THE PYTHON HANDBOOK

Flavio Copes

Table of Contents

Introduction

The Python Handbook follows the 80/20 rule: learn 80% of the topic in 20% of the time.

I find this approach gives a well-rounded overview.

This book does not try to cover everything under the sun related to Python. It focuses on the core of the language, trying to simplify the more complex topics.

I hope the contents of this book will help you achieve what you want: **learn the basics of Python**.

This book is written by Flavio. I **publish web development tutorials** every day on my website flaviocopes.com.

You can reach me on Twitter @flaviocopes.

Enjoy!

Introduction to Python

Python is literally eating the programming world. It is growing in popularity and usage in ways that are pretty much unprecedented in the history of computers.

There is a huge variety of scenarios that Python excels in. **Shell scripting**, **task automation**, **Web development** are just some basic examples.

Python is the language of choice for **data analysis** and **machine learning**, but it can also adapt to create games and work with embedded devices.

Most importantly, it's the language of choice for introductory **computer science courses** in universities all around the world.

Many students learn Python as their first programming language. Many are learning it right now, many will learn it in the future. And for many of them, Python will be the only programming language they need.

Thanks to this unique position, Python is likely going to grow even more in the future.

The language is simple, expressive, and it's quite straightforward.

The ecosystem is huge. There seems to be a library for everything you can imagine.

Python is a high-level programming language suitable for beginners thanks to its intuitive syntax, its huge community and vibrant ecosystem.

It is also appreciated by professionals across many different fields.

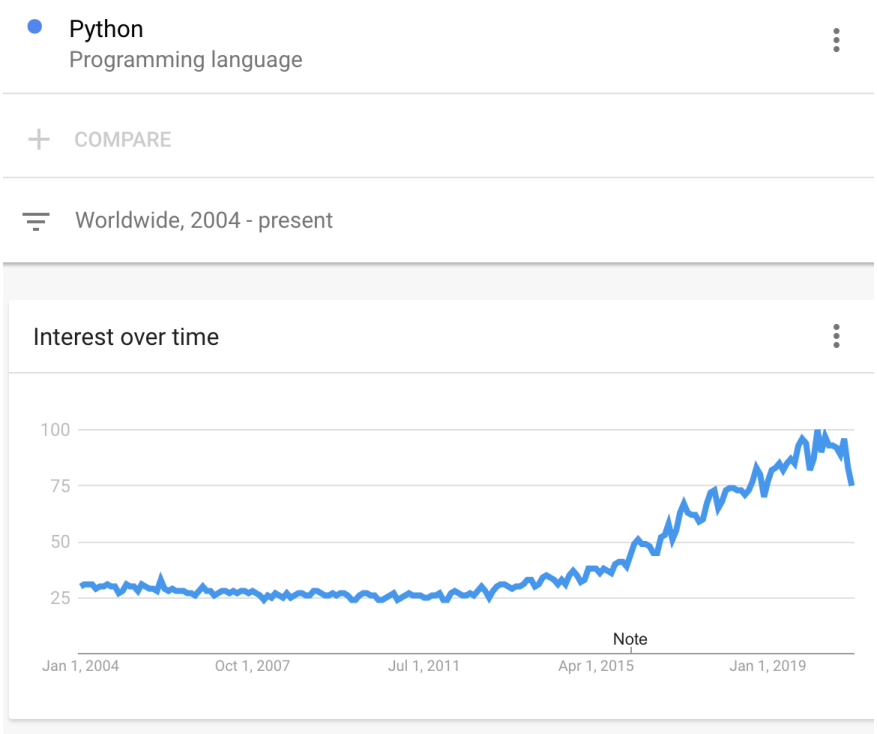
Technically speaking it is an interpreted language that does not have an intermediate compilation phase like a compiled language, for example C or Java.

And like many interpreted languages, it is dynamically typed, which means that you do not have to indicate the types of the variables you use, and variables are not tied to a specific type.

This has pros and cons. In particular we can mention that you write programs faster, but on the other hand you have less help from the tools to prevent possible bugs and you will find out about some kinds of issues only by executing the program at runtime.

Python supports a wide variety of different programming paradigms, including procedural programming, object oriented programming and functional programming. It's flexible enough to adapt to a lot of different needs.

Created in 1991 by Guido van Rossum, it's been rising in popularity - especially in the past 5 years, as this Google Trends infographic shows:



Starting with Python is very easy. All you need is to install the official package from python.org, for Windows, macOS or Linux, and you're ready to go.

If you are new to programming, in the following posts I will guide you to go from zero to becoming a Python programmer.

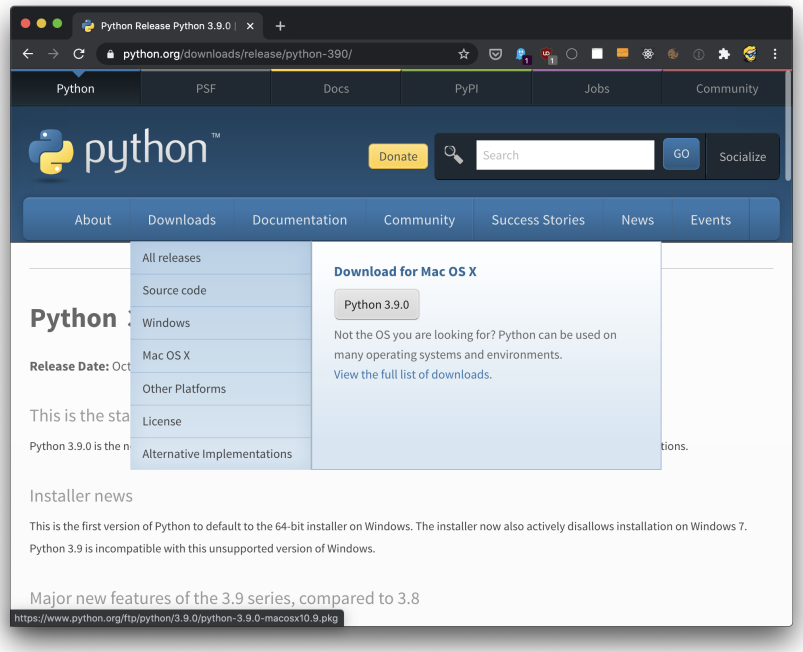
And even if you are currently a programmer specialized into another language, Python is a language worth knowing because I think we're just at the start.

Lower level languages like C++ and Rust might be great for expert programmers, but daunting to begin, and they take a long time to master. Python, on the other hand, is a programming language for programmers, of course, but also for the non-programmers. The students, the people doing their day job with Excel, the scientists.

The language everyone interested in coding should learn first.

Installing Python

Go to https://www.python.org, choose the Downloads menu, choose your operating system and a panel with a link to download the official package will appear:



Make sure you follow the specific instructions for your operating system. On macOS you can find a detailed guide on https://flaviocopes.com/python-installation-macos/.

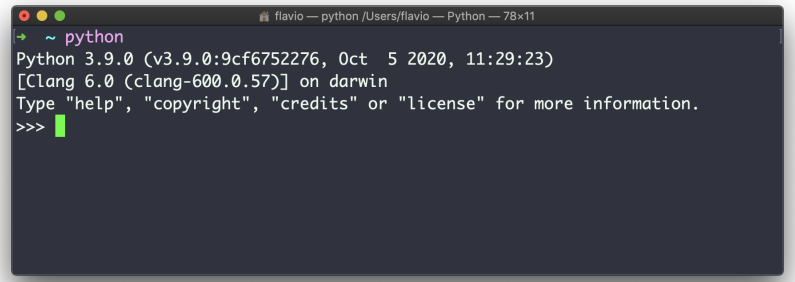
Running Python programs

There are a few different ways to run Python programs.

In particular, you have a distinction between using interactive prompts, where you type Python code and it's immediately executed, and saving a Python program into a file, and executing that.

Let's start with interactive prompts.

If you open your terminal and type `python`, you will see a screen like this:



This is the Python REPL (Read-Evaluate-Print-Loop)

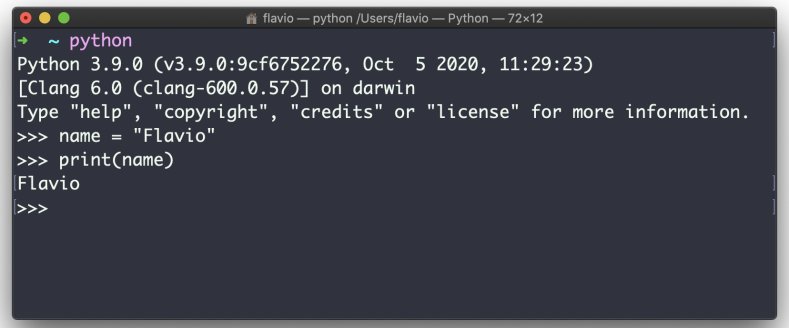
Notice the `>>>` symbol, and the cursor after that. You can type any Python code here, and press the `enter` key to run it.

For example try defining a new variable using

```
name = "Flavio"
```

and then print its value, using `print()`:

```
print(name)
```

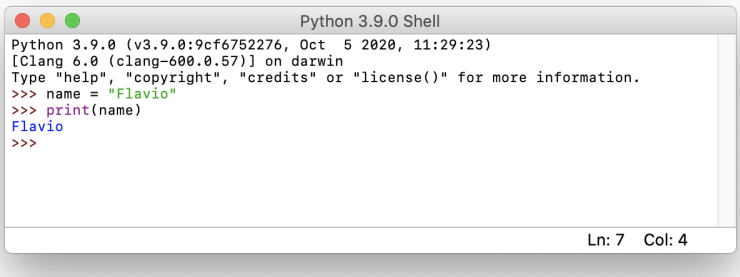


> Note: in the REPL, you can also just type `name`, press the `enter` key and you'll get the value back. But in a program, you are not going to see any output if you do so - you need to use `print()` instead.

Any line of Python you write here is going to be executed immediately.

Type `quit()` to exit this Python REPL.

You can access the same interactive prompt using the IDLE application that's installed by Python automatically:



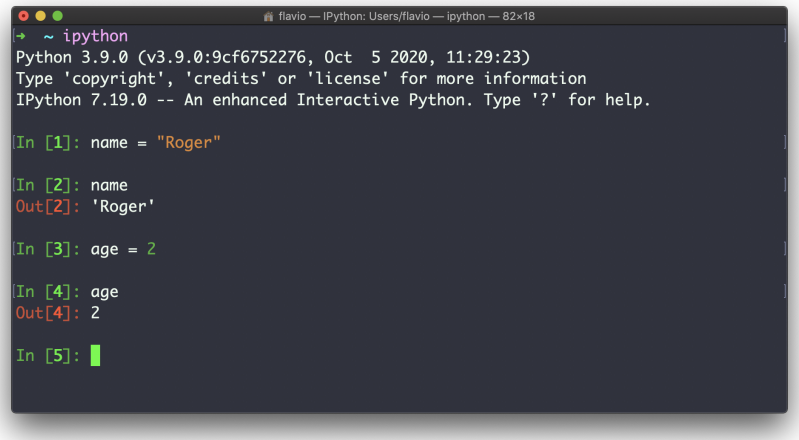
This might be more convenient for you because with the mouse you can move around and copy/paste more easily than with the terminal.

Those are the basics that come with Python by default. However I recommend to install IPython, probably the best command line REPL application you can find.

Install it with

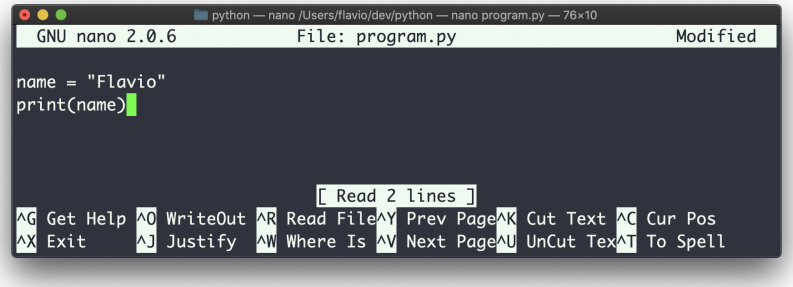
```
pip install ipython
```

Make sure the pip binaries are in your path, then run `ipython`:



`ipython` is another interface to work with a Python REPL, and provides some nice features like syntax highlighting, code completion, and much more.

The second way to run a Python program is to write your Python program code into a file, for example `program.py`:



and then run it with `python program.py`

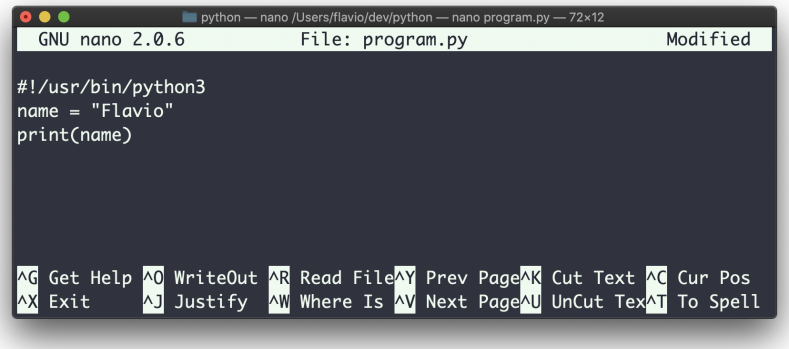
> Note that we save Python programs with the `.py` extension, that's a convention.

In this case the program is executed as a whole, not one line at a time. And that's typically how we run programs.

We use the REPL for quick prototyping and for learning.

On Linux and macOS a Python program can also be transformed into a shell script, by prepending all its content with a special line that indicates which executable to use to run it.

On my system the Python executable is located in `/usr/bin/python3`, so I type `#!/usr/bin/python3` in the first line:

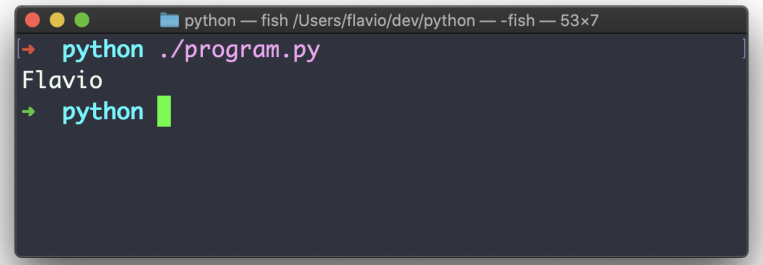


Then I can set execution permission on the file:

```
chmod u+x program.py
```

and I can run the program with

```
./program.py
```



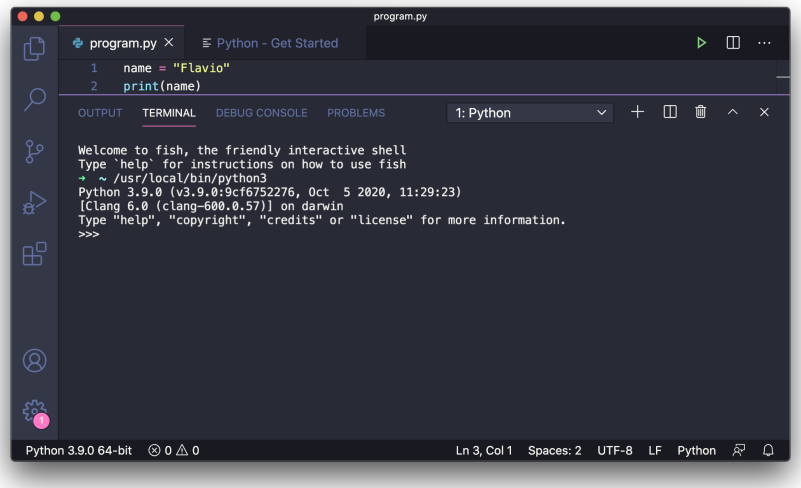
This is especially useful when you write scripts that interact with the terminal.

We have many other ways to run Python programs.

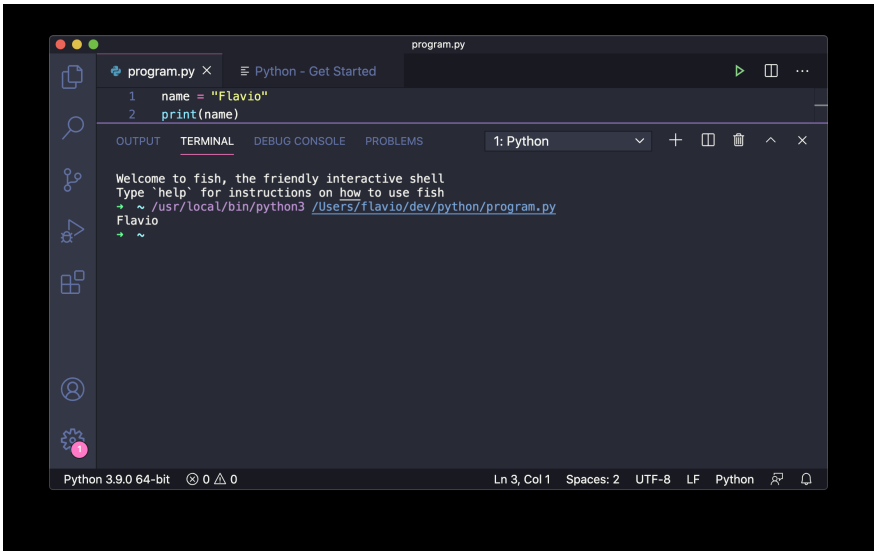
One of them is using VS Code, and in particular the official Python extension from Microsoft:

After installing this extension you will have Python code autocompletion and error checking, automatic formatting and code linting with `pylint`, and some special commands, including:

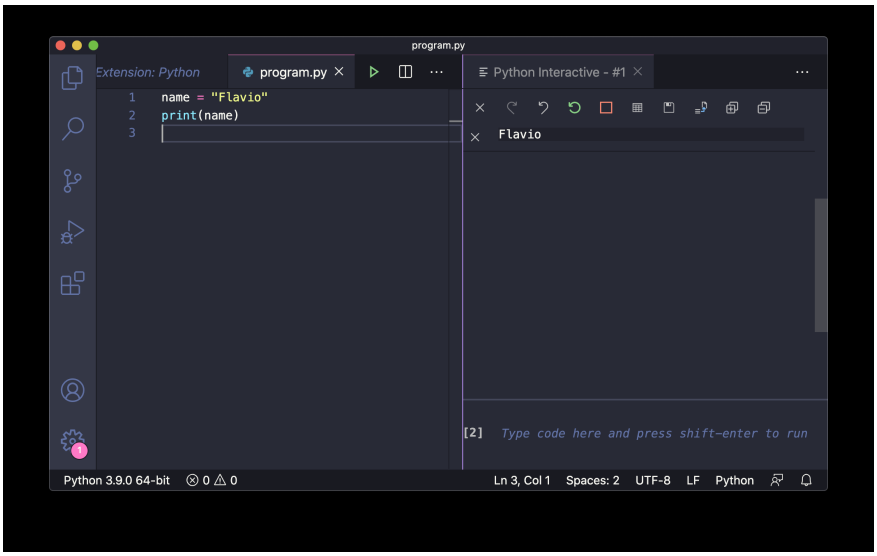
Python: Start REPL to run the REPL in the integrated terminal:



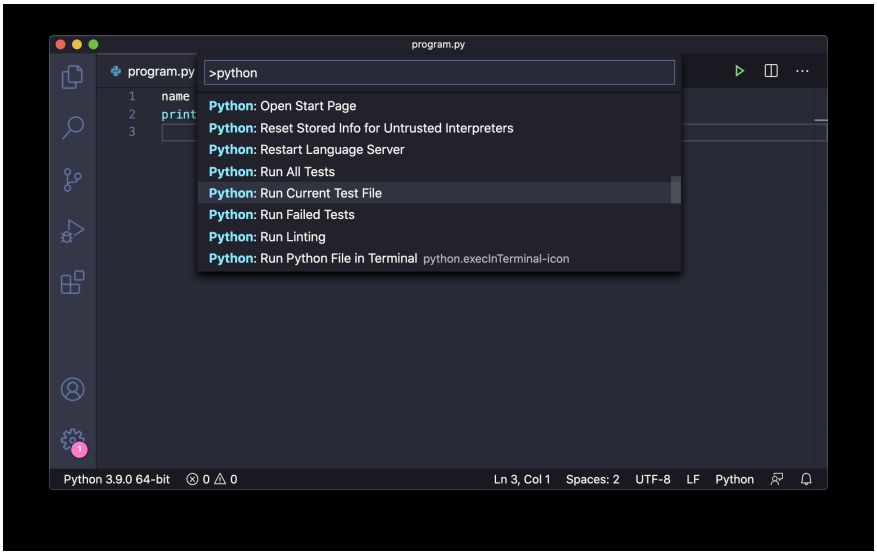
Python: Run Python File in Terminal to run the current file in the terminal:



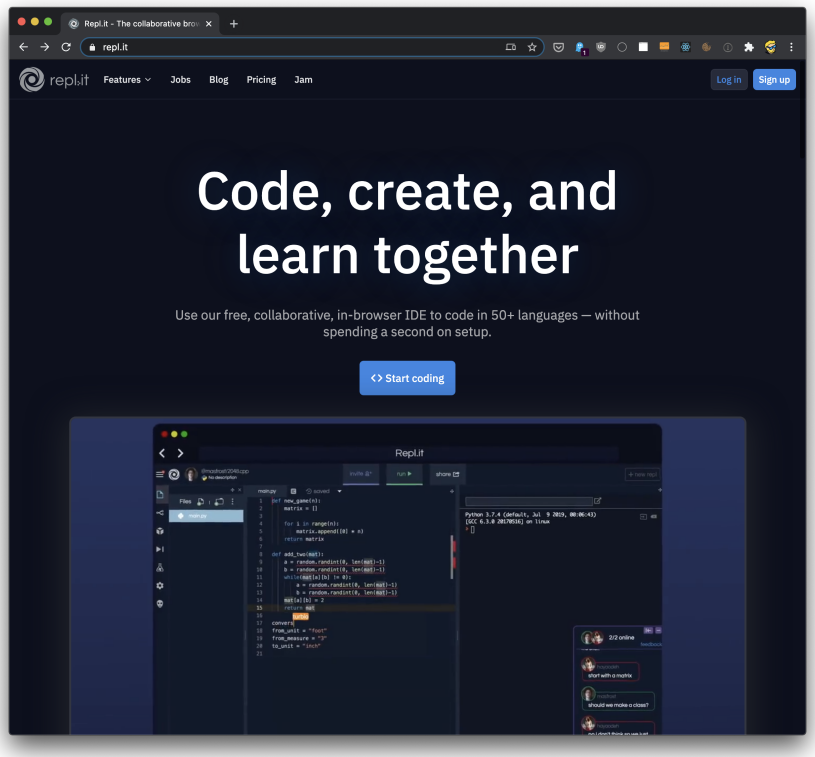
Python: Run Current File in Python Interactive Window:



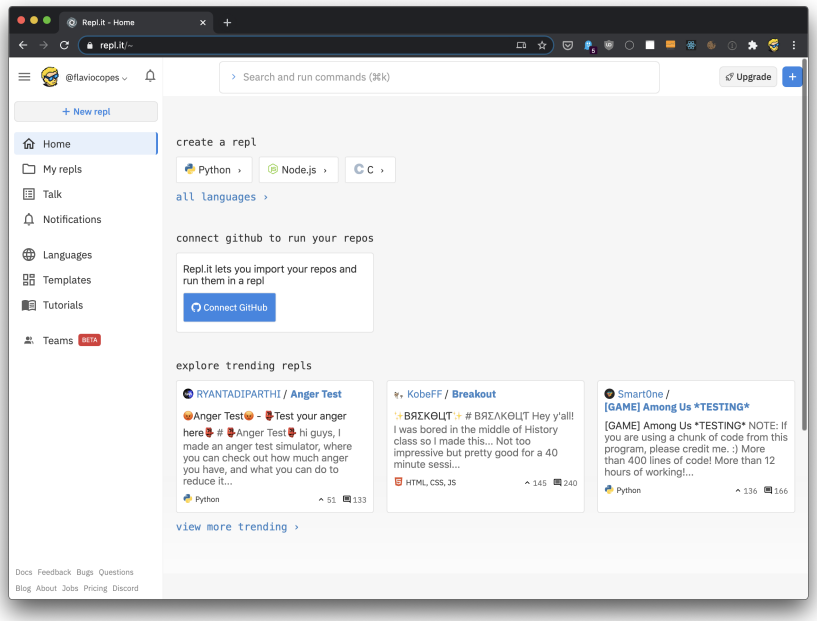
and many more. Just open the command palette (View -> Command Palette, or Cmd-Shift-P) and type `python` to see all the Python-related commands:



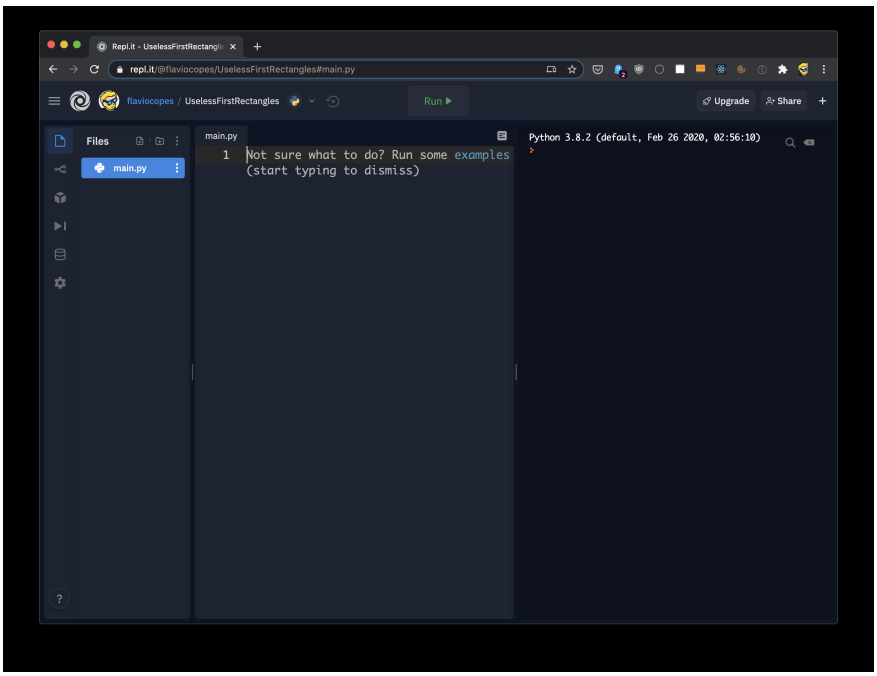
Another way to easily run Python code is to use repl.it, a very nice website that provides a coding environment you can create and run your apps on, in any language, Python included:



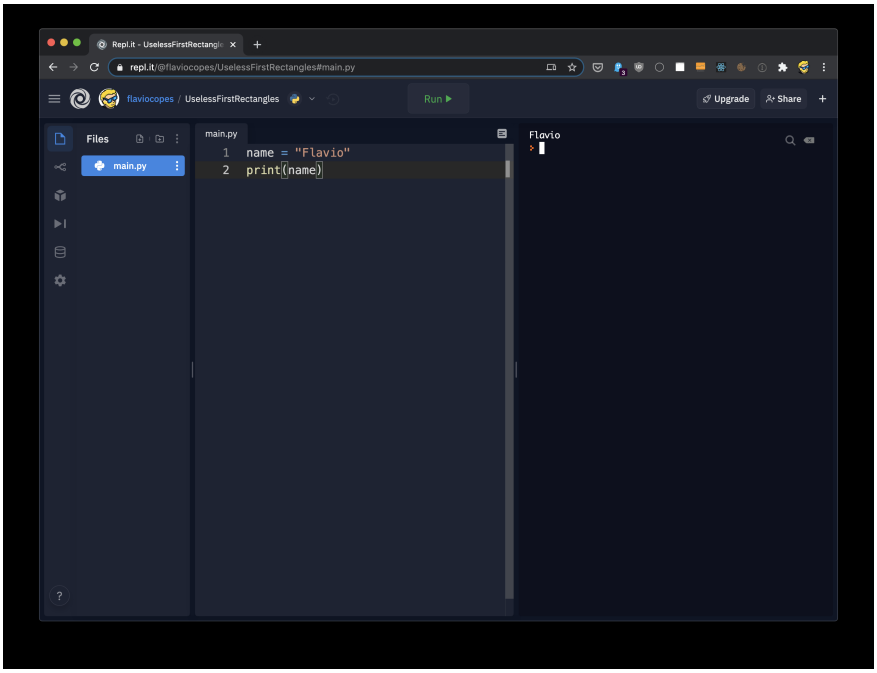
Signup (it's free), then under "create a repl" click Python:



and you will be immediately shown an editor with a `main.py` file, ready to be filled with a lot of Python code:



Once you have some code, click "Run" to run it on the right side of the window:



I think repl.it is handy because:

- you can easily share code just by sharing the link
- multiple people can work on the same code
- it can host long-running programs
- you can install packages
- it provides you a key-value database for more complex applications

Python 2 vs Python 3

One key topic to talk about, right from the start, is the Python 2 vs Python 3 discussion.

Python 3 was introduced in 2008, and it's been in development as the main Python version, while Python 2 continued being maintained with bug fixes and security patches until early 2020.

On that date, Python 2 support was discontinued.

Many programs are still written using Python 2, and organizations still actively work on those, because the migration to Python 3 is not trivial and those programs would require a lot of work to upgrade those programs. And large and important migrations always introduce new bugs.

But new code, unless you have to adhere to rules set by your organization that forces Python 2, should always be written in Python 3.

> This book focuses on Python 3.

The basics of working with Python

Variables

We can create a new Python variable by assigning a value to a label, using the `=` assignment operator.

In this example we assign a string with the value "Roger" to the `name` label:

```
name = "Roger"
```

Here's an example with a number:

```
age = 8
```

A variable name can be composed by characters, numbers, the `_` underscore character. It can't start with a number. These are all **valid** variable names:

```
name1
AGE
aGE
a11111
my_name
_name
```

These are **invalid** variable names:

```
123
test!
name%
```

Other than that, anything is valid unless it's a Python **keyword**. There are some keywords like `for`, `if`, `while`, `import` and more.

There's no need to memorize them, as Python will alert you if you use one of those as a variable, and you will gradually recognize them as part of the Python programming language syntax.

Expressions and statements

We can *expression* any sort of code that returns a value. For example

```
1 + 1
"Roger"
```

A statement on the other hand is an operation on a value, for example these are 2 statements:

```
name = "Roger"
print(name)
```

A program is formed by a series of statements. Each statement is put on its own line, but you can use a semicolon to have more than one statement on a single line:

```
name = "Roger"; print(name)
```

Comments

In a Python program, everything after a hash mark is ignored, and considered a comment:

```
#this is a commented line

name = "Roger" # this is an inline comment
```

Indentation

Indentation in Python is meaningful.

You cannot indent randomly like this:

```
name = "Flavio"
    print(name)
```

Some other languages do not have meaningful whitespace, but in Python, indentation matters.

In this case, if you try to run this program you would get a `IndentationError: unexpected indent` error, because indenting has a special meaning.

Everything indented belongs to a block, like a control statement or conditional block, or a function or class body. We'll see more about those later on.

Data types

Python has several built-in types.

If you create the `name` variable assigning it the value "Roger", automatically this variable is now representing a **String** data type.

```
name = "Roger"
```

You can check which type a variable is using the `type()` function, passing the variable as an argument, and then comparing the result to `str`:

```
name = "Roger"
type(name) == str #True
```

Or using `isinstance()`:

```
name = "Roger"
isinstance(name, str) #True
```

> Notice that to see the `True` value in Python, outside of a REPL, you need to wrap this code inside `print()`, but for clarity reasons I avoid using it

We used the `str` class here, but the same works for other data types.

First, we have numbers. Integer numbers are represented using the `int` class. Floating point numbers (fractions) are of type `float`:

```
age = 1
type(age) == int #True
```

```
fraction = 0.1
type(fraction) == float #True
```

You saw how to create a type from a value literal, like this:

```
name = "Flavio"
age = 20
```

Python automatically detects the type from the value type.

You can also create a variable of a specific type by using the class constructor, passing a value literal or a variable name:

```
name = str("Flavio")
anotherName = str(name)
```

You can also convert from one type to another by using the class constructor. Python will try to determine the correct value, for example extracting a number from a string:

```
age = int("20")
print(age) #20

fraction = 0.1
intFraction = int(fraction)
print(intFraction) #0
```

This is called **casting**. Of course this conversion might not always work depending on the value passed. If you write `test` instead of `20` in the above string, you'll get a `ValueError: invalid literal for int() with base 10: 'test'` error.

Those are just the basics of types. We have a lot more types in Python:

- `complex` for complex numbers
- `bool` for booleans
- `list` for lists
- `tuple` for tuples
- `range` for ranges
- `dict` for dictionaries
- `set` for sets

and more!

We'll explore them all soon.

Operators

Python operators are symbols that we use to run operations upon values and variables.

We can divide operators based on the kind of operation they perform:

- assignment operator
- arithmetic operators
- comparison operators
- logical operators
- bitwise operators

plus some interesting ones like `is` and `in`.

Assignment operator

The assignment operator is used to assign a value to a variable:

```
age = 8
```

Or to assign a variable value to another variable:

```
age = 8
anotherVariable = age
```

Since Python 3.8, the `:=` *walrus operator* is used to assign a value to a variable as part of another operation. For example inside an `if` or in the conditional part of a loop. More on that later.

Arithmetic operators

Python has a number of arithmetic operators: `+`, `-`, `*`, `/` (division), `%` (remainder), `**` (exponentiation) and `//` (floor division):

```
1 + 1 #2
2 - 1 #1
2 * 2 #4
4 / 2 #2
4 % 3 #1
4 ** 2 #16
4 // 2 #2
```

> Note that you don't need a space between the operands, but it's good for readability.

`-` also works as a unary minus operator:

```
print(-4) #-4
```

`+` is also used to concatenate String values:

```
"Roger" + " is a good dog"
#Roger is a good dog
```

We can combine the assignment operator with arithmetic operators:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- ..and so on

Example:

```
age = 8
age += 1
# age is now 9
```

Comparison operators

Python defines a few comparison operators:

- `==`
- `!=`
- `>`
- `<`
- `>=`
- `<=`

You can use those operators to get a boolean value (`True` or `False`) depending on the result:

```
a = 1
b = 2

a == b #False
a != b #True
a > b #False
a <= b #True
```

Boolean operators

Python gives us the following boolean operators:

- `not`
- `and`
- `or`

When working with `True` or `False` attributes, those work like logical AND, OR and NOT, and are often used in the `if` conditional expression evaluation:

```
condition1 = True
condition2 = False

not condition1 #False
condition1 and condition2 #False
condition1 or condition2 #True
```

Otherwise, pay attention to a possible source of confusion.

`or` used in an expression returns the value of the first operand that is not a falsy value (`False` , `0` , `''` , `[]` ..). Otherwise it returns the last operand.

```
print(0 or 1) ## 1
print(False or 'hey') ## 'hey'
print('hi' or 'hey') ## 'hi'
print([] or False) ## 'False'
print(False or []) ## '[]'
```

The Python docs describe it as `if x is false, then y, else x` .

`and` only evaluates the second argument if the first one is true. So if the first argument is falsy (`False` , `0` , `''` , `[]` ..), it returns that argument. Otherwise it evaluates the second argument:

```
print(0 and 1) ## 0
print(1 and 0) ## 0
print(False and 'hey') ## False
print('hi' and 'hey') ## 'hey'
print([] and False ) ## []
print(False and [] ) ## False
```

The Python docs describe it as `if x is false, then x, else y`.

Bitwise operators

Some operators are used to work on bits and binary numbers:

- `&` performs binary AND
- `|` performs binary OR
- `^` performs a binary XOR operation
- `~` performs a binary NOT operation
- `<<` shift left operation
- `>>` shift right operation

Bitwise operators are rarely used, only in very specific situations, but they are worth mentioning.

`is` and `in`

`is` is called the **identity operator**. It is used to compare two objects and returns true if both are the same object. More on objects later.

`in` is called the **membership operator**. Is used to tell if a value is contained in a list, or another sequence. More on lists and other sequences later.

The Ternary Operator

The ternary operator in Python allows you to quickly define a conditional.

Let's say you have a function that compares an `age` variable to the `18` value, and return True or False depending on the result.

Instead of writing:

```
def is_adult(age):
    if age > 18:
        return True
    else:
        return False
```

You can implement it with the ternary operator in this way:

```
def is_adult(age):
    return True if age > 18 else False
```

First you define the result if the condition is True, then you evaluate the condition, then you define the result if the condition is false:

```
<result_if_true> if <condition> else <result_if_fals
```

Strings

A string in Python is a series of characters enclosed into quotes or double quotes:

```
"Roger"
'Roger'
```

You can assign a string value to a variable:

```
name = "Roger"
```

You can concatenate two strings using the `+` operator:

```
phrase = "Roger" + " is a good dog"
```

You can append to a string using `+=`:

```
name = "Roger"
name += " is a good dog"

print(name) #Roger is a good dog
```

You can convert a number to a string using the `str` class constructor:

```
str(8) #"8"
```

This is essential to concatenate a number to a string:

```
print("Roger is " + str(8) + " years old") #Roger is
```

A string can be multi-line when defined with a special syntax, enclosing the string in a set of 3 quotes:

```
print("""Roger is

    8

years old
""")

#double quotes, or single quotes

print('''
Roger is

    8

years old
''')
```

A string has a set of built-in methods, like:

- `isalpha()` to check if a string contains only characters and is not empty
- `isalnum()` to check if a string contains characters or digits and is not empty
- `isdecimal()` to check if a string contains digits and is not empty
- `lower()` to get a lowercase version of a string
- `islower()` to check if a string is lowercase
- `upper()` to get an uppercase version of a string
- `isupper()` to check if a string is uppercase
- `title()` to get a capitalized version of a string

- `startsswith()` to check if the string starts with a specific substring
- `endswith()` to check if the string ends with a specific substring
- `replace()` to replace a part of a string
- `split()` to split a string on a specific character separator
- `strip()` to trim the whitespace from a string
- `join()` to append new letters to a string
- `find()` to find the position of a substring

and many more.

None of those methods alter the original string. They return a new, modified string instead. For example:

```
name = "Roger"
print(name.lower()) #"roger"
print(name) #"Roger"
```

You can use some global functions to work with strings, too.

In particular I think of `len()`, which gives you the length of a string:

```
name = "Roger"
print(len(name)) #5
```

The `in` operator lets you check if a string contains a substring:

```
name = "Roger"
print("ger" in name) #True
```

Escaping is a way to add special characters into a string.

For example, how do you add a double quote into a string that's wrapped into double quotes?

```
name = "Roger"
```

`"Ro"Ger"` will not work, as Python will think the string ends at `"Ro"` .

The way to go is to escape the double quote inside the string, with the `\` backslash character:

```
name = "Ro\"ger"
```

This applies to single quotes too `\'` , and for special formatting characters like `\t` for tab, `\n` for new line and `\\` for the backslash.

Given a string, you can get its characters using square brackets to get a specific item, given its index, starting from 0:

```
name = "Roger"
name[0] #'R'
name[1] #'o'
name[2] #'g'
```

Using a negative number will start counting from the end:

```
name = "Roger"
name[-1] #"r"
```

You can also use a range, using what we call **slicing**:

```
name = "Roger"
name[0:2] #"Ro"
name[:2] #"Ro"
name[2:] #"ger"
```

Booleans

Python provides the `bool` type, which can have two values: `True` and `False` (capitalized)

```
done = False
done = True
```

Booleans are especially useful with conditional control structures like `if` statements:

```
done = True

if done:
    # run some code here
else:
    # run some other code
```

When evaluating a value for `True` or `False`, if the value is not a `bool` we have some rules depending on the type we're checking:

- numbers are always `True` unless for the number `0`
- strings are `False` only when empty
- lists, tuples, sets, dictionaries are `False` only when empty

You can check if a value is a boolean in this way:

```
done = True
type(done) == bool #True
```

Or using `isinstance()`, passing 2 arguments: the variable, and the `bool` class:

```
done = True
isinstance(done, bool) #True
```

The global `any()` function is also very useful when working with booleans, as it returns `True` if any of the values of the iterable (list, for example) passed as argument are `True`:

```
book_1_read = True
book_2_read = False

read_any_book = any([book_1_read, book_2_read])
```

The global `all()` function is same, but returns `True` if all of the values passed to it are `True`:

```
ingredients_purchased = True
meal_cooked = False

ready_to_serve = all([ingredients_purchased, meal_c
```

Numbers

Numbers in Python can be of 3 types: `int`, `float` and `complex`.

Integer numbers

Integer numbers are represented using the `int` class. You can define an integer using a value literal:

```
age = 8
```

You can also define an integer number using the `int()` constructor:

```
age = int(8)
```

To check if a variable is of type `int`, you can use the `type()` global function:

```
type(age) == int #True
```

Floating point numbers

Floating point numbers (fractions) are of type `float`. You can define an integer using a value literal:

```
fraction = 0.1
```

Or using the `float()` constructor:

```
fraction = float(0.1)
```

To check if a variable is of type `float`, you can use the `type()` global function:

```
type(fraction) == float #True
```

Complex numbers

Complex numbers are of type `complex`.

You can define them using a value literal:

```
complexNumber = 2+3j
```

or using the `complex()` constructor:

```
complexNumber = complex(2, 3)
```

Once you have a complex number, you can get its real and imaginary part:

```
complexNumber.real #2.0
complexNumber.imag #3.0
```

Again, to check if a variable is of type `complex`, you can use the `type()` global function:

```
type(complexNumber) == complex #True
```

Arithmetic operations on numbers

You can perform arithmetic operations on numbers, using the arithmetic operators: `+`, `-`, `*`, `/` (division), `%` (remainder), `**` (exponentiation) and `//` (floor division):

```
1 + 1 #2
2 - 1 #1
2 * 2 #4
4 / 2 #2
4 % 3 #1
4 ** 2 #16
4 // 2 #2
```

and you can use the compound assignment operators

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- ..and so on

to quickly perform operations on variables, too:

```
age = 8
age += 1
```

Built-in Functions

There are 2 built-in functions that help with numbers:

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

`round()` given a number, returns its value rounded to the nearest integer:

```
round(0.12) #0
```

You can specify a second parameter to set the decimal points precision:

```
round(0.12, 1) #0.1
```

Several other math utility functions and constants are provided by the Python standard library:

- the `math` package provides general math functions and constants
- the `cmath` package provides utilities to work with complex numbers.
- the `decimal` package provides utilities to work with decimals and floating point numbers.
- the `fractions` package provides utilities to work with rational numbers

We'll explore some of those separately later on.

Constants

Python has no way to enforce a variable to be a constant.

The nearest you can go is to use an enum:

```
class Constants(Enum):
    WIDTH = 1024
    HEIGHT = 256
```

And get to each value using for example `Constants.WIDTH.value`.

No one can reassign that value.

Otherwise if you want to rely on naming conventions, you can adhere to this one: declare variables that should never change uppercase:

```
WIDTH = 1024
```

No one will prevent to overwrite this value, and Python will not stop it.

That's what does most Python code you will see.

Enums

Enums are readable names that are bound to a constant value.

To use enums, import `Enum` from the `enum` standard library module:

```
from enum import Enum
```

Then you can initialize a new enum in this way:

```
class State(Enum):
    INACTIVE = 0
    ACTIVE = 1
```

Once you do so, you can reference `State.INACTIVE` and `State.ACTIVE`, and they serve as constants.

Now if you try to print `State.ACTIVE` for example:

```
print(State.ACTIVE)
```

it will not return `1`, but `State.ACTIVE`.

The same value can be reached by the number assigned in the enum: `print(State(1))` will return `State.ACTIVE`. Same for using the square brackets notation `State['ACTIVE']`.

You can however get the value using `State.ACTIVE.value`.

You can list all the possible values of an enum:

```
list(State) # [<State.INACTIVE: 0>, <State.ACTIVE: 1
```

You can count them:

```
len(State) # 2
```

User Input

In a Python command line application you can display information to the user using the `print()` function:

```
name = "Roger"
print(name)
```

We can also accept input from the user, using `input()`:

```
print('What is your age?')
age = input()
print('Your age is ' + age)
```

This approach gets input at runtime, meaning the program will stop execution and will wait until the user types something and presses the `enter` key.

You can also do more complex input processing and accept input at program invocation time, and we'll see how to do that later on.

This works for command line applications. Other kinds of applications will need a different way of accepting input.

Control statements

What's interesting to do with booleans, and expressions that return a boolean in particular, is that we can make decisions and take different roads depending on their `True` or `False` value.

In Python we do so using the `if` statement:

```
condition = True

if condition == True:
    # do something
```

When the condition test resolves to `True`, like in the above case, its block gets executed.

What is a block? A block is that part that is indented one level (4 spaces usually) on the right:

```
condition = True

if condition == True:
    print("The condition")
    print("was true")
```

The block can be formed by a single line, or multiple lines as well, and it ends when you move back to the previous indentation level:

```
condition = True

if condition == True:
    print("The condition")
    print("was true")

print("Outside of the if")
```

In combination with `if` you can have an `else` block, that's executed if the condition test of `if` results to `False`:

```
condition = True

if condition == True:
    print("The condition")
    print("was True")
else:
    print("The condition")
    print("was False")
```

And you can have different linked `if` checks with `elif`, that's executed if the previous check was `False`:

```
condition = True
name = "Roger"

if condition == True:
    print("The condition")
    print("was True")
elif name == "Roger":
    print("Hello Roger")
else:
    print("The condition")
    print("was False")
```

The second block in this case is executed if `condition` is `False`, and the `name` variable value is "Roger".

In a `if` statement you can have just one `if` and `else` checks, but multiple series of `elif` checks:

```
condition = True
name = "Roger"

if condition == True:
    print("The condition")
    print("was True")
elif name == "Roger":
    print("Hello Roger")
elif name == "Syd":
    print("Hello Syd")
elif name == "Flavio":
    print("Hello Flavio")
else:
    print("The condition")
    print("was False")
```

`if` and `else` can also be used in an inline format, which lets us return a value or another based on a condition.

Example:

```
a = 2
result = 2 if a == 0 else 3
print(result) # 3
```

Lists

Lists are an essential Python data structure.

The allow you to group together multiple values and reference them all with a common name.

For example:

```
dogs = ["Roger", "Syd"]
```

A list can hold values of different types:

```
items = ["Roger", 1, "Syd", True]
```

You can check if an item is contained into a list with the `in` operator:

```
print("Roger" in items) # True
```

A list can also be defined as empty:

```
items = []
```

You can reference the items in a list by their index, starting from zero:

```
items[0] # "Roger"
items[1] # 1
items[3] # True
```

Using the same notation you can change the value stored at a specific index:

```
items[0] = "Roger"
```

You can also use the `index()` method:

```
items.index(0) # "Roger"
items.index(1) # 1
```

As with strings, using a negative index will start searching from the end:

```
items[-1] # True
```

You can also extract a part of a list, using slices:

```
items[0:2] # ["Roger", 1]
items[2:] # ["Syd", True]
```

Get the number of items contained in a list using the `len()` global function, the same we used to get the length of a string:

```
len(items) #4
```

You can add items to the list by using a list `append()` method:

```
items.append("Test")
```

or the extend() method:

```
items.extend(["Test"])
```

You can also use the `+=` operator:

```
items += ["Test"]

# items is ['Roger', 1, 'Syd', True, 'Test']
```

> Tip: with `extend()` or `+=` don't forget the square brackets. Don't do `items += "Test"` or `items.extend("Test")` or Python will add 4 individual characters to the list, resulting in `['Roger', 1, 'Syd', True, 'T', 'e', 's', 't']`

Remove an item using the `remove()` method:

```
items.remove("Test")
```

You can add multiple elements using

```
items += ["Test1", "Test2"]

#or

items.extend(["Test1", "Test2"])
```

These append the item to the end of the list.

To add an item in the middle of a list, at a specific index, use the `insert()` method:

```
items.insert("Test", 1) # add "Test" at index 1
```

To add multiple items at a specific index, you need to use slices:

```
items[1:1] = ["Test1", "Test2"]
```

Sort a list using the `sort()` method:

```
items.sort()
```

> Tip: sort() will only work if the list holds values that can be compared. Strings and integers for example can't be compared, and you'll get an error like `TypeError: '<' not supported between instances of 'int' and 'str'` if you try.

The `sort()` methods orders uppercase letters first, then lowercased letters. To fix this, use:

```
items.sort(key=str.lower)
```

instead.

Sorting modifies the original list content. To avoid that, you can copy the list content using

```
itemscopy = items[:]
```

or use the `sorted()` global function:

```
print(sorted(items, key=str.lower))
```

that will return a new list, sorted, instead of modifying the original list.

Tuples

Tuples are another fundamental Python data structure.

They allow you to create immutable groups of objects. This means that once a tuple is created, it can't be modified. You can't add or remove items.

They are created in a way similar to lists, but using parentheses instead of square brackets:

```
names = ("Roger", "Syd")
```

A tuple is ordered, like a list, so you can get its values referencing an index value:

```
names[0] # "Roger"
names[1] # "Syd"
```

You can also use the `index()` method:

```
items.index("Roger") # 0
items.index("Syd") # 2
```

As with strings and lists, using a negative index will start searching from the end:

```
names[-1] # True
```

You can count the items in a tuple with the `len()` function:

```
len(names) # 2
```

You can check if an item is contained into a tuple with the `in` operator:

```
print("Roger" in names) # True
```

You can also extract a part of a tuple, using slices:

```
names[0:2] # ('Roger', 'Syd')
names[1:] # ('Syd',)
```

Get the number of items in a tuple using the `len()` global function, the same we used to get the length of a string:

```
len(names) #2
```

You can create a sorted version of a tuple using the `sorted()` global function:

```
sorted(names)
```

You can create a new tuple from existing tuples using the `+` operator:

```
newTuple = names + ("Vanille", "Tina")
```

Dictionaries

Dictionaries are a very important Python data structure.

While lists allow you to create collections of values, dictionaries allow you to create collections of **key / value pairs**.

Here is a dictionary example with one key/value pair:

```
dog = { 'name': 'Roger' }
```

The key can be any immutable value like a string, a number or a tuple. The value can be anything you want.

A dictionary can contain multiple key/value pairs:

```
dog = { 'name': 'Roger', 'age': 8 }
```

You can access individual key values using this notation:

```
dog['name'] # 'Roger'
dog['age']  # 8
```

Using the same notation you can change the value stored at a specific index:

```
dog['name'] = 'Syd'
```

And another way is using the `get()` method, which has an option to add a default value:

```
dog.get('name') # 'Roger'
dog.get('test', 'default') # 'default'
```

The `pop()` method retrieves the value of a key, and subsequently deletes the item from the dictionary:

```
dog.pop('name') # 'Roger'
```

The `popitem()` method retrieves and removes the last key/value pair inserted into the dictionary:

```
dog.popitem()
```

You can check if a key is contained into a dictionary with the `in` operator:

```
'name' in dog # True
```

Get a list with the keys in a dictionary using the `keys()` method, passing its result to the `list()` constructor:

```
list(dog.keys()) # ['name', 'age']
```

Get the values using the `values()` method, and the key/value pairs tuples using the `items()` method:

```
print(list(dog.values()))
# ['Roger', 8]

print(list(dog.items()))
# [('name', 'Roger'), ('age', 8)]
```

Get a dictionary length using the `len()` global function, the same we used to get the length of a string or the items in a list:

```
len(dog) #2
```

You can add a new key/value pair to the dictionary in this way:

```
dog['favorite food'] = 'Meat'
```

You can remove a key/value pair from a dictionary using the `del` statement:

```
del dog['favorite food']
```

To copy a dictionary, use the copy() method:

```
dogCopy = dog.copy()
```

Sets

Sets are another important Python data structure.

We can say they work like tuples, but they are not ordered, and they are **mutable**. Or we can say they work like dictionaries, but they don't have keys.

They also have an immutable version, called `frozenset`.

You can create a set using this syntax:

```
names = {"Roger", "Syd"}
```

Sets work well when you think about them as mathematical sets.

You can intersect two sets:

```
set1 = {"Roger", "Syd"}
set2 = {"Roger"}

intersect = set1 & set2 #{'Roger'}
```

You can create a union of two sets:

```
set1 = {"Roger", "Syd"}
set2 = {"Luna"}

union = set1 | set2
#{'Syd', 'Luna', 'Roger'}
```

You can get the difference between two sets:

```
set1 = {"Roger", "Syd"}
set2 = {"Roger"}

difference = set1 - set2 #{'Syd'}
```

You can check if a set is a superset of another (and of course if a set is a subset of another)

```
set1 = {"Roger", "Syd"}
set2 = {"Roger"}

isSuperset = set1 > set2 # True
```

You can count the items in a set with the `len()` global function:

```
names = {"Roger", "Syd"}
len(names) # 2
```

You can get a list from the items in a set by passing the set to the `list()` constructor:

```
names = {"Roger", "Syd"}
list(names) #['Syd', 'Roger']
```

You can check if an item is contained into a set with the `in` operator:

```
print("Roger" in names) # True
```

Functions

A function lets us create a set of instructions that we can run when needed.

Functions are essential in Python and in many other programming languages to create meaningful programs, because they allow us to decompose a program into manageable parts, they promote readability and code reuse.

Here is an example function called `hello` that prints "Hello!":

```
def hello():
    print('Hello!')
```

This is the function **definition**. Thereis a name (`hello`) and a body, the set of instructions, which is the part that follows the colon and it's indented one level on the right.

To run this function, we must call it. This is the syntax to call the function:

```
hello()
```

We can execute this function once, or multiple times.

The name of the function, `hello` , is very important. It should be descriptive, so anyone calling it can imagine what the function does.

A function can accept one or more parameters:

```
def hello(name):
    print('Hello ' + name + '!')
```

In this case we call the function passing the argument

```
hello('Roger')
```

> We call *parameters* the values accepted by the function inside the function definition, and *arguments* the values we pass to the function when we call it. It's common to get confused about this distinction.

An argument can have a default value that's applied if the argument is not specified:

```
def hello(name='my friend'):
    print('Hello ' + name + '!')

hello()
#Hello my friend!
```

Here's how we can accept multiple parameters:

```
def hello(name, age):
    print('Hello ' + name + ', you are ' + str(age)
```

In this case we call the function passing a set of arguments:

```
hello('Roger', 8)
```

Parameters are passed by reference. All types in Python are objects but some of them are immutable, including integers, booleans, floats, strings, and tuples. This means that if you pass them as parameters and you modify their value inside the function, the new value is not reflected outside of the function:

```
def change(value):
    value = 2

val = 1
change(val)

print(val) #1
```

If you pass an object that's not immutable, and you change one of its properties, the change will be reflected outside.

A function can return a value, using the `return` statement. For example in this case we return the `name` parameter name:

```
def hello(name):
    print('Hello ' + name + '!')
    return name
```

When the function meets the `return` statement, the function ends.

We can omit the value:

```
def hello(name):
    print('Hello ' + name + '!')
    return
```

We can have the return statement inside a conditional, which is a common way to end a function if a starting condition is not met:

```
def hello(name):
    if not name:
        return
    print('Hello ' + name + '!')
```

If we call the function passing a value that evaluates to `False`, like an empty string, the function is terminated before reaching the `print()` statement.

You can return multiple values by using comma separated values:

```
def hello(name):
    print('Hello ' + name + '!')
    return name, 'Roger', 8
```

In this case calling `hello('Syd')` the return value is a tuple containing those 3 values: `('Syd', 'Roger', 8)`.

Objects

Everything in Python is an object.

Even values of basic primitive types (integer, string, float..) are objects. Lists are objects, tuples, dictionaries, everything.

Objects have **attributes** and **methods** that can be accessed using the dot syntax.

For example, try defining a new variable of type `int`:

```
age = 8
```

`age` now has access to the properties and methods defined for all `int` objects.

This includes, for example, access to the real and imaginary part of that number:

```
print(age.real) # 8
print(age.imag) # 0

print(age.bit_length()) #4

# the bit_length() method returns the number of bits
```

A variable holding a list value has access to a different set of methods:

```
items = [1, 2]
items.append(3)
items.pop()
```

The methods depend on the type of value.

The `id()` global function provided by Python lets you inspect the location in memory for a particular object.

```
id(age) # 140170065725376
```

> Your memory value will change, I am only showing it as an example

If you assign a different value to the variable, its address will change, because the content of the variable has been replaced with another value stored in another location in memory:

```
age = 8

print(id(age)) # 140535918671808

age = 9

print(id(age)) # 140535918671840
```

But if you modify the object using its methods, the address stays the same:

```
items = [1, 2]

print(id(items)) # 140093713593920

items.append(3)

print(items) # [1, 2, 3]
print(id(items)) # 140093713593920
```

The address only changes if you reassign a variable to another value.

Some objects are *mutable*, some are *immutable*. This depends on the object itself. If the object provides methods to change its content, then it's mutable. Otherwise it's immutable. Most types defined by Python are immutable. For example an `int` is immutable. There are no methods to change its value. If you increment the value using

```
age = 8
age = age + 1

#or

age += 1
```

and you check with `id(age)` you will find that `age` points to a different memory location. The original value has not mutated, we switched to another value.

Loops

Loops are one essential part of programming.

In Python we have 2 kinds of loops: **while loops** and **for loops**.

`while` loops

`while` loops are defined using the `while` keyword, and they repeat their block until the condition is evaluated as `False`:

```
condition = True
while condition == True:
    print("The condition is True")
```

This is an **infinite loop**. It never ends.

Let's halt the loop right after the first iteration:

```
condition = True
while condition == True:
    print("The condition is True")
    condition = False

print("After the loop")
```

In this case, the first iteration is ran, as the condition test is evaluated to `True`, and at the second iteration the condition test evaluates to `False`, so the control goes to the next instruction, after the loop.

It's common to have a counter to stop the iteration after some number of cycles:

```
count = 0
while count < 10:
    print("The condition is True")
    count = count + 1

print("After the loop")
```

`for` loops

Using `for` loops we can tell Python to execute a block for a pre-determined amount of times, up front, and without the need of a separate variable and conditional to check its value.

For example we can iterate the items in a list:

```
items = [1, 2, 3, 4]
for item in items:
    print(item)
```

Or, you can iterate a specific amount of times using the `range()` function:

```
for item in range(04):
    print(item)
```

`range(4)` creates a sequence that starts from 0 and contains 4 items: `[0, 1, 2, 3]` .

To get the index, you should wrap the sequence into the `enumerate()` function:

```
items = [1, 2, 3, 4]
for index, item in enumerate(items):
    print(index, item)
```

Break and continue

Both `while` and `for` loops can be interrupted inside the block, using two special keywords: `break` and `continue`.

`continue` stops the current iteration and tells Python to execute the next one.

`break` stops the loop altogether, and goes on with the next instruction after the loop end.

The first example here prints `1, 3, 4`. The second example prints `1`:

```
items = [1, 2, 3, 4]
for item in items:
    if item == 2:
        continue
    print(item)
```

```
items = [1, 2, 3, 4]
for item in items:
    if item == 2:
        break
    print(item)
```

Classes

Defining new objects in Python using classes

In addition to using the Python-provided types, we can declare our own classes, and from classes we can instantiate objects.

An object is an instance of a class. A class is the type of an object.

Define a class in this way:

```
class <class_name>:
    # my class
```

For example let's define a Dog class

```
class Dog:
    # the Dog class
```

A class can define methods:

```
class Dog:
    # the Dog class
    def bark(self):
        print('WOF!')
```

> `self` as the argument of the method points to the current object instance, and must be specified when defining a method.

We create an instance of a class, an **object**, using this syntax:

```
roger = Dog()
```

Now `roger` is a new object of type Dog.

If you run

```
print(type(roger))
```

You will get `<class '__main__.Dog'>`

A special type of method, `__init__()` is called constructor, and we can use it to initialize one or more properties when we create a new object from that class:

```
class Dog:
    # the Dog class
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print('WOF!')
```

We use it in this way:

```
roger = Dog('Roger', 8)
print(roger.name) # 'Roger'
print(roger.age)  # 8

roger.bark() # 'WOF!'
```

One important features of classes is inheritance.

We can create an Animal class with a method `walk()`:

```
class Animal:
    def walk(self):
        print('Walking..')
```

and the Dog class can inherit from Animal:

```
class Dog(Animal):
    def bark(self):
        print('WOF!')
```

Now creating a new object of class `Dog` will have the `walk()` method as that's inherited from `Animal`:

```
roger = Dog()
roger.walk() # 'Walking..'
roger.bark() # 'WOF!'
```

Modules

Every Python file is a module.

You can import a module from other files, and that's the base of any program of moderate complexity, as it promotes a sensible organization and code reuse.

In the typical Python program, one file acts as the entry point. The other files are modules and expose functions that we can call from other files.

The file `dog.py` contains this code:

```
def bark():
    print('WOF!')
```

We can import this function from another file using `import`, and once we do, we can reference the function using the dot notation, `dog.bark()`:

```
import dog

dog.bark()
```

Or, we can use the `from .. import` syntax and call the function directly:

```
from dog import bark

bark()
```

The first strategy allows us to load everything defined in a file.

The second strategy lets us pick the things we need.

Those modules are specific to your program, and importing depends on the location of the file in the filesystem.

Suppose you put `dog.py` in a `lib` subfolder.

In that folder, you need to create an empty file named `__init__.py`. This tells Python the folder contains modules.

Now you can choose, you can import `dog` from `lib`:

```
from lib import dog

dog.bark()
```

or you can reference the `dog` module specific function importing from `lib.dog`:

```
from lib.dog import bark

bark()
```

The Python Standard Library

Python exposes a lot of built-in functionality through its **standard library**.

The standard library is a huge collection of all sort of utilities, ranging from math utilities to debugging to creating graphical user interfaces.

You can find the full list of standard library modules here: https://docs.python.org/3/library/index.html

Some of the important modules are:

- `math` for math utilities
- `re` for regular expressions
- `json` to work with JSON
- `datetime` to work with dates
- `sqlite3` to use SQLite
- `os` for Operating System utilities
- `random` for random number generation
- `statistics` for statistics utilities
- `requests` to perform HTTP network requests
- `http` to create HTTP servers
- `urllib` to manage URLs

Let's introduce how to *use* a module of the standard library. You already know how to use modules you create, importing from other files in the program folder.

Well that's the same with modules provided by the standard library:

```
import math

math.sqrt(4) # 2.0
```

or

```
from math import sqrt

sqrt(4) # 2.0
```

We'll soon explore the most important modules individually to understand what we can do with them.

The PEP8 Python style guide

When you write code, you should adhere to the conventions of the programming language you use.

If you learn the right naming and formatting conventions right from the start, it will be easier to read code written by other people, and people will find your code easier to read.

Python defines its conventions in the PEP8 style guide. PEP stands for *Python Enhancement Proposals* and it's the place where all Python language enhancements and discussions happen. There are a lot of PEP proposals, all available at https://www.python.org/dev/peps/.

PEP8 is one of the first ones, and one of the most important, too. It defines the formatting and also some rules on how to write Python in a "pythonic" way.

You can read its full content here: https://www.python.org/dev/peps/pep-0008/ but here's a quick summary of the important points you can start with:

- Indent using spaces, not tabs
- Indent using 4 spaces.
- Python files are encoded in UTF-8
- Use maximum 80 columns for your code
- Write each statement on its own line

- Functions, variable names and file names are lowercase, with underscores between words (snake_case)
- Class names are capitalized, separate words are written with the capital letter too, (CamelCase)
- Package names are lowercase and do not have underscores between words
- Variables that should not change (constants) are written in uppercase
- Variable names should be meaningful
- Add useful comments, but avoid obvious comments
- Add spaces around operators
- Do not use unnecessary whitespace
- Add a blank line before a function
- Add a blank line between methods in a class
- Inside functions/methods, blank lines can be used to separate related blocks of code to help readability

Debugging

Debugging is one of the best skills you can learn, as it will help you in many difficult situations.

Every language has its debugger. Python has `pdb`, available through the standard library.

You debug by adding one breakpoint into your code:

```
breakpoint()
```

> You can add more breakpoints if needed.

When the Python interpreter hits a breakpoint in your code, it will stop, and it will tell you what is the next instruction it will run.

Then and you can do a few things.

You can type the name of any variable to inspect its value.

You can press `n` to step to the next line in the current function. If the code calls functions, the debugger does not get into them, and consider them "black boxes".

You can press `s` to step to the next line in the current function. If the next line is a function, the debugger goes into that, and you can then run one instruction of that function at a time.

You can press `c` to continue the execution of the program normally, without the need to do it step-by-step.

You can press `q` to stop the execution of the program.

Debugging is useful to evaluate the result of an instruction, and it's especially good to know how to use it when you have complex iterations or algorithms that you want to fix.

Variables scope

When you declare a variable, that variable is visible in parts of your program, depending on where you declare it.

If you declare it outside of any function, the variable is visible to any code running after the declaration, including functions:

```
age = 8

def test():
    print(age)

print(age) # 8
test() # 8
```

We call it a **global variable**.

If you define a variable inside a function, that variable is a **local variable**, and it is only visible inside that function. Outside the function, it is not reachable:

```
def test():
    age = 8
    print(age)

test() # 8

print(age)
# NameError: name 'age' is not defined
```

Accept arguments from the command line

Python offers several ways to handle arguments passed when we invoke the program from the command line.

So far you've run programs either from a REPL, or using

```
python <filename>.py
```

You can pass additional arguments and options when you do so, like this:

```
python <filename>.py <argument1>
python <filename>.py <argument1> <argument2>
```

A basic way to handle those arguments is to use the `sys` module from the standard library.

You can get the arguments passed in the `sys.argv` list:

```
import sys
print(len(sys.argv))
print(sys.argv)
```

The `sys.argv` list contains as the first item the name of the file that was ran, e.g. `['main.py']`.

This is a simple way, but you have to do a lot of work. You need to validate arguments, make sure their type is correct, you need to print feedback to the user if they are not using the program correctly.

Python provides another package in the standard library to help you: `argparse`.

First you import `argparse` and you call `argparse.ArgumentParser()`, passing the description of your program:

```
import argparse

parser = argparse.ArgumentParser(
    description='This program prints the name of my
)
```

Then you proceed to add arguments you want to accept. For example in this program we accept a `-c` option to pass a color, like this: `python program.py -c red`

```
import argparse

parser = argparse.ArgumentParser(
    description='This program prints a color HEX va
)

parser.add_argument('-c', '--color', metavar='color

args = parser.parse_args()

print(args.color) # 'red'
```

If the argument is not specified, the program raises an error:

```
➜  python python program.py
usage: program.py [-h] -c color
program.py: error: the following arguments are requ
```

You can set an option to have a specific set of values, using `choices`:

```
parser.add_argument('-c', '--color', metavar='color
```

```
➜  python python program.py -c blue
usage: program.py [-h] -c color
program.py: error: argument -c/--color: invalid cho
```

There are more options, but those are the basics.

And there are community packages that provide this functionality, too, like Click and Python Prompt Toolkit.

Lambda functions

Lambda functions (also called anonymous functions) are tiny functions that have no name and only have one expression as their body.

In Python they are defined using the `lambda` keyword:

```
lambda <arguments> : <expression>
```

The body must be a single expression. Expression, not a statement.

> This difference is important. An expression returns a value, a statement does not.

The simplest example of a lambda function is a function that doubles that value of a number:

```
lambda num : num * 2
```

Lambda functions can accept more arguments:

```
lambda a, b : a * b
```

Lambda functions cannot be invoked directly, but you can assign them to variables:

```
multiply = lambda a, b : a * b

print(multiply(2, 2)) # 4
```

The utility of lambda functions comes when combined with other Python functionality, for example in combination with `map()`, `filter()` and `reduce()`.

Recursion

A function in Python can call itself. That's what recursion is. And it can be pretty useful in many scenarios.

The common way to explain recursion is by using the factorial calculation.

The factorial of a number is the number `n` mutiplied by `n-1`, multiplied by `n-2` ... and so on, until reaching the number `1`:

```
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Using recursion we can write a function that calculates the factorial of any number:

```
def factorial(n):
    if n == 1: return 1
    return n * factorial(n-1)

print(factorial(3)) #   6
print(factorial(4)) #  24
print(factorial(5)) # 120
```

If inside the `factorial()` function you call `factorial(n)` instead of `factorial(n-1)`, you are going to cause an infinite recursion. Python by default will halt recursions at 1000 calls, and when this limit is reached, you will get a `RecursionError` error.

Recursion is helpful in many places, and it helps us simplify our code when there's no other optimal way to do it, so it's good to know this technique.

Nested functions

Functions in Python can be nested inside other functions.

A function defined inside a function is visible only inside that function.

This is useful to create utilities that are useful to a function, but not useful outside of it.

You might ask: why should I be "hiding" this function, if it does not harm?

One, because it's always best to hide functionality that's local to a function, and not useful elsewhere.

Also, because we can make use of closures (more on this later).

Here is an example:

```
def talk(phrase):
    def say(word):
        print(word)

    words = phrase.split(' ')
    for word in words:
        say(word)

talk('I am going to buy the milk')
```

If you want to access a variable defined in the outer function from the inner function, you first need to declare it as `nonlocal`:

```
def count():
    count = 0

    def increment():
        nonlocal count
        count = count + 1
        print(count)

    increment()

count()
```

This is useful especially with closures, as we'll see later.

Closures

If you return a nested function from a function, that nested function has access to the variables defined in that function, even if that function is not active any more.

Here is a simple counter example.

```
def counter():
    count = 0

    def increment():
        nonlocal count
        count = count + 1
        return count

    return increment

increment = counter()

print(increment()) # 1
print(increment()) # 2
print(increment()) # 3
```

We return the `increment()` inner function, and that has still access to the state of the `count` variable even though the `counter()` function has ended.

Decorators

Decorators are a way to change, enhance or alter in any way how a function works.

Decorators are defined with the `@` symbol followed by the decorator name, just before the function definition.

Example:

```
@logtime
def hello():
    print('hello!')
```

This `hello` function has the `logtime` decorator assigned.

Whenever we call `hello()`, the decorator is going to be called.

A decorator is a function that takes a function as a parameter, wraps the function in an inner function that performs the job it has to do, and returns that inner function. In other words:

```
def logtime(func):
    def wrapper():
        # do something before
        val = func()
        # do something after
        return val
    return wrapper
```

Docstrings

Documentation is hugely important, not just to communicate to other people what is the goal of a function/class/method/module, but also to yourself.

When you'll come back to your code 6 or 12 months from now, you might not remember all the knowledge you are holding in your head, and reading your code and understanding what it is supposed to do, will be much more difficult.

Comments are one way to do so:

```
# this is a comment

num = 1 #this is another comment
```

Another way is to use **docstrings**.

The utility of docstrings is that they follow conventions and as such they can be processed automatically.

This is how you define a docstring for a function:

```
def increment(n):
    """Increment a number"""
    return n + 1
```

This is how you define a docstring for a class and a method:

```
class Dog:
    """A class representing a dog"""
    def __init__(self, name, age):
        """Initialize a new dog"""
        self.name = name
        self.age = age

    def bark(self):
        """Let the dog bark"""
        print('WOF!')
```

Document a module by placing a docstring at the top of the file, for example supposing this is `dog.py`:

```
"""Dog module

This module does ... bla bla bla and provides the fo

- Dog
...
"""

class Dog:
    """A class representing a dog"""
    def __init__(self, name, age):
        """Initialize a new dog"""
        self.name = name
        self.age = age

    def bark(self):
        """Let the dog bark"""
        print('WOF!')
```

Docstrings can span over multiple lines:

```
def increment(n):
    """Increment
    a number
    """
    return n + 1
```

Python will process those and you can use the `help()` global function to get the documentation for a class/method/function/module.

For example calling `help(increment)` will give you this:

```
Help on function increment in module
__main__:

increment(n)
    Increment
    a number
```

There are many different standards to format docstrings, and you can choose to adhere to your favorite one.

I like Google's standard: https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings

Standard allows to have tools to extract docstrings and automatically generate documentation for your code.

Introspection

Functions, variables and objects can be analyzed using **introspection**.

First, using the `help()` global function we can get the documentation if provided in form of docstrings.

Then, you can use print() to get information about a function:

```
def increment(n):
    return n + 1

print(increment)

# <function increment at 0x7f420e2973a0>
```

or an object:

```
class Dog():
    def bark(self):
        print('WOF!')

roger = Dog()

print(roger)

# <__main__.Dog object at 0x7f42099d3340>
```

The `type()` function gives us the type of an object:

```
print(type(increment))
# <class 'function'>

print(type(roger))
# <class '__main__.Dog'>

print(type(1))
# <class 'int'>

print(type('test'))
# <class 'str'>
```

The `dir()` global function lets us find out all the methods and attributes of an object:

```
print(dir(roger))

# ['__class__', '__delattr__', '__dict__', '__dir__
```

The `id()` global function shows us the location in memory of any object:

```
print(id(roger)) # 140227518093024
print(id(1))     # 140227521172384
```

It can be useful to check if two variables point to the same object.

The `inspect` standard library module gives us more tools to get information about objects, and you can check it out here: https://docs.python.org/3/library/inspect.html

Annotations

Python is dynamically typed. We do not have to specify the type of a variable or function parameter, or a function return value.

Annotations allow us to (optionally) do that.

This is a function without annotations:

```
def increment(n):
    return n + 1
```

This is the same function with annotations:

```
def increment(n: int) -> int:
    return n + 1
```

You can also annotate variables:

```
count: int = 0
```

Python will ignore those annotations. A separate tool called `mypy` can be run standalone, or integrated by IDE like VS Code or PyCharm to automatically check for type errors statically, while you are coding, and it will help you catch type mismatch bugs before even running the code.

A great help especially when your software becomes large and you need to refactor your code.

Exceptions

It's important to have a way to handle errors.

Python gives us exception handling.

If you wrap lines of code into a `try:` block:

```
try:
    # some lines of code
```

If an error occurs, Python will alert you and you can determine which kind of error occurred using a `except` blocks:

```
try:
    # some lines of code
except <ERROR1>:
    # handler <ERROR1>
except <ERROR2>:
    # handler <ERROR2>
```

To catch all exceptions you can use `except` without any error type:

```
try:
    # some lines of code
except <ERROR1>:
    # handler <ERROR1>
except:
    # catch all other exceptions
```

The `else` block is ran if no exceptions were found:

```
try:
    # some lines of code
except <ERROR1>:
    # handler <ERROR1>
except <ERROR2>:
    # handler <ERROR2>
else:
    # no exceptions were raised, the code ran succes
```

A `finally` block lets you perform some operation in any case, regardless if an error occurred or not

```
try:
    # some lines of code
except <ERROR1>:
    # handler <ERROR1>
except <ERROR2>:
    # handler <ERROR2>
else:
    # no exceptions were raised, the code ran succes
finally:
    # do something in any case
```

The specific error that's going to occur depends on the operation you're performing.

For example if you are reading a file, you might get an `EOFError`. If you divide a number by zero you will get a `ZeroDivisionError`. If you have a type conversion issue you might get a `TypeError`.

Try this code:

```
result = 2 / 0
print(result)
```

The program will terminate with an error

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    result = 2 / 0
ZeroDivisionError: division by zero
```

and the lines of code after the error will not be executed.

Adding that operation in a `try:` block lets us recover gracefully and move on with the program:

```
try:
    result = 2 / 0
except ZeroDivisionError:
    print('Cannot divide by zero!')
finally:
    result = 1

print(result) # 1
```

You can raise exceptions in your own code, too, using the `raise` statement:

```
raise Exception('An error occurred!')
```

This raises a general exception, and you can intercept it using:

```
try:
    raise Exception('An error occurred!')
except Exception as error:
    print(error)
```

You can also define your own exception class, extending from Exception:

```
class DogNotFoundException(Exception):
    pass
```

> `pass` here means "nothing" and we must use it when we define a class without methods, or a function without code, too.

```
try:
    raise DogNotFoundException()
except DogNotFoundException:
    print('Dog not found!')
```

The with statement

The `with` statement is very helpful to simplify working with exception handling.

For example when working with files, each time we open a file, we must remember to close it.

`with` makes this process transparent.

Instead of writing:

```
filename = '/Users/flavio/test.txt'

try:
    file = open(filename, 'r')
    content = file.read()
    print(content)
finally:
    file.close()
```

You can write:

```
filename = '/Users/flavio/test.txt'

with open(filename, 'r') as file:
    content = file.read()
    print(content)
```

In other words we have built-in implicit exception handling, as `close()` will be called automatically for us.

`with` is not just helpful to work with files. The above example is just meant to introduce its capabilities.

Installing 3rd party packages using pip

The Python standard library contains a huge number of utilities that simplify our Python development needs, but nothing can satisfy *everything*.

That's why individuals and companies create packages, and make them available as open source software for the entire community.

Those modules are all collected in a single place, the **Python Package Index** available at https://pypi.org, and they can be installed on your system using `pip`.

There are more than 270.000 packages freely available, at the time of writing.

> You should have `pip` already installed if you followed the Python installation instructions.

Install any package using the command `pip install`:

```
pip install <package>
```

or, if you do have troubles, you can also run it through `python -m`:

```
python -m pip install <package>
```

For example you can install the `requests` package, a popular HTTP library:

```
pip install requests
```

and once you do, it will be available for all your Python scripts, because packages are installed globally.

The exact location depends on your operating system.

On macOS, running Python 3.9, the location is `/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages`.

Upgrade a package to its latest version using:

```
pip install -U <package>
```

Install a specific version of a package using:

```
pip install <package>==<version>
```

Uninstall a package using:

```
pip uninstall <package>
```

Show an installed package details, including version, documentation website and author information using:

```
pip show <package>
```

List comprehensions

List comprehensions are a way to create lists in a very concise way.

Suppose you have a list:

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

You can create a new list using a list comprehension, composed by the `numbers` list elements, power 2:

```
numbers_power_2 = [n**2 for n in numbers]
# [1, 4, 9, 16, 25]
```

List comprehensions are a syntax that's sometimes preferred over loops, as it's more readable when the operation can be written on a single line:

```
numbers_power_2 = []
for n in numbers:
    numbers_power_2.append(n**2)
```

and over `map()`:

```
numbers_power_2 = list(map(lambda n : n**2, numbers)
```

Polymorphism

Polymorphism generalizes a functionality so it can work on different types. It's an important concept in object-oriented programming.

We can define the same method on different classes:

```
class Dog:
    def eat():
        print('Eating dog food')

class Cat:
    def eat():
        print('Eating cat food')
```

Then we can generate objects and we can call the `eat()` method regardless of the class the object belongs to, and we'll get different results:

```
animal1 = Dog()
animal2 = Cat()

animal1.eat()
animal2.eat()
```

We built a generalized interface and we now do not need to know that an animal is a Cat or a Dog.

Operator Overloading

Operator overloading is an advanced technique we can use to make classes comparable and to make them work with Python operators.

Let's take a class Dog:

```
class Dog:
    # the Dog class
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Let's create 2 Dog objects:

```
roger = Dog('Roger', 8)
syd = Dog('Syd', 7)
```

We can use operator overloading to add a way to compare those 2 objects, based on the `age` property:

```
class Dog:
    # the Dog class
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __gt__(self, other):
        return True if self.age > other.age else Fa
```

Now if you try running `print(roger > syd)` you will get the result `True`.

In the same way we defined `__gt__()` (which means greater than), we can define the following methods:

- `__eq__()` to check for equality
- `__lt__()` to check if an object should be considered lower than another with the `<` operator
- `__le__()` for lower or equal (`<=`)
- `__ge__()` for greater or equal (`>=`)
- `__ne__()` for not equal (`!=`)

Then you have methods to interoperate with arithmetic operations:

- `__add__()` respond to the `+` operator
- `__sub__()` respond to the `-` operator
- `__mul__()` respond to the `*` operator
- `__truediv__()` respond to the `/` operator
- `__floordiv__()` respond to the `//` operator
- `__mod__()` respond to the `%` operator
- `__pow__()` respond to the `**` operator
- `__rshift__()` respond to the `>>` operator
- `__lshift__()` respond to the `<<` operator
- `__and__()` respond to the `&` operator
- `__or__()` respond to the `|` operator
- `__xor__()` respond to the `^` operator

There are a few more methods to work with other operators, but you got the idea.

Virtual Environments

It's common to have multiple Python applications running on your system.

When applications require the same module, at some point you will reach a tricky situation where an app needs a version of a module, and another app a different version of that same module.

To solve this, you use **virtual environments**.

We'll use `venv`. Other tools work similarly, like `pipenv`.

Create a virtual environment using

```
python -m venv .venv
```

in the folder where you want to start the project, or where you already have an existing project.

Then run

```
source .venv/bin/activate
```

> Use `source .venv/bin/activate.fish` on the Fish shell

Executing the program will activate the Python virtual environment. Depending on your configuration you might also see your terminal prompt change.

Mine changed from

```
➜ folder
```

to

```
(.venv) ➜ folder
```

Now running `pip` will use this virtual environment instead of the global environment.

Conclusion

Thanks a lot for reading this book.

I hope it will inspire you to know more about Python.

For more on Python and programming in general, check out my blog flaviocopes.com.

Send any feedback, errata or opinions at flavio@flaviocopes.com