding.
Guido's home on the web is http://www.python.org/~guido/.
```

Now save and let's start python and read it. One way to open a file is to call the **open** function with the name or path to the file. Without any other arguments, it will open the file in read mode and return a file object which we call **file**. To read the contents of the file, call the **read()** method. This will return all the text in the file. Finally, close the file.

```
>>> file = open("guido_bio.txt")
>>> text = file.read()
>>> file.close()
>>> print(text)
Guido van Rossum is the creator of the Python programming language. He grew
up in the Netherlands and studied at the University of Amsterdam, where he
graduated with a Master's Degree in Mathematics and Computer Science. His
first job after college was as a programmer at CWI, where he worked on the
ABC language, the Amoeba distributed operating system, and a variety of
multimedia projects. During this time he created Python as side project. He
then moved to the United States to take a job at a non-profit research lab
in Virginia, married a Texan, worked for several other startups, and moved
to California. In 2005 he joined Google, where he obtained the rank of Senior
Staff Engineer, and in 2013 he started working for Dropbox as a Principal
Engineer. In October 2019 he retired. Until 2018 he was Python's BDFL
(Benevolent Dictator For Life), and he is still deeply involved in the
```

(continues on next page)

(continued from previous page)

```
Python community. Guido, his wife and their teenager live in Silicon Valley,
where they love hiking, biking and birding.
Guido's home on the web is http://www.python.org/~guido/.
>>>
```

We can print the text to see that everything worked. This method of opening, reading and closing a file is simple. It only took three lines. But there is a danger. What if something goes wrong before you close the file? You do not want to litter your operating system's memory with open files. Because of this, there is a better and safer way that is also 33% shorter. This method is to open a file using the **with** keyword. As before, the open function will create a file object and assign it to the name you specify. You can then read the contents like before, but with this technique you do not need to close the file. That is done for you. In fact, Python will close the file even if an exception occurs in the code block. That is first class service, indeed.

```
>>> with open("guido_bio.txt") as file:
...     contents = file.read()
...
>>> print(contents)
Guido van Rossum is the creator of the Python programming language. He grew
up in the Netherlands and studied at the University of Amsterdam, where he
graduated with a Master's Degree in Mathematics and Computer Science. His
first job after college was as a programmer at CWI, where he worked on the
ABC language, the Amoeba distributed operating system, and a variety of
multimedia projects. During this time he created Python as side project. He
then moved to the United States to take a job at a non-profit research lab
in Virginia, married a Texan, worked for several other startups, and moved
to California. In 2005 he joined Google, where he obtained the rank of Senior
Staff Engineer, and in 2013 he started working for Dropbox as a Principal
Engineer. In October 2019 he retired. Until 2018 he was Python's BDFL
(Benevolent Dictator For Life), and he is still deeply involved in the
Python community. Guido, his wife and their teenager live in Silicon Valley,
where they love hiking, biking and birding.
Guido's home on the web is http://www.python.org/~guido/.
>>>
```

Just to be sure, print the text to check that everything worked.

But what happens if you try to open a file that does not exist? Then Python will raise a *file not found* error. One way to handle this is to wrap the file code in a **try** block, where we can handle a *file not found* exception. For example if the file does not exist, you might want *text* to store the value **None**.

```
try:
    with open ("lottery_numbers.txt") as f:
        text = f.read()
except FileNotFoundError:
    text = None

print(text)
```

If you run this it displays *None*, since the file was not found. And if you replace the file name with the one we created earlier ("guido_bio.txt"), it successfully reads the existing file.

If you don't provide any argument to the **read**() method, it will read the entire content of the file in memory. Typically, for text files this is the recommended method. But if you know you need to handle huge files (larger than your computer memory for example) you may want to read and process the text file line by line. Luckily Python file objects already have a couple of methods, please have a look at the official file documentation.

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
>>>
```

If you want to read all the lines of a file in a list you can also use the **list()** constructor:

```
>>> all_lines = list(f)
>>>
```

or the **readlines()** method:

```
>>> all_lines = f.readlines()
>>>
```

2.11.2 Write

Let us now see how to write a text file. For an example, we will write the names of the five oceans to a txt file. In this case, you need an additional argument to the **open()** function: mode **"w"** for write. If you do not specify a mode, Python will open the file in *read mode* by default. To write the names of the oceans to the file, loop over the list and use the **write()** method. Remember, when using the **with** technique for opening files, Python will automatically close the files for you.

```
oceans = ["Pacific", "Atlantic", "Indian", "Arctic", "Southern"]
with open ("oceans.txt", "w") as f:
    for ocean in oceans:
        f.write(ocean)
```

Here, if the file *oceans.txt* does not exist, Python will create it. And if the file already exists, Python will overwrite it. So be careful.

Let's open the file and see if this worked.

Listing 2: oceans.txt

```
PacificAtlanticIndianArcticSouthern
```

While it did write the ocean name, there are no line separators. That is because we did not write one, and Python does not assume we want them. One way to fix this is two write a new line after writing the name of each ocean.

```
oceans = ["Pacific", "Atlantic", "Indian", "Arctic", "Southern"]
with open ("oceans.txt", "w") as f:
    for ocean in oceans:
        f.write(ocean)
        f.write("\n")
```

Now, if you run the code, and then open the file, you will see each name is on a separate line.

Listing 3: oceans.txt

```
Pacific
Atlantic
Indian
```

(continues on next page)

(continued from previous page)

```
Arctic
Southern
```

There is another way to ensure that each string is on a separate line. You can use the **print()** function and have it to print to the file by using the *file* keyword argument.

```
oceans = ["Pacific", "Atlantic", "Indian", "Arctic", "Southern"]
with open ("oceans.txt", "w") as f:
    for ocean in oceans:
        print(ocean, file=f)
```

If you run this, and look at the file, you get the same result. Each name is on a separate line.

Listing 4: oceans.txt

```
Pacific
Atlantic
Indian
Arctic
Southern
```

2.11.3 Append

Notice that every time we ran our code, any text in the file was overwritten. This is because we open the file in *write* mode. But what if you would like to write to a file without overwriting any existing text?

For this, you open the file in *append mode* by using an **"a"** after the file name. Append mode will create the file if it does not exist, but if the file does exist, then Python will append your text *to the end*. It will not overwrite any existing text.

```
oceans = ["Pacific", "Atlantic", "Indian", "Arctic", "Southern"]
with open ("oceans.txt", "w") as f:
    for ocean in oceans:
        print(ocean, file=f)

with open("oceans.txt", "a") as f:
    print("=======================", file=f)
    print("These are the 5 oceans.", file=f)
```

Let us see that this is the case.

Listing 5: oceans.txt

```
Pacific
Atlantic
Indian
Arctic
Southern
=======================
These are the 5 oceans.
```

2.12 JSON data

JSON (*JavaScript Object Notation*) is a small lightweight, data format. A packet of JSON data is almost identical to a Python *dictionary*. It is shorter than XML and can be quickly parsed by humans and most importantly any web browsers since it uses JavaScript syntax. This makes JSON an ideal format for transporting data between a client and a server. If your client is not a browser, don't worry. Android, iOS and other mobile operating systems, all come equipped with tools for parsing and working with JSON.

In this lesson I will show you how to use Python's built-in JSON library to send and receive JSON data.

2.12.1 A Little Vocabulary

The process of encoding JSON is usually called *serialization*. This term refers to the transformation of data into *a series of bytes* (hence serial) to be stored or transmitted across a network. You may also hear the term *marshaling*, but that's a whole other discussion.

Naturally, *deserialization* is the reciprocal process of decoding data that has been stored or delivered in the JSON standard.

> Yikes! This sounds pretty technical. Definitely. But in reality, all we're talking about here is *reading* and *writing*. Think of it like this: encoding is for writing data to disk, while decoding is for reading data into memory.

Simple Python objects are translated to JSON according to a fairly intuitive conversion:

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Compare to XML

Here is a typical JSON data packet.

```
{
    "title": "Gattaca",
    "release_year": 1997,
```

(continues on next page)

(continued from previous page)

```
    "is_awesome": true,
    "won_oscar": false,
    "actors": ["Ethan Hawke", "Uma Thurman", "Alan Arkin", "Loren Dean"],
    "budget": null,
    "credits": {
        "director": "Andrew Niccol",
        "writer": "Andrew Niccol",
        "composer": "Michael Nyman",
        "cinematographer": "Slawomir Idziak"
    }
}
```

This JSON object contains examples of all possible data types. All the keys are *strings*, but the values can be *strings*, *numbers*, *booleans*, *list*, *null* or even another JSON object.

Now, compare it with the XML version:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<actors>
    <element>Ethan Hawke</element>
    <element>Uma Thurman</element>
    <element>Alan Arkin</element>
    <element>Loren Dean</element>
</actors>
<budget null="true" />
<credits>
    <cinematographer>Slawomir Idziak</cinematographer>
    <composer>Michael Nyman</composer>
    <director>Andrew Niccol</director>
    <writer>Andrew Niccol</writer>
</credits>
<is_awesome>true</is_awesome>
<release_year>1997</release_year>
<title>Gattaca</title>
<won_oscar>false</won_oscar>
</root>
```

The increased size of the XML data is largely due to the end tags repeating the text of the openning tags. A popular sport is debating the merits of JSON versus XML. But instead of arguing, I recommend you learn the pros and cons of both formats, then choose the one which is best for your project.

2.12.2 Reading JSON

First, let's save the sample JSON data to a text file, and then we'll use the context manager to open up the existing `data_file.json` in read mode.

The **json** library provides two methods for turning JSON encoded data into Python objects:

- the *load()* method allows to read (load) JSON data directly from a file
- while the *loads()* methods allows to read (load) JSON data from a string.

  That's why there is an extra **s** in the method name: *s* for *string*.

Now, let's load the JSON data, from the file created earlier, using the *load()* method.

```
import json

with open("data_file.json", "rt") as data_file:
    data = json.load(data_file)
```

If you display the object you will see a dictionary containing all the data:

```
>>> print(data)
{'title': 'Gattaca', 'release_year': 1997, 'is_awesome': True, 'won_oscar': False,
↪'actors': ['Ethan Hawke', 'Uma Thurman', 'Alan Arkin', 'Loren Dean'], 'budget':␣
↪None, 'credits': {'director': 'Andrew Niccol', 'writer': 'Andrew Niccol', 'composer
↪': 'Michael Nyman', 'cinematographer': 'Slawomir Idziak'}}
>>> type(data)
<class 'dict'>
>>>
```

If you look at the type, you will see it is, in fact, a dictionary. Also, notice how the *true*, *false* and *null* were correctly parsed into Python's *True*, *False* and *None*.

Because this is a dictionary, you can access the data by key. We can see the title, the list of actors and so on:

```
>>> data["title"]
'Gattaca'
>>> data["actors"]
['Ethan Hawke', 'Uma Thurman', 'Alan Arkin', 'Loren Dean']
>>> data["release_year"]
1997
>>>
```

Now, let's focus on the **loads()** method, which must be used if the data you need to process arrives in the form of a string. This is common in client-server applications where data is sent over the internet. As an illustration let's create a string with a JSON formatted value:

```
encoded_value = """
    {
        "title": "Tron: Legacy",
        "composer": "Daft Punk",
        "release_year": 2010,
        "budget": 170000000,
        "actors": null,
        "won_oscar": false
    }
"""
tron = json.loads(encoded_value)
```

If you look at the result, we have a valid python dictionary with all data properly converted, *false* is now a Python boolean and *null* is converted to **None**:

```
>>> print(tron)
{'title': 'Tron: Legacy', 'composer': 'Daft Punk', 'release_year': 2010, 'budget':␣
↪170000000, 'actors': None, 'won_oscar': False}
>>>
```

2.12.3 Writing JSON

Suppose you want to store the data about Gattaca movie in a database, or send it to a remote user. To convert this dictionary int a valid JSON string you use the **dumps()** method (read as *dump-s*):

```
>>> print(movie)
{'title': 'Gattaca', 'release_year': 1997, 'is_awesome': True, 'won_oscar': False,
↪'actors': ['Ethan Hawke', 'Uma Thurman', 'Alan Arkin', 'Loren Dean'], 'budget':␣
↪None, 'credits': {'director': 'Andrew Niccol', 'writer': 'Andrew Niccol', 'composer
↪': 'Michael Nyman', 'cinematographer': 'Slawomir Idziak'}}
>>> json.dumps(movie)
'{"title": "Gattaca", "release_year": 1997, "is_awesome": true, "won_oscar": false,
↪"actors": ["Ethan Hawke", "Uma Thurman", "Alan Arkin", "Loren Dean"], "budget":␣
↪null, "credits": {"director": "Andrew Niccol", "writer": "Andrew Niccol", "composer
↪": "Michael Nyman", "cinematographer": "Slawomir Idziak"}}'
>>>
```

When you call the method simply pass in the dictionary. The result is a string in proper JSON format. Notice that *true* and *false* are both lowercase, and that *None* was converted to *null*.

Let's now create a new object, convert it to JSON, and write it to a file. We start by creating a dictionary. For this example, we will use data for the movie *Minority Report*, directed by Steven Spielberg, with a soundtrack by John Williams… This is a must-see movie for any Python programmer.

```
>>> cool_movie = {}
>>> cool_movie["title"] = "Minority Report"
>>> cool_movie["director"] = "Steven Spielberg"
>>> cool_movie["composer"] = "John Williams"
>>> cool_movie["actors"] = ["Tom Cruise", "Colin Farrel", "Samantha Morton", "Max von␣
↪Sydow"]
>>> cool_movie["is_awesome"] = True
>>> cool_movie["budget"] = 102000000
>>> cool_movie["cinematographer"] = "Janus Kaminski"
>>>
```

To write this object to a file in JSON format, we must first open a file. Next call the **dump()** method, passing in the dictionary as the first argument and the file second:

```
>>> with open("cool_movie.json", "wt") as outfile:
...     json.dump(cool_movie, outfile)
...
>>>
```

If we open the file we see that all the data is in there, everything is properly formatted:

```
{"title": "Minority Report", "director": "Steven Spielberg", "composer": "John␣
↪Williams", "actors": ["Tom Cruise", "Colin Farrel", "Samantha Morton", "Max von␣
↪Sydow"], "is_awesome": true, "budget": 102000000, "cinematographer": "Janus Kaminski
↪"}
```

2.12.4 Pretty printing JSON

To analyze and debug JSON data, we may need to print it in a more readable format. This can be done by passing additional parameters **indent** and **sort_keys** to **json.dumps()** and **json.dump()** method.

```python
import json

person_string = '{"name": "Bob", "languages": "English", "numbers": [2, 1.6, null]}'

# Getting dictionary
person_dict = json.loads(person_string)

# Pretty Printing JSON string back
print(json.dumps(person_dict, indent=4, sort_keys=True))
```

This will make sure the output is indented and keys are ordered in ascending order. If you do not specify any of these arguments, the default value of **indent** is **None**, and for **sort_keys** is **False**.

```
{
    "languages": "English",
    "name": "Bob",
    "numbers": [
        2,
        1.6,
        null
    ]
}
```

2.13 Lambda functions

In Python, a *lambda function* refers to a small, anonymous function. We call the functions *anonymous* because technically it has no name, and we don't define it with the standard **def** keyword that we normally use in Python. Instead, lambda functions are defined as one-liners that execute a single expression.

> Before we get started, a quick note: throughout this lesson I will use the terms *anonymous functions*, *lamda functions* or *lambda expressions* interchangeably. They all mean the same thing.

Although they look different, lambda functions behave in the same way as regular functions that are declared using the *def* keyword. They are executed in a similar way as regular Python functions, with the exception that they strictly execute a single expression.

Lambda functions are mainly used for creating small, single-use functions. You'll often see them in-place of what might otherwise be a fully defined function, but written as a lambda to save time and space.

For a concrete description, lambda functions can be understood through the following 3 points:

- A lambda function must always execute a single expression
- An expression is a Python code run by the lambda function, which may or may not return any value
- A lambda function can take any number of input arguments and return any number of output arguments, as long as the function is maintained as a single expression

2.13.1 Definition

Here is the general way to create a lambda expression: you type the keyword `lambda` followed by zero or more inputs, next type a **:** (colon), then finally, you enter a single expression. This expression is the return value. Just like functions, it is perfectly acceptable to have anonymous functions with no inputs. But remember, you cannot use lambda expressions for multi-line functions.

```
lambda arguments: expression
```

For example, let's say we want to declare a lambda function that computes the remainder of a division operation. Of course, we could do this without a function with Python's `%` operator, but it's not very readable when going through the code. It's easy to miss and not intuitive to catch when reading through for the first time.

When we use a lambda function however, we're able to clean things up for better readability and cleaner code:

```
compute_remainder = lambda x, y: x % y
```

Our lambda function takes on 2 arguments, `x` and `y`, and then computes the remainder of those 2 using Python's `%` operator via `x % y`. To call the function, all we have to do is apply it like any other regular Python function by passing the arguments and saving the return value in a variable.

```
### Prints out 1
r = compute_remainder(10, 3)
print(r)
```

Our code using the lambda function is simple and contained. Let us now see another example. Suppose you are processing user data from a web registration form, and would like to combine the first and last names into a single string *full name* for displaying on the user interface.

We will call this lambda expression `full_name`, which will take two arguments *first* and *last* names, and before combining them together we will clean up any extra whitespace characters from the names, using **strip()**, and apply the titlecase transformation using **title()**. This is necessary because humans are sloppy when typing.

```
full_name = lambda first, last: first.strip().title() + " " + last.strip().title()
print(full_name("   leonhard", "EULER"))
# Will print out: 'Leonhard Euler'
```

We should not judge Euler's typing skills, as this is the first time he has ever used a computer.

2.13.2 Trully anonymous

Let us now see a common use of lambda expressions where we do not give it a name. Suppose we have a list of science fiction authors. We would like to sort this list by last name. Notice that some of these authors have a middle name, while others have initials.

```
>>> scifi_authors = ["Isaac Asimov", "Ray Bradbury", "Robert Heinlein", "Arthur C.␣
↪Clarke", "Frank Herbert", "Orson Scott Card", "Douglas Adams", "H. G. Wells",
↪"Leigh Brackett"]
```

Our strategy will be to create an anonymous function that extracts the last name and uses that as the sorting criteria.

> To access the last name, split the string into pieces wherever it has a space.
>
> Next, access the last piece by index **-1**
>
> And, as a final precaution, convert the string to lowercase. This way the sorting is not case-sensitive. Trust me, some people do not know how to use the shift key.

```
>>> ordered_authors = sorted(scifi_authors, key=lambda name: name.split("
↪")[-1].lower())
>>> print(ordered_authors)
```

(continues on next page)

(continued from previous page)

```
['Douglas Adams', 'Isaac Asimov', 'Leigh Brackett', 'Ray Bradbury', 'Orson␣
↪Scott Card', 'Arthur C. Clarke', 'Robert Heinlein', 'Frank Herbert', 'H. G.
↪ Wells']
>>>
```

The list is now in alphabetical order. These names are a pleasure to read.

2.13.3 Higher order functions

We must go deeper. The power of lambda is better shown when we define a function that makes functions. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def create_multiplier_function(n):
    return lambda x: x * n
```

Use that function definition to make a function that always *doubles* the number you send in. Or, use the same function definition to make a function that always *triples* the number you send in

```
>>> my_doubler = create_multiplier_function(2)
>>> print(my_doubler(11))
22
>>> my_tripler = create_multiplier_function(3)
>>> print(my_tripler(11))
33
>>>
```

Suppose you are working with quadratic functions. Perhaps you are computing the trajectories of cannonballs. To do this, let's write a function called **make_quadratic_function**, and its inputs are the three coefficients **a**, **b** and **c**:

```
def make_quadratic_function(a, b, c):
    """Creates a function f(x) = ax^2 + bx + c"""
    return lambda x: a*x**2 + b*x + c
```

And use this function definition to create a function that always doubles the number you send in:

Let's test this by creating the function $2x^2 + 3x - 5$

```
>>> f = make_quadratic_function(2, 3, -5)
>>> f(0)
-5
>>> f(1)
0
>>> f(2)
9
>>>
```

You can see this function works correctly.

Lambda expressions are quite useful when you need a short, throwaway function. Something simple that you will only use once. Common applications are sorting and filtering data.

2.14 Exceptions

Long before the computing era, Benjamin Franklin once said: *"If you fail to plan, you are planning to fail."* As you may have already noticed, when running computer programs, something unexpected always happens. Even when they are really simple, like the ones in these lessons: maybe a file is not there when you try to read it, or you type a string where a number was needed or simply you press <*Enter*> when you should have entered some text.

If there is a way for code to break, then somehow, somewhere, someone will break it. Using *exceptions* you can manage these problems in a responsible way: if a client asks you to capitalize an integer you can decline their bizarre suggestion. Once you learn how to do this, you will become an exceptional Python programmer.

An *exception* is an error that is *raised* by Python, when the execution of our code results in an unexpected outcome. Normally, an exception will have a *type* and a *message*. For example, when python prints such exceptions:

```
ZeroDivisionError: division by zero
TypeError: must be str, not int
IndexError: list index out of range
```

The **ZeroDivisionError**, **TypeError** and **IndexError** represent the *error type* and the text that follows the colon is the *error message*. This message usually describes the cause of the error in more detail and/or the expected vs. actual input.

2.14.1 The traceback

To illustrate the exceptions I am going to deliberately make mistakes in my code. The mere thought of this causes my neural nets to rebel, but this is for a good cause.

Let us first write some code to print `Hello, world!` three times:

```
for i in range(3)
    print("Hello, world!")
```

Then run:

```
macbook$ python3 mistakes.py
File "/Users/python-1o1/examples/mistakes.py", line 1
    for i in range(3)
                    ^
SyntaxError: invalid syntax
```

Instead of our welcoming, we see a problem: the last line displays the error type and also a description. A **syntax error** means you did not follow the rules for how to write valid Python code. You will encounter this a lot when you're first learning Python. Above you will see some text that tells you where the problem occurred, this text is called a **traceback**. Here, we're missing a colon at the end of the for loop, if we correct and run again everything works as expected:

```
Hello, world!
Hello, world!
Hello, world!
```

2.14.2 Common exceptions

Let us see a few more common exceptions before we learn how to raise and handle them. Open the **x-files.txt** and read it. This should be exciting to read.

```
with open("x-files.txt") as textfile:
    the_truth = textfile.read()
print(the_truth)
```

Why am I not surprised:

```
Traceback (most recent call last):
File "/Users/python-1o1/examples/mistakes.py", line 4, in <module>
    with open("x-files.txt") as textfile:
FileNotFoundError: [Errno 2] No such file or directory: 'x-files.txt'
```

Whenever we try to read a non-exitent file Python raises a **FileNotFoundError**. The truth must still be out there.

What if you try to add one, two and three but instead of the number 3 you use the word **three**:

```
>>> 1 + 2 + "three"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

How I laughed while typing that. Here you get a **TypeError** with a more helpful description. Python lets you know that you cannot add an integer and a string. This exception is very common, it occurs when you expect one type of data but receive another.

```
first_name = "Benjamin"
print(fist_name)
```

```
Traceback (most recent call last):
File "/Users/python-1o1/examples/mistakes.py", line 5, in <module>
    print(fist_name)
NameError: name 'fist_name' is not defined
```

The **NameError** exception is thrown whenever we try to use a variable that does not exist, or a function that was not defined. The error message is quite clear, you should be able to fix it easily. Most often there is a misspelled name or a missing import.

2.14.3 Handling exceptions

Python provides us with the **try except** construction to handle exceptions that might be raised by our code. The basic anatomy of the **try except** clause is as follows

```
try:
    # this code runs first
    ...
except:
    # runs when the code above fails
    ...
```

In plain English, the try except clause is basically saying: *Try to do this, except if there's an error, then do this instead.*

There are a few options on what to do with the thrown exception from the try block. Let's discuss them.

2.14.4 Catch specific exceptions

Python allows us to define which exception types we want to *catch* explictily. To do this, we need to specify the type to the **except** block.

```
a = 42
b = "ten"
try:
    sum = a + b
except TypeError:
    print("Cannot sum the variables, please pass numbers only.")
```

And when we run this code we get:

```
macbook$ python3 mistakes.py
Cannot sum the variables, please pass numbers only.
```

It is starting to look better, isn't it? Now, we can actually **log** or **print** the exception itself:

```
a = 42
b = "ten"
try:
    sum = a + b
except TypeError as err:
    print(f"Cannot sum the variables, reason: {err}.")
```

```
macbook$ python3 mistakes.py
Cannot sum the variables, reason: unsupported operand type(s) for +: 'int' and 'str'.
```

Furthermore, we can catch multiple exception types in one clause if we want to handle them in the same way:

```
a = 42
b = 10
try:
    sum = a + b + c
except (TypeError, NameError) as err:
    print(f"Cannot sum the variables, reason: {err}.")
```

```
macbook$ python3 mistakes.py
Cannot sum the variables, reason: name 'c' is not defined.
```

Basically, we just need to pass a **tuple** containing the exception types that we need to handle.

Sometimes, we may need another **except** block with no specific exception type. The purpose of this is to catch all exception types. A typical use case would be when making a request to a third party API, we might not know all of the possible exception types, however, we still want to catch and handle them all.

```
a = 42
b = 10
c = 0
try:
    remainder = (a + b) % c
except (TypeError, NameError) as err:
    print(f"Cannot compute the result, reason: {err}.")
except Exception as err:
    print(f"A {type(err)} occurred, please inspect: {err}.")
```

```
macbook$ python3 mistakes.py
A <class 'ZeroDivisionError'> occurred, please inspect: integer division or modulo by␣
↪zero.
```

In the case our code raises any exceptions other than TypeError and NameError, it will run last except block and print the message.

Notice that we used the **Exception** keyword to specify the generic error type that will match any exceptions.

See also:

Python comes with a large collection of built-in exceptions. Here you can explore them organized in a logical hierarchy. Notice how most of these class names end in **Error** and not in **Exceptions**, like in many other languages.

Just so you know, we can still extend our try except clause by adding **else** and **finally** blocks to it. More on this in a future lesson. Or if you want to read about it yourself, you can checkout the official documentation.

2.15 Requests

There are more pages on the web than people on Earth. And while I have not checked, I am sure each on is full of original, high quality content that would make our ancestors proud.

Most people access web pages through a browser, but as programmers we have other methods. . . Today, we will learn how to use Python to send GET requests to web servers, and then parse the response. This way, you can write software to read websites for you, giving you more time to [. . .] browse the internet.

URL

In a browser, you access a web page by typing the *URL* in the address bar. URL stands for **Uniform Resource Locator** and this string can hold a lot of information, for example: `https://en.wikipedia.org/wiki/Pythonidae?filter=none&select=42#Relationship_with_humans` contains the following components:

- Protocol: `https`
- Hostname: `en.wikipedia.org`
- Path: `wiki/Pythonidae`
- Querystring: `?filter=none&select=42`
- Fragment: `#Relationship_with_humans`

The **requests** library is the de facto standard for making HTTP requests in Python. It hides the complexities of making requests behind a beautiful and simple API, so you can focus on interacting with services and consuming data in your application.

See also:

Though I've tried to include as much information as you need to understand the features and examples included in this lesson, I do assume a very basic general knowledge of HTTP requests.

2.15.1 Getting started

Let's begin by installing the **requests** module. To do so, run the following command:

```
macbook$ pip install requests
```

Once installed, please check the installation by trying to import the module in your python console:

```
>>> import requests
>>>
```

If Python does not report any error it means you're all set, and it is time to begin your journey through web.

2.15.2 The GET request

The HTTP methods such as GET and POST, determine what kind of action you want the web server to perform. One of the most common methods is GET. This indicates that you want to retrieve data from a specified location (or resource). To make such a request in your application all you need is calling `requests.get()` and specifying a URL (or an address).

To test this out, let's make a request to GitHub's root API:

```
>>> requests.get("https://api.github.com")
<Response [200]>
>>>
```

This is it. You've made your first request. Notice that we got a `<Response [200]>` message which is the representation of a response object with code *200* (the http numeric code for success).

2.15.3 The Response

A *Response* is a powerful object for inspecting the results of the request. Let's make that same request again, but this time store the return value in a variable so that you can get a closer look at its attributes and behaviors:

```
>>> response = requests.get("https://api.github.com")
>>>
```

Status Code

The first bit of information that you can gather from **Response** is the **status code**. A status code informs you of the status of the request.

For example, a **200** status means that your request was successful, whereas a **404** status means that the resource you were looking for was *not found*. There are many other possible status codes as well to give you specific insights into what happened with your request.

Sometimes, you might want to use this information to make decisions in your code:

```
if response.status_code == 200:
    print("Request completed successfully.")
elif response.status_code == 404:
    print("Resource not found.")
elif response.status_code == 500:
    print("The server reported an internal error. Please try again later.")
```

The **requests** module goes one step further in simplifying this process for you. If you use a **Response** object in a conditional expression, it will evaluate to *True* if the status code was between 200 and 400, and *False* otherwise. Therefore you can simplify the last example to:

```
if response:
    print("Request completed successfully.")
else:
    print(f"Request has failed, with code {response.status_code}.")
```

Keep in mind that this method is not verifying that the status code is equal to **200**. The reason for this is that other status codes within the 200 to 400 range, such as **204 NO CONTENT** and **304 NOT MODIFIED**, are also considered successful in the sense that they provide some workable response. For example, the 204 tells you that the response was successful, but there's no content to return in the message body.

So, make sure you use this convenient shorthand only if you want to know if the request was generally successful and then, if necessary, handle the response appropriately based on the status code.

Content

Now, you know a lot about how to deal with the status code of the response you got back from the server. However, when you make a GET request, you rarely only care about the status code of the response. Usually, you want to see more.

The response of a GET request often has some valuable information, known as a payload, in the message body. Using the attributes and methods of Response, you can view the payload in a variety of different formats.

To see the response's content in bytes, you use `.content` attribute:

```
>>> response = requests.get('https://api.github.com')
>>> response.content
b'{"current_user_url":"https://api.github.com/user","current_user_authorizations_html_
↪url":"https://github.com/settings/connections/applications{/client_id}",
↪"authorizations_url":"https://api.github.com/authorizations","code_search_url":
↪"https://api.github.com/search/code?q={query}{&page,per_page,sort,order}","commit_
↪search_url":"https://api.github.com/search/commits?q={query}{&page,per_page,sort,
↪order}","emails_url":"https://api.github.com/user/emails","emojis_url":"https://api.
↪github.com/emojis","events_url":"https://api.github.com/events","feeds_url":"https:/
↪/api.github.com/feeds","followers_url":"https://api.github.com/user/followers",
↪"following_url":"https://api.github.com/user/following{/target}","gists_url":
↪"https://api.github.com/gists{/gist_id}","hub_url":"https://api.github.com/hub",
↪"issue_search_url":"https://api.github.com/search/issues?q={query}{&page,per_page,
↪sort,order}","issues_url":"https://api.github.com/issues","keys_url":"https://api.
↪github.com/user/keys","notifications_url":"https://api.github.com/notifications",
↪"organization_repositories_url":"https://api.github.com/orgs/{org}/repos{?type,page,
↪per_page,sort}","organization_url":"https://api.github.com/orgs/{org}","public_
↪gists_url":"https://api.github.com/gists/public","rate_limit_url":"https://api.
↪github.com/rate_limit","repository_url":"https://api.github.com/repos/{owner}/{repo}
↪","repository_search_url":"https://api.github.com/search/repositories?q={query}{&
↪page,per_page,sort,order}","current_user_repositories_url":"https://api.github.com/
↪user/repos{?type,page,per_page,sort}","starred_url":"https://api.github.com/user/
↪starred{/owner}{/repo}","starred_gists_url":"https://api.github.com/gists/starred",
↪"team_url":"https://api.github.com/teams","user_url":"https://api.github.com/users/
↪{user}","user_organizations_url":"https://api.github.com/user/orgs","user_
↪repositories_url":"https://api.github.com/users/{user}/repos{?type,page,per_page,
↪sort}","user_search_url":"https://api.github.com/search/users?q={query}{&page,per_
↪page,sort,order}"}'
>>>
```

While `.content` gives you access to the raw bytes of the response payload, you will often want to convert them into a string using a character encoding such as *UTF-8*. **response** will do that for you when you access `.text`:

```
>>> response.text
'{"current_user_url":"https://api.github.com/user","current_user_authorizations_html_
↪url":"https://github.com/settings/connections/applications{/client_id}",
↪"authorizations_url":"https://api.github.com/authorizations","code_search_url":
↪"https://api.github.com/search/code?q={query}{&page,per_page,sort,order}","commit_
↪search_url":"https://api.github.com/search/commits?q={query}{&page,per_page,sort,
↪order}","emails_url":"https://api.github.com/user/emails","emojis_url":"https://api.
↪github.com/emojis","events_url":"https://api.github.com/events","feeds_url":"https:/
↪/api.github.com/feeds","followers_url":"https://api.github.com/user/followers",
↪"following_url":"https://api.github.com/user/following{/target}","gists_url":
↪"https://api.github.com/gists{/gist_id}","hub_url":"https://api.github.com/hub",
↪"issue_search_url":"https://api.github.com/search/issues?q={query}{&page,per_page,
↪sort,order}","issues_url":"https://api.github.com/issues","keys_url":"https://api.
↪github.com/user/keys","label_search_url":"https://api.github.com/search/labels?q=
↪{query}&repository_id={repository_id}{&page,per_page}","notifications_url":"https://
↪api.github.com/notifications","organization_url":"https://api.github.com/orgs/{org}
↪","organization_repositories_url":"https://api.github.com/orgs/{org}/repos{?type,
↪page,per_page,sort}","organization_teams_url":"https://api.github.com/orgs/{org}/
↪teams","public_gists_url":"https://api.github.com/gists/public","rate_limit_url":
↪"https://api.github.com/rate_limit","repository_url":"https://api.github.com/repos/
↪{owner}/{repo}","repository_search_url":"https://api.github.com/search/repositories?
↪q={query}{&page,per_page,sort,order}","current_user_repositories_url":"https://api.
↪github.com/user/repos{?type,page,per_page,sort}","starred_url":"https://api.github.
↪com/user/starred{/owner}{/repo}","starred_gists_url":"https://api.github.com/gists/
↪starred","user_url":"https://api.github.com/users/{user}","user_organizations_url":
↪"https://api.github.com/user/orgs","user_repositories_url":"https://api.github.com/
↪users/{user}/repos{?type,page,per_page,sort}","user_search_url":"https://api.github.
↪com/search/users?q={query}{&page,per_page,sort,order}"}'
>>>
```

Because the decoding of *bytes* to a *string* requires an encoding scheme, requests will try to guess the encoding based on the response's headers if you do not specify one. You can provide an explicit encoding by setting `.encoding` before accessing `.text` attribute:

```
>>> response.encoding = 'utf-8' # Optional: requests infers this internally
>>> response.text
'{"current_user_url":"https://api.github.com/user","current_user_authorizations_html_
↪url":"https://github.com/settings/connections/applications{/client_id}",
↪"authorizations_url":"https://api.github.com/authorizations","code_search_url":
↪"https://api.github.com/search/code?q={query}{&page,per_page,sort,order}","commit_
↪search_url":"https://api.github.com/search/commits?q={query}{&page,per_page,sort,
↪order}","emails_url":"https://api.github.com/user/emails","emojis_url":"https://api.
↪github.com/emojis","events_url":"https://api.github.com/events","feeds_url":"https:/
↪/api.github.com/feeds","followers_url":"https://api.github.com/user/followers",
↪"following_url":"https://api.github.com/user/following{/target}","gists_url":
↪"https://api.github.com/gists{/gist_id}","hub_url":"https://api.github.com/hub",
↪"issue_search_url":"https://api.github.com/search/issues?q={query}{&page,per_page,
↪sort,order}","issues_url":"https://api.github.com/issues","keys_url":"https://api.
↪github.com/user/keys","label_search_url":"https://api.github.com/search/labels?q=
↪{query}&repository_id={repository_id}{&page,per_page}","notifications_url":"https://
↪api.github.com/notifications","organization_url":"https://api.github.com/orgs/{org}
↪","organization_repositories_url":"https://api.github.com/orgs/{org}/repos{?type,
↪page,per_page,sort}","organization_teams_url":"https://api.github.com/orgs/{org}/
↪teams","public_gists_url":"https://api.github.com/gists/public","rate_limit_url":
↪"https://api.github.com/rate_limit","repository_url":"https://api.github.com/repos/
↪{owner}/{repo}","repository_search_url":"https://api.github.com/search/repositories?
↪q={query}{&page,per_page,sort,order}","current_user_repositories_url":"https://api.
↪github.com/user/repos{?type,page,per_page,sort}","starred_url":"https://api.github.
↪com/user/starred{/owner}{/repo}","starred_gists_url":"https://api.github.com/gists/
↪starred","user_url":"https://api.github.com/users/{user}","user_organizations_url":
↪"https://api.github.com/user/orgs","user_repositories_url":"https://api.github.com/
↪users/{user}/repos{?type,page,per_page,sort}","user_search_url":"https://api.github.
↪com/search/users?q={query}{&page,per_page,sort,order}"}'
```

(continues on next page)

(continued from previous page)

```
>>>
```

If you take a look at the response, you'll see that it is actually serialized JSON content. To get a dictionary, you could take the string you retrieved from `.text` and deserialize it using `json.loads()`. However, a simpler way to accomplish this task is to use `.json()` method:

```
>>> response.json()
{'current_user_url': 'https://api.github.com/user', 'current_user_authorizations_html_
↪url': 'https://github.com/settings/connections/applications{/client_id}',
↪'authorizations_url': 'https://api.github.com/authorizations', 'code_search_url':
↪'https://api.github.com/search/code?q={query}{&page,per_page,sort,order}', 'commit_
↪search_url': 'https://api.github.com/search/commits?q={query}{&page,per_page,sort,
↪order}', 'emails_url': 'https://api.github.com/user/emails', 'emojis_url': 'https://
↪api.github.com/emojis', 'events_url': 'https://api.github.com/events', 'feeds_url':
↪'https://api.github.com/feeds', 'followers_url': 'https://api.github.com/user/
↪followers', 'following_url': 'https://api.github.com/user/following{/target}',
↪'gists_url': 'https://api.github.com/gists{/gist_id}', 'hub_url': 'https://api.
↪github.com/hub', 'issue_search_url': 'https://api.github.com/search/issues?q={query}
↪{&page,per_page,sort,order}', 'issues_url': 'https://api.github.com/issues', 'keys_
↪url': 'https://api.github.com/user/keys', 'label_search_url': 'https://api.github.
↪com/search/labels?q={query}&repository_id={repository_id}{&page,per_page}',
↪'notifications_url': 'https://api.github.com/notifications', 'organization_url':
↪'https://api.github.com/orgs/{org}', 'organization_repositories_url': 'https://api.
↪github.com/orgs/{org}/repos{?type,page,per_page,sort}', 'organization_teams_url':
↪'https://api.github.com/orgs/{org}/teams', 'public_gists_url': 'https://api.github.
↪com/gists/public', 'rate_limit_url': 'https://api.github.com/rate_limit',
↪'repository_url': 'https://api.github.com/repos/{owner}/{repo}', 'repository_search_
↪url': 'https://api.github.com/search/repositories?q={query}{&page,per_page,sort,
↪order}', 'current_user_repositories_url': 'https://api.github.com/user/repos{?type,
↪page,per_page,sort}', 'starred_url': 'https://api.github.com/user/starred{/owner}{/
↪repo}', 'starred_gists_url': 'https://api.github.com/gists/starred', 'user_url':
↪'https://api.github.com/users/{user}', 'user_organizations_url': 'https://api.
↪github.com/user/orgs', 'user_repositories_url': 'https://api.github.com/users/{user}
↪/repos{?type,page,per_page,sort}', 'user_search_url': 'https://api.github.com/
↪search/users?q={query}{&page,per_page,sort,order}'}
>>> response.json()["current_user_url"]
'https://api.github.com/user'
>>> response.json()["events_url"]
'https://api.github.com/events'
>>>
```

The type of the return value of `.json()` is a *dictionary*, so you can access values in the object by key.

Headers

You can do a lot with status codes and message bodies. But, if you need more information, like metadata about the response itself, you'll need to look at the response's headers. They contain a lot of useful information, such as the *content type* of the response payload and a time limit on how long to cache the response. To view these headers, access the `.headers` attribute:

```
>>> response.headers
{'Server': 'GitHub.com', 'Date': 'Fri, 04 Jun 2021 07:42:54 GMT', 'Cache-Control':
↪'public, max-age=60, s-maxage=60', 'Vary': 'Accept, Accept-Encoding, Accept, X-
↪Requested-With', 'ETag': '
↪"27278c3efffccc4a7be1bf315653b901b14f2989b2c2600d7cc2e90a97ffbf60"', 'Access-
↪Control-Expose-Headers': 'ETag, Link, Location, Retry-After, X-GitHub-OTP, X-
↪RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Used, X-RateLimit-Resource, X-
↪RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-GitHub-
↪Media-Type, Deprecation, Sunset', 'Access-Control-Allow-Origin': '*', 'Strict-
↪Transport-Security': 'max-age=31536000; includeSubdomains; preload', 'X-Frame-
↪Options': 'deny', 'X-Content-Type-Options': 'nosniff', 'X-XSS-Protection': '0',
↪'Referrer-Policy': 'origin-when-cross-origin, strict-origin-when-cross-origin',
↪'Content-Security-Policy': "default-src 'none'", 'Content-Type': 'application/json;
```

(continues on next page)

(continued from previous page)

```
>>>
```

`.headers` returns a dictionary-like object, allowing you to access header values by key. For example, to see the content type of the response payload, you can access `Content-Type` key:

```
>>> response.headers["Content-Type"]
'application/json; charset=utf-8'
>>>
```

There is something special about this dictionary-like headers object, though. The HTTP specifications defines headers to be case-insensitive, which means we are able to access these headers without worrying about their capitalization:

```
>>> response.headers["content-type"]
'application/json; charset=utf-8'
>>>
```

Whether you use the key **content-type** or **Content-Type** key, you will get the same value.

Now, you've learned the basics about Response. You've seen its most useful attributes and methods in action. Let's take a step back and see how your responses change when you customize your GET requests.

2.15.4 QueryString parameters

One common way to customize a GET request is to pass values through query string parameters in the URL. To do this using `get()`, you pass data through params argument. For example, you can use GitHub's Search API to look for the requests library:

```python
import requests

# Search GitHub's repositories for requests
response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "requests+language:python"},
)

# Inspect some attributes of the `requests` repository
json_response = response.json()
repository = json_response["items"][0]
print(f'Repository name: {repository["name"]}')  # Python 3.6+
print(f'Repository description: {repository["description"]}')  # Python 3.6+
```

```
macbook$ python3 web_requests.py
Repository name: grequests
Repository description: Requests + Gevent = <3
macbook$ _
```

By passing the dictionary `{"q": "requests+language:python"}` to the **params** argument of `.get()`, you are able to modify the results that come back from the Search API.

You can pass params to **get()** in the form of a dictionary, as you have just done, or as a list of tuples:

2.15.5 Other HTTP Methods

Aside from GET, other popular HTTP methods include POST, PUT, DELETE, HEAD, PATCH, and OPTIONS. **requests** provides a method, with a similar signature to **get()**, for each of these HTTP methods:

```
>>> requests.post('https://httpbin.org/post', data={'key':'value'})
>>> requests.put('https://httpbin.org/put', data={'key':'value'})
>>> requests.delete('https://httpbin.org/delete')
>>> requests.head('https://httpbin.org/get')
>>> requests.patch('https://httpbin.org/patch', data={'key':'value'})
>>> requests.options('https://httpbin.org/get')
```

Each function call makes a request to the *httpbin service* using the corresponding HTTP method. For each method, you can inspect their responses in the same way you did before:

```
>>> response = requests.head('https://httpbin.org/get')
>>> response.headers['Content-Type']
'application/json'
```

```
>>> response = requests.delete('https://httpbin.org/delete')
>>> json_response = response.json()
>>> json_response['args']
{}
```

Headers, response bodies, status codes, and more are returned in the Response for each method.

Note: **httpbin.org** is a great resource created by the author of requests, Kenneth Reitz. It's a service that accepts test requests and responds with data about the requests

The Message Body

According to the HTTP specification, POST, PUT, and the less common PATCH requests pass their data through the message body rather than through parameters in the query string. Using requests, you'll pass the payload to the corresponding function's data parameter.

The `data` argument takes a *dictionary*, a *list of tuples*, *bytes*, or a *file-like object*. You'll want to adapt the data you send in the body of your request to the specific needs of the service you're interacting with.

For example, if your request's content type is **application/x-www-form-urlencoded**, you can send the form data as a dictionary, but also as a list of tuples:

```
>>> requests.post('https://httpbin.org/post', data={'key':'value'})
<Response [200]>
>>> requests.post('https://httpbin.org/post', data=[('key', 'value')])
<Response [200]>
>>>
```

If, however, you need to send JSON data, you can use the `json` parameter. When you pass JSON data via `json`, requests will serialize your data and add the correct **Content-Type** header for you.

```
>>> response = requests.post("https://httpbin.org/post", json={"key":"value"})
>>> json_response = response.json()
>>> json_response["data"]
'{"key": "value"}'
>>> json_response["headers"]["Content-Type"]
'application/json'
>>>
```

2.16 Indices and tables

- genindex
- modindex
- search