Python Introduction IMCBio

Valentine Gilbart

2025-03-19

Table of contents

1 Python introduction

1.1 Description

- Public: Biology PhD students
- Modalities: 3 x 3 hours of class

1.2 Material

Lecture	Link
Lesson 1 - Introduction	lecture/lesson-1.html
Lesson 2 - pandas	lecture/lesson-2.html
Lesson 3 - matplotlib	lecture/lesson-3.html

1.3 Old Material

1.3.1 2024

Lecture	Link
Lesson 1	lecture/2024-lesson-1.html
Lesson 2	lecture/2024-lesson-2.html
Lesson 3	lecture/2024-lesson-3.html

2 Lesson 1 - Introduction, Data types, Operators

3 Introduction

```
import os
os.getcwd()
```

```
'/home/runner/work/python-intro/python-intro'
```

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

Figure 3.1: Python logo

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.

3.3 What is Python?

Figure 3.2: Guido van Rossum

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.

```
# 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 (`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)

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]`.

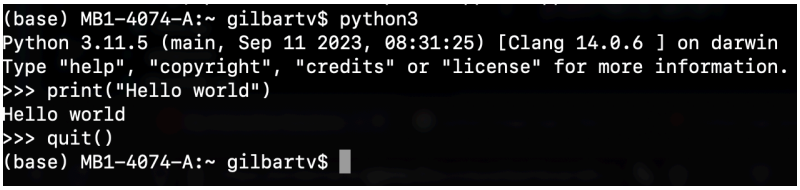

Figure 3.3: Interactive mode

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.

> **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 4.1 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
`quit()` or `exit()`	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 "Object-oriented 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.

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('+')`

> **Browse 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` ($\mathbb{C}$), 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
`.startswith()`	Returns true if the string starts with the specified value
`.endswith()`	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]
```

```
'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
`.extend()`	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

1. Create a list `l = ['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 `{}`.

```
seq = {'BRCA1', 'TP53', 'EGFR', 'MYC'}
```

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
`.remove()`	Remove the specified item
`.pop()`	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"]
```

```
'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
`int()`, `float()`	Convert between integer or float types
`complex()`	Convert to complex type
`str()`	Convert to string type
`list()`, `tuple()`, `set()`	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}
```

```
dict((('a', 1),
      ('f', 2),
      ('g', 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

> ⚠ 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
> ```
>
> ```
> 5.300000000000001
> ```
>
> 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, 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 present in the object
`not in`	Returns True if a sequence 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 not,in,not in,not`	Comparisons, identity, and 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.")
```

```
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.")
```

```
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', 'ACT', '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', 'AU
```

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)
```

```
0 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.

> **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.

```
seq = 'ACCATTTTTTGGGGGGGCGGGGGGAGGGGGGG'
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
  if i+3 > len(seq): # Get out of the loop if we parsed the full seq
    print('Codon not found in sequence.')
    break
else:
  print('The start codon starts at index', i)
```

```
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.
Analyzing sequence 'ATGGCTAGCTA' for protein coding regions...
Analyzing sequence 'ATGTAGCTAGC' for protein coding regions...
All sequences were processed.
```

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

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:

```
# 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
```

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
- w3schools
- pythonforbiologists
- justinbois's Bootcamp

8