

Getting up to speed with Python

João Ventura

v0.2

Contents

1	Introduction	2
2	Installation	3
2.1	Installing on Windows	3
2.2	Installing on macOS	5
2.3	Installing on Linux	5
3	Basic datatypes	7
3.1	Exercises with numbers	8
3.2	Exercises with strings	9
3.3	Exercises with lists	9
4	Modules and functions	10
4.1	Exercises with the math module	11
4.2	Exercises with functions	11
4.3	Recursive functions	11
4.4	Exercises with recursive functions	12
5	Iteration and loops	13
5.1	Exercises with the for loop	15
5.2	Exercises with the while statement	15
6	Dictionaries	16
6.1	Exercises with dictionaries	17
6.2	Exercises with sub-dictionaries	18
7	Classes	19
7.1	Exercises with classes	20
7.2	Class inheritance	20
7.3	Exercises with inheritance	20

Chapter 1

Introduction

This book aims to teach the basics of the Python programming language using a practical approach. Its method is quite basic though: after a very simple introduction to each topic, the reader is invited to learn 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.

This book is divided in the following chapters: in chapter 2 I will provide the basic installation instructions and execution of the Python interpreter. In chapter 3 we will talk about the most basic data types, numbers and strings. In chapter 4 we will start tinkering with functions, and in chapter 5 the topic is about "loops". In chapter 6 we will work with dictionaries and finally, in chapter 7 we will finish the book with some exercises about classes and object oriented programming.

Please note that this book is a work in progress and as such may contain quite a few spelling errors that may be corrected in the future. However it is made available 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.

This book is made available in github (check it at <https://github.com/joaovventura/full-speed-python>) so I welcome any pull requests to correct misspellings, to suggest new exercises or clarification of the current content.

All the best,

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

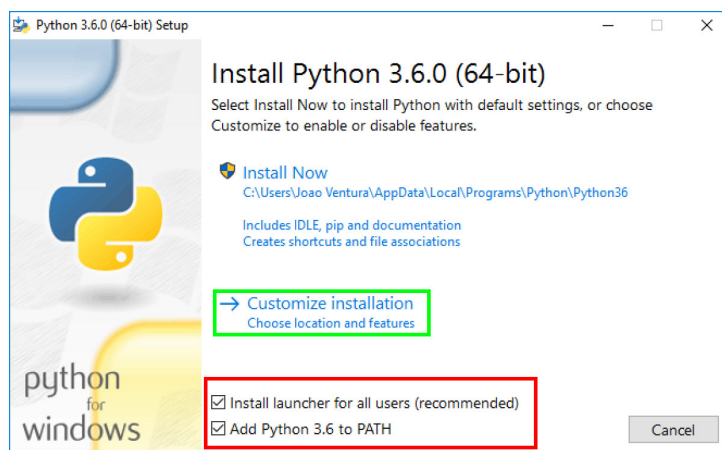
Chapter 2

Installation

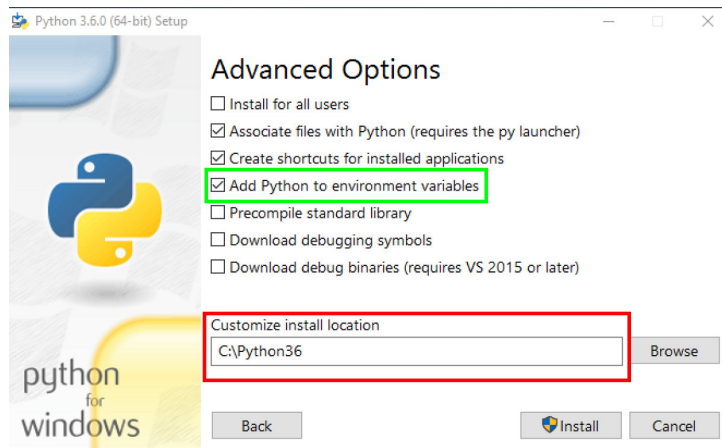
In this chapter we will install and run the Python interpreter in your local computer.

2.1 Installing on Windows

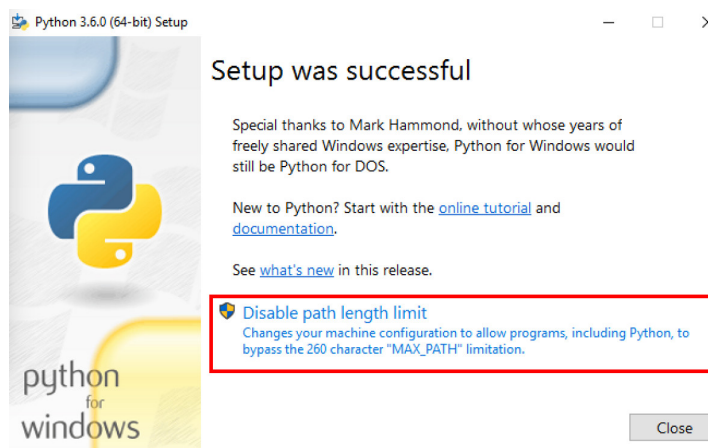
1. Download the latest Python 3 release for Windows on <https://www.python.org/downloads/windows/> and execute the installer. At the time of writing, this is Python 3.6.4.
2. Make sure that the "Install launcher for all users" and "Add Python to PATH" settings are selected and choose "Customize installation".



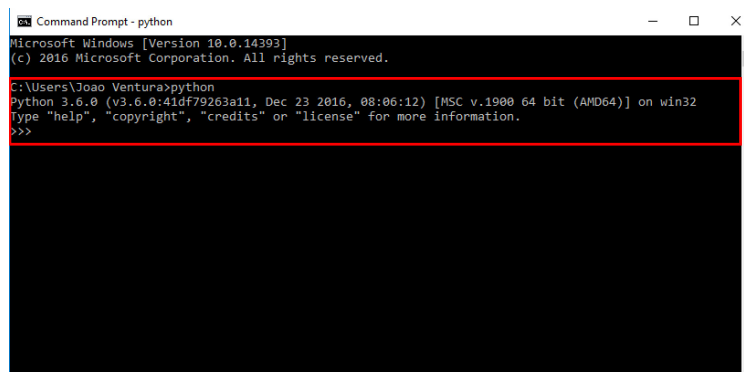
3. In the next screen "Optional Features", you can install everything, but it is essential to install "pip" and "pylauncher (for all users)". Pip is the Python package manager that allows you to install several Python packages and libraries.
4. In the Advanced Options, make sure that you select "Add 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.



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.

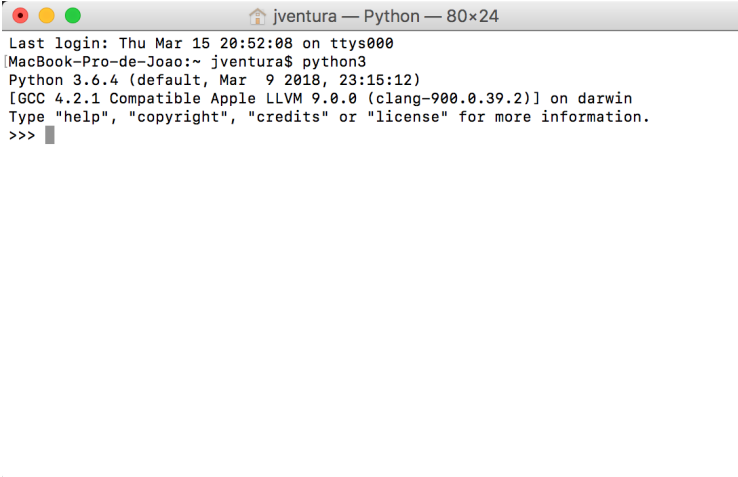


6. Now, open the command line (cmd) and execute "python" or "python3". If everything was correctly installed, you should see the Python REPL. The REPL (from Read, Evaluate, Print and Loop) is an environment that you can use to program small snippets of Python code. Execute `exit()` to exit.



2.2 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 MacPorts package manager (<https://www.macports.org/>) and command "`port install python36`".

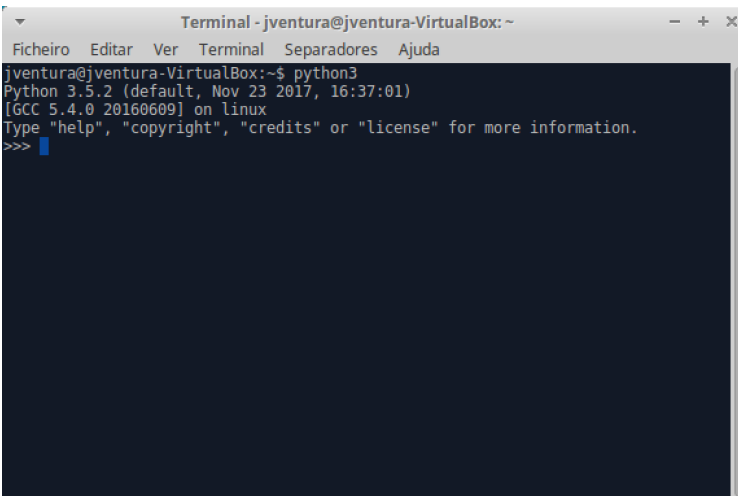
A screenshot of a macOS terminal window titled "jventura — Python — 80x24". The terminal shows the output of running the command `python3`. The output text is: "Last login: Thu Mar 15 20:52:08 on ttys000", "MacBook-Pro-de-Joao:~ jventura\$ python3", "Python 3.6.4 (default, Mar 9 2018, 23:15:12)", "[GCC 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.", and finally the prompt ">>>".

```
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.
>>>
```

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.

2.3 Installing on Linux

For Linux, you can download the latest Python 3 binary releases from <https://www.python.org/downloads/linux/> or use your package manager to install it. To make sure you have Python 3 installed on your system, run *python3* in your terminal.

A screenshot of a Linux terminal window titled "Terminal - jventura@jventura-VirtualBox: ~". The terminal shows the output of running the command `python3`. 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", "Type \"help\", \"copyright\", \"credits\" or \"license\" for more information.", and finally the prompt ">>>".

```
Terminal - jventura@jventura-VirtualBox: ~
Ficheiro Editar Ver Terminal Separadores Ajuda
jventura@jventura-VirtualBox:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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.

Chapter 3

Basic datatypes

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

```
>>> 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 seems to work as repetition:

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

Finally, Python also supports the list datatype. Lists are data structures that allows us to group some values. Lists can have values of several types and you can also mix different types within the same list although usually all values are usually of the same datatype.

Lists are created by starting and ending with square brackets and separated by commas. The values in a list can be accessed by its position where 0 is the index of the first value:

```
>>> 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 using the start and end indexes on the sublist:

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

Finally, there's also some arithmetic that you can do on lists, 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]
```

3.1 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 $\frac{4}{3}\pi r^3$. Calculate the volume of a sphere of radius 5. Suggestion: create a variable named "pi" with the value of 3.1415.

4. Use the module 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.

3.2 Exercises with strings

Using the Python documentation on strings (<https://docs.python.org/3/library/string.html>), solve the following exercises:

1. Initialize the string "abc" on a variable named "s":
 - (a) Use a function to get the length of the string.
 - (b) 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":
 - (a) Use a function that allows you to find the first occurrence of "b" in the string, and the first occurrence of "ccc".
 - (b) 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 find the "replace" function.
 - (a) "AAA BBB CCC"
 - (b) "AAA bbb CCC"

3.3 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].

Chapter 4

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
```

4.1 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).

4.2 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.

4.3 Recursive functions

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

$$f(x) = \begin{cases} 1, & \text{if } x = 0. \\ x \times f(x - 1), & \text{otherwise.} \end{cases} \quad (4.1)$$

As an example, take the factorial of 5:

$$\begin{aligned} 5! &= 5 \times 4! \\ &= 5 \times 4 \times 3! \\ &= 5 \times 4 \times 3 \times 2! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned} \quad (4.2)$$

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.

4.4 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 $fib(n)$.

$$fib(n) = \begin{cases} 0, & \text{if } x = 0. \\ 1, & \text{if } x = 1. \\ fib(n-1) + fib(n-2), & \text{otherwise.} \end{cases} \quad (4.3)$$

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

Chapter 5

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
1
```

Notice that it never prints 0...

5.1 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 keep the last maximum value found so far, you need to keep also 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 by increasing 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.

5.2 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 6

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, keys don't need to be necessarily strings and integers but can be any objects:

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

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

Even more, you can 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.

6.1 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 return 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.

6.2 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 the name of all students which address matches the address in the argument. For instance, invoking "find_students(students, 'Lisbon')" should return Peter and Anna.

Chapter 7

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
```

7.1 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).

7.2 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.

7.3 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.