Python Introduction IMCBio

Valentine Gilbart

2025 - 03 - 20

Table of contents

1	Python introduction 3		
2	Less	son 1 - Introduction, Data types, Operators	4
3	Intr	oduction	5
	3.1	Aim of the class	5
	3.2	Requirements	5
	3.3	What is Python?	6
	3.4	Why use Python?	6
	3.5	How can I program in Python?	7
		3.5.1 Interactive mode	7
		3.5.2 Script mode	8
4	Bas	ic concepts	10
	4.1	Values and variables	10
	4.2	Function calls	11
	4.3	Getting help	13
	4.4	Comment your code	13
5	Hov	v can I represent data?	14
	5.1	Simple data types	14
		5.1.1 Boolean	14
		5.1.2 Numeric	14
		5.1.3 Text	15
	5.2	Data structures	17
		5.2.1 List	17
		5.2.2 Tuple	19
		5.2.3 Set	20
		5.2.4 Dictionary	21
	5.3	Conversion between types	22
6	Hov	v can I manipulate data?	24
	6.1	•	24
		6.1.1 Arithmetic operators	24

		6.1.2	Assignment operators	25
		6.1.3	Comparison operators	25
		6.1.4	Logical operators	27
		6.1.5	Membership operators	27
		6.1.6	Operator precedence	
	6.2	Condi	$tionals \dots \dots \dots \dots \dots \dots$	29
	6.3	Notes	on indentation	31
	6.4	Iterati	ions	32
		6.4.1	For loops	32
		6.4.2	Iterators	34
		6.4.3	While loops	36
		6.4.4		
		6.4.5		
		6.4.6	Exercises	39
7	Con	clusion		41
P۵	feren	cos		42
IVE.	ieren	ces		42
8	Less	on 2 -	Functions, Scientific Packages	43
9	Intro	ductio	on	44
	9.1	Aim o	of the class	44
10	Fund	ction		45
	10.1	Syntar	x	40
		D y II UU	21	
	10.2	-		45
		Docur	mentation	45 46
	10.3	Docur Argun	$egin{array}{llllllllllllllllllllllllllllllllllll$	45 46 46
	$10.3 \\ 10.4$	Docur Argun Outpu	mentation	45 46 46 47
11	10.3 10.4 10.5	Docur Argun Outpu Exerci	mentation	45 46 46 47 49
11	10.3 10.4 10.5	Docur Argun Outpu Exerci	mentation	45 46 46 47 49
11	10.3 10.4 10.5	Docur Argun Outpu Exerci ntific p	mentation	45 46 47 49 50 51
11	10.3 10.4 10.5	Docur Argun Outpu Exerci ntific p Panda 11.1.1	mentation	45 46 47 49 50 51
11	10.3 10.4 10.5	Docur Argun Outpu Exerci ntific p Panda 11.1.1 11.1.2	mentation	45 46 47 49 50 51 51
11	10.3 10.4 10.5	Argun Outpu Exerci ntific p Panda 11.1.1 11.1.2 11.1.3	mentation	45 46 47 49 50 51 54 56
11	10.3 10.4 10.5 Scie e 11.1	Docur Argun Outpu Exerci ntific p Panda 11.1.1 11.1.2 11.1.3 11.1.4	mentation	46 46 47 49 50 51 54 56 57
11	10.3 10.4 10.5 Scie e 11.1	Docur Argun Outpu Exerci ntific p Panda 11.1.1 11.1.2 11.1.3 11.1.4 Matpl	mentation ments it ise coackages as Create pandas data Useful methods Learn More Exercise lotlib	45 46 46 47 49 50 51 54 56 57
11	10.3 10.4 10.5 Scie l 11.1	Argun Outpu Exerci ntific p Panda 11.1.1 11.1.2 11.1.3 11.1.4 Matpl Exerci	mentation	45 46 46 47 49 50 51 51 54 56 57 62

Re	feren	ces	65
13	Less	on 3 - Jupyter Notebook	66
14	14.1	Aim of the class	67 67
ı	Arc	chive 2024	69
16	Less	on 1 - Introduction, Data types, Operators	70
17	Intro	oduction	71
	17.1	Aim of the class	71
	17.2	Requirements	71
	17.3	What is Python?	72
		Why use Python?	72
	17.5	How can I program in Python?	73
		17.5.1 Interactive mode	73
		17.5.2 Script mode	74
18	Basi	c concepts	76
	18.1	Values and variables	76
	18.2	Function calls	77
	18.3	Getting help	79
	18.4	Comment your code	79
19	How	can I represent data?	81
	19.1	Simple data types	81
		19.1.1 Boolean	81
		19.1.2 Numeric	81
		19.1.3 Text	82
	19.2	Data structures	84
		19.2.1 List	84
		19.2.2 Tuple	86
		19.2.3 Set	87
		19.2.4 Dictionary	88
	19.3	Conversion between types	89

20.1 Operators 91 20.1.1 Arithmetic operators 91 20.1.2 Assignment operators 92 20.1.3 Comparison operators 92 20.1.4 Logical operators 94 20.1.5 Membership operators 94 20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24.1 Syntax 112 24.2 Documentation 113 24.3 Arguments 113	:	20.1 Operators	91
20.1.2 Assignment operators 92 20.1.3 Comparison operators 92 20.1.4 Logical operators 94 20.1.5 Membership operators 94 20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			_
20.1.3 Comparison operators 92 20.1.4 Logical operators 94 20.1.5 Membership operators 94 20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.1.1 Arithmetic operators	91
20.1.4 Logical operators 94 20.1.5 Membership operators 94 20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.1.2 Assignment operators	92
20.1.4 Logical operators 94 20.1.5 Membership operators 94 20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.1.3 Comparison operators	92
20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
20.1.6 Operator precedence 95 20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.1.5 Membership operators	94
20.2 Conditionals 96 20.3 Notes on indentation 98 20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
20.4 Iterations 99 20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
20.4.1 For loops 99 20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.3 Notes on indentation	98
20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113	•	20.4 Iterations	99
20.4.2 Iterators 101 20.4.3 While loops 103 20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.4.1 For loops	99
20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
20.4.4 Break statement 104 20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		20.4.3 While loops	103
20.4.5 Continue statement 105 20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		-	
20.4.6 Exercises 106 21 Conclusion 108 References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
References 109 22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113			
22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113	21	Conclusion	108
tific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113	Ref	erences	109
tific Packages 110 23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113	22	esson 2 - Functions From File Handling Scien-	
23 Introduction 111 23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		_	110
23.1 Aim of the class 111 23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113		ine i dekuges	110
23.2 Requirements 111 24 Function 112 24.1 Syntax 112 24.2 Documentation 113	23	ntroduction	111
24 Function 112 24.1 Syntax 112 24.2 Documentation 113		23.1 Aim of the class	111
24.1 Syntax		23.2 Requirements	111
24.1 Syntax			
24.2 Documentation			
24.3 Arguments			
		24.3 Arguments	
24.4 Output		-	
24.5 Exercise		$24.5 Exercise \dots \dots$	116
OF F .: 11 II: 117	25	Excentions Handling	117
25 Excentions Handling 117		· ·	
25.1 Syntax			

26	User-defined input	122
	26.1 input	. 122
	26.2 sys.argv	. 123
	26.3 argparse	. 123
27	File Handling	125
	27.1 Reading	
	27.2 Writting	
	27.3 os module	. 129
	27.4 Regular expression	
	27.5 Exercise	. 132
28	Scientific packages	134
	28.1 Pandas	
	28.1.1 Create pandas data	
	28.1.2 Useful methods	
	28.1.3 Learn More	
	28.1.4 Exercise	
	28.2 Matplotlib	
	28.3 Exercise	
	28.4 More packages	. 142
29	Final tips and resources	144
Re	ferences	145
30	Lesson 3 - Conway's Game of Life	146
31	Introduction	147
	31.1 Aim of the class	. 147
	31.2 Requirements	. 147
32	Conway's Game of Life (basic)	148
	32.1 Instructions	. 148
	32.2 Rules	
	32.3 Functions to create	. 149
	32.4 Optional function	
	32.5 Exemple of input and output	. 150
	32.6 Tips	
33	Conway's Game of Life (advanced)	154
	33.1 Instructions	154

34	Solutions	155
35	References	156

1 Python introduction

2 Lesson 1 - Introduction, Data types, Operators

3 Introduction

3.1 Aim of the class

At the end of this class, you will:

- Be familiar with the Python environment
- Understand the major data types in Python
- Manipulate variables with operators and built-in functions



3.2 Requirements

You need to have a computer, and either:

• install Python 3.0.0 (or above) and install a text editor (Word is not a text editor!).

Note

An IDE (integrated development environment) is an improved text editor. It is a software that provides functionalities like syntax highlighting, auto completion, help, debugger... For example Visual Studio Code (install and learn how to use it with Python), but any other IDE will work.

• have a github account, create a new codespace, and select the Repository vgilbart/python-intro to copy from. This is a free solution up to 60 hours of computing and 15 GB per month.

Figure 3.1: Python logo

3.3 What is Python?

Python is a programming language first released in 1991 and implemented by Guido van Rossum.

It is widely used, with various applications, such as:

- software development
- web development
- data analysis
- ...

It supports different types of programming paradigms (i.e. way of thinking) including the procedural programming paradigm. In this approach, the program moves through a linear series of instructions.

Figure 3.2: Guido van Rossum

```
# Create a string seq
seq = 'ATGAAGGGTCC'
# Call the function len() to retrieve the length of the string
size = len(seq)
# Call the function print() to print a text
print('The sequence has', size, 'bases.')
```

The sequence has 11 bases.

3.4 Why use Python?

- Easy-to-use and easy-to-read syntax
- Large standard library for many applications (pandas for tables, matplotlib for graphs, scikit-learn for machine learning...)
- Interactive mode making it easy to test short snippets of code
- Large community (stackoverflow)

Me when people ask me how I learned programming:



Figure 3.3: Just google (or Chat-GPT/Copilot) it!

3.5 How can I program in Python?

Python is an interpreted language, this means that all scripts written in Python need a software to be run. This software is called an interpreter, which "translate" each line of the code, into instructions that the computer can understand. By extension, the interpreter that is able to read Python scripts is also called Python. So, whenever you want your Python code to run, you give it to the Python interpreter.

3.5.1 Interactive mode

One way to launch the Python interpreter is to type the following, on the command line of a terminal:

python3

Note

You can also try python, /usr/bin/env python3, /usr/bin/python3... There are many ways to call python! You can see where your current python is located by running which python3.

From this, you can start using python interactively, e.g. run:

print("Hello world")

Hello world

To get out of the Python interpreter, type quit() or exit(), followed by enter. Alternatively, on Linux/Mac press [ctrl + d], on Windows press [ctrl + z].

```
(base) MB1-4074-A:~ gilbartv$ python3
Python 3.11.5 (main, Sep 11 2023, 08:31:25) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
>>> quit()
(base) MB1-4074-A:~ gilbartv$
```

Figure 3.4: Interactive mode

3.5.2 Script mode

To run a script, create a folder named script, in which a file named intro.py contains:

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
print("Hello world")
```

and run

```
./script/intro.py
```

You should get the same output as before, that is:

Hello world

The shebang #! followed by the interpreter /usr/bin/env python3 can be put at the beginning of the script in order to ommit calling python3 in command-line. If you don't put it, you will have to run python3 script/intro.py instead of simply ./script/intro.py.

The -*- coding: UTF-8 -*- specify the type of encoding to use. UTF-8 is used by default (which means that this line in the script is not necessary). This accepts characters from all languages. Other valid encoding are available, such as ascii (English characters only).

⚠ Warning

Some common errors can occur at this step:

• bash: script/intro.py: No such file or directory i.e. you are not in the right directory to run the file.

Solution: run ls */ and make sure you can find script/: intro.py, if not go to the correct directory by running cd <insert directory name here>

• bash: script/intro.py: Permission denied i.e. you don't have the right to execute your script.

Solution: run ls -l script/intro.py and make sure you have at least -rwx (read, write, exectute rights) as the first 4 characters, if not run chmod 744 script/intro.py to change your rights.

4 Basic concepts

4.1 Values and variables

You will manipulate values such as integers, characters or dictionaries. These values can be stored in memory using variables. To assign a value to a variable, use the = operator as follow:

```
seq = 'ATGAAGGGTCC'
```

To output the variable value, either type the variable name or use a function like print():

```
seq
```

'ATGAAGGGTCC'

```
print(seq)
```

ATGAAGGGTCC

We can change a variable value by assigning it a new one:

```
seq = seq + 'AAAA' # The + operator can be used to concatenate strings
seq
```

'ATGAAGGGTCCAAAA'

A variable can have a short name (like x and y) or a more descriptive name (seq, motif, genome_file). Rules for Python variable names:

- must start with a letter or the underscore character
- cannot start with a number
- can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- are case-sensitive (seq, Seq and SEQ are three different variables)
- cannot be any of the Python keywords (run help('keywords') to find the list of keywords).

Exercise

Are the following variables names legal?

- 2_sequences
- _sequence
- seq-2
- seq 2

You can try to assign a value to these variable names to be sure of your answer!

4.2 Function calls

A function stores a piece of code that performs a certain task, and that gets run when called. It takes some data as input (parameters that are required or optional), and returns an output (that can be of any type). Some functions are predefined (but we will also learn how to create our own later on).

To run a function, write its name followed by parenthesis. Parameters are added inside the parenthesis as follow:

```
# round(number, ndigits=None)
x = round(number = 5.76543, ndigits = 2)
print(x)
```

5.77

Here the function round() needs as input a numerical value. As an option, one can add the number of decimal places to be used with digits. If an option is not provided, a default value is given. In the case of the option ndigits, None is the default. The function returns a numerical value, that corresponds to the rounded value. This value, just like any other, can be stored in a variable.

To get more information about a function, use the help() function.

i Note

If you provide the parameters in the exact same order as they are defined, you don't have to name them. If you name the parameters you can switch their order. As good practice, put all required parameters first.

```
round(5.76543, 2)

5.77

round(ndigits = 2, number = 5.76543)

5.77
```

In Table 18.1 you will find some basic but useful python functions:

Table 4.1: List of useful Python functions.

Function	Description
print()	Print into the screen the values given in argument.
help()	Execute the built-in help system
<pre>quit() or exit()</pre>	Exit from Python
len()	Return the length of an object
round()	Round a numbers

4.3 Getting help

To get more information about a function or an operator, you can use the help() function. For example, in interactive mode, run help(print) to display the help of the print() function, giving you information about the input and output of this function. If you need information about an operator, you will have to put it into quotes, e.g. help('+')

Prowse the help

If the help is long, press [enter] to get the next line or [space] to get the next 'page' of information. To quit the help, press q.

4.4 Comment your code

Except for the shebang and coding specifications seen before, all things after a hashtag # character will be ignored by the interpreter until the end of the line. This is used to add comments in your code.

Comments are used to:

- explain assumptions
- justify decisions in the code
- expose the problem being solved
- inactivate a line to help debug

5 How can I represent data?

Each programming language has its own set of data types, from the most basics (bool, int, string) to more complex structures (list, tuple, set...).

5.1 Simple data types

5.1.1 Boolean

Booleans represent one of two values: True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9)
```

True

5.1.2 Numeric

Python provides three kinds of numerical type:

- int (\mathbb{Z}) , integers
- float (\mathbb{R}) , real numbers
- complex (C), complex numbers

Python will assign a numerical type automatically.

```
x = 1

y = 2.8

z = 1j + 2 # j is the convention in electrical engineering
```

```
type(x)
```

int

```
type(y)
```

float

```
type(z)
```

complex

5.1.3 Text

String type represents textual data composed of letters, numbers, and symbols. The character string must be expressed between quotes.

```
"""my string"""
"my string"
"my string"
'my string'
```

are all the same thing. The difference with triple quotes is that it allows a string to extend over multiple lines. You can also use single quotes and double quotes freely within the triple quotes.

```
# A multi-line string
my_str = '''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
''''
print(my_str)
```

```
This is a multi-line string. This is the first line. This is the second line.

"What's your name?," I asked.

He said "Bond, James Bond."
```

You can get the number of characters inside a string with len().

```
print(seq)
len(seq)
```

ATGAAGGGTCCAAAA

15

Strings have specific methods (i.e. functions specific to this class of object). Here are a few:

Method	Description
.count()	Returns the number of times
	a specified value occurs in a
	string
<pre>.startswith()</pre>	Returns true if the string
	starts with the specified value
<pre>.endswith()</pre>	Returns true if the string
	ends with the specified value
.find()	Searches the string for a
	specified value and returns
	the position of where it was
	found
.replace()	Returns a string where a
-	specified value is replaced
	with a specified value

They are called like this:

seq.count('A')

7



Tip

To get the help() of the .count() method, you need to run help(str.count).

Exercise

- 1. Check if the sequence seq starts with the codon ATG
- 2. Replace all T into U in seq

5.2 Data structures

Data structures are a collection of data types and/or data structures, organized in some way.

5.2.1 List

List is a collection which is ordered and changeable. It allows duplicate members. They are created using square brackets [].

```
seq = ['ATGAAGGGTCCAAAA', 'AGTCCCCGTATGAT', 'ACCT', 'ACCT']
```

List items are indexed, the first item has index [0], the second item has index [1] etc.

```
seq[1]
```

^{&#}x27;AGTCCCCGTATGAT'

Tip

You can count backwards, with the index [-1] that retrieves the last item.

As a list is changeable, we can change, add, and remove items in a list after it has been created.

```
seq[1] = 'ATG'
seq
```

```
['ATGAAGGGTCCAAAA', 'ATG', 'ACCT', 'ACCT']
```

You can specify a range of indexes by specifying the start (included) and the end (not included) of the range.

```
seq[0:2]
```

['ATGAAGGGTCCAAAA', 'ATG']

Tip

By leaving out the start value, the range will start at the first item:

seq[:2]

['ATGAAGGGTCCAAAA', 'ATG']

Similarly, by leaving out the end value, the range will end at the last item.

Note

Indexes also conveniently work on str types.

```
print(seq[0])
print(seq[0][0:5])
print(seq[0][2])
print(seq[0][-1])

ATGAAGGGTCCAAAA
ATGAA
G
A
```

You can get how many items are in a list with len().

```
len(seq)
```

4

Lists have specific methods. Here are a few:

Method	Description
.append()	Inserts an item at the end
.insert()	Inserts an item at the specified index
<pre>.extend()</pre>	Append elements from another list to the current list
.remove()	Removes the first occurance of a specified item
.pop()	Removes the specified (by default last) index

Exercise

- Create a list 1 = ['AAA', 'AAT', 'AAC'], and add
 AAG at the end, using .append().
- 2. Replace all T into U in the element AAT, using .replace().

5.2.2 Tuple

Tuple is a collection which is ordered and unchangeable. It allows duplicate members. Tuples are written with round brackets ().

my_favorite_amino_acid = ('Y', 'Tyr', 'Tyrosine')

Just like for the list, you can get items with their index. The only difference is that you cannot change a tuple that has been created.

Tuples have specific methods. Here are a few:

Method	Description
.count()	Returns the number of times a specified value occurs
.index()	Searches for a specified value and returns the position of where it was found

Exercise

Try to change the value of the first element of my_favorite_amino_acid and see what happens.

5.2.3 Set

Set is a collection which is unordered and unindexed. It does not allow duplicate members (they will be ignored). Sets are written with curly brackets {}.

Once a set is created, you cannot change its items directly (as they don't have index), but you modify the set by removing and adding items.

Sets have specific methods. Here are a few:

Method	Description
.add()	Adds an element to the set

Method	Description
.difference()	Returns a set containing the difference between two sets
.intersection()	Returns a set containing the intersection between two sets
.union()	Returns a set containing the union of two sets
<pre>.remove() .pop()</pre>	Remove the specified item Removes a random element

```
Exercise

Get the common genes between the following sets:

organism1_genes = {'BRCA1', 'TP53', 'EGFR', 'MYC'}
organism2_genes = {'TP53', 'MYC', 'KRAS', 'BRAF'}
```

5.2.4 Dictionary

Dictionaries are used to store data values in key: value pairs. A dictionary is a collection which is ordered (as of Python >= 3.7), changeable and does not allow duplicates keys. Dictionaries are written with curly brackets {}, with keys and values.

```
organism1_genes = {
    #key: value;
    'BRCA1': 'DNA repair',
    'TP53': 'Tumor suppressor',
    'EGFR': 'Cell growth',
    'MYC': 'Regulation of gene expression'
}
```

Dictionary items can be referred to by using the key name.

```
organism1_genes["BRCA1"]
```

^{&#}x27;DNA repair'

Dictionaries have specific methods. Here are a few:

Method	Description
.items()	Returns a list containing a
	tuple for each key value pair
.keys()	Returns a list containing the
	dictionary's keys
.values()	Returns a list of all the values
	in the dictionary
.pop()	Removes the element with
	the specified key
.get()	Returns the value of the
-	specified key

Exercise

From the dictionary organism1_genes created as example, get the value of the key BRCA1. If the key does not exist, return Unknown by default. Try your code before and after removing the BRCA1 key:value pair. Check the help of get by running help(dict.get).

5.3 Conversion between types

You can get the data type of any object by using the function type(). You can (more or less easily) convert between data types.

Function	Description
bool()	Convert to boolean type
<pre>int(), float()</pre>	Convert between integer or
	float types
complex()	Convert to complex type
str()	Convert to string type
<pre>list(), tuple(), set()</pre>	Convert between list, tuple,
	and set types

Function	Description
dict()	Convert a tuple of order (key, value) into a dictionary type

bool(1)

True

```
int(5.8)
```

5

```
str(1)
```

'1'

```
list({1, 2, 3})
```

[1, 2, 3]

```
set([1, 2, 3, 3])
```

{1, 2, 3}

{'a': 1, 'f': 2, 'g': 3}

6 How can I manipulate data?

In the previous section we have learned how data can be represented in different types and gathered in various data structures. In this section we will see how we can manipulate data in order to do more complex tasks.

6.1 Operators

Operators are used to perform operations on variables and values. We will present a few common ones here.

6.1.1 Arithmetic operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name
+	Addition
-	Substraction
*	Multiplication
/	Division
**	Power



A Warning

Do not use the ^ operator to raise to a power. That is actually the operator for bitwise XOR, which we will not cover.

Python will convert data type according to what is necessary. Thus, when you divide two int you will obtain a float number, if you add a float to an int, you will get a float, ...

```
# Example
2/10
```

0.2

```
Note

+ also conveniently work on str types.

'AC' + 'AT'

'ACAT'
```

6.1.2 Assignment operators

Assignment operators are used to assign values to variables:

Operator	Example as	Same as
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5

Note

The same principle applies to multiplication, division and power, but are less commonly used.

6.1.3 Comparison operators

Comparison operators are used to compare two values:

Operator	Name
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

```
# Example
2 == 1 + 1
```

True

⚠ Warning

You should never use equalty operators (==or !=) with floats or complex values.

```
# Example
2.1 + 3.2 == 5.3
```

False

This is a floating point arithmetic problem seen in other programming languages. It is due to the difficulty of having a fixed number of binary digits (bits) to accurately represent some decimal number. This leads to small rounding errors in calculations.

2.1 + 3.2

5.30000000000001

If you need to use equalty operators, do it with a degree of freedom:

```
tol = 1e-6; abs((2.1 + 3.2) - 5.3) < tol
```

True

6.1.4 Logical operators

Logical operators are used to combine conditional statements:

Operator	Description
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

```
# Example
```

False and False, False and True, True and False, True and True

(False, False, True)

```
# Example
```

False or False, False or True, True or False, True or True

(False, True, True, True)

Example

True or not True

True

6.1.5 Membership operators

Operator	Description
in	Returns True if a sequence with the specified value is
not in	present in the object Returns True if a sequence
not in	with the specified value is not present in the object

```
# Example
'ACCT' in seq
```

False

6.1.6 Operator precedence

Operator precedence describes the order in which operations are performed.

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parenthesis
**	Power
* /	Multiplication, division
+ -	Addition, substraction
==,!=,>,>=,<,<=,is,is	Comparisons, identity, and
not,in,not in,	membership operators
not	Logical NOT
and	AND
or	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Exercise

Try to guess what will output the following expressions:

- 1+1 == 2 and "actg" == "ACTG"
- True or False and True and False
- "Homo sapiens" == "Homo" + "sapiens"
- 'Tumor suppressor' in organism1_genes

Verify with Python.

6.2 Conditionals

Conditionals allows you to make decisions in your code based on certain conditions.

```
if something is true:
    do task a
otherwise:
    do task b
```

The comparison (==, !=, >, >=, <, <=), logical (and, or, not) and membership (in, not in) operators can be used as conditions.

In Python, this is written with an if ... elif ... else statement like so:

```
# Define gene expression levels
gene1_expression = 100
gene2_expression = 50

# Analyze gene expression levels
if gene1_expression > gene2_expression:
   print("Gene 1 has higher expression level.")
elif gene1_expression < gene2_expression:
   print("Gene 2 has higher expression level.")
else:
   print("Gene 1 and Gene 2 have the same expression level.")</pre>
```

Gene 1 has higher expression level.

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition". The following code is equivalent to the one before:

```
# Analyze gene expression levels
if gene1_expression > gene2_expression:
  print("Gene 1 has higher expression level.")
else:
```

```
if gene1_expression < gene2_expression:
   print("Gene 2 has higher expression level.")
else:
   print("Gene 1 and Gene 2 have the same expression level.")</pre>
```

Gene 1 has higher expression level.

```
Exercise
Are these two codes equivalent?
# Code A
if "ATG" in dna_sequence:
 print("Start codon found.")
elif "TAG" in dna_sequence:
 print("Stop codon found.")
else:
  print("No interesting codon not found.")
# Code B
if "ATG" in dna_sequence:
 print("Start codon found.")
  if "TAG" in dna_sequence:
   print("Stop codon found.")
else:
 print("No interesting codon not found.")
```

An if statement cannot be empty, but if for some reason you have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200

if b > a:
   pass
```

6.3 Notes on indentation

Note

Python relies on **indentation** (the spaces at the beginning of the lines).

Indentation is not just for readability. In Python, you use spaces or tabs to indent code blocks. Python uses it to determine the scope of functions, loops, conditional statements, and classes.

Any code that is at the same level of indentation is considered part of the same block. Blocks of code are typically defined by starting a line with a colon (:) and then indenting the following lines.

When you have nested structures like a conditional statement inside another conditional statement, you must further to show the hierarchy. Each level of indentation represents a deeper level of nesting.

It's essential to be consistent with your indentation throughout your code. Mixing tabs and spaces can lead to errors, so it's recommended to choose one and stick with it.

Exercise

Here are three codes, they all are incorrect, can you tell why?

Of course, you can run them and read the error that Python gives!

```
amino_acid_list = ["MET", "ARG", "THR", "GLY"]

if "MET" in amino_acid_list:
   print("Start codon found.")
   if "GLY" in amino_acid_list:
      print("Glycine found.")

else:
print("Start codon not found.")
```

```
dna_sequence = "ATGCTAGCTAGCTAG"

if "ATG" in dna_sequence:
   print("Start codon found.")

if "TAG" in dna_sequence
   print("Stop codon found.")

x = 7

if x > 5:
   print("x is greater than 5")
   if y > 10:
        print("x is greater than 10")
   elif y = 10:
        print("x equals 10")
   else:
        print("x is less than 10")
```

6.4 Iterations

Iteration involves repeating a set of instructions or a block of code multiple times.

There are two types of loops in python, for and while.

Iterating through data structures like lists allows you to access each element individually, making it easier to perform operations on them.

6.4.1 For loops

When using a for loop, you iterate over a sequence of elements, such as a list, tuple, or dictionary.

```
for item in data_structure:
    do task a
```

The loop will execute the indented block of code for each element in the sequence until all elements have been processed. This is particularly useful when you know the number of times you need to iterate.

```
all_codons = [
    'AAA', 'AAC', 'AAG', 'AAT',
    'ACA', 'ACC', 'ACG', 'ACT',
    'AGA', 'AGC', 'AGG', 'AGT',
    'ATA', 'ATC', 'ATG', 'ATT',
    'CAA', 'CAC', 'CAG', 'CAT',
    'CCA', 'CCC', 'CCG', 'CCT',
    'CGA', 'CGC', 'CGG', 'CGT',
    'CTA', 'CTC', 'CTG', 'CTT',
    'GAA', 'GAC', 'GAG', 'GAT',
    'GCA', 'GCC', 'GCG', 'GCT',
    'GGA', 'GGC', 'GGG', 'GGT',
    'GTA', 'GTC', 'GTG', 'GTT',
    'TAA', 'TAC', 'TAG', 'TAT',
    'TCA', 'TCC', 'TCG', 'TCT',
    'TGA', 'TGC', 'TGG', 'TGT',
    'TTA', 'TTC', 'TTG', 'TTT'
]
count = 0
for codon in all codons:
  if codon[1] == 'T':
    count += 1
print(count, 'codons have a T as a second nucleotide.')
```

16 codons have a T as a second nucleotide.

What it does is the following: it processes each element in the list all_codons, called in the following code codon. If the codon has as a second character a T, it adds 1 to a counter (the variable called count).

⚠ Warning

You cannot modify an element of a list that way.

```
for codon in all_codons:
   if 'T' in codon :
      codon = codon.replace('T', 'U')
print(all_codons)
```

['AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'AGT', 'AGA', 'AGC', 'AGG', 'AGT', 'ATA',

This is because all_codons was converted to an iterator in the for statement.

6.4.2 Iterators

An iterator is a special object that gives values in succession.

In the previous example, the iterator returns a copy of the item in a list, not a reference to it. Therefore, the **codon** inside the **for** block is not a view into the original list, and changing it does not do anything.

A way to modify the list would be to use an iterable to access the original data. The range(start, stop) function creates an iterable to count from one integer to another.

```
for i in range(2, 10):
    print(i, end=' ')
```

2 3 4 5 6 7 8 9

We could count from 0 to the size of the list, loop though every element of the list by calling them by their index, and modify them if necessary. That's what the following code does:

```
for i in range(0, len(all_codons)):
   if 'T' in all_codons[i] :
     all_codons[i] = all_codons[i].replace('T', 'U')
print(all_codons)
```

```
['AAA', 'AAC', 'AAG', 'AAU', 'ACA', 'ACC', 'ACG', 'ACU', 'AGA', 'AGC', 'AGG', 'AGU', 'AUA', 'A
```

Another useful function that returns an iterator is enumerate(). It is an iterator that generates pairs of index and value. It is commonly used when you need to access both the index and value of items simultaneously.

```
seq = 'ATGCATGC'

# Print index and identity of bases
for i, base in enumerate(seq):
    print(i, base)
```

O A
1 T
2 G
3 C
4 A
5 T
6 G
7 C

```
# Loop through sequence and print index of G's
for i, base in enumerate(seq):
    if base in 'G':
        print(i, end=' ')
```

2 6

6.4.3 While loops

A while loop continues executing a set of statement as long as a condition is true.

```
while condition is true:
    do task a
```

This type of loop is handy when you're not sure how many iterations you'll need to perform or when you need to repeat a block of code until a certain condition is met.

```
seq = 'TACTCTGTCGATCGTACGTATGCAAGCTGATGCATGATTGACTTCAGTATCGAGCGCAGCA'
start_codon = 'ATG'
# Initialize sequence index
i = 0
# Scan sequence until we hit the start codon
while seq[i:i+3] != start_codon:
  i += 1
# Show the result
print('The start codon begins at index', i)
```

The start codon begins at index 19



Warning

Remember to increment i, or you'll get stuck in a loop.

Actually, the previous code is quite dangerous. You can also get stuck in a loop... if the start_codon does not appear in seq at all.

Indeed, even when you go above the given length of seq, the condition seq[i:i+3] != start_codon will still be true because seq[i:i+3] will output an empty string.



Figure 6.1: Hopefully not you!

```
seq[9999:9999+3]
```

So, once the end of the sequence is reached, the condition seq[i:i+3] != start_codon will always be true, and you'll get stuck in an infinite loop.

```
i Note

To get interrupt a process, press [ctrl + c].
```

6.4.4 Break statement

Iteration stops in a for loop when the iterator is exhausted. It stops in a while loop when the conditional evaluates to False. There is another way to stop iteration: the break keyword. Whenever break is encountered in a for or while loop, the iteration stops and execution continues outside the loop.

Codon not found in sequence.

Note

Also, note that the else statement can be used in for and while loops. In for loops it is executed when the loop is finished. In while loops, it is executed when the condition is no longer true. In both case, the loops need to not encounter a break to enter in the else block.

6.4.5 Continue statement

In addition to the break statement, there is also the continue statement in Python that can be used to alter the flow of iteration in loops. When continue is encountered within a loop, it skips the remaining code inside the loop for the current iteration and moves on to the next iteration.

Here's an example showcasing the continue statement in a loop:

```
# List of DNA sequences
dna_sequences = ['ATGCTAGCTAG', 'ATCGATCGATC', 'ATGGCTAGCTA', 'ATGTAGCTAGC']

# Find sequences starting with a start codon
for sequence in dna_sequences:
    if sequence[:3] != 'ATG': # Check if the sequence does not start with a start codon
        print(f"Sequence '{sequence}' does not start with a start codon. Skipping analysis.")
        continue # Skip further analysis for this sequence
    print(f"Analyzing sequence '{sequence}' for protein coding regions...")
    # Additional analysis code here
else:
    print('All sequences were processed.')

Analyzing sequence 'ATGCTAGCTAG' for protein coding regions...
Sequence 'ATCGATCGATC' does not start with a start codon. Skipping analysis.
```

The continue statement in this example skips the analysis code for sequence that does not start with a start codon.

All sequences were processed.

Analyzing sequence 'ATGGCTAGCTA' for protein coding regions...
Analyzing sequence 'ATGTAGCTAGC' for protein coding regions...

6.4.6 Exercises

Exercise 1

Given a list of DNA sequences, find the first sequence that contains a specific motif 'TATA', print the sequence, and stop the process. If no sequence contains the motif, print a message accordingly. You must use only one for loop. With the input given below, the output should look like this:

```
With the input given below, the output should look like
this:
# List of DNA sequences with a TATA
dna_sequences = [
'ATGCTACAGCTAG',
'ATCGATATAATC', # TATA
'ATGGCTAGCTA',
'ATGTAGCTAGC',
'ATGTAGCTATA' # TATA
for ...
# Your code here
Sequence 'ATCGATATAATC' contains the 'TATA' motif.
# List of DNA sequences without a TATA
dna_sequences = [
'ATGCTACAGCTAG',
'ATCGATACAATC',
'ATGGCTAGCTA',
'ATGTAGCTAGC'
for ...
```

No sequence contains the 'TATA' motif.

Your code here

Exercise 2

Analyze a DNA sequence to count the number of consecutive 'A' nucleotides. You must use only one while loop. With the input given below, the output should look like this:

```
# DNA sequence to analyze
dna_sequence = 'ATGATAAGAGAAAGTAAAAGCGATCGAAAAAA'
while ...
# Your code here
```

Number of consecutive 'A's: 6

7 Conclusion

Congrats! You now know the (very) basics of Python programming.

If you want to keep on practising with simple exercises, you can check out w3schools.

For more biology-related exercises check out pythonforbiologist.org, they have exercises availables in each chapters.

For french speakers, the AFPy (Association Francophone Python) has a learning tool called HackInScience.

Or keep on googling for more python exercises!

To continue your learning journey, follow lesson 2.

References

A python conference organized by the AFPy (Association Francophone Python) is held in Strasbourg in the end of October 2024!

Here are some references and ressources that (greatly) inspired this class:

- Python doc
- \bullet w3schools
- pythonforbiologists
- justinbois's Bootcamp

8 Lesson 2 - Functions, Scientific Packages

```
import os
os.getcwd()
os.listdir('exercise/data')

['sample.csv',
   'my-figure-2.png',
   'example.txt',
   'my-figure.png',
   'some_file.txt',
   'example.fasta']
```

9 Introduction

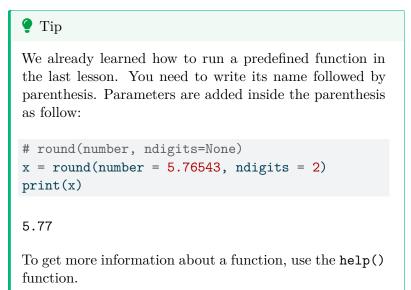
9.1 Aim of the class

At the end of this class, you will be able to:

- Create simple functions
- Handle some errors
- Ask for input from the user
- Upload, modify and download files into Python
- Import packages (and use in a simple manner some scientific packages)

10 Function

A function stores a piece of code that performs a certain task, and that gets run when called. It takes some data as input (parameters that are required or optional), and returns an output (that can be of any type).



We will now learn how to create our own function.

10.1 Syntax

In python, a function is declared with the keyword def followed by its name, and the arguments inside parenthesis. The next block of code, corresponding to the content of the function, must be indented. The output is defined by the return keyword.

```
def hello(name):
    """Presenting myself."""
    presentation = f"Hello, my name is {name}."
    return presentation
```

```
text = hello(name = "Valentine")
print(text)
```

Hello, my name is Valentine.

10.2 Documentation

As you may have noticed, you can also add a description of the function directly after the function definition. It is the message that will be shown when running help(). As it can be along text over multiple lines, it is common to put it inside triple quotes """.

```
help(hello)

Help on function hello in module __main__:
hello(name)
    Presenting myself.
```

10.3 Arguments

You can have several arguments. They can be mandatory or optional. To make them optional, they need to have a default value assigned inside the function definition, like so:

```
def hello(name, french = True):
    """Presenting myself."""
    if french:
        presentation = f"Bonjour, je m'appelle {name}."
    else:
```

```
presentation = f"Hello, my name is {name}."
return presentation
```

The parameter name is mandatory, but french is optional.

```
hello("Valentine")
"Bonjour, je m'appelle Valentine."
hello(french = False)
TypeError: hello() missing 1 required positional argument: 'name'
                                            Traceback (most recent call last)
TypeError
Cell In[8], line 1
----> 1 hello(french = False)
TypeError: hello() missing 1 required positional argument: 'name'
 Note
  Reminder: if you provide the parameters in the exact
  same order as they are defined, you don't have to name
  them. If you name the parameters you can switch their
  order. As good practice, put all required parameters first.
  hello(french = False, name = "Valentine")
  'Hello, my name is Valentine.'
  hello("Valentine", False)
  'Hello, my name is Valentine.'
```

10.4 Output

If no return statement is given, then no output will be returned, but the function will still be run.

```
def hello(name):
    """Presenting myself."""
    print("We are inside the 'hello()' function.")
    presentation = f"Hello, my name is {name}."
```

```
print(hello("Valentine"))
```

We are inside the 'hello()' function. None

The output can be of any type. If you have a lot of things to return, you might want to return a list or a dict for example.

```
def multiple_of_3(list_of_numbers):
    """Returns the number that are multiple of 3."""
    multiples = []
    for num in list_of_numbers:
        if num % 3 == 0:
            multiples.append(num)
    return multiples

multiple_of_3(range(1, 20, 2))
```

[3, 9, 15]

```
i Note
This could be written as a one-liner.

def multiple_of_3(list_of_numbers):
    """Returns the number that are multiple of 3."""
    multiples = [num for num in list_of_numbers if num % 3 == 0]
    return multiples

multiple_of_3(range(1, 20, 2))

[3, 9, 15]
```

10.5 Exercise

Exercise

Write a function called nucl_freq to compute nucleotide frequency of a sequence. Given a sequence as input, it outputs a dictionnary with keys being the nucleotides A, T, C and G, and values being their frequency in the sequence. With the input given below, the output should be:

```
def ...
    # Your code here
nucl_freq("ATTCCCGGGGG")

{'C': 0.3, 'A': 0.1, 'T': 0.2, 'G': 0.4}
```

11 Scientific packages

A python package contains a set of function to perform specific tasks.

A package needs to be **installed** to your computer one time.

You can install a package with pip. It should have been automatically installed with your python, to make sure that you have it you can run:

```
#| eval: false

# In Linux/MacOS

python -m pip --version

# In Windows

py -m pip --version
```

If it does not work, check out pip documentation.

To install a package called pandas, you must run:

```
#| eval: false

# In Linux/MacOS
python -m pip install pandas
# In Windows
py -m pip install pandas
```

To get more information about pip, check out the full documentation.



Installing a package is done outside of the python interpreter, in command line in a terminal.

When you wish to use a package in a python script, you'll need to import it, by writing inside of you script:

```
import pandas
```

11.1 Pandas

Pandas is a package used to work with data sets, in order to easily clean, manipulate, explore and analyze data.

11.1.1 Create pandas data

Pandas provides two types of classes for handling data:

• Series: a one-dimensional labeled array holding data of any type such as integers or strings. It is like a column in a table.

```
# If nothing else is specified, the values are labeled with their index number (starting from
myseries = pandas.Series([1, 7, 2], index = ["x", "y", "z"])
print(myseries)
```

```
x 1
y 7
z 2
dtype: int64
```

• DataFrame: a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns. It is like a table.

```
data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

df = pandas.DataFrame(data)
print(df)
```

```
calories duration
0 420 50
1 380 40
2 390 45
```

You can also create a DataFrame from a file.

```
# Make sure this is the correct path for you! You are in the directory from where you execute
df = pandas.read_csv('exercise/data/sample.csv')
print(df)
```

	Gene	Expression_Level	Tissue
0	${\tt GeneA}$	8.7	Heart
1	${\tt GeneB}$	3.2	Heart
2	${\tt GeneA}$	7.0	Brain
3	${\tt GeneB}$	10.2	Brain
4	${\tt GeneA}$	6.6	Liver
5	GeneB	7.6	Liver

You get access to the index and column names with:

```
df.columns
df.index
```

RangeIndex(start=0, stop=6, step=1)

You can rename index and column names:

```
df = df.rename(index={0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5 : 'f'})
df.index
```

```
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
```

You can select rows:

```
# Select one row by its label
print(df.loc[['a']])
# Select one row by its index
print(df.iloc[[0]])

# Select several rows by labels
print(df.loc[['a','c']])
# Select one row by index
print(df.iloc[[0,2]])
```

```
Gene Expression_Level Tissue
a GeneA
                      8.7 Heart
   Gene Expression_Level Tissue
a GeneA
                      8.7 Heart
   Gene Expression_Level Tissue
a GeneA
                      8.7 Heart
 {\tt GeneA}
                      7.0 Brain
   Gene Expression_Level Tissue
                      8.7 Heart
a GeneA
                      7.0 Brain
c GeneA
```

You can select columns:

```
# Select one column by label
df['Tissue'] # Series
df[['Tissue']] # DataFrame

# Select several columns
df[['Gene', 'Expression_Level']]

# Select several columns by index
df.iloc[:,[0,1]]
```

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.py
return method()

	Gene	Expression_Level
a	GeneA	8.7
b	GeneB	3.2
\mathbf{c}	GeneA	7.0
d	GeneB	10.2
\mathbf{e}	$\operatorname{Gene} A$	6.6
f	GeneB	7.6

You can select rows and columns as follows:

```
df.loc[['b'], ['Gene', 'Expression_Level']]
```

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pg return method()

	Gene	Expression_Level
b	GeneB	3.2

You can filter based on a condition as follows:

```
df[df['Expression_Level'] > 6]
```

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pg return method()

	Gene	Expression_Level	Tissue
a	GeneA	8.7	Heart
\mathbf{c}	GeneA	7.0	Brain
d	GeneB	10.2	Brain
\mathbf{e}	GeneA	6.6	Liver
f	GeneB	7.6	Liver

11.1.2 Useful methods

To explore the data set, use the following methods:

df.info()

<class 'pandas.core.frame.DataFrame'>

Index: 6 entries, a to f

Data columns (total 3 columns):

#	Column	Non-Null Count	Dtype
0	Gene	6 non-null	object
1	Expression_Level	6 non-null	float64
2	Tissue	6 non-null	object

dtypes: float64(1), object(2)
memory usage: 364.0+ bytes

df.describe()

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pg return method()

	Expression_Level
count	6.000000
mean	7.216667
std	2.358319
\min	3.200000
25%	6.700000
50%	7.300000
75%	8.425000
max	10.200000

df.head()

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pg return method()

	Gene	Expression_Level	Tissue
a	GeneA	8.7	Heart
b	GeneB	3.2	Heart
\mathbf{c}	$\operatorname{Gene} A$	7.0	Brain
d	GeneB	10.2	Brain
\mathbf{e}	$\operatorname{Gene}A$	6.6	Liver

```
df.sort_values(by="Gene")
```

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pg return method()

	Gene	Expression_Level	Tissue
a	GeneA	8.7	Heart
\mathbf{c}	$\operatorname{Gene} A$	7.0	Brain
\mathbf{e}	$\operatorname{Gene} A$	6.6	Liver
b	GeneB	3.2	Heart
d	GeneB	10.2	Brain
f	GeneB	7.6	Liver

```
df['Expression_Level'].mean()

df.groupby("Gene")[['Expression_Level']].mean()
```

/opt/hostedtoolcache/Python/3.10.16/x64/lib/python3.10/site-packages/IPython/core/formatters.pyreturn method()

	Expression_Level	
Gene		
GeneA	7.433333	
GeneB	7.000000	

11.1.3 Learn More

To get more information on how to use pandas, check out:

- the documentation
- the cheat sheet
- any useful tutorial

11.1.4 Exercise

Exercise

- 1. Create a pandas DataFrame from the file containing the frequency of each nucleotide per sequences (exercise/data/example.txt).
- Make sure that df.index contains the name of the sequences, and df.columns contains the nucleotides.
- 3. Use pandas.melt() (see the doc) to get the data in the following format:

```
nucl freq
Seq
        A 0.46
seq1
           0.20
seq2
          0.16
seq3
seq4
           0.18
seq5
           0.26
           0.22
seq1
seq2
           0.12
```

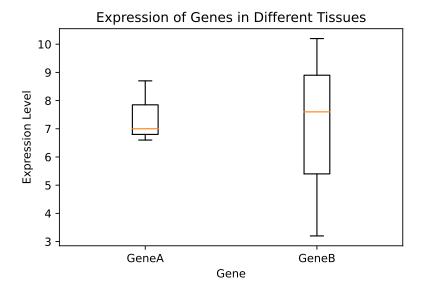
- 4. Get the mean value of all nucleotide frequencies.
- 5. Get the mean value of frequencies per nucleotide.
- 6. Filter to remove values of seq1.
- 7. Recompute the mean value of frequencies per nucleotide.

11.2 Matplotlib

Matplotlib is a package to create visualizations in Python widely used in science.

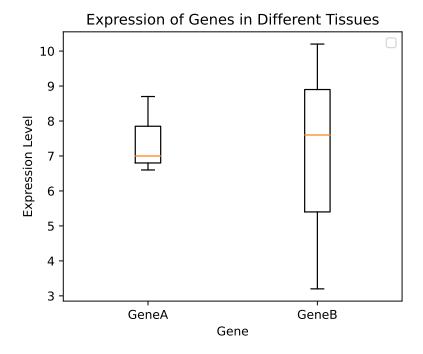
To shorten the name of the package when we call its functions, we can import it as follows:

```
import matplotlib.pyplot as plt
df = pandas.read_csv('exercise/data/sample.csv')
# The data for GeneA and GeneB is extracted from the DataFrame 'df'
serieA = df[df['Gene'] == 'GeneA']['Expression_Level']
serieB = df[df['Gene'] == 'GeneB']['Expression_Level']
# Create a new figure
fig = plt.figure()
# Create a boxplot showing the expression levels of GeneA and GeneB
plt.boxplot([serieA, serieB], # List of series
            labels=['GeneA', 'GeneB'])
# Set the label for the x-axis
plt.xlabel('Gene')
# Set the label for the y-axis
plt.ylabel('Expression Level')
# Set the title of the plot
plt.title('Expression of Genes in Different Tissues')
# Display the boxplot
plt.show()
# Save the plot as a PNG file with a resolution of 300 dots per inch (dpi)
# The file will be saved in the specified location
fig.savefig('exercise/data/my-figure.png', dpi=300)
```



The following code is equivalent.

No artists with labels found to put in legend. Note that artists whose label start with an unconstant



Note

The first way of plotting is function-oriented, and the second is object-oriented. You might encounter both styles of coding.

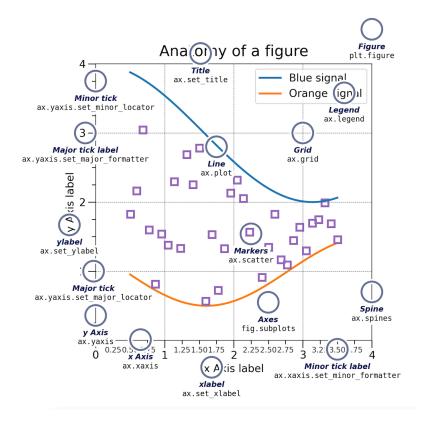


Figure 11.1: Anatomy of a matplotlib plot

Many visualizations are available (static, animated, interactive). For more information, check out:

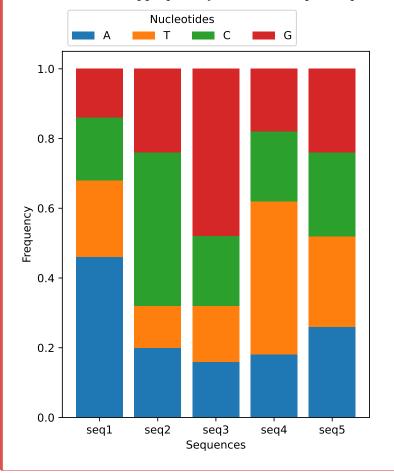
- the documentation
- the cheat sheet
- any useful tutorial
- some inspiration

11.3 Exercise

Exercise

Create a script that gets nucleotide frequency data from a file in the format of exercise/data/example.txt, and visualizes it using Matplotlib and Pandas.

Your script should read the data, create a stacked bar chart showing the nucleotide frequencies for each sequence, and label the axes appropriately. Here's the expected plot:



11.4 More packages

There are MANY packages available, here's a short list of some that might interest you:

Package	Usage	Example of usage
	nComputational	Sequence handling,
Dioi y tho	molecular biology	access to NCBI
	molecular blology	databases
NumPy	Numerical arrays	Data manipulation,
Nullii y	Numerical arrays	mathematical
		operations, linear
C 1	II: -1- 11 :+f f	algebra
Seaborn	High-level interface for	Data visualization,
TITO	drawing plots	statistical graphics
HTSeq	High throughput	Quality and coverage,
	sequencing	counting reads, read
		alignment
Scanpy	Single-Cell Analysis	Preprocessing,
		visualization, clustering
SciPy	Mathematical	Clustering, ODE,
	algorithms	Fourier Transforms
Scikit-	Image processing	Image enhancement,
image		segmentation, feature
		extraction
Scikit-	Machine learning	Classification,
learn		regression, clustering,
		dimensionality reduction
TensorFlowDeep learning		Neural networks,
and Py-	- ·	natural language
Torch		processing, computer
		vision

12 Final tips and resources

Here are a couple of tips:

- Leave comments (think of your future self!)
- Be consistent (quotes, indents...)
- Don't re-invent the wheel, for common tasks, it's likely that a function already exists
- Read the documentation when using a new package or function!
- Google It! Use the correct programming vocabulary to increase your chances of finding an answer. If you don't find anything, try wording it differently.
- The easiest way to learn is by example, so follow a tutorial with the example data, and then try to apply it to your own!

You can follow some free tutorials on:

- Code Academy
- EdX
- Youtube!

Finally, you should able to use Github Copilot (AI coding assistant), as it is free for students: https://education.github.com/benefits.

Trying random stuff for hours instead of reading the documentation



Figure 12.1: Please, read the doc

References

A python conference organized by the AFPy (Association Francophone Python) is held in Strasbourg in the end of October 2024!

Here are some references and ressources that inspired this class:

- Python doc
- \bullet w3schools
- pythonforbiologists
- justinbois's Bootcamp

13 Lesson 3 - Jupyter Notebook

14 Introduction

14.1 Aim of the class

This last class is to pratice the notions learned on a couple of (larger) exercises.

The exercises have one provided solution, but it is not the only one. We all have a different coding style, and it evolves with time. So do not worry if you have a different solution, as long as your result is correct, that's already great!

"the best way to learn a language is to speak to natives" the guy learning Python:



Figure 14.1: pssssss

14.2 Requirements

Remembering some of lesson 1 and 2.

Part I Archive 2024

16 Lesson 1 - Introduction, Data types, Operators

17 Introduction

17.1 Aim of the class

At the end of this class, you will:

- Be familiar with the Python environment
- Understand the major data types in Python
- Manipulate variables with operators and built-in functions



17.2 Requirements

You need to have a computer, and either:

• install Python 3.0.0 (or above) and install a text editor (Word is not a text editor!).

Note

An IDE (integrated development environment) is an improved text editor. It is a software that provides functionalities like syntax highlighting, auto completion, help, debugger... For example Visual Studio Code (install and learn how to use it with Python), but any other IDE will work.

• have a github account, create a new codespace, and select the Repository vgilbart/python-intro to copy from. This is a free solution up to 60 hours of computing and 15 GB per month.

Figure 17.1: Python logo

17.3 What is Python?

Python is a programming language first released in 1991 and implemented by Guido van Rossum.

It is widely used, with various applications, such as:

- software development
- web development
- data analysis
- ...

It supports different types of programming paradigms (i.e. way of thinking) including the procedural programming paradigm. In this approach, the program moves through a linear series of instructions.



Figure 17.2: Guido van Rossum

```
# Create a string seq
seq = 'ATGAAGGGTCC'
# Call the function len() to retrieve the length of the string
size = len(seq)
# Call the function print() to print a text
print('The sequence has', size, 'bases.')
```

The sequence has 11 bases.

17.4 Why use Python?

- Easy-to-use and easy-to-read syntax
- Large standard library for many applications (numpy for tables/matrices, matplotlib for graphs, scikit-learn for machine learning...)
- Interactive mode making it easy to test short snippets of code
- Large community (stackoverflow)

Me when people ask me how I learned programming:



Figure 17.3: Just google it!

17.5 How can I program in Python?

Python is an interpreted language, this means that all scripts written in Python need a software to be run. This software is called an interpreter, which "translate" each line of the code, into instructions that the computer can understand. By extension, the interpreter that is able to read Python scripts is also called Python. So, whenever you want your Python code to run, you give it to the Python interpreter.

17.5.1 Interactive mode

One way to launch the Python interpreter is to type the following, on the command line of a terminal:

python3

Note

You can also try python, /usr/bin/env python3, /usr/bin/python3... There are many ways to call python! You can see where your current python is located by running which python3.

From this, you can start using python interactively, e.g. run:

print("Hello world")

Hello world

To get out of the Python interpreter, type quit() or exit(), followed by enter. Alternatively, on Linux/Mac press [ctrl + d], on Windows press [ctrl + z].

```
(base) MB1-4074-A:~ gilbartv$ python3
Python 3.11.5 (main, Sep 11 2023, 08:31:25) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
>>> quit()
(base) MB1-4074-A:~ gilbartv$
```

Figure 17.4: Interactive mode

17.5.2 Script mode

To run a script, create a folder named script, in which a file named intro.py contains:

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
print("Hello world")
```

and run

```
./script/intro.py
```

You should get the same output as before, that is:

Hello world

The shebang #! followed by the interpreter /usr/bin/env python3 can be put at the beginning of the script in order to ommit calling python3 in command-line. If you don't put it, you will have to run python3 script/intro.py instead of simply ./script/intro.py.

The -*- coding: UTF-8 -*- specify the type of encoding to use. UTF-8 is used by default (which means that this line in the script is not necessary). This accepts characters from all languages. Other valid encoding are available, such as ascii (English characters only).

⚠ Warning

Some common errors can occur at this step:

• bash: script/intro.py: No such file or directory i.e. you are not in the right directory to run the file.

Solution: run ls */ and make sure you can find script/: intro.py, if not go to the correct directory by running cd <insert directory name here>

• bash: script/intro.py: Permission denied i.e. you don't have the right to execute your script.

Solution: run ls -l script/intro.py and make sure you have at least -rwx (read, write, exectute rights) as the first 4 characters, if not run chmod 744 script/intro.py to change your rights.

18 Basic concepts

18.1 Values and variables

You will manipulate values such as integers, characters or dictionaries. These values can be stored in memory using variables. To assign a value to a variable, use the = operator as follow:

```
seq = 'ATGAAGGGTCC'
```

To output the variable value, either type the variable name or use a function like print():

```
seq
```

'ATGAAGGGTCC'

```
print(seq)
```

ATGAAGGGTCC

We can change a variable value by assigning it a new one:

```
seq = seq + 'AAAA' # The + operator can be used to concatenate strings
seq
```

'ATGAAGGGTCCAAAA'

A variable can have a short name (like x and y) or a more descriptive name (seq, motif, genome_file). Rules for Python variable names:

- must start with a letter or the underscore character
- cannot start with a number
- can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- are case-sensitive (seq, Seq and SEQ are three different variables)
- cannot be any of the Python keywords (run help('keywords') to find the list of keywords).

Exercise

Are the following variables names legal?

- 2_sequences
- _sequence
- seq-2
- seq 2

You can try to assign a value to these variable names to be sure of your answer!

18.2 Function calls

A function stores a piece of code that performs a certain task, and that gets run when called. It takes some data as input (parameters that are required or optional), and returns an output (that can be of any type). Some functions are predefined (but we will also learn how to create our own later on).

To run a function, write its name followed by parenthesis. Parameters are added inside the parenthesis as follow:

```
# round(number, ndigits=None)
x = round(number = 5.76543, ndigits = 2)
print(x)
```

5.77

Here the function round() needs as input a numerical value. As an option, one can add the number of decimal places to be used with digits. If an option is not provided, a default value is given. In the case of the option ndigits, None is the default. The function returns a numerical value, that corresponds to the rounded value. This value, just like any other, can be stored in a variable.

To get more information about a function, use the help() function.

Note

If you provide the parameters in the exact same order as they are defined, you don't have to name them. If you name the parameters you can switch their order. As good practice, put all required parameters first.

```
round(5.76543, 2)
5.77

round(ndigits = 2, number = 5.76543)
5.77
```

In Table 18.1 you will find some basic but useful python functions:

Table 18.1: List of useful Python functions.

Function	Description
print()	Print into the screen the values given in argument.
help()	Execute the built-in help system
<pre>quit() or exit()</pre>	Exit from Python
len()	Return the length of an object
round()	Round a numbers

Note

In python, you will also hear about methods. This vocabulary belongs to a programming paradigm called "Objectoriented programming" (OOP).

A method is a function that belongs to a specific class of objects. It is defined within a class and operates only on objects from that class. Methods can access and modify the object's state.

18.3 Getting help

To get more information about a function or an operator, you can use the help() function. For example, in interactive mode, run help(print) to display the help of the print() function, giving you information about the input and output of this function. If you need information about an operator, you will have to put it into quotes, e.g. help('+')

Prowse the help

If the help is long, press [enter] to get the next line or [space] to get the next 'page' of information.

To quit the help, press q.

18.4 Comment your code

Except for the shebang and coding specifications seen before, all things after a hashtag # character will be ignored by the interpreter until the end of the line. This is used to add comments in your code.

Comments are used to:

- explain assumptions
- justify decisions in the code
- expose the problem being solved
- inactivate a line to help debug

• ...

19 How can I represent data?

Each programming language has its own set of data types, from the most basics (bool, int, string) to more complex structures (list, tuple, set...).

19.1 Simple data types

19.1.1 Boolean

Booleans represent one of two values: True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9)
```

True

19.1.2 Numeric

Python provides three kinds of numerical type:

- int (\mathbb{Z}) , integers
- float (\mathbb{R}) , real numbers
- complex (C), complex numbers

Python will assign a numerical type automatically.

```
x = 1

y = 2.8

z = 1j + 2 # j is the convention in electrical engineering
```

```
type(x)
```

int

```
type(y)
```

float

```
type(z)
```

complex

19.1.3 Text

String type represents textual data composed of letters, numbers, and symbols. The character string must be expressed between quotes.

```
"""my string"""
"my string"
"my string"
'my string'
```

are all the same thing. The difference with triple quotes is that it allows a string to extend over multiple lines. You can also use single quotes and double quotes freely within the triple quotes.

```
# A multi-line string
my_str = '''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
''''
print(my_str)
```

```
This is a multi-line string. This is the first line. This is the second line.

"What's your name?," I asked.

He said "Bond, James Bond."
```

You can get the number of characters inside a string with len().

```
print(seq)
len(seq)
```

ATGAAGGGTCCAAAA

15

Strings have specific methods (i.e. functions specific to this class of object). Here are a few:

Method	Description
.count()	Returns the number of times
	a specified value occurs in a
	string
<pre>.startswith()</pre>	Returns true if the string
	starts with the specified value
<pre>.endswith()</pre>	Returns true if the string
	ends with the specified value
.find()	Searches the string for a
	specified value and returns
	the position of where it was
	found
.replace()	Returns a string where a
	specified value is replaced
	with a specified value

They are called like this:

seq.count('A')

7



Tip

To get the help() of the .count() method, you need to run help(str.count).

Exercise

- 1. Check if the sequence seq starts with the codon ATG
- 2. Replace all T into U in seq

19.2 Data structures

Data structures are a collection of data types and/or data structures, organized in some way.

19.2.1 List

List is a collection which is ordered and changeable. It allows duplicate members. They are created using square brackets [].

```
seq = ['ATGAAGGGTCCAAAA', 'AGTCCCCGTATGAT', 'ACCT', 'ACCT']
```

List items are indexed, the first item has index [0], the second item has index [1] etc.

seq[1]

^{&#}x27;AGTCCCCGTATGAT'

Tip

You can count backwards, with the index [-1] that retrieves the last item.

As a list is changeable, we can change, add, and remove items in a list after it has been created.

```
seq[1] = 'ATG'
seq
```

```
['ATGAAGGGTCCAAAA', 'ATG', 'ACCT', 'ACCT']
```

You can specify a range of indexes by specifying the start (included) and the end (not included) of the range.

```
seq[0:2]
```

['ATGAAGGGTCCAAAA', 'ATG']

Tip

By leaving out the start value, the range will start at the first item:

seq[:2]

['ATGAAGGGTCCAAAA', 'ATG']

Similarly, by leaving out the end value, the range will end at the last item.

Note

Indexes also conveniently work on str types.

```
print(seq[0])
print(seq[0][0:5])
print(seq[0][2])
print(seq[0][-1])

ATGAAGGGTCCAAAA
ATGAA
G
A
```

You can get how many items are in a list with len().

```
len(seq)
```

4

Lists have specific methods. Here are a few:

Method	Description
.append()	Inserts an item at the end
.insert()	Inserts an item at the specified index
<pre>.extend()</pre>	Append elements from another list to the current list
.remove()	Removes the first occurance of a specified item
.pop()	Removes the specified (by default last) index

Exercise

- Create a list 1 = ['AAA', 'AAT', 'AAC'], and add
 AAG at the end, using .append().
- 2. Replace all T into U in the element AAT, using .replace().

19.2.2 Tuple

Tuple is a collection which is ordered and unchangeable. It allows duplicate members. Tuples are written with round brackets ().

my_favorite_amino_acid = ('Y', 'Tyr', 'Tyrosine')

Just like for the list, you can get items with their index. The only difference is that you cannot change a tuple that has been created.

Tuples have specific methods. Here are a few:

Method	Description
.count()	Returns the number of times a specified value occurs
.index()	Searches for a specified value and returns the position of where it was found

Exercise

Try to change the value of the first element of my_favorite_amino_acid and see what happens.

19.2.3 Set

Set is a collection which is unordered and unindexed. It does not allow duplicate members (they will be ignored). Sets are written with curly brackets {}.

Once a set is created, you cannot change its items directly (as they don't have index), but you modify the set by removing and adding items.

Sets have specific methods. Here are a few:

Method	Description
.add()	Adds an element to the set

Method	Description
.difference()	Returns a set containing the difference between two sets
.intersection()	Returns a set containing the intersection between two sets
.union()	Returns a set containing the union of two sets
<pre>.remove() .pop()</pre>	Remove the specified item Removes a random element

```
Exercise

Get the common genes between the following sets:

organism1_genes = {'BRCA1', 'TP53', 'EGFR', 'MYC'}
organism2_genes = {'TP53', 'MYC', 'KRAS', 'BRAF'}
```

19.2.4 Dictionary

Dictionaries are used to store data values in key: value pairs. A dictionary is a collection which is ordered (as of Python >= 3.7), changeable and does not allow duplicates keys. Dictionaries are written with curly brackets {}, with keys and values.

```
organism1_genes = {
    #key: value;
    'BRCA1': 'DNA repair',
    'TP53': 'Tumor suppressor',
    'EGFR': 'Cell growth',
    'MYC': 'Regulation of gene expression'
}
```

Dictionary items can be referred to by using the key name.

```
organism1_genes["BRCA1"]
```

^{&#}x27;DNA repair'

Dictionaries have specific methods. Here are a few:

Method	Description
.items()	Returns a list containing a tuple for each key value pair
.keys()	Returns a list containing the dictionary's keys
.values()	Returns a list of all the values in the dictionary
.pop()	Removes the element with the specified key
.get()	Returns the value of the specified key

Exercise

From the dictionary organism1_genes created as example, get the value of the key BRCA1. If the key does not exist, return Unknown by default. Try your code before and after removing the BRCA1 key:value pair. Check the help of get by running help(dict.get).

19.3 Conversion between types

You can get the data type of any object by using the function type(). You can (more or less easily) convert between data types.

Function	Description
bool()	Convert to boolean type
<pre>int(), float()</pre>	Convert between integer or
	float types
complex()	Convert to complex type
str()	Convert to string type
<pre>list(), tuple(), set()</pre>	Convert between list, tuple,
	and set types

Function	Description
dict()	Convert a tuple of order (key, value) into a dictionary type

```
bool(1)
```

True

```
int(5.8)
```

5

```
str(1)
```

'1'

```
list({1, 2, 3})
```

[1, 2, 3]

```
set([1, 2, 3, 3])
```

{1, 2, 3}

{'a': 1, 'f': 2, 'g': 3}

20 How can I manipulate data?

In the previous section we have learned how data can be represented in different types and gathered in various data structures. In this section we will see how we can manipulate data in order to do more complex tasks.

20.1 Operators

Operators are used to perform operations on variables and values. We will present a few common ones here.

20.1.1 Arithmetic operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name
+	Addition
_	Substraction
*	Multiplication
/	Division
**	Power



⚠ Warning

Do not use the ^ operator to raise to a power. That is actually the operator for bitwise XOR, which we will not cover.

Python will convert data type according to what is necessary. Thus, when you divide two int you will obtain a float number, if you add a float to an int, you will get a float, ...

```
# Example
2/10
```

0.2

```
Note

+ also conveniently work on str types.

'AC' + 'AT'

'ACAT'
```

20.1.2 Assignment operators

Assignment operators are used to assign values to variables:

Operator	Example as	Same as
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5

Note

The same principle applies to multiplication, division and power, but are less commonly used.

20.1.3 Comparison operators

Comparison operators are used to compare two values:

Operator	Name
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

```
# Example
2 == 1 + 1
```

True

⚠ Warning

You should never use equalty operators (==or !=) with floats or complex values.

```
# Example
2.1 + 3.2 == 5.3
```

False

This is a floating point arithmetic problem seen in other programming languages. It is due to the difficulty of having a fixed number of binary digits (bits) to accurately represent some decimal number. This leads to small rounding errors in calculations.

2.1 + 3.2

5.30000000000001

If you need to use equalty operators, do it with a degree of freedom:

```
tol = 1e-6; abs((2.1 + 3.2) - 5.3) < tol
```

True

20.1.4 Logical operators

Logical operators are used to combine conditional statements:

Operator	Description
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

Example

False and False, False and True, True and False, True and True

(False, False, True)

Example

False or False, False or True, True or False, True or True

(False, True, True, True)

Example

True or not True

True

20.1.5 Membership operators

Operator	Description
in	Returns True if a sequence with the specified value is
not in	present in the object Returns True if a sequence with the specified value is not
	present in the object

```
# Example
'ACCT' in seq
```

False

20.1.6 Operator precedence

Operator precedence describes the order in which operations are performed.

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parenthesis
**	Power
* /	Multiplication, division
+ -	Addition, substraction
==,!=,>,>=,<,<=,is,is	Comparisons, identity, and
not,in,not in,	membership operators
not	Logical NOT
and	AND
or	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Exercise

Try to guess what will output the following expressions:

- 1+1 == 2 and "actg" == "ACTG"
- True or False and True and False
- "Homo sapiens" == "Homo" + "sapiens"
- 'Tumor suppressor' in organism1_genes

Verify with Python.

20.2 Conditionals

Conditionals allows you to make decisions in your code based on certain conditions.

```
if something is true:
    do task a
otherwise:
    do task b
```

The comparison (==, !=, >, >=, <, <=), logical (and, or, not) and membership (in, not in) operators can be used as conditions.

In Python, this is written with an if ... elif ... else statement like so:

```
# Define gene expression levels
gene1_expression = 100
gene2_expression = 50

# Analyze gene expression levels
if gene1_expression > gene2_expression:
   print("Gene 1 has higher expression level.")
elif gene1_expression < gene2_expression:
   print("Gene 2 has higher expression level.")
else:
   print("Gene 1 and Gene 2 have the same expression level.")</pre>
```

Gene 1 has higher expression level.

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition". The following code is equivalent to the one before:

```
# Analyze gene expression levels
if gene1_expression > gene2_expression:
  print("Gene 1 has higher expression level.")
else:
```

```
if gene1_expression < gene2_expression:
   print("Gene 2 has higher expression level.")
else:
   print("Gene 1 and Gene 2 have the same expression level.")</pre>
```

Gene 1 has higher expression level.

```
Exercise
Are these two codes equivalent?
# Code A
if "ATG" in dna_sequence:
 print("Start codon found.")
elif "TAG" in dna_sequence:
 print("Stop codon found.")
else:
  print("No interesting codon not found.")
# Code B
if "ATG" in dna_sequence:
 print("Start codon found.")
  if "TAG" in dna_sequence:
   print("Stop codon found.")
else:
 print("No interesting codon not found.")
```

An if statement cannot be empty, but if for some reason you have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200

if b > a:
   pass
```

20.3 Notes on indentation

Note

Python relies on **indentation** (the spaces at the beginning of the lines).

Indentation is not just for readability. In Python, you use spaces or tabs to indent code blocks. Python uses it to determine the scope of functions, loops, conditional statements, and classes.

Any code that is at the same level of indentation is considered part of the same block. Blocks of code are typically defined by starting a line with a colon (:) and then indenting the following lines.

When you have nested structures like a conditional statement inside another conditional statement, you must further to show the hierarchy. Each level of indentation represents a deeper level of nesting.

It's essential to be consistent with your indentation throughout your code. Mixing tabs and spaces can lead to errors, so it's recommended to choose one and stick with it.

Exercise

Here are three codes, they all are incorrect, can you tell why?

Of course, you can run them and read the error that Python gives!

```
amino_acid_list = ["MET", "ARG", "THR", "GLY"]

if "MET" in amino_acid_list:
   print("Start codon found.")
   if "GLY" in amino_acid_list:
      print("Glycine found.")

else:
print("Start codon not found.")
```

```
dna_sequence = "ATGCTAGCTAGCTAG"

if "ATG" in dna_sequence:
   print("Start codon found.")

if "TAG" in dna_sequence
   print("Stop codon found.")

x = 7

if x > 5:
   print("x is greater than 5")
   if y > 10:
        print("x is greater than 10")

elif y = 10:
        print("x equals 10")
   else:
        print("x is less than 10")
```

20.4 Iterations

Iteration involves repeating a set of instructions or a block of code multiple times.

There are two types of loops in python, for and while.

Iterating through data structures like lists allows you to access each element individually, making it easier to perform operations on them.

20.4.1 For loops

When using a for loop, you iterate over a sequence of elements, such as a list, tuple, or dictionary.

```
for item in data_structure:
    do task a
```

The loop will execute the indented block of code for each element in the sequence until all elements have been processed. This is particularly useful when you know the number of times you need to iterate.

```
all_codons = [
    'AAA', 'AAC', 'AAG', 'AAT',
    'ACA', 'ACC', 'ACG', 'ACT',
    'AGA', 'AGC', 'AGG', 'AGT',
    'ATA', 'ATC', 'ATG', 'ATT',
    'CAA', 'CAC', 'CAG', 'CAT',
    'CCA', 'CCC', 'CCG', 'CCT',
    'CGA', 'CGC', 'CGG', 'CGT',
    'CTA', 'CTC', 'CTG', 'CTT',
    'GAA', 'GAC', 'GAG', 'GAT',
    'GCA', 'GCC', 'GCG', 'GCT',
    'GGA', 'GGC', 'GGG', 'GGT',
    'GTA', 'GTC', 'GTG', 'GTT',
    'TAA', 'TAC', 'TAG', 'TAT',
    'TCA', 'TCC', 'TCG', 'TCT',
    'TGA', 'TGC', 'TGG', 'TGT',
    'TTA', 'TTC', 'TTG', 'TTT'
]
count = 0
for codon in all codons:
  if codon[1] == 'T':
    count += 1
print(count, 'codons have a T as a second nucleotide.')
```

16 codons have a T as a second nucleotide.

What it does is the following: it processes each element in the list all_codons, called in the following code codon. If the codon has as a second character a T, it adds 1 to a counter (the variable called count).

```
⚠ Warning
```

You cannot modify an element of a list that way.

```
for codon in all_codons:
   if 'T' in codon :
      codon = codon.replace('T', 'U')

print(all_codons)

['AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG',
```

This is because 22 and an arranged to an iteration

'ACT', 'AGA', 'AGC', 'AGG', 'AGT', 'ATA',

This is because all_codons was converted to an iterator in the for statement.

20.4.2 Iterators

An iterator is a special object that gives values in succession.

In the previous example, the iterator returns a copy of the item in a list, not a reference to it. Therefore, the **codon** inside the **for** block is not a view into the original list, and changing it does not do anything.

A way to modify the list would be to use an iterable to access the original data. The range(start, stop) function creates an iterable to count from one integer to another.

```
for i in range(2, 10):
    print(i, end=' ')
```

2 3 4 5 6 7 8 9

We could count from 0 to the size of the list, loop though every element of the list by calling them by their index, and modify them if necessary. That's what the following code does:

```
for i in range(0, len(all_codons)):
    if 'T' in all_codons[i] :
        all_codons[i] = all_codons[i].replace('T', 'U')

print(all_codons)

['AAA', 'AAC', 'AAG', 'AAU', 'ACA', 'ACC', 'ACG', 'ACU', 'AGA', 'AGC', 'AGG', 'AGU', 'AUA', '
```

```
seq = 'ATGCATGC'

# Print index and identity of bases
for i, base in enumerate(seq):
    print(i, base)
```

commonly used when you need to access both the index and

value of items simultaneously.

```
O A
1 T
2 G
3 C
4 A
5 T
6 G
7 C
```

```
# Loop through sequence and print index of G's
for i, base in enumerate(seq):
    if base in 'G':
        print(i, end=' ')
```

2 6

20.4.3 While loops

A while loop continues executing a set of statement as long as a condition is true.

```
while condition is true:
    do task a
```

This type of loop is handy when you're not sure how many iterations you'll need to perform or when you need to repeat a block of code until a certain condition is met.

```
seq = 'TACTCTGTCGATCGTACGTATGCAAGCTGATGCATGATTGACTTCAGTATCGAGCGCAGCA'
start_codon = 'ATG'
# Initialize sequence index
i = 0
# Scan sequence until we hit the start codon
while seq[i:i+3] != start_codon:
  i += 1
# Show the result
print('The start codon begins at index', i)
```

The start codon begins at index 19



Warning

Remember to increment i, or you'll get stuck in a loop.

Actually, the previous code is quite dangerous. You can also get stuck in a loop... if the start_codon does not appear in seq at all.

Indeed, even when you go above the given length of seq, the condition seq[i:i+3] != start_codon will still be true because seq[i:i+3] will output an empty string.



Figure 20.1: Hopefully not you!

```
seq[9999:9999+3]
```

So, once the end of the sequence is reached, the condition seq[i:i+3] != start_codon will always be true, and you'll get stuck in an infinite loop.

```
i Note

To get interrupt a process, press [ctrl + c].
```

20.4.4 Break statement

Iteration stops in a for loop when the iterator is exhausted. It stops in a while loop when the conditional evaluates to False. There is another way to stop iteration: the break keyword. Whenever break is encountered in a for or while loop, the iteration stops and execution continues outside the loop.

Codon not found in sequence.

Note

Also, note that the else statement can be used in for and while loops. In for loops it is executed when the loop is finished. In while loops, it is executed when the condition is no longer true. In both case, the loops need to not encounter a break to enter in the else block.

20.4.5 Continue statement

In addition to the break statement, there is also the continue statement in Python that can be used to alter the flow of iteration in loops. When continue is encountered within a loop, it skips the remaining code inside the loop for the current iteration and moves on to the next iteration.

Here's an example showcasing the continue statement in a loop:

```
# List of DNA sequences
dna_sequences = ['ATGCTAGCTAG', 'ATCGATCGATC', 'ATGGCTAGCTA', 'ATGTAGCTAGC']

# Find sequences starting with a start codon
for sequence in dna_sequences:
    if sequence[:3] != 'ATG': # Check if the sequence does not start with a start codon
        print(f"Sequence '{sequence}' does not start with a start codon. Skipping analysis.")
        continue # Skip further analysis for this sequence
    print(f"Analyzing sequence '{sequence}' for protein coding regions...")
    # Additional analysis code here
else:
    print('All sequences were processed.')

Analyzing sequence 'ATGCTAGCTAG' for protein coding regions...
Sequence 'ATCGATCGATC' does not start with a start codon. Skipping analysis.
Analyzing sequence 'ATGGCTAGCTA' for protein coding regions...
```

The continue statement in this example skips the analysis code for sequence that does not start with a start codon.

All sequences were processed.

Analyzing sequence 'ATGTAGCTAGC' for protein coding regions...

20.4.6 Exercises

Exercise 1

Given a list of DNA sequences, find the first sequence that contains a specific motif 'TATA', print the sequence, and stop the process. If no sequence contains the motif, print a message accordingly. You must use only one for loop. With the input given below, the output should look like this:

```
# List of DNA sequences with a TATA
dna_sequences = [
'ATGCTACAGCTAG',
'ATCGATATAATC', # TATA
'ATGGCTAGCTA',
'ATGTAGCTAGC',
'ATGTAGCTATA' # TATA
for ...
# Your code here
Sequence 'ATCGATATAATC' contains the 'TATA' motif.
```

```
# List of DNA sequences without a TATA
dna_sequences = [
'ATGCTACAGCTAG',
'ATCGATACAATC',
'ATGGCTAGCTA',
'ATGTAGCTAGC'
for ...
# Your code here
```

No sequence contains the 'TATA' motif.

Exercise 2

Analyze a DNA sequence to count the number of consecutive 'A' nucleotides. You must use only one while loop. With the input given below, the output should look like this:

```
# DNA sequence to analyze
dna_sequence = 'ATGATAAGAGAAAGTAAAAGCGATCGAAAAAA'
while ...
# Your code here
```

Number of consecutive 'A's: 6

21 Conclusion

Congrats! You now know the (very) basics of Python programming.

If you want to keep on practising with simple exercises, you can check out w3schools.

For more biology-related exercises check out pythonforbiologist.org, they have exercises availables in each chapters.

For french speakers, the AFPy (Association Francophone Python) has a learning tool called HackInScience.

Or keep on googling for more python exercises!

To continue your learning journey, follow lesson 2.

References

A python conference organized by the AFPy (Association Francophone Python) is held in Strasbourg in the end of October 2024!

Here are some references and ressources that (greatly) inspired this class:

- Python doc
- \bullet w3schools
- pythonforbiologists
- justinbois's Bootcamp

22 Lesson 2 - Functions, Errors, File Handling, Scientific Packages

23 Introduction

23.1 Aim of the class

At the end of this class, you will be able to:

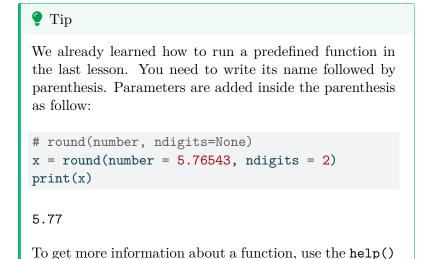
- Create simple functions
- Handle some errors
- Ask for input from the user
- Upload, modify and download files into Python
- Import packages (and use in a simple manner some scientific packages)

23.2 Requirements

Remembering some of lesson 1.

24 Function

A function stores a piece of code that performs a certain task, and that gets run when called. It takes some data as input (parameters that are required or optional), and returns an output (that can be of any type).



We will now learn how to create our own function.

24.1 Syntax

function.

In python, a function is declared with the keyword def followed by its name, and the arguments inside parenthesis. The next block of code, corresponding to the content of the function, must be indented. The output is defined by the return keyword.

```
def hello(name):
    """Presenting myself."""
    presentation = "Hello, my name is {0}.".format(name)
    return presentation
```

```
text = hello(name = "Valentine")
print(text)
```

Hello, my name is Valentine.

24.2 Documentation

As you may have noticed, you can also add a description of the function directly after the function definition. It is the message that will be shown when running help(). As it can be along text over multiple lines, it is common to put it inside triple quotes """.

```
help(hello)

Help on function hello in module __main__:
hello(name)
    Presenting myself.
```

24.3 Arguments

You can have several arguments. They can be mandatory or optional. To make them optional, they need to have a default value assigned inside the function definition, like so:

```
def hello(name, french = True):
    """Presenting myself."""
    if french:
        presentation = "Bonjour, je m'appelle {0}."
    else:
```

```
presentation = "Hello, my name is {0}."
return presentation.format(name)
```

The parameter name is mandatory, but french is optional.

```
hello("Valentine")
"Bonjour, je m'appelle Valentine."
hello(french = False)
TypeError: hello() missing 1 required positional argument: 'name'
                                            Traceback (most recent call last)
TypeError
Cell In[7], line 1
----> 1 hello(french = False)
TypeError: hello() missing 1 required positional argument: 'name'
 Note
 Reminder: if you provide the parameters in the exact
 same order as they are defined, you don't have to name
 them. If you name the parameters you can switch their
 order. As good practice, put all required parameters first.
 hello(french = False, name = "Valentine")
  'Hello, my name is Valentine.'
 hello("Valentine", False)
  'Hello, my name is Valentine.'
```

24.4 Output

If no return statement is given, then no output will be returned, but the function will still be run.

```
def hello(name):
    """Presenting myself."""
    print("We are inside the 'hello()' function.")
    presentation = "Hello, my name is {0}.".format(name)
```

```
print(hello("Valentine"))
```

We are inside the 'hello()' function. None

The output can be of any type. If you have a lot of things to return, you might want to return a list or a dict for example.

```
def multiple_of_3(list_of_numbers):
    """Returns the number that are multiple of 3."""
    multiples = []
    for num in list_of_numbers:
        if num % 3 == 0:
            multiples.append(num)
    return multiples

multiple_of_3(range(1, 20, 2))
```

[3, 9, 15]

```
i Note
This could be written as a one-liner.

def multiple_of_3(list_of_numbers):
    """Returns the number that are multiple of 3."""
    multiples = [num for num in list_of_numbers if num % 3 == 0]
    return multiples

multiple_of_3(range(1, 20, 2))

[3, 9, 15]
```

24.5 Exercise

Exercise

Write a function called nucl_freq to compute nucleotide frequency of a sequence. Given a sequence as input, it outputs a dictionnary with keys being the nucleotides A, T, C and G, and values being their frequency in the sequence. With the input given below, the output should be:

```
def ...
    # Your code here

nucl_freq("ATTCCCGGGG")

{'G': 0.4, 'A': 0.1, 'C': 0.3, 'T': 0.2}
```

25 Exceptions Handling

25.1 Syntax

It is possible to handle errors (in python, they are also called exceptions), using the following statements:

- try to test a block of code for errors
- except to handle the error
- else to excute code if there is no error
- finally to excute code, regardless of the result of the try and except blocks

```
try:
   print(some_undefined_variable)
except:
   print("Oops... Something went wrong")
else:
   print("Nothing went wrong")
finally:
   print("The 'try except' is finished")
```

Oops... Something went wrong
The 'try except' is finished

25.2 Raising exceptions

Here is a table of some of the built-in exceptions in python.

Exception	Description
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c or
NameError	Delete). Raised when a variable is not found in the local or global
TypeError	scope. Raised when a function or operation is applied to an
ValueError	object of an incorrect type. Raised when a function receives an argument of the
RuntimeError	correct type but of an incorrect value. Raised when an error occurs that do not belong to any specific exceptions.
Exception	Base class of exceptions.

You can use them to be more specific about the type of exception occurring.

```
try:
    print(some_undefined_variable)
except NameError:
    print("A variable is not defined")
except:
    print("Oops... Something went wrong")
else:
    print("Nothing went wrong")
finally:
    print("The 'try except' is finished")
```

A variable is not defined The 'try except' is finished

You can also use them to throw an exception if a condition occurs, by using the raise keyword.

```
try:
   if not isinstance(x, int):
      raise TypeError("Only integers are allowed")
   if x < 0:
      raise ValueError("Sorry, no numbers below zero")
   print(x, "is a positive integer.")
except NameError:
   print("A variable is not defined")
else:
   print("Nothing went wrong")
finally:
   print("The 'try except' is finished")</pre>
```

The 'try except' is finished

TypeError: Only integers are allowed

TypeError

Traceback (most recent call last)

```
Cell In[20], line 4
        2 try:
        3   if not isinstance(x, int):
----> 4      raise TypeError("Only integers are allowed")
        5   if x < 0:
        6      raise ValueError("Sorry, no numbers below zero")
TypeError: Only integers are allowed</pre>
```

25.3 Exercise

Exercise Let's make our previous function even better by adding some exception handling. Raise a TypeError if the input is not a string. Raise a ValueError if the input string contains something else than the nucleotides A, C, T, With the input given below, the output and errors should be: def ... # Your code here nucl_freq(5474) nucl_freq("ATTCXCCGGGG") nucl_freq("ATTCCCGGGG") TypeError: Input must be a string. Traceback (most recent call last) TypeError Cell In[21], line 15 12 freq[nucl] = seq.count(nucl)/n return freq 13 ---> 15 nucl_freq(5474) 16 nucl_freq("ATTCXCCGGGG") 17 nucl_freq("ATTCCCGGGGG") Cell In[21], line 3, in nucl_freq(seq) 1 def nucl_freq(seq): if not isinstance(seq, str):

```
----> 3 raise TypeError("Input must be a string.")

4 valid_nucl = "A", "T", "C", "G"

5 seq_nucl = set(seq)

TypeError: Input must be a string.
```

26 User-defined input

There are some interesting ways to get input from the user:

- input() receives input from the keyboard. This means that the input is defined *while* the python script is being executed.
- sys.argv takes arguments provided in command line after the name of the program. This means that the input is defined *before* the python script is being executed.
- argparse is similar to sys.argv, with the advantage of being able to give specific names to arguments.

26.1 input

Python stops executing when it comes to the input() function, and continues when the user has given some input.

In a file called username-1.py, write the following:

```
username = input("Enter username: ")
print("Username is: " + username)
```

Then in the terminal, run:

```
#| eval: False
python username-1.py
```

You should be asked, in command line, to enter a username. When you write it, and press Enter, it gets printed.

Enter username: vgilbart Username is: vgilbart

26.2 sys.argv

To use sys.argv you need to import a module called sys. It is part of the standard python library, so you should not have to install anything in particular.

In a file called username-2.py, write the following:

```
import sys
print("Username is: " + sys.argv[1])
```

Then in the terminal, run:

```
#| eval: False
python username-2.py vgilbart
```

Arguments are given in command line, seperated by [space].

Username is: vgilbart

Note

What is the type of sys.argv? Remember that in python index begins at 0. What do you think is sys.argv[0]? Verify!

Also, what happens if you run python username-2.py valentine gilbart?

26.3 argparse

Just like for sys, you need to import argparse.

In a file called username-3.py, write the following:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--username', action="store")

args = parser.parse_args()
print("Username is: " + args.username)
```

Then in the terminal, run:

```
#| eval: False
python username-3.py --username vgilbart
```

Arguments are given in command line, but they have specific names.

Note

argparse is a very useful module when creating programs! You can easily specify the expected type of argument, whether it is optional or not, and create a help for your script. Check their tutorial for more information.

27 File Handling

The key function to work with files in open(). It has two parameters file and mode.

```
# Write the correct path for you!
fasta_file = 'exercise/data/example.fasta'
f = open(fasta_file, mode = 'r')
```

The modes can be one of the following:

Mode	Description
r	Opens a file for reading, error if the file does not exist (default)
a	Opens a file for appending, creates the file if it does not exist
W	Opens a file for writing, creates the file if it does not exist
x	Creates the specified file, returns an error if the file exists

27.1 Reading

The open() function returns a file object, which has a read() method for reading the content of the file:

```
print(f.read())
```

>seq1

The parameter **size** = can be added to specify the number of bytes (~ characters) to return.

```
# We need to re-open it because we have already parsed the whole file
f = open(fasta_file, mode = 'r')
print(f.read(2))
```

>s

You can return one line by using the .readline() method. By calling it two times, you can read the two first lines:

```
f = open(fasta_file, mode = 'r')
print(f.readline())
print(f.readline())
```

>seq1

By looping through the lines of the file, you can read the whole file, line by line:

```
for i, line in enumerate(f):
   print(i, line)
```

- 0 > seq2
- 2 >seq3

- 3 CCGGGCGGTCGATGGATGGAGGGAGCGATCGATCGGTCGATCGGTG
- 4 >seq4
- 5 GATCGATCGATCTTTTATCGATCGATTGTTCTTTCGATCGTTCTATCGA
- 6 >seq5

It is a good practice to close the file when you are done with it.

f.close()



⚠ Warning

In some cases, changes made to a file may not show until you close the file.

Note

>seq1

A common syntax to handle files that you might encounter

```
with open(fasta_file, 'r') as f:
  print(f.readline())
>seq1
```

This code is equivalent to

```
f = open(fasta_file, 'r')
 print(f.readline())
finally:
 f.close()
```

The with statement is an example of a context manager, i.e. it allows to allocate and release resources precisely, by cleaning up the resources once they are no longer needed.

27.2 Writting

To write into a file, you must have it open under a w, a mode.

Then, the method write() can be used.

```
txt_file = "exercise/data/some_file.txt"
f = open(txt_file, "w")
f.write("Woops! I have deleted the content!\n")
f.close()
# Read the current content of the file
f = open(txt_file, "r")
print(f.read())
```

Woops! I have deleted the content!

⚠ Warning

Be very careful when opening a file in write mode as you can delete its content without any way to retrieve the original file!

As you may have noticed, write() returns the number of characters written. You can prevent it from being printed by assigning the return value to a variable that will not be used.

```
f = open(txt_file, "a")
_ = f.write("Now the file has more content!\n")
f.close()
# Read the current content of the file
f = open(txt_file, "r")
print(f.read())
```

Woops! I have deleted the content! Now the file has more content!

Note

You must specify a newline with the character:

- \n in Linus/MacOS
- \r\n in Windows
- \r in MacOS before X

27.3 os module

Python has a built-in package called **os**, to interact with the operating system.

```
import os

print("Current working directory:", os.getcwd())
  os.chdir('../')
print("Current working directory:", os.getcwd())
```

Current working directory: /home/runner/work/python-intro/python-intro Current working directory: /home/runner/work/python-intro

Here are some useful functions from the os package.

Function	Description
<pre>getcwd() chdir()</pre>	Returns the current working directory Change the current working directory
listdir()	Returns a list of the names of the entries in a
mkdir()	directory Creates a directory
mkdirs()	Creates a directory recursively

27.4 Regular expression

A regular expression is a sequence of characters that forms a search pattern.

Python has a built-in package called **re**, to work with regular expressions.

```
import re
x = re.findall("hello", "hello world, hello you!")
print(x)
```

['hello', 'hello']

Here are some useful functions from the re package.

Function	Description
findall()	Returns a list containing all matches
search()	Returns a Match object if there is a match
	anywhere in the string
split()	Returns a list where the string has been split at each match
sub()	Replaces one or many matches with a string

To be more specific about a sequence search, regular expression uses metacharacters (i.e characters with sepecial meaning)

Metachara Description		Example
	A set of characters	[a-m]
\	Signals a special sequence (can	\n
	also be used to escape special	
	characters)	
•	Any character (except newline	heo
	character)	
^	Starts with	^hello
\$	Ends with	hello\$
*	Zero or more occurrences	he.*o

Metachara Description		Example
+ ? {}	One or more occurrences Zero or one occurrences Exactly the specified number of occurrences	he.+o he.?o he.{2}o
()	Either or Captures and group	hello bonjour hello (.+) \1 in which \1 correspond to what is being captured in (.+)

Note

To build and test a regex, you can use regex101.com, or any website equivalent, in which you can write your regex, and some string to test, to see how it matches.

A Match Object is an object containing information about the search and the result.

```
x = re.search("hello .*",
"""
hello world
hello you
bonjour
""")
print(x)
```

<re.Match object; span=(1, 12), match='hello world'>

The Match object has methods used to retrieve information about the search, and the result:

- .span() returns a tuple containing the start and end positions of the match.
- $\bullet\,$.group() returns the part of the string where there was a match

print(x.group())

hello world

Exercise

From the list dna_sequences = ["ATGCGAATTCAC", "ATGAC", "ATGCCCGGGTAA", "ATGACGTACGTC", "ATGAGGGGTTCA"],

- 1. Extract all sequences that start with ATG and end with AC or AA.
- 2. Extract all sequences that contain either G or C repeated three times consecutively.

You should get the following results:

```
Sequences starting with 'ATG' and ending with 'AC' or 'AA':
['ATGCGAATTCAC', 'ATGAC', 'ATGCCCGGGTAA']
Sequences containing 'G' or 'C' repeated three times consecutively:
['ATGCCCGGGTAA', 'ATGAGGGGTTCA']
```

27.5 Exercise

Exercise

Create a program, that you can run on command line as follow ./analyse_fasta.py path/to/fasta/file path/to/output/file. It should:

- read the fasta file,
- calculate the nucleotide frequency for each sequence (using the previously defined function)
- create a new file as follow:

```
Seq A C T G
seq1 0.1 0.2 0.3 0.4
seq2 0.4 0.3 0.2 0.1
```

. . .

To make this easier, consider that the sequences in the fasta file are only in one line.

You might make good use of the method str.strip(). You can take as input the file in exercise/data/example.fasta you should get the same result as exercise/data/example.txt.

28 Scientific packages

A python package contains a set of function to perform specific tasks.

A package needs to be **installed** to your computer one time.

You can install a package with pip. It should have been automatically installed with your python, to make sure that you have it you can run:

```
#| eval: false
# In Linux/MacOS
python -m pip --version
# In Windows
py -m pip --version
```

If it does not work, check out pip documentation.

To install a package called pandas, you must run:

```
#| eval: false
# In Linux/MacOS
python -m pip install pandas
# In Windows
py -m pip install pandas
```

To get more information about pip, check out the full documentation.



Warning

Installing a package is done outside of the python interpreter, in command line in a terminal.

When you wish to use a package in a python script, you'll need to import it, by writing inside of you script:

```
import pandas
```

28.1 Pandas

Pandas is a package used to work with data sets, in order to easily clean, manipulate, explore and analyze data.

28.1.1 Create pandas data

Pandas provides two types of classes for handling data:

• Series: a one-dimensional labeled array holding data of any type such as integers or strings. It is like a column in a table.

```
# If nothing else is specified, the values are labeled with their index number (starting from
myseries = pandas.Series([1, 7, 2], index = ["x", "y", "z"])
print(myseries)
```

```
x 1
y 7
z 2
dtype: int64
```

 DataFrame: a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns. It is like a table.

```
data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

df = pandas.DataFrame(data)
print(df)
```

```
calories duration
0 420 50
1 380 40
2 390 45
```

You can also create a DataFrame from a file.

```
# Make sure this is the correct path for you! You are in the directory from where you execute
df = pandas.read_csv('exercise/data/sample.csv')
print(df)
```

You get access to the index and column names with:

```
df.columns
df.index
```

You can rename index and column names:

```
df = df.rename(index={0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5 : 'f'})
df.index
```

You can select rows:

```
# Select one row by its label
print(df.loc[['a']])
# Select one row by its index
print(df.iloc[[0]])

# Select several rows by labels
print(df.loc[['a','c']])
# Select one row by index
print(df.iloc[[0,2]])
```

You can select columns:

```
# Select one column by label
df['Tissue'] # Series
df[['Tissue']] # DataFrame

# Select several columns
df[['Gene','Expression_Level']]

# Select several columns by index
df.iloc[:,[0,1]]
```

You can select rows and columns as follows:

```
df.loc[['b'], ['Gene', 'Expression_Level']]
```

You can filter based on a condition as follows:

```
df[df['Expression_Level'] > 6]
```

28.1.2 Useful methods

To explore the data set, use the following methods:

```
df.info()

df.describe()

df.head()

#| eval: false

df.sort_values(by="Gene")

df['Expression_Level'].mean()

df.groupby("Gene")[['Expression_Level']].mean()
```

28.1.3 Learn More

To get more information on how to use pandas, check out:

- the documentation
- the cheat sheet
- any useful tutorial

28.1.4 Exercise

Exercise

- 1. Create a pandas DataFrame from the file containing the frequency of each nucleotide per sequences (exercise/data/example.txt).
- 2. Make sure that df.index contains the name of the sequences, and df.columns contains the nucleotides.
- 3. Use pandas.melt() (see the doc) to get the data in the following format:

```
nucl freq
Seq
seq1
            0.46
seq2
            0.20
            0.16
seq3
seq4
            0.18
            0.26
seq5
seq1
        Τ
            0.22
            0.12
seq2
. . .
```

- 4. Get the mean value of all nucleotide frequencies.
- 5. Get the mean value of frequencies per nucleotide.
- 6. Filter to remove values of seq1.
- 7. Recompute the mean value of frequencies per nucleotide.

28.2 Matplotlib

Matplotlib is a package to create visualizations in Python widely used in science.

To shorten the name of the package when we call its functions, we can import it as follows:

```
import matplotlib.pyplot as plt
df = pandas.read_csv('exercise/data/sample.csv')
# The data for GeneA and GeneB is extracted from the DataFrame 'df'
serieA = df[df['Gene'] == 'GeneA']['Expression_Level']
serieB = df[df['Gene'] == 'GeneB']['Expression_Level']
# Create a new figure
fig = plt.figure()
# Create a boxplot showing the expression levels of GeneA and GeneB
plt.boxplot([serieA, serieB], # List of series
            labels=['GeneA', 'GeneB'])
# Set the label for the x-axis
plt.xlabel('Gene')
# Set the label for the y-axis
plt.ylabel('Expression Level')
# Set the title of the plot
plt.title('Expression of Genes in Different Tissues')
# Display the boxplot
plt.show()
# Save the plot as a PNG file with a resolution of 300 dots per inch (dpi)
# The file will be saved in the specified location
fig.savefig('exercise/data/my-figure.png', dpi=300)
```

The following code is equivalent.

```
# Create a new figure
fig, ax = plt.subplots(1, figsize=(5, 4))
```

Note

The first way of plotting is function-oriented, and the second is object-oriented. You might encounter both styles of coding.

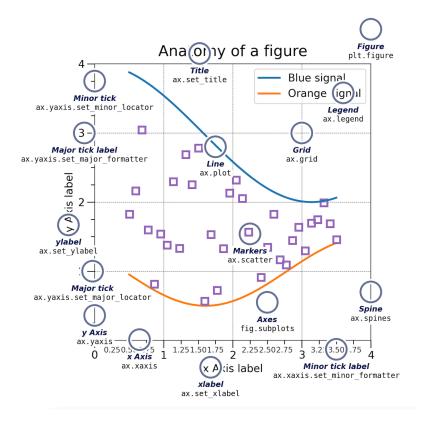


Figure 28.1: Anatomy of a matplotlib plot

Many visualizations are available (static, animated, interactive). For more information, check out:

- the documentation
- the cheat sheet
- any useful tutorial
- some inspiration

28.3 Exercise

Exercise

Create a script that gets nucleotide frequency data from a file in the format of exercise/data/example.txt, and visualizes it using Matplotlib and Pandas.

Your script should read the data, create a stacked bar chart showing the nucleotide frequencies for each sequence, and label the axes appropriately. Here's the expected plot:

28.4 More packages

There are MANY packages available, here's a short list of some that might interest you:

	TT	D. I. C.
Package	Usage	Example of usage
BioPythonComputational		Sequence handling,
	molecular biology	access to NCBI
		databases
NumPy	Numerical arrays	Data manipulation,
		mathematical
		operations, linear
		algebra
Seaborn	High-level interface for	Data visualization,
	drawing plots	statistical graphics
HTSeq	High throughput	Quality and coverage,
	sequencing	counting reads, read
		alignment
Scanpy	Single-Cell Analysis	Preprocessing,
		visualization, clustering
SciPy	Mathematical	Clustering, ODE,
	algorithms	Fourier Transforms
Scikit-	Image processing	Image enhancement,
image		segmentation, feature
		extraction

Package	Usage	Example of usage
Scikit- learn	Machine learning	Classification, regression, clustering, dimensionality reduction
TensorFlowDeep learning and Py- Torch		Neural networks, natural language processing, computer vision

29 Final tips and resources

Here are a couple of tips:

- Leave comments (think of your future self!)
- Be consistent (quotes, indents...)
- Don't re-invent the wheel, for common tasks, it's likely that a function already exists
- Read the documentation when using a new package or function!
- Google It! Use the correct programming vocabulary to increase your chances of finding an answer. If you don't find anything, try wording it differently.
- The easiest way to learn is by example, so follow a tutorial with the example data, and then try to apply it to your own!

You can follow some free tutorials on:

- Code Academy
- EdX
- Youtube!

Finally, you should able to use Github Copilot (AI coding assistant), as it is free for students: https://education.github.com/benefits.

Trying random stuff for hours instead of reading the documentation



Figure 29.1: Please, read the doc

References

A python conference organized by the AFPy (Association Francophone Python) is held in Strasbourg in the end of October 2024!

Here are some references and ressources that inspired this class:

- Python doc
- \bullet w3schools
- pythonforbiologists
- justinbois's Bootcamp

30 Lesson 3 - Conway's Game of Life

31 Introduction

31.1 Aim of the class

This last class is to pratice the notions learned on a couple of (larger) exercises.

The exercises have one provided solution, but it is not the only one. We all have a different coding style, and it evolves with time. So do not worry if you have a different solution, as long as your result is correct, that's already great!

"the best way to learn a language is to speak to natives" the guy learning Python:



Figure 31.1: pssssss

31.2 Requirements

Remembering some of lesson 1 and 2.

32 Conway's Game of Life (basic)

32.1 Instructions

Create an implementation of Conway's Game of Life in a script called conway_life_basic.py.

The game consists in initializing a 2D matrix of binary value (0 or 1), and, by following certain rules, observing its evolution at each generation.

Each value represent a cell, that can either be live (0) or dead (1).

32.2 Rules

Cells interact with their neighbors such that:

- Any live cell with fewer (<) than two live neighbors dies (as if by underpopulation)
- Any live cell with more than three (>) live neighbors dies (as if by overpopulation)
- Any live cell with two or three live neighbors lives on to the next generation
- Any dead cell with exactly three live neighbors becomes a live cell (as if by reproduction)

The new matrix created corresponds to a new generation.

The original game is played on a infinite board, but we'll implement it to a finite board. When a cell is in a corner, it has 3 neighbors. When a cell is on a side it has 5 neighbors.

32.3 Functions to create

You should create:

- a count_neighbors function that counts the number of live neighbors around a cell
- a survival function that determines if a cell survives or dies based on the rules of the game
- a generation function that generate the next generation of the game
- an animate_life function that animate the game of life

You should use basic python, and print each generation to the terminal as follows:

- live cells are represented by a *
- dead cells are represented by a .

e.g. the following 3-by-3 2D matrix has one live cell in the center:

· · · ·

You will need to initialize a matrix to begin the game, then it should run on its own for a defined amount of generations.

32.4 Optional function

As an option, you can create a function called initialize_universe to initialize the grid with one of the following seeds that have specific proprieties:

```
# Dictionary containing different seed patterns for the game
seeds = {
    "diehard": [
        [0, 0, 0, 0, 0, 1, 0],
        [1, 1, 0, 0, 0, 0, 0],
```

```
[0, 1, 0, 0, 0, 1, 1, 1],
    ],
    "boat": [[1, 1, 0], [1, 0, 1], [0, 1, 0]],
    "r_pentomino": [[0, 1, 1], [1, 1, 0], [0, 1, 0]],
    "pentadecathlon": [
        [1, 1, 1, 1, 1, 1, 1],
        [1, 0, 1, 1, 1, 1, 0, 1],
        [1, 1, 1, 1, 1, 1, 1],
    ],
    "beacon": [[1, 1, 0, 0], [1, 1, 0, 0], [0, 0, 1, 1], [0, 0, 1, 1]],
    "acorn": [[0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [1, 1, 0, 0, 1, 1, 1]],
    "spaceship": [[0, 0, 1, 1, 0], [1, 1, 0, 1, 1], [1, 1, 1, 1, 0], [0, 1, 1, 0, 0]],
    "block_switch_engine": [
        [0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0, 1, 1],
        [0, 0, 0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0],
        [1, 0, 1, 0, 0, 0, 0, 0],
    ],
    "infinite": [
        [1, 1, 1, 0, 1],
        [1, 0, 0, 0, 0],
        [0, 0, 0, 1, 1],
        [0, 1, 1, 0, 1],
        [1, 0, 1, 0, 1],
    ],
}
```

32.5 Exemple of input and output

Here's an example of input, and the desired output:

```
# Initialize by hand
universe = [
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 1, 1, 1, 0, 1, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
  [0, 0, 0, 1, 1, 0, 1, 0, 0, 0],
  [0, 0, 1, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
animate_life(universe, generations = 3, delay=0.5)
. . . * . . . . . .
. . * * . . . . . .
. . . * * * * . . .
    . * * . * * . .
. * * . . . . . . .
. * * . . . . . . .
. . . * * * * . . .
```

The same matrix can be initialized by the following (if you are doing the optional initialize_universe function):

```
# Initialize with a seed
universe = initialize_universe(universe_size = (10, 10), seed = "infinite", seed_position = (2
for row in universe:
    print(row)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

With this matrix, if the number of generations increases (>30), it should stabilize over:

32.6 Tips

- Take an exemple and do it by hand to better understand the game.
- Before jumping into coding, try to have a plan of how you will implement it all. Imagine what will be the input and output of each function.
- To visualize the evolution of the grid, you can print it, and then can clean the terminal by using os.system('cls' if os.name == 'nt' else 'clear') (you will need to import os at the beginning of your script).
- To wait between two generations you can use time.sleep(delay) (you will need to import time at the beginning of your script).

33 Conway's Game of Life (advanced)

33.1 Instructions

Create another implementation of Conway's Game of Life, with the following characteristics:

- the file parses arguments from command line (if you use argparse, you can check the help of your function by running in the terminal python conway_life_advanced.py --help)
- use pandas (or numpy as there it is only numerical data) to deal with the matrix
- use matplotlib to plot each generation, and create a final gif

Example of a final gif generated from the same universe matrix as the one from basic implementation

34 Solutions

Both solutions are available in the folder exercise/script/.

35 References