RENAN MOURA

THE PYTHON GUIDE FOR BEGINNERS

Start coding in Python 3

The Python Guide for Beginners

Renan Moura

The Python Guide for Beginners

1 Preface

Python has become one of the fastest-growing programming languages over the past few years.

Not only it is widely used, it is also an awesome language to tackle if you want to get into the world of programming.

This Python Guide for Beginners allows you to learn the core of the language in a matter of hours instead of weeks.

The intention of this book is not to be an exhaustive manual on everything Python has to offer as one of the major languages in modern programming.

I focus on what you will need to use most of the time to solve most of the problems as a beginner.

I deeply believe that you should be able to learn the core of any programming language and then go from there to dive into specifics and details as needed.

I'm Renan Moura and I write about Software Development on renanmf.com.

You can also find me as @renanmouraf on:

- Twitter: https://twitter.com/renanmouraf
- Linkedin: https://www.linkedin.com/in/renanmouraf
- Instagram: https://www.instagram.com/renanmouraf

2 Introduction to Python

Python was created in 1990 by Guido Van Rossum in Holland.

One of the objectives of the language was to be accessible to non-programmers.

Python was also designed to be a second language for programmers to learn due to its low learning curve and ease of use.

Python runs on Mac, Linux, Windows, and many other platforms.

Python is:

- Interpreted: it can execute at runtime, and changes in a program are instantly perceptible. To be very technical, Python has a compiler. The difference when compared to Java or C++ is how transparent and automatic it is. With Python, we don't have to worry about the compilation step as it's done in real-time. The tradeoff is that interpreted languages are usually slower than compiled ones.
- Semantically Dynamic: you don't have to specify types for variables and there is nothing that makes you do it.
- Object-Oriented: everything in Python is an object. But you can choose to write code in an object-oriented, procedural, or even functional way.
- High level: you don't have to deal with low-level machine details.

Python has been growing a lot recently partly because of its many uses in the following areas:

- System scripting: it's a great tool to automate everyday repetitive tasks.
- Data Analysis: it is a great language to experiment with and has tons of libraries and tools to handle data, create models, visualize results and even deploy solutions. This is used in areas like Finance, E-commerce, and Research.
- Web Development: frameworks like Django and Flask allow the development of web applications, API's, and websites.
- Machine Learning: Tensorflow and Pytorch are some of the libraries that allow scientists and the industry to develop and deploy Artificial Intelligence solutions in Image Recognition, Health, Self-driving cars, and many other fields.

You can easily organize your code in modules and reuse them or share them with others.

Finally, we have to keep in mind that Python had breaking changes between versions 2 and 3. And since Python 2 support ended in 2020, this article is solely based on Python 3.

So let's get started.

3 Installing Python 3

If you use a Mac or Linux you already have Python installed. But Windows doesn't come with Python installed by default.

You also might have Python 2, and we are going to use Python 3. So you should check to see if you have Python 3 first.

Type the following in your terminal.

```
python3 -V
```

Notice the uppercase `V`.

If your result is something similar to 'Python 3.x.y', for instance, `Python 3.8.1`, then you are ready to go.

If not, follow the next instructions according to your Operating System.

3.1 Installing Python 3 on Windows

Go to https://www.python.org/downloads/.

Download the latest version.

After the download, double-click the installer.

On the first screen, check the box indicating to "Add Python 3.x to PATH" and then click on "Install Now".

Wait for the installation process to finish until the next screen with the

message "Setup was successful".

Click on "Close".

3.2 Installing Python 3 on Mac

Install XCode from the App Store.

Install the command line tools by running the following in your terminal.

```
xcode-select --install
```

I recommend using Homebrew. Go to https://brew.sh/ and follow the instructions on the first page to install it.

After installing Homebrew, run the following `brew` commands to install Python 3.

```
brew update
brew install python3
```

Homebrew already adds Python 3 to the PATH, so you don't have to do anything else.

3.3 Installing Python 3 on Linux

To install using `apt`, available in Ubuntu and Debian, enter the following:

```
sudo apt install python3
```

To install using `yum`, available in RedHat and CentOS, enter the

following:

```
sudo yum install python3
```

4 Running Code

You can run Python code directly in the terminal as commands or you can save the code in a file with the `.py` extension and run the Python file.

4.1 Terminal

Running commands directly in the terminal is recommended when you want to run something simple.

Open the command line and type `python3`:

```
renan@mypc:~$ python3
```

You should see something like this in your terminal indicating the version (in my case, Python 3.6.9), the operating system (I'm using Linux), and some basic commands to help you.

The `>>>` tells us we are in the Python console.

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Let's test it by running our first program to perform basic math and add two numbers.

```
>>> 2 + 2
```

The output is:

```
4
```

To exit the Python console simply type `exit()`.

```
>>> exit()
```

4.2 Running `.py` files

If you have a complex program, with many lines of code, the Python console isn't the best option.

The alternative is simply to open a text editor, type the code, and save the file with a `.py` extension.

Let's do that, create a file called `second_program.py` with the following content.

```
print('Second Program')
```

The `print()` function prints a message on the screen.

The message goes inside the parentheses with either single quotes or double quotes, both work the same.

To run the program, on your terminal do the following:

```
renan@mypc:~$ python3 second_program.py
```

The output is:

```
Second Program
```

5 Syntax

Python is known for its clean syntax.

The language avoids using unnecessary characters to indicate some specificity.

5.1 Semicolons

Python makes no use of semicolons to finish lines. A new line is enough to tell the interpreter that a new command is beginning.

The `print()` function will display something.

In this example, we have two commands that will display the messages inside the single quotes.

```
print('First command')
print('Second command')
```

Output:

```
First command
Second command
```

This is **wrong** due to the semicolons in the end:

```
print('First command');
print('Second command');
```

5.2 Indentation

Many languages use curly-brackets to define scopes.

Python's interpreter uses only indentation to define when a scope ends and another one starts.

This means you have to be aware of white spaces at the beginning of each line -- they have meaning and might break your code if misplaced.

This definition of a function works:

```
def my_function():
    print('First command')
```

This **doesn't work** because the indentation of the second line is missing and will throw an error:

```
def my_function():
print('First command')
```

5.3 Case sensitivity and variables

Python is case sensitive. So the variables `name` and `Name` are not the same thing and store different values.

```
name = 'Renan'
Name = 'Moura'
```

As you could see, variables are easily created by just assigning values to them using the `=` symbol.

This means `name` stores 'Renan' and `Name` stores 'Moura'.

5.4 Comments

Finally, to comment something in your code, use the hash mark `#`.

The commented part does not influence the program flow.

```
# this function prints something
def my_function():
    print('First command')
```

This was an overview, minor details on each of these will become more clear in the next chapters with examples and broader explanations.

6 Comments

The purpose of comments is to explain what is happening in the code.

Comments are written along with your code but do not influence your program flow.

When you work by yourself, maybe comments don't feel like something you should write. After all, at the moment, you know the whys of every single line of code.

But what if new people come on board your project after a year and the project has 3 modules, each with 10,000 lines of code?

Think about people who don't know a thing about your app and who are suddenly having to maintain it, fix it, or add new features.

Remember, there is no single solution for a given problem. Your way of solving things is yours and yours only. If you ask 10 people to solve the same problem, they will come up with 10 different solutions.

If you want others to fully understand your reasoning, good code design is mandatory, but comments are an integral part of any codebase.

6.1 How to Write Comments in Python

The syntax of comments in Python is rather easy: just use the hash mark `#` symbol in front of the text you want to be a comment.

```
#This is a comment and it won't influence my program flow
```

You can use a comment to explain what some piece of code does.

```
#calculates the sum of any given two numbers
a + b
```

6.2 Multiline Comments

Maybe you want to comment on something very complex or describe how some process works in your code.

In these cases, you can use multiline comments.

To do that, just use a single hash mark `#` for each line.

```
#Everything after the hash mark # is a comment
#This is a comment and it won't influence my program flow

#Calculates the cost of the project given variables a and b
#a is the time in months it will take until the project is finished
#b is how much money it will cost per month
a + b * 10
```

7 Variables

In any program, you need to store and manipulate data to create a flow or some specific logic.

That's what variables are for.

You can have a variable to store a name, another one to store the age of a person, or even use a more complex type to store all of this at once like a dictionary.

7.1 Creating also known as Declaring

Declaring a variable is a basic and straightforward operation in Python.

Just pick a name and attribute a value to it use the `=` symbol.

```
name='Bob'

age=32
```

You can use the `print()` function to show the value of a variable.

```
print(name)

print(age)
```

```
Bob

32
```

Notice that in Python there is no special word to declare a variable.

The moment you assign a value, the variable is created in memory.

Python also has dynamic typing, which means you don't have to tell it if your variable is a text or a number, for instance.

The interpreter infers the typing based on the value assigned.

If you need it, you can also re-declare a variable just by changing its value.

```
#declaring name as a string
name='Bob'
#re-declaring name as an int
name = 32
```

Keep in my mind, though, that this is not recommended since variables must have meaning and context.

If I have a variable called `name` I don't expect it to have a number stored in it.

7.2 Naming Convention

Let's continue from the last section when I talked about meaning and context.

Don't use random variable names like `x` or `y`.

Say you want to store the time of a party, just call it `party_time`.

Oh, did you notice the underscore `_`?

By convention, if you want to use a variable name that is composed of

two or more words, you separate them by underscores. This is called Snake Case.

Another option would be using CamelCase as in `partyTime`. This is very common in other languages, but not the convention in Python as stated before.

Variables are case sensitive, so `party_time` and `Party_time` are not the same. Also, keep in mind that the convention tells us to always use lower case.

Remember, use names that you can recall inside your program easily. Bad naming can cost you a lot of time and cause annoying bugs.

In summary, variable names:

- Are Case sensitive: `time` and `TIME` are not the same
- Have to start with an underscore `_` or a letter (DO NOT start with a number)
- Are allowed to have only numbers, letters and underscores. No special characters like: #, $, &, @, etc.

This, for instance, is **not** allowed: `party#time`, `10partytime`.

8 Types

To store data in Python you need to use a variable. And every variable has its type depending on the value of the data stored.

Python has dynamic typing, which means you don't have to explicitly declare the type of your variable -- but if you want to, you can.

Lists, Tuples, Sets, and Dictionaries are all data types and have dedicated chapters later on with more details, but we'll look at them briefly here.

This way I can show you the most important aspects and operations of each one in their own chapter while keeping this chapter more concise and focused on giving you a broad view of the main data types in Python.

8.1 Determining the Type

First of all, let's learn how to determine the data type.

Just use the `type()` function and pass the variable of your choice as an argument, like the example below.

```
print(type(my_variable))
```

8.2 Boolean

The boolean type is one of the most basic types of programming.

A boolean type variable can only represent either *True* or *False*.

```
my_bool = True
print(type(my_bool))

my_bool = bool(1024)
print(type(my_bool))
```

```
<class 'bool'>
<class 'bool'>
```

8.3 Numbers

There are three numeric types: int, float, and complex.

8.3.1 Integer

```
my_int = 32
print(type(my_int))

my_int = int(32)
print(type(my_int))
```

```
<class 'int'>
<class 'int'>
```

8.3.2 Float

```
my_float = 32.85
print(type(my_float))

my_float = float(32.85)
print(type(my_float))
```

```
<class 'float'>
```

```
<class 'float'>
```

8.3.3 Complex

```
my_complex_number = 32+4j
print(type(my_complex_number))

my_complex_number = complex(32+4j)
print(type(my_complex_number))
```

```
<class 'complex'>
<class 'complex'>
```

8.4 String

The text type is one of the most commons types out there and is often called *string* or, in Python, just `str`.

```
my_city = "New York"
print(type(my_city))

#Single quotes have exactly
#the same use as double quotes
my_city = 'New York'
print(type(my_city))

#Setting the variable type explicitly
my_city = str("New York")
print(type(my_city))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
```

You can use the `+` operator to concatenate strings.

Concatenation is when you have two or more strings and you want to

join them into one.

```
word1 = 'New '
word2 = 'York'

print(word1 + word2)
```

```
New York
```

The string type has many built-in methods that let us manipulate them. I will demonstrate how some of these methods work.

The `len()` function returns the length of a string.

```
print(len('New York'))
```

```
8
```

The `replace()` method replaces a part of the string with another. As an example, let's replace 'New' for 'Old'.

```
print('New York'.replace('New', 'Old'))
```

```
Old York
```

The `upper()` method will return all characters as uppercase.

```
print('New York'.upper())
```

```
NEW YORK
```

The `lower()` method does the opposite, and returns all characters as lowercase.

```
print('New York'.lower())
```

```
new york
```

8.5 Lists

A list has its items ordered and you can add the same item as many times as you want. An important detail is that lists are mutable.

Mutability means you can change a list after its creation by adding items, removing them, or even just changing their values. These operations will be demonstrated later in the chapter dedicated to Lists.

```
my_list = ["bmw", "ferrari", "maclaren"]
print(type(my_list))

my_list = list(("bmw", "ferrari", "maclaren"))
print(type(my_list))
```

```
<class 'list'>
<class 'list'>
```

8.6 Tuples

A tuple is just like a list: ordered, and allows repetition of items.

There is just one difference: a tuple is immutable.

Immutability means you can't change a tuple after its creation. If you try to add an item or update one, for instance, the Python interpreter will show you an error. I will show that these errors occur later in the chapter dedicated to Tuples.

```
my_tuple = ("bmw", "ferrari", "maclaren")
print(type(my_tuple))

my_tuple = tuple(("bmw", "ferrari", "maclaren"))
print(type(my_tuple))
```

```
<class 'tuple'>
<class 'tuple'>
```

8.7 Sets

Sets don't guarantee the order of the items and are not indexed.

A key point when using sets: they don't allow repetition of an item.

```
my_set = {"bmw", "ferrari", "maclaren"}
print(type(my_set))


my_set = set(("bmw", "ferrari", "maclaren"))
print(type(my_set))
```

```
<class 'set'>
<class 'set'>
```

8.8 Dictionaries

A dictionary doesn't guarantee the order of the elements and is mutable.

One important characteristic in dictionaries is that you can set your own access keys for each element.

```
my_dict = {"country" : "France", "worldcups" : 2}
print(type(my_dict))

my_dict = dict(country="France", worldcups=2)
print(type(my_dict))
```

```
<class 'dict'>
<class 'dict'>
```

9 Typecasting

Typecasting allows you to convert between different types.

This way you can have an `int` turned into a `str`, or a `float` turned into an `int`, for instance.

9.1 Explicit conversion

To cast a variable to a string just use the `str()` function.

```
# this is just a regular explicit intialization
my_str = str('32')
print(my_str)

# int to str
my_str = str(32)
print(my_str)

# float to str
my_str = str(32.0)
print(my_str)
```

```
32
32
32.0
```

To cast a variable to an integer just use the `int()` function.

```
# this is just a regular explicit intialization
my_int = int(32)
print(my_int)

# float to int: rounds down to 3
my_int = int(3.2)
print(my_int)

# str to int
my_int = int('32')
```

```
print(my_int)
```

```
32
3
32
```

To cast a variable to a float just use the `float()` function.

```
# this is an explicit intialization
my_float = float(3.2)
print(my_float)

# int to float
my_float = float(32)
print(my_float)

# str to float
my_float = float('32')
print(my_float)
```

```
3.2
32.0
32.0
```

What I did before is called an *explicit* type conversion.

In some cases you don't need to do the conversion explicitly, since Python can do it by itself.

9.2 Implicit conversion

The example below shows implicit conversion when adding an `int` and a `float`.

Notice that `my_sum` is `float`. Python uses `float` to avoid data loss since the `int` type can not represent the decimal digits.

```
my_int = 32
my_float = 3.2

my_sum = my_int + my_float

print(my_sum)

print(type(my_sum))
```

```
35.2
<class 'float'>
```

On the other hand, in this example, when you add an `int` and a `str`, Python will not be able to make the implicit conversion, and the explicit type conversion is necessary.

```
my_int = 32
my_str = '32'

# explicit conversion works
my_sum = my_int + int(my_str)
print(my_sum)

#implicit conversion throws an error
my_sum = my_int + my_str
```

```
64

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

The same error is thrown when trying to add `float` and `str` types without making an explicit conversion.

```
my_float = 3.2
my_str = '32'

# explicit conversion works
my_sum = my_float + float(my_str)
print(my_sum)
```

```
#implicit conversion throws an error
my_sum = my_float + my_str
```

```
35.2

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

10 User Input

If you need to interact with a user when running your program in the command line (for example, to ask for a piece of information), you can use the `input()` function.

```
country = input("What is your country? ") #user enters 'Brazil'

print(country)
```

```
Brazil
```

The captured value is always `string`. Just remember that you might need to convert it using typecasting.

```
age = input("How old are you? ") #user enters '29'

print(age)

print(type(age))

age = int(age)

print(type(age))
```

The output for each `print()` is:

```
29

<class 'str'>

<class 'int'>
```

Notice the age 29 is captured as `string` and then converted explicitly

to `int`.

11 Operators

In a programming language, operators are special symbols that you can apply to your variables and values in order to perform operations such as arithmetic/mathematical and comparison.

Python has lots of operators that you can apply to your variables and I will demonstrate the most used ones.

11.1 Arithmetic Operators

Arithmetic operators are the most common type of operators and also the most recognizable ones.

They allow you to perform simple mathematical operations.

They are:

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `**`: Exponentiation
- `//`: Floor Division, rounds down the result of a division
- `%`: Modulus, gives you the remainder of a division

Let's see a program that shows how each of them is used.

```
print('Addition:', 5 + 2)
print('Subtraction:', 5 - 2)
print('Multiplication:', 5 * 2)
```

```
print('Division:', 5 / 2)
print('Floor Division:', 5 // 2)
print('Exponentiation:', 5 ** 2)
print('Modulus:', 5 % 2)
```

```
Addition: 7

Subtraction: 3

Multiplication: 10

Division: 2.5

Floor Division: 2

Exponentiation: 25

Modulus: 1
```

11.1.1 Concatenation

Concatenation is when you have two or more strings and you want to join them into one.

This is useful when you have information in multiple variables and want to combine them.

For instance, in this next example I combine two variables that contain my first name and my last name respectively to have my full name.

The `+` operator is also used to concatenate.

```
first_name = 'Renan '
last_name = 'Moura'

print(first_name + last_name)
```

```
Renan Moura
```

Since concatenation is applied to strings, to concatenate strings with other types, you have to do an explicit typecast using `str()`.

I have to typecast the `int` value 30 to string with `str()` to concatenate it with the rest of the string.

```
age = 'I have ' + str(30) + ' years old'

print(age)
```

```
I have 30 years old
```

11.2 Comparison Operators

Use comparison operators to compare two values.

These operators return either `True` or `False`.

They are:

- `==`: Equal
- `!=`: Not equal
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to

Let's see a program that shows how each of them is used.

```
print('Equal:', 5 == 2)
print('Not equal:', 5 != 2)
print('Greater than:', 5 > 2)
print('Less than:', 5 < 2)
```

```
print('Greater than or equal to:', 5 >= 2)
print('Less than or equal to:', 5 <= 2)
```

```
Equal: False

Not equal: True

Greater than: True

Less than: False

Greater than or equal to: True

Less than or equal to: False
```

11.3 Assignment Operators

As the name implies, these operators are used to assign values to variables.

`x = 7` in the first example is a direct assignment storing the number `7` in the variable `x`.

The assignment operation takes the value on the right and assigns it to the variable on the left.

The other operators are simple shorthands for the Arithmetic Operators.

In the second example `x` starts with `7` and `x += 2` is just another way to write `x = x + 2`, which means the previous value of `x` is added by `2` and reassigned to `x` that is now equals to `9`.

- `=`: simple assignment

```
x = 7
print(x)
```

```
7
```

- `+=`: addition and assignment

```
x = 7
x += 2
print(x)
```

```
9
```

- `-=`: subtraction and assignment

```
x = 7
x -= 2
print(x)
```

```
5
```

- `*=`: multiplication and assignment

```
x = 7
x *= 2
print(x)
```

```
14
```

- `/=`: division and assignment

```
x = 7
x /= 2
print(x)
```

```
3.5
```

- `%=`: modulus and assignment

```
x = 7
x %= 2
print(x)
```

```
1
```

- `//=`: floor division and assignment

```
x = 7
x //= 2
print(x)
```

```
3
```

- `**=`: exponentiation and assignment

```
x = 7
x **= 2
print(x)
```

```
49
```

11.4 Logical Operators

Logical operators are used to combine statements applying boolean algebra.

They are:

- `and`: `True` only when both statements are true
- `or`: `False` only when both x and y are false
- `not`: The `not` operator simply inverts the input, `True` becomes `False` and vice versa.

Let's see a program that shows how each one is used.

```
x = 5
y = 2

print(x == 5 and y > 3)

print(x == 5 or y > 3)

print(not (x == 5))
```

```
False

True

False
```

11.5 Membership Operators

These operators provide an easy way to check if a certain object is present in a sequence: `string`, `list`, `tuple`, `set`, and `dictionary`.

They are:

- `in`: returns `True` if the object is present
- `not in`: returns `True` if the object is not present

Let's see a program that shows how each one is used.

```
number_list = [1, 2, 4, 5, 6]

print( 1 in number_list)

print( 5 not in number_list)

print( 3 not in number_list)
```

```
True

False

True
```

12 Conditionals

Conditionals are one of the cornerstones of any programming language.

They allow you to control the program flow according to specific conditions you can check.

12.1 The `if` statement

The way you implement a conditional is through the `if` statement.

The general form of an `if` statement is:

```
if expression:
    statement
```

The `expression` contains some logic that returns a boolean, and the `statement` is executed only if the return is `True`.

A simple example:

```
bob_age = 32
sarah_age = 29

if bob_age > sarah_age:
    print('Bob is older than Sarah')
```

```
Bob is older than Sarah
```

We have two variables indicating the ages of Bob and Sarah. The

condition in plain English says "if Bob's age is greater than Sarah's age, then print the phrase 'Bob is older than Sarah'".

Since the condition returns `True`, the phrase will be printed on the console.

12.2 The `if else` and `elif` statements

In our last example, the program only does something if the condition returns `True`.

But we also want it to do something if it returns `False` or even check a second or third condition if the first one wasn't met.

In this example, we swapped Bob's and Sarah's age. The first condition will return `False` since Sarah is older now, and then the program will print the phrase after the `else` instead.

```
bob_age = 29
sarah_age = 32

if bob_age > sarah_age:
    print('Bob is older than Sarah')
else:
    print('Bob is younger than Sarah')
```

```
Bob is younger than Sarah
```

Now, consider the example below with the `elif`.

```
bob_age = 32
sarah_age = 32

if bob_age > sarah_age:
    print('Bob is older than Sarah')
```

```
elif bob_age == sarah_age:
    print('Bob and Sarah have the same age')
else:
    print('Bob is younger than Sarah')
```

```
Bob and Sarah have the same age
```

The purpose of the `elif` is to provide a new condition to be checked before the `else` is executed.

Once again we changed their ages and now both are 32 years old.

As such, the condition in the `elif` is met. Since both have the same age the program will print "Bob and Sarah have the same age".

Notice you can have as many `elif`s as you want, just put them in sequence.

```
bob_age = 32
sarah_age = 32

if bob_age > sarah_age:
    print('Bob is older than Sarah')
elif bob_age < sarah_age:
    print('Bob is younger than Sarah')
elif bob_age == sarah_age:
    print('Bob and Sarah have the same age')
else:
    print('This one is never executed')
```

```
Bob and Sarah have the same age
```

In this example, the `else` is never executed because all the possibilities are covered in the previous conditions and thus could be removed.

12.3 Nested conditionals

You might need to check more than one conditional for something to happen.

In this case, you can nest your `if` statements.

For instance, the second phrase "Bob is the oldest" is printed only if both `if`s pass.

```
bob_age = 32
sarah_age = 28
mary_age = 25

if bob_age > sarah_age:
    print('Bob is older than Sarah')
    if bob_age > mary_age:
        print('Bob is the oldest')
```

```
Bob is the oldest
```

Or, depending on the logic, make it simpler with Boolean Algebra.

This way, your code is smaller, more readable and easier to maintain.

```
bob_age = 32
sarah_age = 28
mary_age = 25

if bob_age > sarah_age and bob_age > mary_age:
    print('Bob is the oldest')
```

```
Bob is the oldest
```

Or even:

```
bob_age = 32
sarah_age = 28
```

```
mary_age = 25

if bob_age > sarah_age > mary_age:
    print('Bob is the oldest')
```

```
Bob is the oldest
```

12.4 Ternary Operator

The ternary operator is a one-line `if` statement.

It's very handy for simple conditions.

This is how it looks:

```
<expression> if <condition> else <expression>
```

Consider the following Python code:

```
a = 25
b = 50
x = 0
y = 1

result = x if a > b else y

print(result)
```

```
1
```

Here we use four variables, `a` and `b` are for the condition, while `x` and `y` represent the expressions.

`a` and `b` are the values we are checking against each other to evaluate

some condition. In this case, we are checking if `a` is greater than `b`.

If the expression holds true, i.e., `a` is greater than `b` then the value of `x` will be attributed to `result` which will be equal to 0.

However, if `a` is less than `b`, then we have the value of `y` assigned to `result`, and `result` will hold the value `1`.

Since `a` is less than `b`, 25 < 50, `result` will have `1` as final value from `y`.

The easy way to remember how the condition is evaluated is to read it in plain English.

Our example would read: `result` will be `x` if `a` is greater than `b` otherwise `y`.

13 Lists

As promised in the Types chapter, this chapter and the next three about Tuples, Sets, and Dictionaries will have more in-depth explanations of each of them since they are very important and broadly used structures in Python to organize and deal with data.

A list has its items ordered and you can add the same item as many times as you want.

An important detail is that lists are mutable.

Mutability means you can change a list after its creation by adding items, removing them, or even just changing their values.

13.0.1 Initialization

13.0.1.1 Empty List

```
people = []
```

13.0.1.2 List with initial values

```
people = ['Bob', 'Mary']
```

13.0.2 Adding in a List

To add an item in the end of a list, use `append()`.

```
people = ['Bob', 'Mary']
people.append('Sarah')

print(people)
```

```
['Bob', 'Mary', 'Sarah']
```

To specify the position for the new item, use the `insert()` method.

```
people = ['Bob', 'Mary']
people.insert(0, 'Sarah')

print(people)
```

```
['Sarah', 'Bob', 'Mary']
```

13.0.3 Updating in a List

Specify the position of the item to update and set the new value.

```
people = ['Bob', 'Mary']
people[1] = 'Sarah'
print(people)
```

```
['Bob', 'Sarah']
```

13.0.4 Deleting in a List

Use the `remove()` method to delete the item given as an argument.

```
people = ['Bob', 'Mary']
people.remove('Bob')
print(people)
```

```
['Mary']
```

To delete everybody, use the `clear()` method.

```
people = ['Bob', 'Mary']
people.clear()
```

13.0.5 Retrieving in a List

Use the index to reference the item.

Remember that the index starts at 0.

So to access the second item use the index 1.

```
people = ['Bob', 'Mary']
print(people[1])
```

```
Mary
```

13.0.6 Check if a given item already exists in a List

```
people = ['Bob', 'Mary']

if 'Bob' in people:
  print('Bob exists!')
else:
  print('There is no Bob!')
```

14 Tuples

A tuple is similar to a list: it's ordered, and allows repetition of items.

There is just one difference: a tuple is immutable.

Immutability, if you remember, means you can't change a tuple after its creation. If you try to add an item or update one, for instance, the Python interpreter will show you an error.

14.0.1 Initialization

14.0.1.1 Empty Tuple

```
people = ()
```

14.0.1.2 Tuple with initial values

```
people = ('Bob', 'Mary')
```

14.0.2 Adding in a Tuple

Tuples are immutable. This means that if you try to add an item, you will see an error.

```
people = ('Bob', 'Mary')
people[2] = 'Sarah'
```

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

14.0.3 Updating in a Tuple

Update an item will also return an error.

But there is a trick: you can convert into a list, change the item, and then convert it back to a tuple.

```
people = ('Bob', 'Mary')
people_list = list(people)
people_list[1] = 'Sarah'
people = tuple(people_list)
print(people)
```

```
('Bob', 'Sarah')
```

14.0.4 Deleting in a Tuple

For the same reason you can't add an item, you also can't delete an item, since they are immutable.

14.0.5 Retrieving in a Tuple

Use the index to reference an item.

```
people = ('Bob', 'Mary')
print(people[1])
```

```
Mary
```

14.0.6 Check if a given item already exists in a Tuple

```
people = ('Bob', 'Mary')

if 'Bob' in people:
  print('Bob exists!')
else:
  print('There is no Bob!')
```

15 Sets

Sets don't guarantee the order of the items and are not indexed.

A key point when using sets: they don't allow repetition of an item.

15.0.1 Initialization

15.0.1.1 Empty Set

```
people = set()
```

15.0.1.2 Set with initial values

```
people = {'Bob', 'Mary'}
```

15.0.2 Adding in a Set

Use the `add()` method to add one item.

```
people.add('Sarah')
```

Use the `update()` method do add multiple items at once.

```
people.update(['Carol', 'Susan'])
```

Remember, Sets do not allow repetition, so if you add 'Mary' again, nothing changes.

```
people = {'Bob', 'Mary'}

people.add('Mary')

print(people)
```

```
{'Bob', 'Mary'}
```

15.0.3 Updating in a Set

Items in a set are not mutable. You have to either add or delete an item.

15.0.4 Deleting in a Set

To remove Bob from the set:

```
people = {'Bob', 'Mary'}
people.remove('Bob')
print(people)
```

```
{'Mary'}
```

To delete everybody:

```
people.clear()
```

15.0.5 Check if a given item already exists in a set

```
people = {'Bob', 'Mary'}

if 'Bob' in people:
  print('Bob exists!')
else:
  print('There is no Bob!')
```

16 Dictionaries

The dictionary doesn't guarantee the order of the elements and it is mutable.

One important characteristic of dictionaries is that you can set your customized access keys for each element.

16.0.1 Initialization of a Dictionary

16.0.1.1 Empty Dictionary

```
people = {}
```

16.0.1.2 Dictionary with initial values

```
people = {'Bob':30, 'Mary':25}
```

16.0.2 Adding in a Dictionary

If the key doesn't exist yet, it is appended to the dictionary.

```
people['Sarah']=32
```

16.0.3 Updating a Dictionary

If the key already exists, the value is just updated.

```
#Bob's age is 28 now
people['Bob']=28
```

Notice that the code is pretty much the same.

16.0.4 Deleting in a Dictionary

To remove Bob from the dictionary:

```
people.pop('Bob')
```

To delete everybody:

```
people.clear()
```

16.0.5 Retrieving in a Dictionary

```
bob_age = people['Bob']
print(bob_age)
```

```
30
```

16.0.6 Check if a given key already exists in a Dictionary

```
if 'Bob' in people:
  print('Bob exists!')
else:
  print('There is no Bob!')
```

17 `while` Loops

Loops are used when you need to repeat a block of code a certain number of times or apply the same logic over each item in a collection.

There are two types of loops: `for` and `while`.

You will learn about `for` loops in the next chapter.

17.1 Basic Syntax

The basic syntax of a `while` loop is as below.

```
while condition:
    statement
```

The loop will continue *until* the condition is `True`.

17.2 The square of a number is

The example below takes each value of `number` and calculates its squared value.

```
number = 1
while number <= 5:
    print(number, 'squared is', number**2)
    number = number + 1
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

You can use any variable name, but I chose `number` because it makes sense in the context. A common generic choice would be simply `i`.

The loop will go on until `number` (initialized with 1) is less than or equal to 5.

Notice that after the `print()` command, the variable `number` is incremented by 1 to take the next value.

If you don't do the incrementation you will have an infinite loop since `number` will never reach a value greater than 5. This is a very important detail!

17.3 `else` block

When the condition returns `False`, the `else` block will be called.

```
number = 1
while number <= 5:
    print(number, 'squared is', number**2)
    number = number + 1
else:
    print('No numbers left!')
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
No numbers left!
```

Notice the phrase 'No numbers left!' is printed after the loop ends, that is after the condition `number <= 5` evaluates to `False`.

17.4 How to break out of a `while` loop in Python?

Simply use the `break` keyword, and the loop will stop its execution.

```
number = 1
while number <= 5:
    print(number, 'squared is', number**2)
    number = number + 1
    if number == 4:
        break
```

```
1 squared is 1
2 squared is 4
3 squared is 9
```

The loop runs normally, and when `number` reaches 4 the `if` statement evaluates to `True` and the `break` command is called. This finishes the loop before the squared value of the numbers 4 and 5 are calculated.

17.5 How to skip an item in a `while` loop

The `continue` will do that for you.

I had to invert the order of the `if` statement and the `print()` to show how it works properly.

```
number = 0
while number < 5:
    number = number + 1
    if number == 4:
        continue
    print(number, 'squared is', number**2)
```

```
1 squared is 1
2 squared is 4
```

```
  3 squared is 9
  5 squared is 25
```

The program always checks if 4 is the current value of `number`. If it is, the square of 4 won't be calculated and the `continue` will skip to the next iteration when the value of `number` is 5.

18 `for` Loops

`for` loops are similar to `while` loops in the sense that they are used to repeat blocks of code.

The most important difference is that you can easily iterate over sequential types.

18.1 Basic Syntax

The basic syntax of a `for` loop is as below.

```
for item in collection:
    statement
```

18.2 Loop over a list

To loop over a list or any other collection, just proceed as shown in the example below.

```
cars = ['BMW', 'Ferrari', 'McLaren']
for car in cars:
    print(car)
```

```
BMW
Ferrari
McLaren
```

The list of `cars` contains three items. The for loop will iterate over the list and store each item in the `car` variable, and then execute a

statement, in this case `print(car)`, to print each car in the console.

18.3 `range()` function

The range function is widely used in for loops because it gives you a simple way to list numbers.

This code will loop through the numbers 0 to 5 and print each of them.

```
for number in range(5):
    print(number)
```

```
0
1
2
3
4
```

In contrast, without the `range()` function, we would do something like this.

```
numbers = [0, 1, 2, 3, 4]
for number in numbers:
    print(number)
```

```
0
1
2
3
4
```

You can also define a `start` and `stop` using `range()`.

Here we are starting in 5 and stoping in 10. The number you set to stop is not included.

```
for number in range(5, 10):
    print(number)
```

```
5
6
7
8
9
```

Finally, it is also possible to set a step.

Here we starting in 10 and incrementing by 2 until 20, since 20 is the stop, it is not included.

```
for number in range(10, 20, 2):
    print(number)
```

```
10
12
14
16
18
```

18.4 `else` block

When the items in the list are over, the `else` block will be called.

```
cars = ['BMW', 'Ferrari', 'McLaren']
for car in cars:
    print(car)
else:
    print('No cars left!')
```

```
BMW
Ferrari
McLaren
```

```
No cars left!
```

18.5 How to break out of a for loop in Python

Simply use the `break` keyword, and the loop will stop its execution.

```
cars = ['BMW', 'Ferrari', 'McLaren']
for car in cars:
    print(car)
    if car == 'Ferrari':
        break
```

```
BMW
Ferrari
```

The loop will iterate the list and print each car.

In this case, after the loop reaches 'Ferrari', the `break` is called and 'McLaren' won't be printed.

18.6 How to skip an item in a for loop

In this section, we'll learn how `continue` can do this for you.

I had to invert the order of the `if` statement and the `continue` to show how it works properly.

Notice that I always check if 'Ferrari' is the current item. If it is, 'Ferrari' won't be printed, and the `continue` will skip to the next item 'McLaren'.

```
cars = ['BMW', 'Ferrari', 'McLaren']
for car in cars:
    if car == 'Ferrari':
        continue
    print(car)
```

```
BMW
McLaren
```

18.7 Loop over a Loop: Nested Loops

Sometimes you have more complex collections, like a list of lists.

To iterate over these lists, you need nested `for` loops.

In this case, I have three lists: one of BMW models, another on Ferrari models, and finally one with McLaren models.

The first loop iterates over each brand's list, and the second will iterate over the models of each brand.

```
car_models = [
['BMW I8', 'BMW X3',
'BMW X1'],
['Ferrari 812', 'Ferrari F8',
'Ferrari GTC4'],
['McLaren 570S', 'McLaren 570GT',
'McLaren 720S']
]

for brand in car_models:
    for model in brand:
        print(model)
```

```
BMW I8
BMW X3
BMW X1
Ferrari 812
Ferrari F8
Ferrari GTC4
McLaren 570S
McLaren 570GT
McLaren 720S
```

18.8 Loop over other Data Structures

The same logic shown to apply `for` loops over a `list` can be extended to the other data structures: `tuple`, `set`, and `dictionary`.

I will briefly demonstrate how to make a basic loop over each one of them.

18.8.1 Loop over a Tuple

```
people = ('Bob', 'Mary')

for person in people:
  print(person)
```

```
Bob
Mary
```

18.8.2 Loop over a Set

```
people = {'Bob', 'Mary'}

for person in people:
  print(person)
```

```
Bob
Mary
```

18.8.3 Loop over a Dictionary

To print the keys:

```
people = {'Bob':30, 'Mary':25}

for person in people:
  print(person)
```

```
Bob
Mary
```

To print the values:

```
people = {'Bob':30, 'Mary':25}

for person in people:
  print(people[person])
```

```
30
25
```

19 Functions

As the code grows the complexity also grows. And functions help organize the code.

Functions are a handy way to create blocks of code that you can reuse.

19.1 Definition and Calling

In Python use the `def` keyword to define a function.

Give it a name and use parentheses to inform 0 or more arguments.

In the line after the declaration starts, remember to indent the block of code.

Here is an example of a function called `print_first_function()` that only prints a phrase 'My first function!'.

To call the function just use its name as defined.

```
def print_first_function():
    print('My first function!')

print_first_function()
```

```
My first function!
```

19.2 `return` a value

Use the `return` keyword to return a value from the function.

In this example the function `second_function()` returns the string 'My second function!'.

Notice that `print()` is a built-in function and our function is called from inside it.

The string returned by `second_function()` is passed as argument to the `print()` function.

```
def second_function():
    return 'My second function!'

print(second_function())
```

```
My second function!
```

19.3 `return` multiple values

Functions can also return multiple values at once.

`return_numbers()` returns two values simultaneously.

```
def return_numbers():
    return 10, 2

print(return_numbers())
```

```
(10, 2)
```

19.4 Arguments

You can define parameters between the parentheses.

When calling a function with parameters you have to pass arguments according to the parameters defined.

The past examples had no parameters, so there was no need for arguments. The parentheses remained empty when the functions were called.

19.4.1 One Argument

To specify one parameter, just define it inside the parentheses.

In this example, the function `my_number` expects one number as argument defined by the parameter `num`.

The value of the argument is then accessible inside the function to be used.

```
def my_number(num):
    return 'My number is: ' + str(num)

print(my_number(10))
```

```
My number is: 10
```

19.4.2 Two or more Arguments

To define more parameters, just use a comma to separate them.

Here we have a function that adds two numbers called `add`, it expects two arguments defined by `first_num` and `second_num`.

The arguments are added by the `+` operator and the result is then returned by the `return`.

```
def add(first_num, second_num):
    return first_num + second_num

print(add(10,2))
```

```
12
```

This example is very similar to the last one. The only difference is that we have 3 parameters instead of 2.

```
def add(first_num, second_num, third_num):
    return first_num + second_num + third_num

print(add(10,2,3))
```

```
15
```

This logic of defining parameters and passing arguments is the same for any number of parameters.

It is important to point out that the arguments have to be passed in the same order that the parameters are defined.

19.4.3 Default value.

You can set a default value for a parameter if no argument is given using the `=` operator and a value of choice.

In this function, if no argument is given, the number 30 is assumed as

the expected value by default.

```
def my_number(my_number = 30):
    return 'My number is: ' + str(my_number)

print(my_number(10))
print(my_number())
```

```
My number is: 10
My number is: 30
```

19.4.4 Keyword or Named Arguments

When calling a function, the order of the arguments have to match the order of the parameters.

The alternative is if you use keyword or named arguments.

Set the arguments to their respective parameters directly using the name of the parameters and the `=` operators.

This example flips the arguments, but the function works as expected because I tell which value goes to which parameter by name.

```
def my_numbers(first_number, second_number):
    return 'The numbers are: ' + str(first_number) + ' and ' + str(second_number)

print(my_numbers(second_number=30, first_number=10))
```

```
The numbers are: 10 and 30
```

19.4.5 Any number of arguments: `*args`

If you don't want to specify the number of parameters, just use the `*` before the parameter name. Then the function will take as many arguments as necessary.

The parameter name could be anything like `*numbers`, but there is a convention in Python to use `*args` for this definition of a variable number of arguments.

```
def my_numbers(*args):
    for arg in args:
        print(arg)

my_numbers(10,2,3)
```

```
10
2
3
```

19.4.6 Any number of Keyword/Named arguments: `**kwargs`

Similar to `*args`, we can use `**kwargs` to pass as many keyword arguments as we want, as long as we use `**`.

Again, the name could be anything like `**numbers`, but `**kwargs` is a convention.

```
def my_numbers(**kwargs):
    for key, value in kwargs.items():
        print(key)
        print(value)

my_numbers(first_number=30, second_number=10)
```

```
first_number
30
second_number
10
```

19.4.7 Other types as arguments

The past examples used mainly numbers, but you can pass any type as argument and they will be treated as such inside the function.

This example is taking strings as arguments.

```
def my_sport(sport):
    print('I like ' + sport)

my_sport('football')
my_sport('swimming')
```

```
I like football
I like swimming
```

This function takes a list as argument.

```
def my_numbers(numbers):
    for number in numbers:
        print(number)

my_numbers([30, 10, 64, 92, 105])
```

```
30
10
64
92
105
```

20 Scope

The place where a variable is created defines its availability to be accessed and manipulated by the rest of the code. This is known as **scope**.

There are two types of scope: local and global.

20.1 Global Scope

A global scope allows you to use the variable anywhere in your program.

If your variable is outside a function, it has global scope by default.

```
name = 'Bob'

def print_name():
  print('My name is ' + name)

print_name()
```

```
My name is Bob
```

Notice that the function could use the variable `name` and print `My name is Bob`.

20.2 Local Scope

When you declare a variable inside a function, it only exists inside that function and can't be accessed from the outside.

```
def print_name():
    name = "Bob"
    print('My name is ' + name)

print_name()
```

```
My name is Bob
```

The variable `name` was declared inside the function, the output is the same as before.

But this will throw an error:

```
def print_name():
    name = 'Bob'
    print('My name is ' + name)

print(name)
```

The output of the code above is:

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

I tried to print the variable `name` from outside the function, but the scope of the variable was local and could not be found in a global scope.

20.3 Mixing Scopes

If you use the same name for variables inside and outside a function, the function will use the one inside its scope.

So when you call `print_name()`, the `name='Bob'` is used to print the

phrase.

On the other hand, when calling `print()` outside the function scope, `name="Sarah"` is used because of its global scope.

```
name = "Sarah"

def print_name():
    name = 'Bob'
    print('My name is ' + name)

print_name()

print(name)
```

The output of the code above is:

```
My name is Bob

Sarah
```

21 List Comprehensions

Sometimes we want to perform some very simple operations over the items of a list.

List comprehensions give us a succinct way to work on lists as an alternative to other methods of iteration, such as `for` loops.

21.1 Basic syntax

To use a list comprehension to replace a regular for loop, we can make:

```
[expression for item in list]
```

Which is the same as doing:

```
for item in list:
    expression
```

If we want some conditional to apply the expression, we have:

```
[expression for item in list if conditional ]
```

Which is the same as doing:

```
for item in list:
    if conditional:
        expression
```

21.2 Example 1: calculating the cube of a number

Regular way

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

for n in numbers:
    new_list.append(n**3)

print(new_list)
```

```
[1, 8, 27, 64, 125]
```

Using list comprehensions

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

new_list = [n**3 for n in numbers]

print(new_list)
```

```
 [1, 8, 27, 64, 125]
```

21.3 Example 2: calculating the cube of a number only if it is greater than 3

Using the conditional, we can filter only the values we want the expression to be applied to.

Regular way

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

for n in numbers:
    if(n > 3):
        new_list.append(n**3)

print(new_list)
```

```
[64, 125]
```

Using list comprehensions

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

new_list = [n**3 for n in numbers if n > 3]

print(new_list)
```

```
[64, 125]
```

21.4 Example 3: calling functions with list comprehensions

We can also call functions using the list comprehension syntax:

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

def cube(number):
    return number**3

new_list = [cube(n) for n in numbers if n > 3]

print(new_list)
```

```
[64, 125]
```

22 Lambda Functions

A Python lambda function can only have one expression and can't have multiple lines.

It is supposed to make it easier to create some small logic in one line instead of a whole function as it is usually done.

Lambda functions are also anonymous, which means there is no need to name them.

22.1 Basic Syntax

The basic syntax is very simple: just use the `lambda` keyword, define the parameters needed, use ":" to separate the parameters from the expression.

The general form is:

```
lambda arguments : expression
```

22.1.1 One parameter example

Look at this example using only one parameter.

```
cubic = lambda number : number**3
print(cubic(2))
```

```
8
```

22.1.2 Multiple parameter example

If you want, you can also have multiple parameters.

```
exponential = lambda multiplier, number, exponent : multiplier * number**exponent
print(exponential(2, 2, 3))
```

```
16
```

22.1.3 Calling the Lambda Function directly

You don't need to use a variable as we did before, you can make use of parenthesis around the lambda function and another pair of parenthesis around the arguments.

The declaration of the function and the execution will happen in the same line.

```
(lambda multiplier, number, exponent : multiplier * number**exponent)(2, 2, 3)
```

```
16
```

22.2 Examples using lambda functions with other built-in functions

22.2.1 Map

The Map function applies the expression to each item in a list.

Let's calculate the cubic of each number in the list.

```
numbers = [2, 5, 10]
cubics = list(map(lambda number : number**3, numbers))
print(cubics)
```

```
[8, 125, 1000]
```

22.2.2 Filter

The Filter function will filter the list based on the expression.

Let's filter to have only the numbers greater than 5.

```
numbers = [2, 5, 10]
filtered_list = list(filter(lambda number : number > 5, numbers))
print(filtered_list)
```

```
[10]
```

23 Modules

After some time your code starts to get more complex with lots of functions and variables.

To make it easier to organize the code we use Modules.

A well-designed Module also has the advantage of being reusable, so you write code once and reuse it everywhere.

You can write a module with all the mathematical operations and other people can use it.

And, if you need, you can use someone else's modules to simplify your code, speeding up your project.

In other programming languages, these are also referred to as libraries.

23.1 Using a Module

To use a module we use the `import` keyword.

As the name implies we have to tell our program what module to import.

After that, we can use any function available in that module.

Let's see an example using the `math` module.

First, let's see how to have access to a constant, Euler's number.

```
import math

math.e
```

```
2.718281828459045
```

In this second example, we are going to use a function that calculates the square root of a number.

It is also possible to use the `as` keyword to create an alias.

```
import math as m

m.sqrt(121)

m.sqrt(729)
```

```
11
27
```

Finally, using the `from` keyword, we can specify exactly what to import instead of the whole module and use the function directly without the module's name.

This example uses the `floor()` function that returns the largest integer less than or equal to a given number.

```
from math import floor

floor(9.8923)
```

```
9
```

23.2 Creating a Module

Now that we know how to use modules, let's see how to create one.

It is going to be a module with the basic math operations `add`, `subtract`, `multiply`, `divide` and it is gonna be called `basic_operations`.

Create the `basic_operations.py` file with the four functions.

```
def add(first_num, second_num):
    return first_num + second_num

def subtract(first_num, second_num):
    return first_num - second_num

def multiply(first_num, second_num):
    return first_num * second_num

def divide(first_num, second_num):
    return first_num / second_num
```

Then, just import the `basic_operations` module and use the functions.

```
import basic_operations

basic_operations.add(10,2)
basic_operations.subtract(10,2)
basic_operations.multiply(10,2)
basic_operations.divide(10,2)
```

```
12
8
20
5.0
```

24 `if __name__ == '__main__'`

You are on the process of building a module with the basic math operations `add`, `subtract`, `multiply`, `divide` called `basic_operations` saved in the `basic_operations.py` file.

To guarantee everything is fine, you can run some tests.

```
def add(first_num, second_num):
    return first_num + second_num

def subtract(first_num, second_num):
    return first_num - second_num

def multiply(first_num, second_num):
    return first_num * second_num

def divide(first_num, second_num):
    return first_num / second_num

print(add(10, 2))
print(subtract(10,2))
print(multiply(10,2))
print(divide(10,2))
```

After running the code:

```
renan@pro-home:~$ python3 basic_operations.py
```

The output is:

```
12
8
20
5.0
```

The output for those tests are what we expected.

Our code is right and ready to share.

Let's import the new module and run it in the Python console.

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import basic_operations
12
8
20
5.0
>>>
```

When the module is imported our tests are displayed on the screen even though we didn't do anything besides importing `basic_operations`.

To fix that we use `if __name__ == '__main__'` in the `basic_operations.py` file like this:

```
def add(first_num, second_num):
    return first_num + second_num

def subtract(first_num, second_num):
    return first_num - second_num

def multiply(first_num, second_num):
    return first_num * second_num

def divide(first_num, second_num):
    return first_num / second_num

if __name__ == '__main__':
    print(add(10, 2))
    print(subtract(10,2))
    print(multiply(10,2))
    print(divide(10,2))
```

Running the code directly on the terminal will continue to display the

tests. But when you import it like a module, the tests won't show and you can use the functions the way you intended.

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import basic_operations
>>> basic_operations.multiply(10,2)
20
>>>
```

Now that you know how to use the `if __name__ == '__main__'`, let's understand how it works.

The condition tells when the interpreter is treating the code as a module or as a program itself being executed directly.

Python has this native variable called `__name__`.

This variable represents the name of the module which is the name of the `.py` file.

Create a file `my_program.py` with the following and execute it.

```
print(__name__)
```

The output will be:

```
__main__
```

This tells us that when a program is executed directly, the variable `__name__` is defined as `__main__`.

But when it is imported as a module, the value of `__name__` is the name of the module.

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_program
my_program
>>>
```

This is how the Python interpreter differentiates the behavior of an imported module and a program executed directly in the terminal.

25 Files

Creating, deleting, reading, and many other functions applied to files are an integral part of many programs.

As such, it is very important to know how to organize and deal with files directly from your code.

Let's see how to handle files in Python.

25.1 File create

First things first, create!

We are going to use the `open()` function.

This function opens a file and returns its corresponding object.

The first argument is the name of the file we are handling, the second refers to the operation we are using.

The code below creates the file "people.txt", the `x` argument is used when we just want to create the file. If a file with the same name already exists, it will throw an exception.

```
people_file = open("people.txt", "x")
```

You can also we the `w` mode to create a file. Unlike the `x` mode, it will not throw an exception since this mode indicates the *writing* mode. We are opening a file to write data into it and, if the file doesn't exist, it is

created.

```
people_file = open("people.txt", "w")
```

The last one is the `a` mode which stands for *append*. As the name implies, you can append more data to the file, while the `w` mode simply overwrites any existing data.

When appending, if the file doesn't exist, it also creates it.

```
people_file = open("people.txt", "a")
```

25.2 File write

To write data into a file, you simply open a file with the `w` mode.

Then, to add data, you use the object returned by the `open()` function, in this case, the object is called `people_file`. Then call the *write()* function passing the data as argument.

```
people_file = open("people.txt", "w")
people_file.write("Bob\n")
people_file.write("Mary\n")
people_file.write("Sarah\n")
people_file.close()
```

We use `\n` at the end to break the line, otherwise the content in the file will stay in the same line as "BobMarySarah".

One more detail is to *close()* the file. This is not only a good practice, but also ensures that your changes were applied to the file.

Remember that when using `w` mode, the data that already existed in the file will be overwritten by the new data. To add new data without losing what was already there, we have to use the append mode.

25.3 File append

The `a` mode appends new data to the file, keeping the existing one.

In this example, after the first writing with `w` mode, we are using the `a` mode to append. The result is that each name will appear twice in the file "people.txt".

```
#first write
people_file = open("people.txt", "w")
people_file.write("Bob\n")
people_file.write("Mary\n")
people_file.write("Sarah\n")
people_file.close()

#appending more data
#keeping the existing data
people_file = open("people.txt", "a")
people_file.write("Bob\n")
people_file.write("Mary\n")
people_file.write("Sarah\n")
people_file.close()
```

25.4 File read

Reading the file is also very straightforward: just use the `r` mode like so.

If you read the "people.txt" file created in the last example, you should see 6 names in your output.

```
people_file = open("people.txt", "r")
print(people_file.read())
```

```
Bob
Mary
Sarah
Bob
Mary
Sarah
```

The `read()` function reads the whole file at once if you use the `readline()` function, you can read the file line by line.

```
people_file = open("people.txt", "r")
print(people_file.readline())
print(people_file.readline())
print(people_file.readline())
```

```
Bob
Mary
Sarah
```

You can also a loop to read the lines like the example below.

```
people_file = open("people.txt", "r")
for person in people_file:
  print(person)
```

```
Bob
Mary
Sarah
Bob
Mary
Sarah
```

25.5 Delete a File

To delete a file, you also need the os module.

Use the `remove()` method.

```
import os

os.remove('my_file.txt')
```

25.6 Check if a File Exists

Use the `os.path.exists()` method to check the existence of a file.

```
import os

if os.path.exists('my_file.txt'):
  os.remove('my_file.txt')
else:
  print('There is no such file!')
```

25.7 Copy a File

For this one, I like to use the `copyfile()` method from the `shutil` module.

```
from shutil import copyfile

copyfile('my_file.txt','another_file.txt')
```

There are a few options to copy a file, but `copyfile()` is the fastest one.

25.8 Rename and Move a File

If you need to move or rename a file you can use the `move()` method from the `shutil` module.

```
from shutil import move

move('my_file.txt','another_file.txt')
```

26 Classes and Objects

Classes and Objects are the fundamental concepts of Object-Oriented Programming.

In Python, **everything is an object**!

A variable (object) is just an instance of its type (class).

That's why when you check the type of a variable you can see the `class` keyword right next to its type (class).

This code snippet shows that `my_city` is an object and it is an instance of the class `str`.

```
my_city = "New York"
print(type(my_city))
```

```
<class 'str'>
```

26.1 Differentiate Class x Object

The class gives you a standard way to create objects. A class is like a base project.

Say you are an engineer working for Boeing.

Your new mission is to build the new product for the company, a new model called 747-Space. This aircraft flies higher altitudes than other commercial models.

Boeing needs to build dozens of those to sell to airlines all over the world, and the aircrafts have to be all the same.

To guarantee that the aircrafts (objects) follow the same standards, you need to have a project (class) that can be replicable.

The class is a project, a blueprint for an object.

This way you make the project once, and reuse it many times.

In our code example before, consider that every string has the same behavior and the same attributes. So it only makes sense for strings to have a class str to define them.

26.2 Attributes and Methods

Objects have some behavior which is is given by attributes and methods.

In simple terms, in the context of an object, attributes are variables and methods are functions attached to an object.

For example, a string has many built-in methods that we can use.

They work like functions, you just need to them from the objects using a `.`.

In this code snippet, I'm calling the `replace()` method from the string variable `my_city` which is an object, and an instance of the class `str`.

The `replace()` method replaces a part of the string for another and returns a new string with the change. The original string remains the

same.

Let's replace 'New' for 'Old' in 'New York'.

```
my_city = 'New York'
print(my_city.replace('New', 'Old'))
print(my_city)
```

```
Old York
New York
```

26.3 Creating a Class

We have used many objects (instances of classes) like strings, integers, lists, and dictionaries. All of them are instances of predefined classes in Python.

To create our own classes we use the `class` keyword.

By convention, the name of the class matches the name of the `.py` file and the module by consequence. It is also a good practice to organize the code.

Create a file `vehicle.py` with the following class `Vehicle`.

```
class Vehicle:
    def __init__(self, year, model, plate_number, current_speed = 0):
        self.year = year
        self.model = model
        self.plate_number = plate_number
        self.current_speed = current_speed

    def move(self):
        self.current_speed += 1

    def accelerate(self, value):
        self.current_speed += value
```

```
    def stop(self):
        self.current_speed = 0

    def vehicle_details(self):
        return self.model + ', ' + str(self.year) + ', ' + self.plate_number
```

Let's break down the class to explain it in parts.

The `class` keyword is used to specify the name of the class `Vehicle`.

The `__init__` function is a built-in function that all classes have, it is called when an object is created and is often used to initialize the attributes, assigning values to them, similar to what is done to variables.

The first parameter `self` in the `__init__` function is a reference to the object (instance) itself. We call it `self` by convention and it has to be the first parameter in every instance method, as you can see in the other method definitions `def move(self)`, `def accelerate(self, value)`, `def stop(self)`, and `def vehicle_details(self)`.

`Vehicle` has 4 attributes: `year`, `model`, `plate_number`, and `current_speed`.

Inside the `__init__`, each one of them is initialized with the parameters given when the object is instantiated.

Notice that `current_speed` is initialized with `0` by default, meaning that if no value is given, `current_speed` will be equal to 0 when the object is first instantiated.

Finally, we have three methods to manipulate our vehicle regarding its speed: `def move(self)`, `def accelerate(self, value)`, `def stop(self)`.

And one method to give back information about the vehicle: `def vehicle_details(self)`.

The implementation inside the methods work the same way as in functions. You can also have a `return` to give you back some value at the end of the method as demonstrated by `def vehicle_details(self)`.

26.4 Using the Class

Use the class on a terminal, import the `Vehicle` class from the `vehicle` module.

Create an instance called `my_car`, initializing `year` with 2009, `model` with 'F8', `plate_number` with 'ABC1234', and `current_speed` with 100.

The `self` parameter is not taken into consideration when calling methods. The Python interpreter infers its value as being the current object/instance automatically, so we just have to pass the other arguments when instantiating and calling methods.

Now use the methods to `move()` the car which increases its `current_speed` by 1, `accelerate(10)` which increases its `current_speed` by the value given in the argument, and `stop()` which sets the `current_speed` to 0.

Remember to print the value of `current_speed` at every command to see the changes.

To finish the test, call `vehicle_details()` to print the information about our vehicle.

```
>>> from vehicle import Vehicle
>>>
>>> my_car = Vehicle(2009, 'F8', 'ABC1234', 100)
>>> print(my_car.current_speed)
100
>>> my_car.move()
>>> print(my_car.current_speed)
101
>>> my_car.accelerate(10)
>>> print(my_car.current_speed)
111
>>> my_car.stop()
>>> print(my_car.current_speed)
0
>>> print(my_car.vehicle_details())
F8, 2009, ABC1234
```

If we don't set the initial value for `current_speed`, it will be zero by default as stated before and demonstrated in the next example.

```
>>> from vehicle import Vehicle
>>>
>>> my_car = Vehicle(2009, 'F8', 'ABC1234')
>>> print(my_car.current_speed)
0
>>> my_car.move()
>>> print(my_car.current_speed)
1
>>> my_car.accelerate(10)
>>> print(my_car.current_speed)
11
>>> my_car.stop()
>>> print(my_car.current_speed)
0
>>> print(my_car.vehicle_details())
F8, 2009, ABC1234
```

27 Inheritance

Let's define a generic `Vehicle` class and save it inside the `vehicle.py` file.

```
class Vehicle:
    def __init__(self, year, model, plate_number, current_speed):
        self.year = year
        self.model = model
        self.plate_number = plate_number
        self.current_speed = current_speed

    def move(self):
        self.current_speed += 1

    def accelerate(self, value):
        self.current_speed += value

    def stop(self):
        self.current_speed = 0

    def vehicle_details(self):
        return self.model + ', ' + str(self.year) + ', ' + self.plate_number
```

A vehicle has attributes `year`, `model`, `plate_number`, and `current_speed`.

The definition of vehicle in the `Vehicle` is very generic and might not be suitable for trucks for instance because it should include a `cargo` attribute.

On the other hand, a cargo attribute does not make much sense for small vehicles like motorcycles.

To solve this we can use *inheritance*.

When a class (child) inherits another class (parent), all the attributes and methods from to parent class are inherited by the child class.

27.1 Parent and Child

In our case, we want a new `Truck` class to inherit everything from the `Vehicle` class. Then we want it to add its own specific attribute `current_cargo` to control the addition and removal of cargo from the truck.

The `Truck` class is called a *child* class that inherits from its *parent* class `Vehicle`.

A *parent* class is also called a *superclass* while a *child* class is also known as a *subclass*.

Create the class `Truck` and save it inside the `truck.py` file.

```
from vehicle import Vehicle

class Truck(Vehicle):
    def __init__(self, year, model, plate_number, current_speed, current_cargo):
    super().__init__(year, model, plate_number, current_speed)
    self.current_cargo = current_cargo

    def add_cargo(self, cargo):
        self.current_cargo += cargo

    def remove_cargo(self, cargo):
        self.current_cargo -= cargo
```

Let's break down the class to explain it in parts.

The class `Vehicle` inside the parentheses when defining the class `Truck` indicates that the *parent* `Vehicle` is being inherited by its child `Truck`.

The `__init__` method has `self` as its first parameter, as usual.

The parameters `year`, `model`, `plate_number`, and `current_speed` are there to match the ones in the `Vehicle` class.

We added a new parameter `current_cargo` suited for the `Truck` class.

In the first line of the `__init__` method of the `Truck` class we have to call the `__init__` method of the `Vehicle` class.

To do that we use `super()` to make a reference to the *supperclass* `Vehicle`, so when `super().__init__(year, model, plate_number, current_speed)` is called we avoid repetition of our code.

After that, we can assign the value of `current_cargo` normally.

Finally, we have two methods to deal with the `current_cargo`: `def add_cargo(self, cargo):`, and `def remove_cargo(self, cargo):`.

Remember that `Truck` inherits attributes and methods from `Vehicle`, so we also have an implicit access to the methods that manipulate the speed: `def move(self)`, `def accelerate(self, value)`, `def stop(self)`.

27.2 Using the `Truck` class

Use the class in your terminal, import the `Truck` class from the `truck` module.

Create an instance called `my_truck`, initializing `year` with 2015, `model` with 'V8', `plate_number` with 'XYZ1234', `current_speed` with 0, and `current_cargo` with 0.

Use `add_cargo(10)` to increase `current_cargo` by 10, `remove_cargo(4)`, to decrease `current_cargo` by 4.

Remember to print the value of `current_cargo` at every command to see the changes.

By inheritance, we can use the methods from the `Vehicle` class to `move()` the truck which increases its `current_speed` by 1, `accelerate(10)` which increases its `current_speed` by the value given in the argument, and `stop()` which sets the `current_speed` to 0.

Remember to print the value of `current_speed` at every interaction to see the changes.

To finish the test, call `vehicle_details()` inherited from the `Vehicle` class to print the information about our truck.

```
>>> from truck import Truck
>>>
>>> my_truck = Truck(2015, 'V8', 'XYZ1234', 0, 0)
>>> print(my_truck.current_cargo)
0
>>> my_truck.add_cargo(10)
>>> print(my_truck.current_cargo)
10
>>> my_truck.remove_cargo(4)
>>> print(my_truck.current_cargo)
6
>>> print(my_truck.current_speed)
0
>>> my_truck.accelerate(10)
>>> print(my_truck.current_speed)
10
>>> my_truck.stop()
>>> print(my_truck.current_speed)
0
>>> print(my_truck.vehicle_details())
V8, 2015, XYZ1234
```

28 Exceptions

Errors are a part of every programmer's life, and knowing how to deal with them is a skill on its own.

The way Python deals with errors is called 'Exception Handling'.

If some piece of code runs into an error, the Python interpreter will raise an exception.

28.1 Types of Exceptions

Let's try to raise some exceptions on purpose and see the errors they produce.

- `TypeError`

First, try to add a string and an integer:

```
'I am a string' + 32
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

- `IndexError`

Now, try to access an index that doesn't exist in a list.

A common mistake is to forget that sequences are 0-indexed, meaning the first item has index 0, not 1.

In this example, the list `car_brands` ends at index 2.

```
car_brands = ['ford', 'ferrari', 'bmw']
print(car_brands[3])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- `NameError`

If we try to print a variable that doesn't exist.

```
print(my_variable)
```

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

- `ZeroDivisionError`

Math doesn't allow division by zero, trying to do so will raise an error, as expected.

```
32/0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

This was just a sample of the kinds of exceptions you might see on your daily routine and what can cause each of them.

28.2 Exception Handling

Now we know how to cause errors that will crash our code and shows us some message saying something is wrong.

To handle these exceptions just make use of the `try/except` statement.

```
try:
  32/0
except:
  print('Dividing by zero!')
```

```
Dividing by zero!
```

The example above shows the use of the `try` statement.

Put the block of code that may cause an exception inside the `try` scope. If everything runs alright, the `except` block is not invoked. But if an exception is raised, the block of code inside the `except` is executed.

This way the program doesn't crash and if you have some code after the exception, it will keep running if you want to.

28.3 Specific Exception Handling

In the last example the `except` block was generic, meaning it was catching anything.

Good practice it to specify the type of exception we are trying to catch, which helps a lot when debugging the code later.

If you know a block of code can throw an `IndexError`, specify it in the `except`:

```
try:
  car_brands = ['ford', 'ferrari', 'bmw']
  print(car_brands[3])
except IndexError:
  print('There is no such index!')
```

```
There is no such index!
```

You can use a tuple to specify as many exceptions types as you want in a single `except`:

```
try:
  print('My code!')
except(IndexError, ZeroDivisionError, TypeError):
  print('My Excepetion!')
```

28.4 `else`

It is possible to add an `else` command at the end of the `try/except`. It runs only if there are no exceptions.

```
my_variable = 'My variable'
try:
  print(my_variable)
except NameError:
  print('NameError caught!')
else:
  print('No NameError')
```

```
My variable
No NameError
```

28.5 Raising Exceptions

The `raise` command allows you to manually raise an exception.

This is particularly useful if you want to catch an exception and do something with it -- like logging the error in some personalized way like redirecting it to a log aggregator, or ending the execution of the code since the error should not allow the progress of the program.

```
try:
  raise IndexError('This index is not allowed')
except:
  print('Doing something with the exception!')
  raise
```

```
Doing something with the exception!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: This index is not allowed
```

28.6 `finally`

The `finally` block is executed independently of exceptions being raised or not.

They are usually there to allow the program to clean up resources like files, memory, network connections, etc.

```
try:
  print(my_variable)
except NameError:
  print('Except block')
finally:
  print('Finally block')
```

```
Except block
Finally block
```

29 Conclusion

That's it!

Congratulations on reaching the end.

I want to thank you for reading this book.

If you want to learn more, check out my blog renanmf.com.

Let me know if you have any suggestions by reaching out to me at renan@renanmf.com.

You can also find me as @renanmouraf on:

- Twitter: https://twitter.com/renanmouraf
- Linkedin: https://www.linkedin.com/in/renanmouraf
- Instagram: https://www.instagram.com/renanmouraf