

Programming environment

1 How to install python

Python is a interpreted language and to run it it is necessary to install the execution environment and supporting libraries. The way to install the Python environment depends on the Operating Systems (Windows, Linux, MAC OS X).

To develop python applications it is best to use an Integrated Development Environment.

In this course we suggest students to use the Microsoft Visual Studio Code (<https://code.visualstudio.com/>).

The way to install Visual Studio Code is described in

- <https://code.visualstudio.com/docs/setup/setup-overview>

Visual Studio Code is a generic IDE, and as such it is necessary to install the Python extension.

This extension allows the IDE to syntax highlight the code, suggest auto completion, and debug the applications, among other functionalities. The way to setup the python programming environment (including installing python) is described in:

- <https://code.visualstudio.com/docs/python/python-tutorial>

Do not proceed until you concluded the Visual Studio Code Tutorial:

<https://code.visualstudio.com/docs/python/python-tutorial>

<https://code.visualstudio.com/docs/python/python-tutorial>

<https://docs.python.org/3/using/index.html>

2 How to run python

Python can be executed in two different ways in interactive and batch mode.

Most of the examples and exercises in this laboratory, can be executed in interactive or batch mode.

In interactive mode, each line should be typed individually, and if input from the user is required, only after such input the program continues and more python code lines can be typed.

In batch mode, the student creates a python files, writes all the code and executes it inside the IDE.

2.1 Interactive use

In interactive mode the interpreter receives command (python instructions) that are executed as the user/programmer types them.

In the command line do:

```
$ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print ("Hello world")
Hello world
>>> a = 12
>>> print (a*2)
24
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
$
```


In the Visual Studio Code it is also possible to experiment inside a Python execution environment:

- Press **Ctrl-Shift-p**
- Type **python Interactive**
- Select **Python: Create python Interactive Window**
- Type python instructions in the bottom of the window

2.2 Batch mode

In batch mode the python interpreter reads a whole python file and executes it as a regular compiled application:

```
$ python example.py  
Hello World  
24  
$
```

In Visual Studio Code when editing a python file if the programmer presses the Ctrl-F5 key or the button , the IDE starts a python interpreter, that reads the current files and executes it. The results of the execution are printed in the bottom of the window (Terminal Pane).

<https://docs.python.org/3/tutorial/interpreter.html>

<https://code.visualstudio.com/docs/introvideos/basics>

3 Debugging

The Visual Studio Code allows debugging of python application from within it. It allows the regular execution of applications with the added functionalities.

- If the application crashed it is possible to evaluate the state of the application
- it is possible to insert breakpoints
- it is possible to execute the application step by step
- it is possible to verify and change the values of variables
- it is possible to execute python expressions that will change the state of the applications

In order to debug python applications it is necessary to first initialize the environment. To configure the visual Studio code workspace, follow the initial steps on

- <https://code.visualstudio.com/docs/python/debugging>

After the configuration just follow the generic debugging instructions:

- <https://code.visualstudio.com/docs/editor/debugging>

4 Collaborative code editing

<https://docs.microsoft.com/en-us/visualstudio/liveshare/use/vscode>

<https://docs.microsoft.com/en-us/visualstudio/liveshare/quickstart/share>

<https://docs.microsoft.com/en-us/visualstudio/liveshare/quickstart/join>

<https://docs.microsoft.com/en-us/visualstudio/liveshare/quickstart/browser-join>

The Visual Studio Code offers a set of tools that allow multiple user to interact in the same workspace and files.

This allows remote users (in different computers) to control the same workspace:

- All user see the same interface
- All user can change in real time the sema file.
- All user can execute the applications and intercar with them

To activate this functionality it is necessary to install the Visual studio live share.

Follow the instructions in:

- <https://docs.microsoft.com/en-us/visualstudio/liveshare/use/vscode>

You will be required to sign in. You can use your TEAMS/Técnico account to do so.

After sharing a session, the remote partner can access your work from his installed Visua Studio Code or even from the browser:

- <https://docs.microsoft.com/en-us/visualstudio/liveshare/quickstart/join>
- <https://docs.microsoft.com/en-us/visualstudio/liveshare/quickstart/browser-join>

Now all participants can cooperate on the same workspace in real time.

If necessary, an audio connection can be established to help partners to communicate.

5 Documentation and tutorials

Python Documentation

- <https://docs.python.org/3/>

The Python Tutorial

- <https://docs.python.org/3/tutorial/index.html>

“Dive into Python” (Chapters 2 to 4)

- <https://diveinto.org/python3/table-of-contents.html>

The Python Language Reference

- <https://docs.python.org/3/reference/index.html>

The Python Standard Library

- <https://docs.python.org/3/library/index.html>

Visual Studio Code

- <https://code.visualstudio.com/docs>
- <https://code.visualstudio.com/docs/python/python-tutorial>
- <https://code.visualstudio.com/docs/python/debugging>

Visual Studio Live Share

- <https://visualstudio.microsoft.com/services/live-share/>
- <https://docs.microsoft.com/en-us/visualstudio/liveshare/>
- <https://docs.microsoft.com/en-us/visualstudio/liveshare/use/vscode>

Introduction to python

6 Sample of code...

Here is the first example of a Python code.

```
x = 34 - 23          # A comment.  
y = "Hello"        # Another one.  
z = 3.45  
if z == 3.45 or y == "Hello":  
    x = x + 1  
    y = y + " World" # String concat.  
print (x)  
print (y)
```

Some of the lines are familiar to anyone that already programs, but there are some subtle differences

Copy and paste this code to a Visual Studio file and try to execute it.

We'll now start to see the specific characteristics of the Python programming language.

7 Basics to Understand the Python Code

<https://docs.python.org/3/tutorial/introduction.html>

7.1 Basic language rules

Assignment uses `=`

- `z = 3.45` assigns the value 3.45 to the variable z

comparison uses `==`

- `z == 3.45` compares the variable z with the value 3.45 (it is either true or false)

For numbers `+-*/%` work as expected

- in strings `+` is used for string concatenation
- in strings `%` is used for string formatting.

The basic printing command is “print.”

- `print (x)`

Logical operators are words (**and**, **or**, **not**)

- not symbols (`&&`, `||`, `!`).

First assignment to a variable will create it.

- Variable types don't need to be declared.
- Python figures out the variable types from the type of the assignment value

A variable can

- Change value
- Change type !!!

7.2 Basic Datatypes

Integers

- `z = 11 / 2` # result is 5, integer division.

Floats

- `x = 3.456`
- The operations are similar to those of C

Strings

- Can use `"xxx"` or `'xxx'` to specify them
- `"abc"` `'abc'` (Same thing.)
- Unmatched ones can occur within the string. `"matt's"`
- Use triple double-quotes for multi-line strings or strings than contain both `'` and `"` inside of them: `"""a' ' b " c"""`

7.3 Newlines, blocks and spaces

Use a newline to end a line of code.

- Not a semicolon like in C++ or Java.
- Use `\` when to continue on next line

No braces `{ }` to mark blocks of code in Python...

- Use consistent indentation instead
- The first line with a new indentation is considered outside of the block.

```
Out of block
```

```
    block 1
```

```
    block 1
```

```
        block 2
```

```
    block 1
```

```
Out of block
```

If the first line of the block used spaces or tabs to indent it

- all lines of that block should use the same characters

7.4 Comments

Comments start with #

- the rest of line is ignored.

8 Python and Types

Python determines the data types in a program at run time.

- Dynamic Typing
- Not necessary for the programmer to declare it

But it enforces them after defined

- Strong Typing
- For instance, you can't just append an integer to a string.
 - `print("number" + 12) # error!!!`
- You must first convert the integer to a string itself.
 - `print("number" + str(12)) # ok!!!`

```
x = "the answer is " # Decides x is string.
y = 23                # Decides y is integer.
print x + y          # Python will complain about this.
print (y+y)          # prints 46
print (x+"srt")      # prints the answer is 23
```

9 Naming Rules

Names are case sensitive and cannot start with a number.

- Used in variables, classes, functions, modules

They can contain letters, numbers, and underscores.

- **bob Bob _bob _2_bob_ bob_2 BoB**

There are some reserved words:

- **and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while**

9.1 Accessing Non-existent Name

If you try to access a name/variable before it's been properly created

- you'll get an error.

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

Names are created when

- Creating a new variables (first assignment)
- implementing a function
- Declaring a class

```
>>> x = 12
>>> def f():
...     print(14)
...
>>> x
```

```
12
>>> f
<function f at 0x7f3ae1181430>
>>> type(x)
<class 'int'>
>>> type(f)
<class 'function'>
>>>
```

9.2 Multiple Assignment

You can assign multiple values to multiple variables in just one code line

```
>>> a,b = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot unpack non-iterable int object
>>> a,b = 2, 3
>>> a
2
>>> b
3
>>>
```

10 String Operations

<https://docs.python.org/3/library/stdtypes.html#textseq>

<https://docs.python.org/3/tutorial/inputoutput.html>

<https://pyformat.info/>

10.1 Strings are objects

We can use some methods built-in to the string some operations:

```
>>> "hello".upper()  
'HELLO'  
>>>
```

There are many other handy string operations available.

str(Object)

- returns a String representation of the Object
- converts any object to string

```
>>> str(1)  
'1'  
>>> str(1.23)  
'1.23'  
>>> str(str)  
<class 'str'>  
>>> str([])  
'[]'  
>>>
```

10.2 Printing with Python

You can print a string to the screen using **print(args)**

print adds a newline to the end of the output

A list of strings will be concatenated them with a space between them:

```
>>> print(12, "xx", [])
```



```
12 xx []
>>>
```

10.3 String formatting

The % emulates the **printf** functionalities.

The % operator can be applied to strings and replaces any %d %f %s placeholder by a value on a list

The % operator can be used in combination with the **print** command to format the output text

```
>>> "%d %d" % (12, 13)
'12 13'
>>> print ("%s xyz %d" % ("abc", 34) )
abc xyz 34
>>> "%d %f" % (12, 13)
'12 13.000000'
>>>
```

The **format** method replaces the % operator and is applied to a string containing the {} placeholder:

```
>>> s = '{} {}'.format('one', 'two')
>>> s
'one two'
>>>
```

More information in <https://pyformat.info/>

11 Input in Python 3

<https://docs.python.org/3/tutorial/inputoutput.html>

The `input(string)` method returns a line of user input as a string

The parameter is used as a prompt

```
>> s = input()
ola
>>> s
'ola'
>>> ss = input("Please type something: ")
Please type something: hello
>>> ss
'hello'
>>>
```

The string can be converted by using the conversion methods:

- `int(string)`

```
>>> n = int('12')
>>> n
12
>>> type(n)
<class 'int'>
>>> k = int("12.2")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.2'
```

- `float(string)`

```
>>> x = float(12)
>>> x
12.0
>>> type(x)
<class 'float'>
```

```
>>> y = float(12E-1)
>>> y
1.2
>>>
```

- `bool(string)`

```
>>> b1 = bool(0)
>>> b1
False
>>> b2 = bool(12)
>>> b2
True
>>> b3 = bool(True)
>>> b3
True
>>> b4 = boot(false)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'boot' is not defined
>>> b4 = bool("false")
>>> b4
True
>>> b5 = bool("False")
>>> b5
True
>>> b6 = bool("")
>>> b6
False
>>>
```

11.1 Input Example

```
name = input("What's your name? > ")
birthyear = int(input("What year were you born? > "))
"Hi {}! You are {} years old!".format(12, 12)
```

12 Exercise 1

Implement a program that reads two numbers from the keyboard and calculates their average

Boolean and control structures

13 Booleans

0, **None** and **""** are false

Everything else is **True**

True and **False** are aliases for **1** and **0** respectively

Experiment:

```
>>> True+True
```

The basic Boolean operator are **and or not**

13.1 Boolean Expressions

Compound Boolean expressions short circuit

and or return one of the elements in the expression

```
>>> 12 or True
```

```
12
```

```
>>> 12 and 13
```

```
13
```

```
>>>
```

When None is returned the interpreter does not print anything

```
>>> 12 and None
```

```
>>> 12 and ""
```

```
''
```

```
>>> 12 and False
```

```
False
```

```
>>>
```

14 Control structures

<https://docs.python.org/3/tutorial/controlflow.html> (4.1 ... 4.4, 4.8)

https://docs.python.org/3/reference/compound_stmts.html

14.1 Code Blocks / No braces

Python uses indentation instead of braces to define the code blocks

All lines in the same block must be indented the same amount to be part of the scope

- if indented more then become part of an inner scope / inner blocks
- if indented less, then they must follow the indentation rules of the outer block

This forces the programmer to use proper indentation since the indenting is part of the program!

New blocks can only be started on certain instructions (such as **if**)

```
>>> a = 12
>>>     print(a)
File "<stdin>", line 1
    print(a)
    ^
IndentationError: unexpected indent
>>> if a > 13:
...     print(a)
...
>>>
```

14.2 Boolean expressions in control structures

The evaluated expression do not need to be inside parentheses.

Boolean expressions are terminated by :

The block starts in the line after the :

14.3 If Statements

Has the usual **if / else** parts, but also contains the **elif**, allowing multiple expressions to be evaluated and control the execution.

Only one block (**if, elif else**) is executed.

```
import math
x= int(input())
print ( "x = {} is ".format(x))
if x < 0 :
    print("lower than zero")
elif x > 0  :
    print("higher than zero")
else :
    print("zero")
```

14.4 While loops

Uses a Boolean expression to control the iterations. Just like regular C whiles

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

14.5 For Loops

Is different from the C for.

Python for does not evaluate an expression every iteration.

It repeats the block to each one of the values on a list

The python for receives a list of values.

The length of the list defines the number of iterations

```
for x in [1,7,13,2]:
    print x
for x in '[1,7,13,2]':
    print x
```


The range function allows the creation of lists of values.

Experiment:

```
for x in range (12):
    print(x)
for x in range (3, 12):
    print(x)
for x in range (3, 12, 3):
    print(x)
for x in range (20,12, -2):
    print(x)
```

14.6 12.5 Loop Control Statements (else, continue break)

There are various control statements that can be applied to loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

The continue statement, also borrowed from C, continues with the next iteration of the loop

When applied to a loop, the optional else clause runs only if the loop exits normally (not by break)

The Loop Else Clause

```
• x = 1
• while x < 3 :
•     print x
•     x = x + 1
• else:
•     print 'hello'
```

15 Exercise 2

Implement a program that reads 20 positive numbers from the keyboard and calculates their average

Files and exceptions

16 Files

<https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>

16.1 Input

To read a file, first it is necessary to open it (as in C).

The open function return a file object that has a series of methods to access its contents:

- **read** – returns a string with all the file content
- **readline** – returns a single text line. If called various times, each time the method call it the following text line in the file, starting in the first one. After reading the last line, **readline** returns an empty string
- **readlines** – returns an list with all the file text lines. The for can be used to execute some code to each line.
- **close** – closes the file

```
f = open('data', 'r')
v1 = f.readlines()
f.close()
for s in v1:
    print s
f = open('data', 'r')
s1 = f.read()
f.close()
print s1
```

It is also possible to use the for directly to the open file, when in each iteration of the for one line is processed:

```
f = open('data', 'r')
for line in f:
    print(line, end='')
f.close()
```

16.2 Files: Output

To write to a file it is necessary to open the file for writing:

```
f = open('data', 'w')
```

after writing everything it is necessary to close the file.

The methods to write text to the file are:

- **write** – method that write one line to the file. Returns the number of characters written
- **writeline** – method that recives a list of strings and writes them to the file

```
f = open('workfile', 'w')  
f.write('This is a test\n')  
f.close()
```

How to read files in other formats

<https://docs.python.org/3/library/fileformats.html>

How to access the file system

<https://docs.python.org/3/library/filesys.html>

17 Exception

<https://docs.python.org/3/tutorial/errors.html>

<https://docs.python.org/3/library/exceptions.html>

Whenever a method or function does not run as expected an exception is raised.

During the execution of a program if an exception is not caught and handled, the program terminates.

Review the code samples from this document and programs that you executed and list the various exceptions that were generated and crashed the executions.

Fill the following table with the name/exceptions that occurred:

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9

Chapter 12

Chapter 14

To catch such exception and allow the execution of recovery code it is necessary to use the **try/except** instructions:

- the **try** instruction defines a block that executes regular. Any exception that occurs inside such block will be caught and correctly handled if a **except** instruction exist

- The **except** instructions appear after the try block and defines what exception is handled and what code is executed if such exception occurs

```
try:
    file = open('data')
except IOError:
    print "Oops! That file does not exist..."
```

After the execution of the except block the code continues executing normally.

If the exception name is omitted then any exception is caught

```
try:
    ...
except :
    print "Oops! Something happened :/"
```

If multiple exception are to be handled the except instruction can receive various exception names. In this case the same code will be executed for all the exceptions.

```
try:
    ...
except (RuntimeError, TypeError, NameError):
    print("one of those exceptions happened")
```

If for different exceptions the programmer wants to execute different code then, multiple except instructions can be used

```
try:
    ...
except OSError:
    print("OS error")
except ValueError:
    print("Could not convert data to an integer.")
except:
```

```
print("Unexpected error:")
```


18Exercise 3

Implement a program that reads a file containing one number per line

- Prints all the values on the screen
- calculates the average

The first version of the program should work for files with just one number per line.

The second version of the file should read files that can have lines with text:

- If the line just contain one number, then it is processed
- if it contains other text, such line should be ignored.

Implement these programs guaranteeing that all exception are caught and correctly handled.

Complex datatypes

19 Sequence types

<https://docs.python.org/3/library/stdtypes.html#typesseq>

The three basic sequence types are lists, tuples, and range objects.

These datatype allows the storage of ordered sets of data and offer a common set of operations:

Operation	Result
<code>x in s</code>	True if an item of <code>s</code> is equal to <code>x</code> , else False
<code>x not in s</code>	False if an item of <code>s</code> is equal to <code>x</code> , else True
<code>s + t</code>	the concatenation of <code>s</code> and <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to adding <code>s</code> to itself <code>n</code> times
<code>s[i]</code>	<code>i</code> th item of <code>s</code> , origin 0
<code>s[i:j]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code>
<code>s[i:j:k]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code>
<code>len(s)</code>	length of <code>s</code>
<code>min(s)</code>	smallest item of <code>s</code>
<code>max(s)</code>	largest item of <code>s</code>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)
<code>s.count(x)</code>	total number of occurrences of <code>x</code> in <code>s</code>

19.1 Lists

<https://docs.python.org/3/library/stdtypes.html#list>

<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

Ordered collection of data

Data can be of different types

Lists are mutable, elements can be added, removed or replaced.

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a], [a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

Lists have most of mutable sequence methods:

<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code>)
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code>)
<code>s.extend(t)</code> or <code>s += t</code>	extends <i>s</i> with the contents of <i>t</i> (for the most part the same as <code>s[len(s):len(s)] = t</code>)
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code>)
<code>s.pop([i])</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i]</code> is equal to <i>x</i>

s.reverse() reverses the items of s in place

```
>>> x = [1, 'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

19.2 Tuples

<https://docs.python.org/3/library/stdtypes.html#tuple>

<https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as allowing storage in a set or dict instance).

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: **()**
- Using a trailing comma for a singleton tuple: **a,** or **(a,)**
- Separating items with commas: **a, b, c** or **(a, b, c)**

- Using the **tuple()** built-in: **tuple()** or `tuple(iterable)`

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2, )
>>>
```

19.3 Ranges

<https://docs.python.org/3/library/stdtypes.html#range>

<https://docs.python.org/3/tutorial/controlflow.html#the-range-function>

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

Ranges are created using the **range()** function:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

19.4 Strings

<https://docs.python.org/3/library/stdtypes.html#str>

Textual data in Python is handled with **str** objects, or strings. Strings are immutable sequences of Unicode code points.

String literals are written in a variety of ways:

- Single quotes: **'allows embedded "double" quotes'**
- Double quotes: **"allows embedded 'single' quotes"**.
- Triple quoted: **'''Three single quotes''', """Three double quotes"""**

Strings implement all of the common sequence operations, along with the additional methods that create new strings. List of available string methods in

- <https://docs.python.org/3/library/stdtypes.html#str>

20 Exercise 4

Modify the program from **exercise 3** to calculate the standard deviation of the numbers available in a file.

Read the file once and store the numbers in a new list.

Use such list to calculate the average and standard deviation.

21 Sets

<https://docs.python.org/3/library/stdtypes.html#set>
<https://docs.python.org/3/tutorial/datastructures.html#sets>

A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

The **set** type is mutable.

The contents can be changed using methods like **add()** and **remove()**.

Non-empty sets can be created by placing a comma-separated list of elements within braces.

```
>>> sss = {12, 13}
>>> sss
{12, 13}
>>> sss.add(12)
>>> sss
{12, 13}
>>>
```

Empty sets can be created with the `set()` function.

```
>>> sss = set()
>>> sss.add(14)
>>> sss.add(13)
>>> sss.add(14)
>>> sss
{13, 14}
>>>
```

Like other collections, sets support

- **x in set**

- `len(set)`
- `for x in set.`

Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

22 Exercise 5

Implement a program that reads a text file and show at the end of the execution the characters that appear on such file.

23 Mapping Types — dict

<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary.

Dictionaries can be created

- by placing a comma-separated list of key: value pairs within braces, for example:
`{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`,
- by the `dict` constructor.

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

The access to the stored values is done using `[]`

```
>>> a = dict(one=1, two=2, three=3)
>>> a['one']
1
>>>
```

Entries can be modified using the assignment operator

```
>>> a['one']
1
```

```
>>> a['one']="um"
>>> a
{'one': 'um', 'two': 2, 'three': 3}
>>> a['one']
'um'
>>>
```

Assigning to a key that does not exist adds an entry

```
>>> a['four'] = 4
>>> a
{'one': 'um', 'two': 2, 'three': 3, 'four': 4}
>>>
```

The **del** method deletes an element from a dictionary

```
>>> del(a['two'])
>>> a
{'one': 'um', 'three': 3, 'four': 4}
>>>
```

to verify is a key is present in the dictionary use **key in d**

```
>>> 'one' in a
True
>>> 'two' in a
False
>>>
```

Besides accessing individual the elements stored in the dictionary, it is possible to access the all keys and all the values:

- **d.items()** - Return a new view of the dictionary's items ((key, value) pairs).
- **d.keys()** - Return a new view of the dictionary's keys.

- **d.values()** - Return a new view of the dictionary's values.

All these views can be used to iterate over a dictionary:

```
>>> for p in a.items():
...     print (p)
...
('one', 'um')
('three', 3)
('four', 4)
>>>
```

```
>>> for p in a.items():
...     print (p[0])
...
one
three
four
>>>
```

```
>>> for k in a.keys():
...     print (k)
...
one
three
four
>>>
```

```
>>> for k in a:
...     print (k)
...
one
three
four
>>>
```

```
>>> for p in a.items():
...     print (p[1])
...
um
3
4
>>>
```

```
>>> for v in a.values():
...     print (v)
...
um
3
4
>>>
```

24 Exercise 6

Implement a program that reads a text file and show at the end of the execution the number of times each character appeared