University of Antwerp

Chemo-informatics and computational drug design

Prof. Dr. Hans De Winter

University of Antwerp

Campus Drie Eiken, Building A

Universiteitsplein 1, 2610 Wilrijk, Belgium

Gefinancierd door de Europese Unie

NextGenerationEU

Chapter 1. Introduction to Python

1. Introduction

Python is a general-purpose programming language that is very powerful yet also quite easy to learn, read and write. Many applications in Python exist, ranging from web development to machine learning and data science. Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. As Python supports modules and packages, it can be used for a variety of applications. Simply by loading different modules, the user can incorporate a wide variety of different and specific functions into its code base. Statistical, mathematical and machine learning methods are introduced by importing modules such as scikit-learn and scipy, while bioinformatics-approaches are largely governed by packages such as Biopython. Finally, plotting and data visualisation has been made easy with Python packages like Matplotlib.

To work with molecules and drugs, there exist Python packages like RDKit and OpenBabel to help you with that. In fact, as almost all molecule manipulations can be done with Python in combination with a package like RDKit, it is important for the interested chemo-informatician to learn and use Python. For this reason, this course will start with a short introduction on Python. In subsequent chapters, key concepts of the chemo-informatics package RDKit will be introduced.

2. Installing Python

Python can be installed and run from almost every computer, be it a Mac, Windows or Linux machine. There are even implementations of Python on the small Raspberry Pi systems, illustrating how widespread and powerful Python really is.

Python comes in two main flavors, namely version 2 and version 3+. The former has been discontinued since the beginning of 2020, so we will only focus on version 3+ (3.7, 3.8, and 3.9). In order to distinct between both flavors, the name Python2 is often used to denote Python version 2, while the name Python3 referes to Python version 3+. In this course, we will refer to Python 3+ when talking about Python as such.

Installation of Python can be achieved in many ways, but probably the easiest ways are through means of the Anaconda installer for a local installation of Python on your computer, or by using the Google Colab interface for a web-based usage of Python without the need to install software on your system. In the following sections, both approaches are illustrated.

2.1. Anaconda

Anaconda is a free, easy-to-install package manager, environment manager, and Python distribution with a collection of many open source packages. Installation of Anaconda is first required, and subsequently many Python packages can be installed in their own environments, making it safe to try-out different Python packages without introducing the risc that packages might conflict and break already installed packages.

An extensive and well-writing installation manual is available from the Anaconda documentation portal. This installation manual describes the required steps to take for insalling on Windows, macOS and Linux systems.

2.2. Google Colab

Perhaps the easiest way to learn Python is with the Google Colab application. This is a web-based application that allows one to write and execute Python code in your own browser and with no installation or required configuration settings. In order to use Google Colab for the first time, you can look at this nice tutorial and follow the different steps.

3. Python concepts

3.1. Importing modules

One of the powerful aspects of Python is its use of modules. With modules, the user can select what kind of functionality is needed by importing the corresponding modules that contain the required functionalities. Python modules are imported using the `import` statement:

```
>>> import math
>>> math.sqrt(25)
5.0
>>> math.sin(math.pi)
1.2246467991473532e-16
```

In this example, the `math` module is imported. This module contains many mathematical functions and constants (such as `math.pi`).

It is also possible to import only specific functions from the module, as illustrated in the following example in which only the `sqrt` function is loaded:

```
>>> from math import sqrt
>>> sqrt(25)
5.0
>>> math.sin(math.pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

One may of course also import multiple functions at once:

```
>>> from math import sqrt, sin, pi
>>> sqrt(25)
5.0
>>> sin(pi)
1.2246467991473532e-16
```

Aliases (or synonyms) to the imported modules may be provided as desired:

```
>>> import math as mathematics
>>> mathematics.pi
3.141592653589793
```

To know which functions and constants are defined in the spcified module, one can use the `dir()` function:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

3.2. Basic operations

Basic operations in Python include addition, substraction, multiplication and division. The exact effect of each of these operations depends on the underlying data type (see below), but in the case of numbers as data types the results are as expected:

```
>>> 10 + 4
14
>>> 10 - 4
6
>>> 10 * 4
40
>>> 10 ** 4
10000
```

```
>>> 10 / 4
2.5
>>> 5 % 4
1
>>> 10 // 4
2
```

Python works also with `booleans`, which are data types that can have the value of `true` or `false`. Boolean operations include comparisons such as > (larger than), >= (larger or equal than), < (smaller than), <= (smaller or equal than) and == (equal) or != (not equal). In addition, there exist also `and`, `or` and `not` as boolean operations. All data types can automaticallly be converted to a `boolean` type:

```
>>> 5 > 3
True
>>> 5 < 3
False
>>> 5 != 3
True
>>> 5 >= 3 and 10 > 3
True
>>> 5 >= 3 and not 10 > 3
False
>>> True
True
>>> not False
True
```

3.3. Data types

Standard Python contains already a multitude of data types, and these can be extended by the programmer using the `class` data type (below more on classes). The common data types in Python are integer numbers (`int`), fractional numbers (`float`), booleans (`bool`), text (`str`) and the `None` type. The type can be determined with the `type()` function:

```
>>> type(2)
<class 'int'>
>>> type(2.4)
<class 'float'>
>>> type(“two”)
<class 'str'>
>>> type(‘two’)
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```

In order to find out if a given object is of a given type, the `isinstance()` function is useful:

```
>>> isinstance(2.0, int)
False
>>> isinstance(2, int)
True
>>> isinstance(2, float)
False
>>> isinstance(2.0, float)
True
>>> isinstance(2.0, (float, int))
True
```

Objects often can be converted to other types (not always, as shown in the last example):

```
>>> int(2.1)
2
>>> float(2)
2.0
>>> str(2.9)
‘2.9’
>>> int(“two”)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'two'
```

With respect to conversion to a `boolean` type, `0`, `None` and empty containers (see below) are all converted to `False`, and all the other objects are converted to `True`:

```
>>> bool(0)
False
>>> bool(0.00000001)
True
>>> bool(None)
False
>>> bool(“”)
False
>>> bool(1)
True
```

3.4. Variables

Variables in Python are placeholders that can contain certain values. These values are assigned to the variables:

```
>>> a = 3
>>> a
3
>>> b = 7
>>> a + b
10
>>> b ** a
343
>>> b * 2
14
>>> c = a + b
>>> c
10
>>> a = 1
>>> a = 2
>>> a = 3
>>> a
3
>>> a = “another value into a”
>>> a
'another value into a'
```

3.5. Lists

As the name suggests, lists are lists of ordered and multiple data types. Lists are denoted with `[]`, in which `[` denotes the start of the list, and `]` the end. Elements can be added to the list using the `append()` method:

```
>>> my_list = []
>>> my_list
[]
>>> my_list.append(3)
>>> my_list
[3]
>>> my_list.append(4)
>>> my_list
[3, 4]
>>> another_list = ["This", "are", "five", "list", "elements"]
>>> another_list
['This', 'are', 'five', 'list', 'elements']
```

List elements can be accessed with their index (starting at zero). The last element can be accessed with the -1 index and the total number of elements is retrieved with the `len()` function:

```
>>> another_list[0]
'This'
>>> another_list[2]
'five'
```

```
>>> another_list[-1]
'elements'
>>> len(another_list)
5
```

Individual elements can be removed from a list using the `remove()` and `pop()` methods:

```
>>> another_list.remove(‘list’)
>>> another_list
['This', 'are', 'five', 'elements']
>>> another_list.pop(1)
'are'
>>> another_list
['This', 'five', 'elements']
```

Lists can be sorted:

```
>>> another_list.sort()
>>> another_list
['This', 'elements', 'five']
```

There are many more functions and methods that can be used in conjunction with lists. For the interested student, please have a look at the W3 Schools webpage on lists.

3.6. Tuples

Tuples are like lists, but tuples cannot be modified with methods like `append()` or `remove()`. Tuples are initiated and remain then constant:

```
>>> digits = (1,2,3)
>>> digits
(1, 2, 3)
>>> digits[1]
2
>>> len(digits)
3
>>> digits * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> digits.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

3.7. Strings

Strings are sequences of characters. They share some similarities with lists such as accessing elements with indices and the `len()` method:

```
>>> txt = “this is a string”
>>> txt[1]
'h'
>>> txt[1:4]
'his'
>>> len(txt)
16
```

In addition, strings also possess some methods that are typical for text:

```
>>> txt2 = txt.upper()
>>> txt2
'THIS IS A STRING'
>>> txt2.lower()
'this is a string'
>>> txt.split()
['this', 'is', 'a', 'string']
```

The W3 Schools webpage on strings provides examples of many other methods related to strings.

3.8. Dictionaries

Dictionaries are structures which can contain multiple data types which are ordered as key-value pairs. For each (unique) *key*, the dictionary contains an associated *value*. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object:

```
>>> d = {}
>>> d
{}
>>> d = dict()
>>> d = {1: "one", "two": 2, 3: [1,2,3,4], "four": None}
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None}
```

The dictionary elements can be accessed as key/value pairs:

```
>>> d[1]
'one'
>>> d[105] = "OneHundredAndFive"
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None, 105: 'OneHundredAndFive'}
>>> d[105] = None
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None, 105: None}
```

As already shown in the previous example, dictionary elements can be added by providing the key/value pair using square brackets. Deletion of dictionary elements is possible with the `del()` function:

```
>>> del d[105]
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None}
```

For more information, see the W3 Schools webpage on dictionaries.

3.9. Sets

Sets are containers very similar to dictionaries, but sets have only the unique keys and no values. Sets can be used to generate lists of unique elements:

```
>>> l = set()
>>> l.add(1)
>>> l.add(2)
>>> l.add(3)
>>> l.add(3)
>>> l.add(3)
>>> l
{1, 2, 3}
>>> l.remove(3)
>>> l
{1, 2}
```

For more information, see the W3 Schools webpage on sets.

3.10. Control statements

Conditional statements

Conditional statements (`if`, `else`, and `elif`) control the flow of a program depending on the outcome of certain conditions. These conditions are evaluated and depending on the outcome (`True` or `False`) a certain flow direction is followed:

```
>>> x = 3
>>> if x == 3:
...     print("Three")
... elif x == 4:
...     print("Four")
... else:
```

```
...     print(“Not 3 and not 4”)
...
Three
```

Loops

Repetitive processes are implemented using loops. There are two kinds of loops, namely the `for` loop and the `while` loop. Let’s start with the `for` loop to illustrate how one can iterate over all elements in a list:

```
>>> fruits = ["apple", "lemon", "strawberry"]
>>> for fruit in fruits:
...     print(fruit)
...     print(fruit.upper())
...
apple
APPLE
lemon
LEMON
strawberry
STRAWBERRY
```

The `range()` keyboard may come in handy in this context. With this keyword, a range of numbers is generated. For example, `range(10)`, which is equal to `range(0,10)`, returns a list of 10 integers ranging from 0 to 9 inclusive:

```
>>> for i in range(10):
...    print(i)
...
0
1
2
3
4
5
6
7
8
9
```

This feature may be exploited to loop over a list using list indices:

```
>>> for i in range(len(fruits)):
...     print(i, fruits[i])
...
0 apple
1 lemon
2 strawberry
```

The second keyword for looping is `while`. This keywords evaluates the expression given as argument, and as long as this experession converts to `True`, the block contained by the `while` keyword is executed:

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i += 1
...
1
2
3
4
```

Breaks and continuation

To interrupt a loop, the `break` keyword may be used:

```
>>> for i in range(5):
...     print(i)
...     if i == 3:
```

```
...             break
...
0
1
2
3
```

When the `break` keyword is encountered, the loop terminates and the program flow moves out of the current loop. This is comparable to the `continue` keyword, however in that case the program continues at the beginning of the current loop:

```
>>> for i in range(5):
...     if i == 1 or i == 3:
...             print(i)
...     else:
...             continue
...
1
3
```

3.11. Functions

Functions are sets of instructions grouped together. Functions are launched when called, and they can have multiple input values (called the arguments) and a single return value. Functions come in handy when the same set of instructions has to be repeated on different vaiables or values. They are defined using the `def()` keyword:

```
>>> def sum(x,y):
...     s = x + y
...     return s
...
>>> sum(2,3)
5
>>> sum(100,4)
104
```

In the preceding example, a `sum()` function is defined that takes two arguments and returns a single value. It is also possible to define default values for the input arguments:

```
>>> def sum(x=1,y=2):
...     s = x + y
...     return(s)
...
>>> sum()
3
>>> sum(4,5)
9
```

Functions are defined and used extensively in Python, and form the basis of many packages that may be imported in your Python code. For example, the `math.sqrt()` that was mentioned on page 8 is actually a function that takes a single argument and returns the square root of that argument.

Although a function can only return a single value by definition, it is possible to circumvent this limitation by returning a tuple, list or dictionary:

```
>>> def sum_substraction_multiplication(a, b):
...     return(a+b, a-b, a*b)
...
>>> a = sum_substraction_multiplication(2,3)
>>> a
(5, -1, 6)
>>> print("2 + 3 = ", a[0])
2 + 3 =  5
>>> print("2 - 3 = ", a[1])
2 - 3 =  -1
>>> print("2 * 3 = ", a[2])
2 * 3 =  6
```

3.12. Reading and writing files

Writing to text files is achieved using a three-steps procedure: 1) open the file, 2) write to the file, 3) close the file:

```
>>> f = open("test.txt", "w")   # 'w': write to the file
>>> f.write("Hello\n")
6
>>> f.write("Line 2\n")
7
>>> f.write("\n")
1
>>> f.write("\n")
1
>>> f.write("\n")
1
>>> f.write("Last line\n")
10
>>> f.close()
```

This code generates a new file with filename `test.txt`. Should the `text.txt` file already exists, then this existing file is overwritten by the new data. This file contains six lines:

```
Hello
Line 2



Last line
```

Reading a file is also done in three steps: opening, reading, and closing:

```
>>> f = open("test.txt", "r")   # 'r': open the file to read
>>> for line in f.readlines():
...     print(line)
...
Hello

Line 2







Last line

>>> f.close()
```

It is common practice to remove any newline characters at the end of each line [using the `strip()` method], and to skip over empty lines:

```
>>> f = open("test.txt", "r")   # 'r': open the file to read
>>> for line in f.readlines():
...     line = line.strip()
...     if line is None or line == "": continue
...     print(line)
...
Hello
Line 2
Last line
>>> f.close()
```

3.13. Further reading

Python is a rich scripting language with many more functions and options than can ever be covered in this short chapter. Google is a perfect source of information to search for Python functions that you might need (for example, “how to read in text files in python”). In addition, there exist many tutorial sites on the internet, such as W3 Schools for Python.