

# Full Speed Python

João Ventura

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>4</b>
	Installing on Windows . . . . .	4
	Installing on macOS . . . . .	6
	Installing on Linux . . . . .	6
<b>3</b>	<b>Numbers and strings</b>	<b>8</b>
	Exercises with numbers . . . . .	9
	Exercises with strings . . . . .	9
<b>4</b>	<b>Lists</b>	<b>11</b>
	Exercises with lists . . . . .	11
	List comprehensions . . . . .	12
	Exercises with list comprehensions . . . . .	12
<b>5</b>	<b>Modules and functions</b>	<b>14</b>
	Exercises with the math module . . . . .	15
	Exercises with functions . . . . .	15
	Recursive functions . . . . .	15
	Exercises with recursive functions . . . . .	16
<b>6</b>	<b>Iteration and loops</b>	<b>17</b>
	Exercises with the for loop . . . . .	19
	Exercises with the while statement . . . . .	19
<b>7</b>	<b>Dictionaries</b>	<b>21</b>
	Exercises with dictionaries . . . . .	22
	Exercises with sub-dictionaries . . . . .	23
<b>8</b>	<b>Classes</b>	<b>24</b>
	Exercises with classes . . . . .	25
	Class inheritance . . . . .	25
	Exercises with inheritance . . . . .	25
<b>9</b>	<b>Iterators</b>	<b>27</b>
	Iterator classes . . . . .	28
	Exercises with iterators . . . . .	29

<b>10 Generators</b>	<b>30</b>
Exercises with generators . . . . .	31
<b>11 Coroutines</b>	<b>32</b>
Exercises with coroutines . . . . .	33
Pipelines . . . . .	33
Exercises with coroutine pipelines . . . . .	35
<b>12 Asynchronous programming</b>	<b>36</b>
Exercises with asyncio . . . . .	38

# Chapter 1

## Introduction

This book aims to teach the Python programming language using a practical approach. Its method is quite simple: after a short introduction to each topic, the reader is invited to learn more by solving the proposed exercises.

These exercises have been used extensively in my web development and distributed computing classes at the Superior School of Technology of Setúbal. With these exercises, most students are up to speed with Python in less than a month. In fact, students of the distributed computing course, taught in the second year of the software engineering degree, become familiar with Python's syntax in two weeks and are able to implement a distributed client-server application with sockets in the third week.

Please note that this book is a work in progress and, as such, may contain a few spelling errors that may be corrected in the future. However it is made available now as it is so it can be useful to anyone who wants to use it. I sincerely hope you can get something good through it.

The source of this book is available on github (<https://github.com/joaovventura/full-speed-python>). I welcome any pull requests to correct misspellings, suggest new exercises or to provide clarification of the current content.

All the best,

João Ventura - Adjunct Professor at the Escola Superior de Tecnologia de Setúbal.

# Chapter 2

## Installation

In this chapter we will install and run the Python interpreter in your local computer.

### Installing on Windows

1. Download the latest Python 3 release for Windows on <https://www.python.org/downloads/windows/> and execute the installer. At the time of writing, this is Python 3.6.4.
2. Make sure that the “Install launcher for all users” and “Add Python to PATH” settings are selected and choose “Customize installation”.

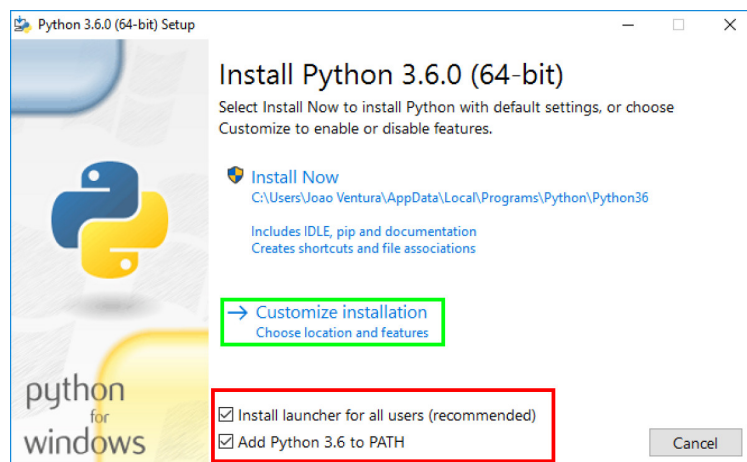


Figure 2.1: Windows installation

3. In the next screen “Optional Features”, you can install everything, but it is essential to install “pip” and “pylauncher (for all users)”. Pip is the Python package manager that allows you to install several Python packages and libraries.
4. In the Advanced Options, make sure that you select “Add Python to environment variables”. Also, I suggest that you change the install location to something like

C:\Python36\ as it will be easier for you to find the Python installation if something goes wrong.

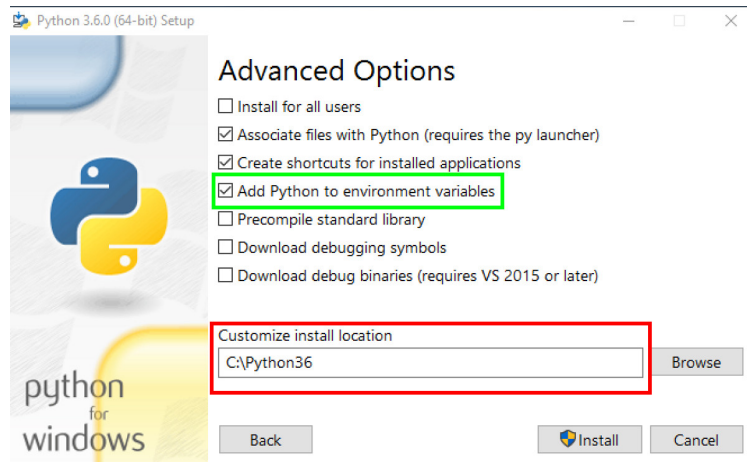


Figure 2.2: Windows installation

5. Finally, allow Python to use more than 260 characters on the file system by selecting “Disable path length limit” and close the installation dialog.

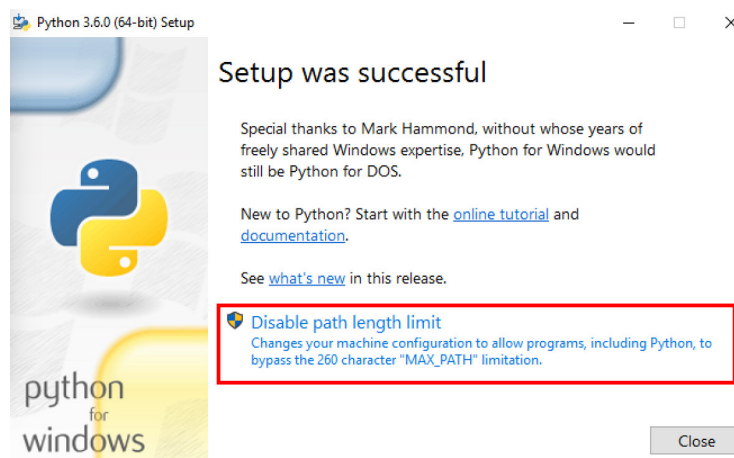
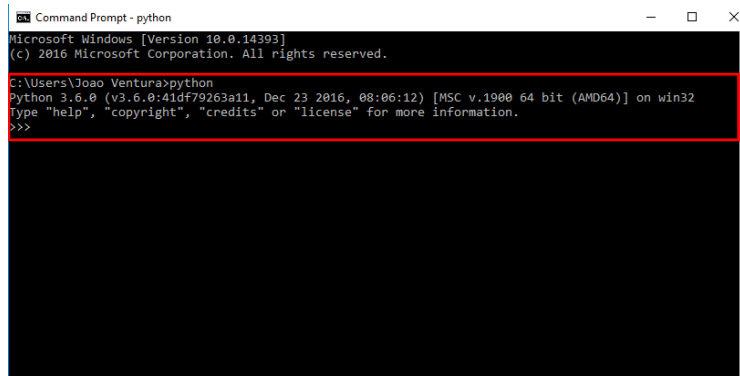


Figure 2.3: Windows installation

6. Now, open the command line (cmd) and execute “python” or “python3”. If everything was correctly installed, you should see the Python REPL. The REPL (from Read, Evaluate, Print and Loop) is a environment that you can use to program small snippets of Python code. Run *exit()* to leave the REPL.



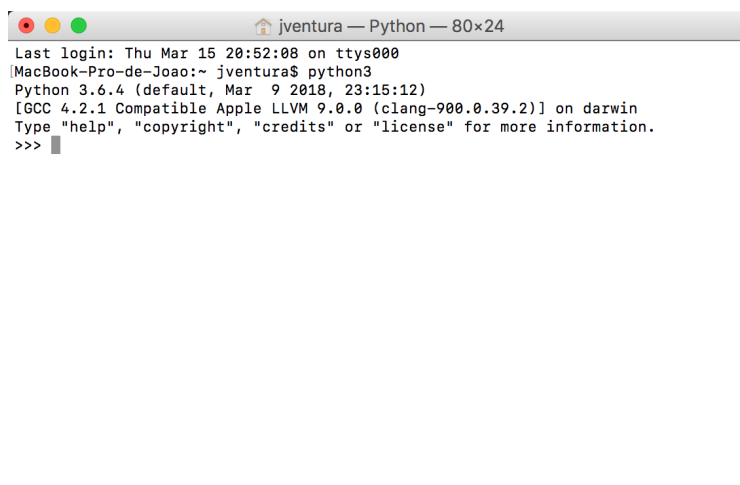
```
Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Joao Ventura>python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 2.4: Python REPL

## Installing on macOS

You can download the latest macOS binary releases from <https://www.python.org/downloads/mac-osx/>. Make sure you download the latest Python 3 release (3.6.4 at the time of writing). You can also use Homebrew, a package manager for macOS (<https://brew.sh/>). To install the latest Python 3 release with Homebrew, just do “**brew install python3**” on your terminal. Another option is to use the MacPorts package manager (<https://www.macports.org/>) and command “**port install python36**”.



```
jventura — Python — 80x24
Last login: Thu Mar 15 20:52:08 on ttys000
MacBook-Pro-de-Joao:~ jventura$ python3
Python 3.6.4 (default, Mar 9 2018, 23:15:12)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 2.5: Python REPL

Finally, open the terminal, execute `python3` and you should see the Python REPL as above. Press `Ctrl+D` or write `exit()` to leave the REPL.

## Installing on Linux

To install Python on Linux, you can download the latest Python 3 source releases from <https://www.python.org/downloads/source/> or use your package manager (`apt-get`,

aptitude, synaptic and others) to install it. To make sure you have Python 3 installed on your system, run `python3 --version` in your terminal.

Finally, open the terminal, execute `python3` and you should see the Python REPL as in the following image. Press Ctrl+D or write `exit()` to leave the REPL.

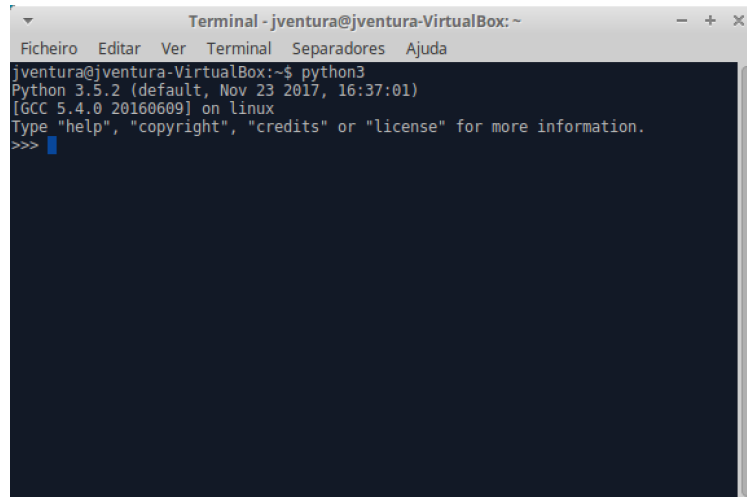
A screenshot of a terminal window titled "Terminal - jventura@jventura-VirtualBox: ~". The window has a menu bar with "Ficheiro", "Editar", "Ver", "Terminal", "Separadores", and "Ajuda". The terminal content shows the command `python3` being executed, resulting in the Python 3.5.2 shell. The output text is: `jventura@jventura-VirtualBox:~$ python3`, `Python 3.5.2 (default, Nov 23 2017, 16:37:01)`, `[GCC 5.4.0 20160609] on linux`, and `Type "help", "copyright", "credits" or "license" for more information.`. The prompt `>>>` is followed by a blue cursor.

Figure 2.6: Python REPL



# Chapter 3

## Numbers and strings

In this chapter we will work with the most basic datatypes, numbers and strings. Start your Python REPL and write the following:

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> b = 2.5
>>> type(b)
<class 'float'>
```

Basically, you are declaring **two variables** (named “a” and “b”) which will hold some numbers: variable “a” is an integer number while variable “b” is a real number. We can now use our variables or any other numbers to do some calculations:

```
>>> a + b
4.5
>>> (a + b) * 2
9.0
>>> 2 + 2 + 4 - 2/3
7.333333333333333
```

Python also has support for string datatypes. Strings are sequences of characters (like words) and can be defined using single or double quotes:

```
>>> hi = "hello"
>>> hi
'hello'
>>> bye = 'goodbye'
>>> bye
'goodbye'
```

You can add strings to concatenate them but you can not mix different datatypes, such as strings and integers.

```
>>> hi + "world"
'helloworld'
>>> "Hello" + 3
```

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

However, multiplication works as repetition:

```
>>> "Hello" * 3
'HelloHelloHello'
```

## Exercises with numbers

1. Try the following mathematical calculations and guess what is happening:  $((3 / 2))$ ,  $((3 // 2))$ ,  $((3 \% 2))$ ,  $((3**2))$ .

Suggestion: check the Python library reference at <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>.

2. Calculate the average of the following sequences of numbers: (2, 4), (4, 8, 9), (12, 14/6, 15)
3. The volume of a sphere is given by  $(4/3 * \pi * r^3)$ . Calculate the volume of a sphere of radius 5. Suggestion: create a variable named “pi” with the value of 3.1415.
4. Use the modulo operator (%) to check which of the following numbers is even or odd: (1, 5, 20, 60/7).

Suggestion: the remainder of  $(x/2)$  is always zero when  $(x)$  is even.

5. Find some values for  $(x)$  and  $(y)$  such that  $(x < 1/3 < y)$  returns “True” on the Python REPL. Suggestion: try  $(0 < 1/3 < 1)$  on the REPL.

## Exercises with strings

Using the Python documentation on strings (<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>), solve the following exercises:

1. Initialize the string “abc” on a variable named “s”:
  1. Use a function to get the length of the string.
  2. Write the necessary sequence of operations to transform the string “abc” in “aaabbbccc”. Suggestion: Use string concatenation and string indexes.
2. Initialize the string “aaabbbccc” on a variable named “s”:
  1. Use a function that allows you to find the first occurrence of “b” in the string, and the first occurrence of “ccc”.
  2. Use a function that allows you to replace all occurrences of “a” to “X”, and then use the same function to change only the first occurrence of “a” to “X”.

3. Starting from the string “aaa bbb ccc”, what sequences of operations do you need to arrive at the following strings? You can use the “replace” function.
1. “AAA BBB CCC”
  2. “AAA bbb CCC”

# Chapter 4

## Lists

Python lists are data structures that group sequences of elements. Lists can have elements of several types and you can also mix different types within the same list although all elements are usually of the same datatype.

Lists are created using square brackets and the elements separated by commas. The elements in a list can be accessed by their positions where 0 is the index of the first element:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[0]
1
>>> l[1]
2
```

Can you access the number 4 in the previous list?

Sometimes you want just a small portion of a list, a sublist. Sublists can be retrieved using a technique called *slicing*, which consists on defining the start and end indexes:

```
>>> l = ['a', 'b', 'c', 'd', 'e']
>>> l[1:3]
['b', 'c']
```

Finally, arithmetic with lists is also possible, like adding two lists together or repeating the contents of a list.

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2] * 2
[1, 2, 1, 2]
```

## Exercises with lists

Create a list named “l” with the following values ([1, 4, 9, 10, 23]). Using the Python documentation about lists (<https://docs.python.org/3.5/tutorial/introduction.html#lists>)

solve the following exercises:

1. Using list slicing get the sublists [4, 9] and [10, 23].
2. Append the value 90 to the end of the list "l". Check the difference between list concatenation and the "append" method.
3. Calculate the average value of all values on the list. You can use the "sum" and "len" functions.
4. Remove the sublist [4, 9].

## List comprehensions

List comprehensions are a concise way to create lists. It consists of square brackets containing an expression followed by the "for" keyword. The result will be a list whose results match the expression. Here's how to create a list with the squared numbers of another list.

```
>>> [x*x for x in [0, 1, 2, 3]]  
[0, 1, 4, 9]
```

Given its flexibility, list comprehensions generally make use of the "range" function which returns a range of numbers:

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

Sometimes you may want to filter the elements by a given condition. The "if" keyword can be used in those cases:

```
>>> [x for x in range(10) if x % 2 == 0]  
[0, 2, 4, 6, 8]
```

The example above returns all even values in range 0..10. More about list comprehensions can be found at <https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>.

## Exercises with list comprehensions

1. Using list comprehensions, create a list with the squares of the first 10 numbers.
2. Using list comprehensions, create a list with the cubes of the first 20 numbers.
3. Create a list comprehension with all the even numbers from 0 to 20, and another one with all the odd numbers.
4. Create a list with the squares of the even numbers from 0 to 20, and sum the list using the "sum" function. The result should be 1140. First create the list using list comprehensions, check the result, then apply the sum to the list comprehension.

5. Make a list comprehension that returns a list with the squares of all even numbers from 0 to 20, but ignore those numbers that are divisible by 3. In other words, each number should be divisible by 2 and not divisible by 3. Search for the “and” keyword in the Python documentation. The resulting list is [4, 16, 64, 100, 196, 256].

# Chapter 5

## Modules and functions

In this chapter we will talk about **modules and functions**. A function is a block of code that is used to perform a single action. A module is a Python file containing variables, functions and many more things.

Start up your Python REPL and let's use the “math” module which provides access to mathematical functions:

```
>>> import math
>>> math.cos(0.0)
1.0
>>> math.radians(275)
4.799655442984406
```

Functions are sequences of instructions that are executed when the function is **invoked**. The following defines the “do\_hello” function that prints two messages when invoked:

```
>>> def do_hello():
...     print("Hello")
...     print("World")
...
>>> do_hello()
Hello
World
```

Make sure that you insert **a tab** before both print expressions in the previous function. Tabs and spaces in Python are relevant and define that a block of code is somewhat dependent on a previous instruction. For instance, the print expressions are “inside” the “do\_hello” function therefore must have a tab.

Functions can also receive parameters a return values (using the “return” keyword):

```
>>> def add_one(val):
...     print("Function got value", val)
...     return val + 1
...
>>> value = add_one(1)
Function got value 1
```

```
>>> value
2
```

## Exercises with the math module

Use the Python documentation about the math module (<https://docs.python.org/3/library/math.html>) to solve the following exercises:

1. Find the greatest common divisor of the following pairs of numbers: (15, 21), (152, 200), (1988, 9765).
2. Compute the base-2 logarithm of the following numbers: 0, 1, 2, 6, 9, 15.
3. Use the “input” function to ask the user for a number and show the result of the sine, cosine and tangent of the number. Make sure that you convert the user input from string to a number (use the `int()` or the `float()` function).

## Exercises with functions

1. Implement the “add2” function that receives two numbers as arguments and returns the sum of the numbers. Then implement the “add3” function that receives and sums 3 parameters.
2. Implement a function that returns the greatest of two numbers given as parameters. Use the “if” statement to compare both numbers: <https://docs.python.org/3/tutorial/controlflow.html#if-statements>.
3. Implement a function named “is\_divisible” that receives two parameters (named “a” and “b”) and returns true if “a” can be divided by “b” or false otherwise. A number is divisible by another when the remainder of the division is zero. Use the modulo operator (“%”).
4. Create a function named “average” that computes the average value of a list passed as parameter to the function. Use the “sum” and “len” functions.

## Recursive functions

In computer programming, a recursive function is simply a function that calls itself. For instance take the factorial function.

$$f(x) = \begin{cases} 1, & \text{if } x = 0. \\ x \times f(x - 1), & \text{otherwise.} \end{cases} \quad (5.1)$$

As an example, take the factorial of 5:



$$\begin{aligned}
5! &= 5 \times 4! \\
&= 5 \times 4 \times 3! \\
&= 5 \times 4 \times 3 \times 2! \\
&= 5 \times 4 \times 3 \times 2 \times 1 \\
&= 120
\end{aligned}
\tag{5.2}$$

Basically, the factorial of 5 is 5 times the factorial of 4, etc. Finally, the factorial of 1 (or of zero) is 1 which breaks the recursion. In Python we could write the following recursive function:

```
def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x-1)
```

The trick with recursive functions is that there must be a “base” case where the recursion must end and a recursive case that iterates towards the base case. In the case of factorial we know that the factorial of zero is one, and the factorial of a number greater than zero will depend on the factorial of the previous number until it reaches zero.

## Exercises with recursive functions

1. Implement the **factorial function** and test it with several different values. Cross-check with a calculator.
2. Implement a recursive function to compute the sum of the (n) first integer numbers (where (n) is a function parameter). Start by thinking about the base case (the sum of the first 0 integers is?) and then think about the recursive case.
3. The Fibonacci sequence is a sequence of numbers in which each number of the sequence matches the sum of the previous two terms. Given the following recursive definition implement (fib(n)).

$$fib(n) = \begin{cases} 0, & \text{if } x = 0. \\ 1, & \text{if } x = 1. \\ fib(n-1) + fib(n-2), & \text{otherwise.} \end{cases}
\tag{5.3}$$

Check your results for the first numbers of the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

# Chapter 6

## Iteration and loops

In this chapter we are going to explore the topics of iteration and loops. Loops are used in computer programming to automate repetitive tasks.

In Python the most common form of iteration is the “for” loop. The “for” loop allows you to iterate over all items of a list such that you can do whatever you want with each item. For instance, let’s create a list and print the square value of each element.

```
>>> for value in [0, 1, 2, 3, 4, 5]:
...     print(value * value)
...
0
1
4
9
16
25
```

It’s quite easy but very powerful! The “for” loop is the basis of many things in programming. For instance, you already know about the “sum(list)” function which sums all the elements of a list, but here’s an example using the “for” loop:

```
>>> mylist = [1,5,7]
>>> sum = 0
>>> for value in mylist:
...     sum = sum + value
...
>>> print(sum)
13
```

Basically, you create the variable “sum” and keep adding each value as it comes from the list.

Sometimes, instead of the values of a list, you may need to work with the indexes themselves, i.e., not with the values, but the positions where they are in the list. Here’s an example that iterates over a list and returns the indexes and the values for each index:

```

>>> mylist = [1,5,7]
>>> for i in range(len(mylist)):
...     print("Index:", i, "Value:", mylist[i])
...
Index: 0 Value: 1
Index: 1 Value: 5
Index: 2 Value: 7

```

You can see that we are not iterating over the list itself but iterating over the “range” of the length of the list. The range function returns a special list:

```

>>> list(range(3))
[0, 1, 2]

```

So, when you use “range” you are not iterating over “mylist” but over a list with some numbers that you’ll use as indexes to access individual values on “mylist”. More about the range function in the Python docs at <https://docs.python.org/3/tutorial/controlflow.html#the-range-function>.

Sometimes you may need both things (indexes and values), and you can use the “enumerate” function:

```

>>> mylist = [1,5,7]
>>> for i, value in enumerate(mylist):
...     print("Index:", i, "Value:", value)
...
Index: 0 Value: 1
Index: 1 Value: 5
Index: 2 Value: 7

```

Remember that the first value on a Python list is always at index 0.

Finally, we also have the “while” statement that allows us to repeat a sequence of instructions while a specified condition is true. For instance, the following example starts “n” at 10 and **while “n” is greater than 0**, it keeps subtracting 1 from “n”. When “n” reaches 0, the condition “n > 0” is false, and the loop ends:

```

>>> n = 10
>>> while n > 0:
...     print(n)
...     n = n-1
...
10
9
8
7
6
5
4
3
2

```

Notice that it never prints 0...

## Exercises with the for loop

For this section you may want to consult the Python docs at <https://docs.python.org/3/tutorial/controlflow.html#for-statements>.

1. Create a function “add” that receives a list as parameter and returns the sum of all elements in the list. Use the “for” loop to iterate over the elements of the list.
2. Create a function that receives a list as parameter and returns the maximum value in the list. As you iterate over the list you may want to keep the maximum value found so far in order to keep comparing it with the next elements of the list.
3. Modify the previous function such that it returns a list with the first element being the maximum value and the second being the index of the maximum value in the list. Besides keeping the maximum value found so far, you also need to keep the position where it occurred.
4. Implement a function that returns the reverse of a list received as parameter. You may create an empty list and keep adding the values in reversed order as they come from the original list. Check what you can do with lists at <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>.
5. Make the function “is\_sorted” that receives a list as parameter and returns True if the list is sorted in ascending order. For instance [1, 2, 2, 3] is ordered while [1, 2, 3, 2] is not. Suggestion: you have to compare a number in the list with the next one, so you can use indexes or you need to keep the previous number in a variable as you iterate over the list.
6. Implement the function “is\_sorted\_dec” which is similar to the previous one but all items must be sorted by decreasing order.
7. Implement the “has\_duplicates” function which verifies if a list has duplicate values. You may have to use two “for” loops, where for each value you have to check for duplicates on the rest of the list.

## Exercises with the while statement

1. Implement a function that receives a number as parameter and prints, in decreasing order, which numbers are even and which are odd, until it reaches 0.

```
>>> even_odd(10)
Even number: 10
Odd number: 9
Even number: 8
Odd number: 7
```

Even number: 6  
Odd number: 5  
Even number: 4  
Odd number: 3  
Even number: 2  
Odd number: 1

# Chapter 7

## Dictionaries

In this chapter we will work with Python dictionaries. Dictionaries are data structures that indexes values by a given key (key-value pairs). The following example shows a dictionary that indexes students ages by name.

```
ages = {  
    "Peter": 10,  
    "Isabel": 11,  
    "Anna": 9,  
    "Thomas": 10,  
    "Bob": 10,  
    "Joseph": 11,  
    "Maria": 12,  
    "Gabriel": 10,  
}  
  
>>> print(ages["Peter"])  
10
```

It is possible to iterate over the contents of a dictionary using “items”, like this:

```
>>> for name, age in ages.items():  
...     print(name, age)  
...  
Peter 10  
Isabel 11  
Anna 9  
Thomas 10  
Bob 10  
Joseph 11  
Maria 12  
Gabriel 10
```

However, dictionary keys don't necessarily need to be strings but can be any immutable object:

```
d = {
    0: [0, 0, 0],
    1: [1, 1, 1],
    2: [2, 2, 2],
}

>>> d[2]
[2, 2, 2]
```

And you can also use other dictionaries as values:

```
students = {
    "Peter": {"age": 10, "address": "Lisbon"},
    "Isabel": {"age": 11, "address": "Sesimbra"},
    "Anna": {"age": 9, "address": "Lisbon"},
}

>>> students['Peter']
{'age': 10, 'address': 'Lisbon'}
>>> students['Peter']['address']
'Lisbon'
```

This is quite useful to structure hierarchical information.

## Exercises with dictionaries

Use the Python documentation at <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict> to solve the following exercises.

Take the following Python dictionary:

```
ages = {
    "Peter": 10,
    "Isabel": 11,
    "Anna": 9,
    "Thomas": 10,
    "Bob": 10,
    "Joseph": 11,
    "Maria": 12,
    "Gabriel": 10,
}
```

1. How many students are in the dictionary? Search for the “len” function.
2. Implement a function that receives the “ages” dictionary as parameter and returns the average age of the students. Traverse all items on the dictionary using the “items” method as above.
3. Implement a function that receives the “ages” dictionary as parameter and returns the name of the oldest student.

4. Implement a function that receives the “ages” dictionary and a number “n” and returns a new dict where each student is (n) years older. For instance, *new\_ages(ages, 10)* returns a copy of “ages” where each student is 10 years older.

## Exercises with sub-dictionaries

Take the following dictionary:

```
students = {  
    "Peter": {"age": 10, "address": "Lisbon"},  
    "Isabel": {"age": 11, "address": "Sesimbra"},  
    "Anna": {"age": 9, "address": "Lisbon"},  
}
```

1. How many students are in the “students” dict? Use the appropriate function.
2. Implement a function that receives the students dict and returns the average age.
3. Implement a function that receives the students dict and an address, and returns a list with names of all students whose address matches the address in the argument. For instance, invoking “find\_students(students, 'Lisbon')” should return Peter and Anna.



# Chapter 8

## Classes

In object oriented programming (OOP), a class is a structure that allows to group together a set of properties (called attributes) and functions (called methods) to manipulate those properties. Take the following class that defines a person with properties “name” and “age” and the “greet” method.

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is %s!" % self.name)
```

Most classes will need the constructor method (“\_\_init\_\_”) to initialize the class’s attributes. In the previous case the constructor of the class receives the person’s name and age and stores that information in the class’s instance (referenced by the *self* keyword). Finally, “greet” method prints the name of the person as stored in a specific class instance (object).

Class instances are used through the instantiation of objects. Here’s how we can instantiate two objects:

```
>>> a = Person("Peter", 20)
>>> b = Person("Anna", 19)

>>> a.greet()
Hello, my name is Peter!
>>> b.greet()
Hello, my name is Anna!

>>> print(a.age)  # We can also access the attributes of an object
20
```

## Exercises with classes

Use the Python documentation on classes at <https://docs.python.org/3/tutorial/classes.html> to solve the following exercises.

1. Implement a class named “Rectangle” to store the coordinates of a rectangle given by (x1, y1) and (x2, y2).
2. Implement the class constructor with the parameters (x1, y1, x2, y2) and store them in the class instances using the “self” keyword.
3. Implement the “width()” and “height()” methods which return, respectively, the width and height of a rectangle. Create two objects, instances of “Rectangle” to test the calculations.
4. Implement the method “area” to return the area of the rectangle (width\*height).
5. Implement the method “circumference” to return the perimeter of the rectangle (2\*width + 2\*height).
6. Do a print of one of the objects created to test the class. Implement the “\_\_str\_\_” method such that when you print one of the objects it print the coordinates as (x1, y1)(x2, y2).

## Class inheritance

In object oriented programming, inheritance is one of the forms in which a subclass can inherit the attributes and methods of another class, allowing it to rewrite some of the super class’s functionalities. For instance, from the “Person” class above we could create a subclass to keep people with 10 years of age:

```
class TenYearOldPerson(Person):  
  
    def __init__(self, name):  
        super().__init__(name, 10)  
  
    def greet(self):  
        print("I don't talk to strangers!!")
```

The indication that the “TenYearOldPerson” class is a subclass of “Person” is given on the first line. Then, we rewrote the constructor of the subclass to only receive the name of the person, but we will eventually call the super class’s constructor with the name of the 10-year-old and the age hardcoded as 10. Finally we reimplemented the “greet” method.

## Exercises with inheritance

Use the “Rectangle” class as implemented above for the following exercises:

1. Create a “Square” class as subclass of “Rectangle”.

2. Implement the “Square” constructor. The constructor should have only the x1, y1 coordinates and the size of the square. Notice which arguments you’ll have to use when you invoke the “Rectangle” constructor when you use “super”.
3. Instantiate two objects of “Square”, invoke the area method and print the objects. Make sure that all calculations are returning correct numbers and that the coordinates of the squares are consistent with the size of the square used as argument.

# Chapter 9

## Iterators

As we saw previously, in Python we use the “for” loop to iterate over the contents of objects:

```
>>> for value in [0, 1, 2, 3, 4, 5]:
...     print(value)
...
0
1
4
9
16
25
```

Objects that can be used with a “for” loop are called iterators. An iterator is, therefore, an object that follows the iteration protocol.

The built-in function “iter” can be used to build iterator objects, while the “next” function can be used to gradually iterate over their content:

```
>>> my_iter = iter([1, 2, 3])
>>> my_iter
<list_iterator object at 0x10ed41cc0>
>>> next(my_iter)
1
>>> next(my_iter)
2
>>> next(my_iter)
3
>>> next(my_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If there are no more elements, the iterator raises a “StopIteration” exception.

## Iterator classes

Iterators can be implemented as classes. You just need to implement the “`__next__`” and “`__iter__`” methods. Here’s an example of a class that mimics the “range” function, returning all values from “a” to “b”:

```
class MyRange:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __iter__(self):
        return self

    def __next__(self):
        if self.a < self.b:
            value = self.a
            self.a += 1
            return value
        else:
            raise StopIteration
```

Basically, on every call to “next” it moves forward the internal variable “a” and returns its value. When it reaches “b”, it raises the `StopIteration` exception.

```
>>> myrange = MyRange(1, 4)
>>> next(myrange)
1
>>> next(myrange)
2
>>> next(myrange)
3
>>> next(myrange)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

But most important, you can use the iterator class in a “for” loop:

```
>>> for value in MyRange(1, 4):
...     print(value)
...
1
2
3
```

## Exercises with iterators

1. Implement an iterator class to return the square of all numbers from “a” to “b”.
2. Implement an iterator class to return all the even numbers from 1 to (n).
3. Implement an iterator class to return all the odd numbers from 1 to (n).
4. Implement an iterator class to return all numbers from (n) down to 0.
5. Implement an iterator class to return the fibonnaci sequence from the first element up to (n). You can check the definition of the fibonnaci sequence in the function’s chapter. These are the first numbers of the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
6. Implement an iterator class to return all consecutive pairs of numbers from 0 until (n), such as (0, 1), (1, 2), (2, 3)...

# Chapter 10

## Generators

If you read the previous chapter, you know that iterators are objects that are regularly used with “for” loops. In other words, iterators are objects that implement the iteration protocol. A Python generator is a convenient way to implement an iterator. Instead of a class, a generator is a function which returns a value each time the “yield” keyword is used. Here’s an example of a generator to count the values between two numbers:

```
def myrange(a, b):  
    while a < b:  
        yield a  
        a += 1
```

Like iterators, generators can be used with the “for” loop:

```
>>> for value in myrange(1, 4):  
...     print(value)  
...  
1  
2  
3
```

Under the hood, generators behave similarly to iterators:

```
>>> seq = myrange(1,3)  
>>> next(seq)  
1  
>>> next(seq)  
2  
>>> next(seq)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration
```

The interesting thing about generators is the “yield” keyword. The “yield” keyword works much like the “return” keyword, but unlike “return”, it allows the function to eventually resume its execution. In other words, each time the next value of a generator is needed, Python wakes up the function and resumes its execution from the “yield” line as if the

function had never exited.

Generator functions can use other functions inside. For instance, it is very common to use the “range” function to iterate over a sequence of numbers:

```
def squares(n):  
    for value in range(n):  
        yield value * value
```

## Exercises with generators

1. Implement a generator called “squares” to yield the square of all numbers from (a) to (b). Test it with a “for” loop and print each of the yielded values.
2. Create a generator to yield all the even numbers from 1 to (n).
3. Create another generator to yield all the odd numbers from 1 to (n).
4. Implement a generator that returns all numbers from (n) down to 0.
5. Create a generator to return the fibonnaci sequence starting from the first element up to (n). The first numbers of the sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
6. Implement a generator that returns all consecutive pairs of numbers from 0 to (n), such as (0, 1), (1, 2), (2, 3)...



# Chapter 11

## Coroutines

Python coroutines are similar to generators but instead of producing data, coroutines are mostly used as data consumers. In other words, coroutines are functions that are resumed everytime a value is sent using the `send` method.

The trick with coroutines is the use of the `yield` keyword on the right side of an assignment expression. Here's an example of a coroutine that just prints the values that are sent to it:

```
def coroutine():
    print('My coroutine')
    while True:
        val = yield
        print('Got', val)

>>> co = coroutine()
>>> next(co)
My coroutine
>>> co.send(1)
Got 1
>>> co.send(2)
Got 2
>>> co.send(3)
Got 3
```

The initial call to `next` is required to move the coroutine forward. You can see that it executes the print statement. Eventually, the function reaches the `yield` expression where it will wait to be resumed. Then, everytime a value is sent (with `send`), the coroutine function resumes from the `yield`, copies the value to `val` and prints it.

Coroutines can be closed with the `close()` method.

```
>>> co.close()
>>> co.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

## Exercises with coroutines

1. Create a coroutine named “square” that prints the square of any sent value.
2. Implement the “minimize” coroutine that keeps and prints the minimum value that is sent to the function.

## Pipelines

Coroutines can be used to implement data pipelines where one coroutine will send data to the next coroutine in the pipeline. Coroutines push data into the pipeline using the `send()` method.



Here’s an example of a small pipeline where the values sent to the producer coroutine are squared and sent to the consumer coroutine for printing:

```
def producer(consumer):
    print("Producer ready")
    while True:
        val = yield
        consumer.send(val * val)

def consumer():
    print("Consumer ready")
    while True:
        val = yield
        print('Consumer got', val)
```

As above, coroutines must be “primed” with `next` before any value can be sent.

```
>>> cons = consumer()
>>> prod = producer(cons)
>>> next(prod)
Producer ready
>>> next(cons)
Consumer ready

>>> prod.send(1)
Consumer got 1
>>> prod.send(2)
Consumer got 4
```

```
>>> prod.send(3)
Consumer got 9
```

Also, with coroutines, data can be sent to multiple destinations. The following example implements two consumers where the first only prints numbers in 0..10 and the second only print numbers in 10..20:

```
def producer(consumers):
    print("Producer ready")
    try:
        while True:
            val = yield
            for consumer in consumers:
                consumer.send(val * val)
    except GeneratorExit:
        for consumer in consumers:
            consumer.close()

def consumer(name, low, high):
    print("%s ready" % name)
    try:
        while True:
            val = yield
            if low < val < high:
                print('%s got' % name, val)
    except GeneratorExit:
        print("%s closed" % name)
```

As before, coroutines must be “primed” before any value can be sent.

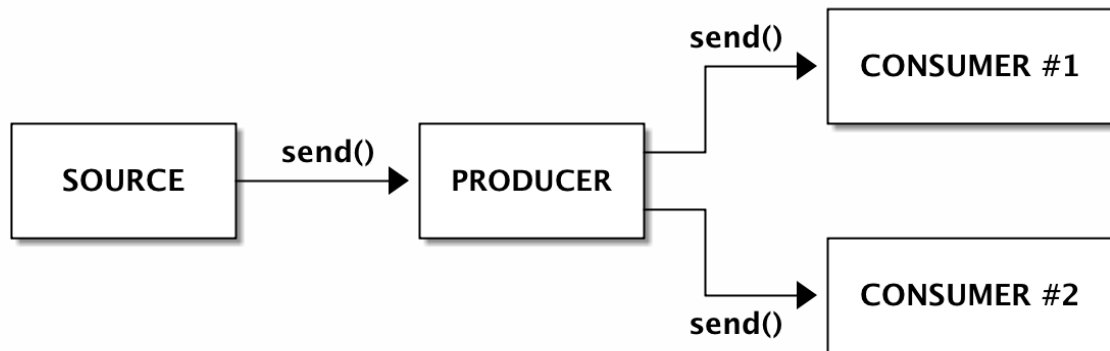
```
>>> con1 = consumer('Consumer 1', 00, 10)
>>> con2 = consumer('Consumer 2', 10, 20)
>>> prod = producer([con1, con2])

>>> next(prod)
Producer ready
>>> next(con1)
Consumer 1 ready
>>> next(con2)
Consumer 2 ready

>>> prod.send(1)
Consumer 1 got 1
>>> prod.send(2)
Consumer 1 got 4
>>> prod.send(3)
Consumer 1 got 9
>>> prod.send(4)
Consumer 2 got 16
```

```
>>> prod.close()
Consumer 1 closed
Consumer 2 closed
```

The data is sent to all consumers but only the second executes the print statement. Notice the use of the `GeneratorExit` exception. Sometimes it can be useful to catch the exception and inform the downstream coroutines that the pipeline is no longer useful.



## Exercises with coroutine pipelines

1. Implement a producer-consumer pipeline where the values squared by the producer are sent to two consumers. One should store and print the minimum value sent so far and the other the maximum value.
2. Implement a producer-consumer pipeline where the values squared by the producer are dispatched to two consumers, one at a time. The first value should be sent to consumer 1, the second value to consumer 2, third value to consumer 1 again, and so on. Closing the producer should force the consumers to print a list with the numbers that each one obtained.

# Chapter 12

## Asynchronous programming

So far we have been doing *synchronous programming*. Synchronous program execution is quite simple: a program starts at the first line, then each line is executed until the program reaches the end. Each time a function is called, the program waits for the function to return before continuing to the next line.

In asynchronous programming, the execution of a function is usually non-blocking. In other words, each time you call a function it returns immediately. However, that function does not necessarily gets executed right way. Instead, there is usually a mechanism (called the “scheduler”) which is responsible for the future execution of the function.

The problem with asynchronous programming is that a program may end before any asynchronous function starts. A common solution for this is for asynchronous functions to return “futures” or “promises”. These are objects that represent the state of execution of an async function. Finally, asynchronous programming frameworks typically have mechanisms to block or wait for those async functions to end based on those “future” objects.

Since Python 3.6, the “asyncio” module combined with the *async* and *await* keyword allows us to implement what is called *co-operative multitasking programs*. In this type of programming, a coroutine function voluntarily yields control to another coroutine function when idle or when waiting for some input.

Consider the following asynchronous function that squares a number and sleeps for one second before returning. Asynchronous functions are declared with **async def**. Ignore the **await** keyword for now:

```
import asyncio

async def square(x):
    print('Square', x)
    await asyncio.sleep(1)
    print('End square', x)
    return x * x

# Create event loop
loop = asyncio.get_event_loop()
```

```
# Run async function and wait for completion
results = loop.run_until_complete(square(1))
print(results)

# Close the loop
loop.close()
```

The event loop (<https://docs.python.org/3/library/asyncio-eventloop.html>) is, among other things, the Python mechanism that schedules the execution of asynchronous functions. We use the loop to run the function until completion. This is a synchronizing mechanism that makes sure the next print statement doesn't execute until we have some results.

The previous example is not a good example of asynchronous programming because we don't need that much complexity to execute only one function. However, imagine that you would need to execute the `square(x)` function three times, like this:

```
square(1)
square(2)
square(3)
```

Since the `square()` function has a sleep function inside, the total execution time of this program would be 3 seconds. However, given that the computer is going to be idle for a full second each time the function is executed, why can't we start the next call while the previous is sleeping? Here's how we do it:

```
# Run async function and wait for completion
results = loop.run_until_complete(asyncio.gather(
    square(1),
    square(2),
    square(3)
))
print(results)
```

Basically, we use `asyncio.gather(*tasks)` to inform the loop to wait for all tasks to finish. Since the coroutines will start at almost the same time, the program will run for only 1 second. Asyncio **gather()** won't necessarily run the coroutines by order although it will return an ordered list of results.

```
$ python3 python_async.py
Square 2
Square 1
Square 3
End square 2
End square 1
End square 3
[1, 4, 9]
```

Sometimes results may be needed as soon as they are available. For that we can use a second coroutine that deals with each result using `asyncio.as_completed()`:

```
(...)

async def when_done(tasks):
    for res in asyncio.as_completed(tasks):
        print('Result:', await res)

loop = asyncio.get_event_loop()
loop.run_until_complete(when_done([
    square(1),
    square(2),
    square(3)
]))
```

This will print something like:

```
Square 2
Square 3
Square 1
End square 3
Result: 9
End square 1
Result: 1
End square 2
Result: 4
```

Finally, async coroutines can call **other async coroutine functions** with the **await** keyword:

```
async def compute_square(x):
    await asyncio.sleep(1)
    return x * x

async def square(x):
    print('Square', x)
    res = await compute_square(x)
    print('End square', x)
    return res
```

## Exercises with asyncio

1. Implement an asynchronous coroutine function to add two variables and sleep for the duration of the sum. Use the asyncio loop to call the function with two numbers.
2. Change the previous program to schedule the execution of two calls to the sum function.