Introduction to python™

Xavier Gendre

October 7, 2016

Contents

1 Preliminaries

1.1 What is Python?

Python is a widely used programming language initially implemented in 1989 by Guido van Rossum who is still playing an important role in the Python community. Its reference implementation, called *CPython*, is managed by the Python Software Fundation and distributed according to the Python Software Foundation License. This licence is compatible with the *GNU General Public License* and approved by the *Open Source Initiative*, thus *CPython* is free and open-source software.

To run Python code, a specific software called *interpreter* is needed. Such softwares are available for many operating systems, allowing Python code to run on a wide variety of systems. Commonly, installing Python amounts to get and install such an interpreter from the Download page of the project.

The Python philosophy is summarized in a sequence of aphorisms known as PEP 20. It gives a great importance to code readability and to the capacity to express concepts in few lines of code. The ability to produce good code in the sense of this philosophy comes with reading and writing code, a lot of code! Anyone who wants to produce Python code has to read the PEP 8 and to deeply thinks about the advices given in this document.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

1.2 About Python version

Nowadays, two distinct versions of Python coexist, the version 2 and version 3. These versions are **not compatible**. This is an important point to keep in mind, especially when help or code examples are searched on the Web. A list of the main incompatibilities (and problems to solve when porting code from Python 2.x to Python 3.x) can be found in an article of Guido van Rossum called What's New In Python 3.0.

The last release of version 2 is Python 2.7 and it was announced in November 2014 that this is the last one, no 2.8 will be released in the future. This version is supported until 2020 but the users are deeply encouraged to move to version 3 as soon as possible.

The current stable release is Python 3.4 and the development version is 3.5. Some attention should be paid to that when documentation pages are browsed. In the sequel of this document, Python 3.4 is silently assumed in all the examples.

1.3 Where to find help?

An undeniable force of Python language is its large community. A lot of tutorials, forum threads, articles, . . . can easily be found on the Web. Again, it is important to ensure version compatiblity when pieces of code are found in such a way.

Official documentation is an endless source of information about the language and its ecosystem. By default, the documentation is the one for the development version of Python (currently, version 3.5.2). In the top left corner of the page, a selector allows to choose what is the required version. Picking `3.4` leads to documentation for the latest stable release, namely version 3.4.5.

As usual with programming languages, a lot of questions are answered in stackoverflow. A good advice when some problem is encountered is to first browse these pages before posting elsewhere.

1.4 Working with Python

As explained previously, to run Python code, an interpreter is needed. With a common installation of Python 3, two softwares are directly at your disposal. The first one is the basic interpreter and can be called from a simple console by the command `python3`. The other one is the software *IDLE* and can be started by `idle3`. Note the number `3` at the end of each command, it indicates that version 3.x is needed, not version 2.x.

IDLE is a simple IDE for Python that provides a window containing the basic interpreter and several usefull features (script edition, debugger, ...). Whatever the software, the interpreter gives some informations about itself:

```
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

After the Python version, some details are given about how the interpreter has been compiled and first instructions to type are proposed. Triple angle brackets `>>>` indicates the Python prompt after which instructions can be typed. In the following example, the function `copyright` is executed (more explanations about functions will be given later):

```
>>> copyright()
Copyright (c) 2001-2014 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

```
>>>
```

Under particular circumstances, Python instructions spread over several lines, the prompt becomes three dots `...` to point out that the instruction is continuing. In such situations, the number of white spaces at the beginning of the line is crucial and an empty line ends the instruction (more explanations will be given soon):

```
>>> if 2 > 1:
...     print(42)
...     print('Of course, 2 > 1!')
...
42
Of course, 2 > 1!
>>>
```

In this example, the function `print` prints the integer `42` and the string `'Of course, 2 > 1!'` to the standard output followed by a carriage return and returns the focus to the user at the end.

Although this is not considered as a good practice, it is possible to run multiple instructions in one line by separating them with `;`. Thus, this example is equivalent to the previous one:

```
>>> if 2 > 1:
...     print(42); print('Of course, 2 > 1!')
...
42
Of course, 2 > 1!
>>>
```

1.5 Working with Spyder

Working with the raw interpreter only is sufficient to run scripts but is a bit austere and several tools should be welcome. As discussed above, the software IDLE offers some useful features but remains stern. There exist a lot of IDE for Python and we introduce Spyder in this section. By installing this software, pay attention to take a version for Python 3.

Here are listed some reasons justifying this choice:

- Spyder is a free software (MIT License),
- Spyder is cross-platform (GNU/Linux, Mac OSX, Microsoft Windows, ...),
- Spyder supports Python 3,
- Spyder is an active project.

Spyder main window is organized as follows : at the top, some menus and buttons to interact with the software; on the left, the script editor; on the right, an interface to get some help and a console.

Let us start with the console, you should note that the content is similar to the one of the interpreter but with a different prompt `In [1]`. This Python interpreter is called IPython and provides more tools than the basic interpreter. From a practical point of view, IPython allows to run Python code in the same way as the classic interpreter and we do not introduce its specific tools in this document. In particular, you can directly type some instructions in IPython console as we did above:

```
In [1]: copyright()
Copyright (c) 2001-2014 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.

In [2]: if 2 > 1:
   ...:     print(42)
   ...:     print('Of course, 2 > 1!')
   ...:
42
Of course, 2 > 1!

In [3]:
```

If you want to keep a classic Python console, you could open a new one through the menu bar.

Above the console, you find an interface to get help. To use it, put your cursor over an instruction (*e.g.* the word `print` in the console) then press `Ctrl+I`. The content of the panel changes to show some informations. The documentation is not as rich as in the official documentation but it can sometimes be useful. Other tabs of this panel allow you to see the variables currently defined (see explanations further) and to browse the file system.

On the left of the window, there is the script editor and this is where you are going to spend a lot of time. Fortunately for us, all the Python instructions do not have to be typed in a console and it is possible to pass a bunch of code to the interpreter via a text file called a *script*. Such a file commonly begins with a particular comment (in Python, a comment starts with the character #) to specify the encoding. After this comment, you also could find a string surrounded by triple quotation marks """. This string is known as a *docstring* and is for documentation (see explanations further). These comment and docstring are not mandatory but this is a good habit to put them. Python code comes after these two lines.

For instance, our examples in a file encoded with UTF-8 (you do not want other encoding, isn't it?) and a small docstring, could be like that:

```
#  -*- coding: utf-8 -*-
"""Useless docstring"""

copyright()
if 2 > 1:
    print(42)
    print('Of course, 2 > 1!')
```

In the sequel of this document, we omit the two first line in the examples to only show the content of the Python code.

To run the whole script, press F5 or use the green arrow button in the top icon bar. If you only want to run a piece of code, highlight it and press Ctrl+Enter. You also can run only the active line by pressing F9. In both case, the code is passed to the active interpreter. It is also possible to divide the code in subsections with the comment line #%% and to run each section aside with Shift+Enter. This last way of writing is comfortable when learning and doing exercises because it easily allows you to reload specific pack of lines. To split our two examples and run them one after the other, we could organize our script in the following way:

```
#  -*- coding: utf-8 -*-
"""Useless docstring"""

# %%

copyright()

# %%

if 2 > 1:
    print(42)
    print('Of course, 2 > 1!')
```

A lot of other features of Spyder could be introduced but such an exhaustive review is beyond the scope of this document.

2 A programming language

2.1 Variables

A *variable* is a symbolic name associated with a value. In Python, the name of a variable has to follow several rules:

- it must start with a letter or an underscore,
- remainder characters may consist of letters, numbers and underscores.

Valid variable name examples are `my_var`, `_var`, `_MyVar01`, `x42_`, . . . Moreover, like everything else in Python, variable names are **case sensitive**, *i.e.* `myVar`, `MyVar` and `mYvaR` are three distinct variable names.

To assign a value to a variable, we use the operator `=` with the variable name on the left and the value to assign on the right. If the variable does not yet exist, it is created. Otherwise, its value is simply modified.

```
a = 17
print(a)
a = 8
print(a)
```

If you try to use a variable that is not defined, you will get an error:

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

The mechanism to handle errors in Python will be cover later in this document.

The content of a variable has a *type*, *i.e.* a category that determines the possible values. Usual types are boolean, integer, real number, string, . . . To know the type of a variable, we use the function `type`:

```
x = True; type(x)
x = 42; type(x)
x = 42.0; type(x)
x = 'Hello World!'; type(x)
```

The properties of common types will be introduced in the next section.

This is important to bear in mind that Python has typed objects but untyped variable names. It means that the content of the variable has a type, not the variable itself. This is why we could assign contents from various types to `x` in the previous example without any trouble. In Python, the assignment of a value to a variable amounts to make the variable names pointing to a specific location in the memory which contains the value itself. This location in the memory is called an *address* and assigning a value with a different type to a variable simply consists in making this address pointing elsewhere in the memory. To get the address, we use the function `id`:

```
x = True; id(x)
x = 1234; id(x)
```

If the same value is assigned to two distinct variable names, both of them can point to the same address, avoiding data replication. To test this mechanism, we can use operators `is` and `is not` to for object identity:

```
x = 42; id(x)
y = 42; id(y)
z = 43; id(z)

x is y
x is not z
```

When a variable name is no longer needed, it is possible to break the link with a memory location and to delete it with the operator `del`:

```
x = 'I exist!'
print(x)
del x
print(x)
```

2.2 Standard types

2.2.1 Boolean

The type `bool` can only take two values: `True` and `False`. Operators `not`, `and` and `or` allow to combine them to produce other boolean values.

```
x = True; type(x)
print(x)
print(not x)

y = False
print(x and y)
print(x and not y)
print(x or y)
print(not x or y)
```

Booleans mainly appear in conditional structures (see explanations further) to test some conditions. Nevertheless, in Python, a lot of not boolean values are considered false:

- special value `None` which is a null value, *i.e.* a value assigned to a variable name only to make it exist,
- zero numeric values `0`, `0.0`, ...,
- any empty sequence `''`, `()`, `[]`, ... (see further),
- empty mapping `{}` (see further),
- any class with a `__bool__` method which returns `False` (see further).

All other values are considered true, so objects of many types are always true.

Some operators also produce boolean values:

- `==` (equal)
- `!=` (not equal)
- `<` (strictly lower than)
- `<=` (lower than or equal)
- `>` (strictly greater than)
- `>=` (greater than or equal)
- `in` and `not in` (belong operator, see further)
- `is` and `is not` (object identity)

```
'Hello' == 'hello'
'aaa' <= 'aab'
17 > 8
```

2.2.2 Numeric types

Three distinct numeric types are at our disposal : `int` for integers, `float` for floating point numbers and `complex` for complex numbers. Floating point numbers are usually implemented using double precision and complex numbers have a real and imaginary part, which are each a floating point number. To produce an imaginary number, the character `j` or `J` has to be appended to a numeric literal.

Common operations on numeric types are available in Python as illustrated in the following examples. When an operation between two numeric types makes sense, note that the type of the result is the more widened (*e.g.* adding a floating and a complex number leads to a complex number).

```
# Various numeric types
x = 5; type(x)
y = 3.14159; type(y)
z = 2+3j; type(z)

# Explicit type
int(x)
float(x)
complex(x,y)

# Complex numbers
print(z.real) # Real part of z
print(z.imag) # Imaginary part of z
print(z.conjugate()) # Conjugate of z

# Common operations
x + y # Sum of x and y
x - y # Difference of x and y
x * y # Product of x and y
x / y # Quotient of x and y
x // y # Floored quotient of x and y
x % y # Remainder of x / y

# Assignment operations
x += 42 # Update x by adding 42 (same as x = x + 42)
x -= 42 # Update x by subtracting 42 (same as x = x - 42)
x *= 2 # Update x by multiplying by 2 (same as x = x * 2)
# Idem for other common operations...

# Useful operations
abs(-x) # Absolute value for an integer or a floating number
abs(z) # Magnitude for a complex number
```

```
pow(x, 3) # x to power 3
x ** 3 # Idem, x to power 3

round(y) # y rounded to closest integer
round(y, 2) # y rounded to 2 digits
```

2.2.3 Sequence types

In Python, sequence types can be *mutable* (*i.e.* values contained in the sequence can be modified) or *immutable* (*i.e.* values contained in the sequence can **not** be modified). The basic sequence types are introduced in this section. Additional ones are described further and offer the same set of operations according to their mutability.

Tuples are immutable sequences and are delimited by a pair of parentheses `()`. This sequence type can contains values with different types and square brackets `[]` are used to access items and for slicing.

```
t = () # Empty tuple
t = (17, 8.0, 'Hello') # Tuple can contains different types

t[0] # First item has index 0
t[2] = 81 # Error because t is immutable
t[-1] # Last item

t = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H')
t[3:5] # Slice of t from 3 to 5 excluded
t[:5] # Slice of t from start to 5 excluded
t[3:] # Slice of t from 3 to end
t[1:6:2] # Slice of t from 1 to 6 excluded with step 2

t[-1] # Negative index starts with the last item ...
t[-1:-6:-2] # ... and you can slice!
```

Examples below list several common operations that we can do with tuples. They remain available for lists introduced further.

```
t = (17, 8, 19, 81, 19)

len(t) # Length of t
min(t) # Smallest item of t
max(t) # Largest item of t
```

```
17 in t # Belong operator
17 not in t # Don't belong operator

t + ('Hello', 'Python') # Concatenation of tuples
t * 3 # Concatenate 3 times t

t.count(19) # Number of occurences in t
```

When it is not muddled, surrounding parentheses of a tuple can be omitted. This trick is mainly used to simultaneously assign values to several variables or to elegantly swap the contents of variables.

```
a, b, c = 17, 8, 'Hello'
print(a)
print(b)
print(c)

a, b = b, a # Smart, isn't it?
print(a)
print(b)
```

Lists are mutable sequences and are delimited by a pair of square brackets `[]`. Accessing items and slicing work like with tuples.

```
l = [] # Empty list
l = [17, 8.0, 'Hello'] # List can contains different types

l = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
l[2]
l[3:5]
```

Besides the operations from the immutable sequences, lists offer operations that modify itself.

```
l = ['Python', 'is', 'quite', 'fun!']
l[2] = 'very' # Replace item at position 2 of l by 'very'
del l[2:3] # Delete items in the slice

s = ['P', 'Y', 'T', 'H', 'O', 'N']
s.sort() # Sort the list in increasing order
s.reverse() # Reverse the items of s
```

```
s.remove('T') # Remove the first item containing 'T' from s
s.pop() # Retrieve the last item and remove it from s
s.pop(1) # Idem for item at position 1
s.clear() # Removes all items from l (same as del l[:])
s.append(42) # Appends 42 to the end of the list

s.extend(l) # Extend s with the items of l
s += l # Idem, extend s with the items of l
s *= 3 # Update s with its contents repeated 3 times
s.insert(5, 'Hello') # Insert 'Hello' into s at the position 5
```

Finally, *ranges* are immutable sequences of numbers. They are commonly used for looping (see further). The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents. Beyond that, a `range` object is handled like any immutable sequence.

```
range(10) # A range object from 0 to 9
list(range(10)) # Same object as a list
list(range(1, 11)) # From 1 to 10
list(range(0, 10, 3)) # From 0 to 9 with step 3
list(range(0, -10, -1)) # From 0 to -9

# Ranges are handled like other sequences
r = range(0, 20, 2)
11 in r
10 in r
r[5]; r[:5]; r[-1]
```

Questions

- What do you get with `t[:-2]` and `t[-2:]`?
- Define `t = (17, 8, 'Hello')`, explain what you obtain with `min(t)`.
- Read the documentation about the function `sort`. How can we sort a list in decreasing order only using `sort`?
- Create the list `[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]` in one line with `range` objects.
- Be careful with a **list of lists** when you fill its content:

```
s = ['Bobby', 'Teddy']
t1 = [s, s, s]
print(s); print(t1)
t1[0][0] = 'Billy'
print(s); print(t1) # (***)
s = 'Cindy'
print(s); print(t1) # (***)

s = ['Bobby', 'Teddy']
t2 = [s[:], s[:], s[:]] # With slicing
print(s); print(t2)
t2[0][0] = 'Billy'
print(s); print(t2) # (***)
s = 'Cindy'
print(s); print(t2) # (***)
```

Compare what you obtain at each line tagged with `(***)` and explain such a behavior. In `t1`, the list `s` is passed by *reference* and, in `t2`, the list `s` is passed by *value*. You have to pay attention to how you handle your data in Python.

2.2.4 Text sequence type

Objects of type `str`, also called *strings*, are used to handle textual data. Strings are immutable sequences of Unicode code points. They can be surrounded by simple quotes `''`, double quotes `""` or triple quoted with `'''` or `"""`. Triple quoted strings may span multiple lines.

```
s = 'Hello'; type(s)
s = "Hello"; type(s)

s_multi = '''A multiple
lines string'''

# Strings are handled like other immutable sequences
s[2]
s[1:3]
s[0] = 'h' # Error, strings are immutable
s + s
s*3

# Common types can be casted to string
str(42)
str([17, 8])
```

Type `str` provides a lot of operations to manipulate strings and it would be beyond the scope of this document to make an exhaustive review. For more details, you should refer to the official documentation. Hereafter, we give examples of some of the most useful operations with strings.

```
s = 'Hello, Python charmers! How are you?'

s.startswith('Hello')
s.endswith('you')

s.find('Python') # Get the first occurence
s.count('ar')

s.split() # Split s on spaces
s.split('ar') # Split s on occurences of 'ar'
s.partition('How')
s.replace('Hello', 'Good morning')

' '.join(['Python', 'is', 'fun!']) # Initial string is the separator
'   Hey!  '.strip() # Remove trailing spaces

# String formatting
'The sum of 1 + 2 is {}'.format(1+2)
'The sum of 1 + 2 is {} and the product 1 * 2 is {}'.format(1+2, 1*2)
'{1} created {0}'.format('Python', 'Guido') # Order is important
'{0} {1} {0}'.format('***', 'Go Python!') # We can reuse arguments
'Name: {first} {last}'.format(first='John', last='Doe')
```

Questions

- How to test if the content of a string is a number?
- How to find the last occurence of a substring in a string?

2.2.5 Mapping types

Python provides only one mapping type, the *dictionaries*. Such objects are mutable and offer a map between (quite common) values to arbitrary objects. To handle dictionaries, we use a syntax similar to the lists.

```
# Simple dictionary creation
a = {'one': 1, 'two': 2, 'three': 3}; type(a)
# Useful zip function
b = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
b.keys() # Get the keys
b.values() # Get the values
```

```
# These dictionaies are equal
a == b

# Similar to mutable objects
len(a)
a['two']
a['three'] = 33
a['four'] = 4 # Append on the fly
b.clear() # Remove all items

# Playing with keys
1 in a
'one' in a
a['five'] # Error, the key doesn't exist
a.get('five', 'Default value') # Safer...

# Nice with tuples
grid = dict() # Empty dictionary
grid[1,3] = 'Player'
grid[2,6] = 'Bomb'
```

2.2.6 Other types

There also exist some special types that play specific roles in Python. Among them, we should mention `None` for the null object and `NotImplemented` which could be returned when some operation is not supported. These types have to be handled with `is` and `is not` operators.

We just introduced the more commonn types in Python. The interested reader will find a lot of additional informations in the official documentation.

2.3 Conditionals

We use the word *conditional* for code that is run according to some boolean value. Such a thing is based on a `if` statement:

```
a = 17
if a > 0:
    print('a is positive')
```

It is **crucial** to understand how this piece of code is structured. The `if` statement ends with the character `:` and the next line has to start with some spaces. If the condition is true, this line is run. If the condition is false, the line is ignored and the run continues. You can choose the number of spaces at the beginning of the line but it has to be strictly positive. These spaces delimit the bunch of code to run if the condition is true and their number must remain the same along all the line of the conditional block.

```
a = 17
if a > 0:
print('a is positive') # Error, no space

a = 17
if a > 0:
    print('a is positive')
    print('Yeah!') # In the conditional block

a = -42
if a > 0:
    print('a is positive')
print('Yeah!') # Out of conditional block

a = 17
if a > 0:
    print('a is positive')
  print('Yeah!') # Error, incorrect number of spaces
```

All instruction blocks in Python have to follow these rules!

Let your IDE help you!

Now, we know how to write an instruction block, we can construct richer conditionals by running some block if the condition is true and an other block otherwise. Such a structure is known as a `if ... else` statement.

```
a = 17
if a > 0:
    print('Condition is true')
    print('a is positive')
else:
    print('Condition is false')
    print('a is nonpositive')
```

Quite often, we use a conditional to give a value to a variable according to some condition:

```
a = 42
if a % 2 == 1:
    val = 'Odd number'
else:
    val = 'Even number'
```

```
print(val)
```

Such a code is a bit long and to reduce its size, Python offers *inline conditional*:

```
# This example is equivalent to the previous one
a = 42
val = 'Odd number' if a % 2 == 1 else 'Even number'
print(val)
```

An `if ... else` statement is adapted to test a single condition but it becomes less easy when multiple nested conditions are needed:

```
a = 17
mod3 = a % 3
if mod3 == 0:
    print('We can divide {} by 3'.format(a))
else:
    if mod3 == 1:
        print('The remainder of {} by 3 is 1'.format(a))
    else:
        if mod3 == 2:
            print('The remainder of {} by 3 is 2'.format(a))
        else:
            print('This eventuality should be impossible')
```

To produce a code easier to read and test the various conditions one by one, we can deal with a `if ... elif ... else` statement. So, the previous example can elegantly be rewritten as follows:

```
a = 17
mod3 = a % 3
if mod3 == 0:
    print('We can divide {} by 3'.format(a))
elif mod3 == 1:
    print('The remainder of {} by 3 is 1'.format(a))
elif mod3 == 2:
    print('The remainder of {} by 3 is 2'.format(a))
else:
    print('This eventuality should be impossible')
```

Questions

- Being given some variable `v`, write a conditional to test if `v` is `None` or not.
- Being given two strings `s1` and `s2`, write a conditional to test if they have the same length.
- For some numeric value `v`, write a `if ... elif ... else` statement that prints different messages according to the positivity, nullity or negativity of `v`.

2.4 Loops

Looping means repeating a bunch of code until some condition becomes false. The first statement to do it with Python is based on `while`. Such a loop starts with `while` and a condition and repeats the following block of code while this condition is true (potentially *ad nauseam* if the condition never change):

```
# Syracuse Sequence
u = 5; print(u)
while u != 1:
    u = u // 2 if u % 2 == 0 else 3*u + 1
    print(u)

# Square
n = 0
while n < 8:
    print(n, '#' * 8)
    n += 1
```

The second example corresponds to a so common loop and there is a specific statement for that. Indeed, `n` is initialized to `0` and while `n` still not reach the value `8`, we increment it by one and `n` takes all the values `0`, `1`, ..., `8`. This loop can be summarized with a `for` statement:

```
# Equivalent to previous 'Square' example
for n in range(8):
    print(n, '#' * 8)

# Useful to browse a sequence from indices
t = ('', '***', '* *', ' * ')
for i in (1, 2, 1, 0, 2, 3, 2, 0)*3:
  print(t[i])
```

Actually, a `for` statement can be used with a lot of objects, called *iterables*, to browse one by one all their items. Among the iterable objects (see further for other examples), we have already introduced some sequence types:

```python
# With a tuple
t = ('I', 'am', 'a', 'tuple')
for item in t:
    print(item)

# With a list
l = ['I', 'am', 'a', 'list']
for item in l:
    print(item)

# With a range
for i in range(2, 10, 3):
    print(i)

# With a string
s = 'How are you, Pythonistas?'
for c in s:
    print(c)

# With a dictionary
d = {'one':1, 'two':2, 'three':3}
for k in d:
    print(k)
```

The example with a dictionary needs some clarifications. As you can notice, the keys are browsed, not the values. We could use the square brackets `[]` to get the values or use `d.keys()` and `d.values` but we also could use `d.items()` iterate over the pairs *key-value*:

```python
# Browsing keys (like previous example)
for k in d.keys():
    print(k)

# Browsing values
for v in d.values():
    print(v)

# Getting value from key
for k in d:
    print('{} --> {}'.format(k, d[k]))

# Browsing the pairs
for (k, v) in d.items():
    print('Key: {} -- Value: {}'.format(k, v))
```

It is sometimes useful to skip some iteration, to consider an endless loop, to quit a not finished loop, to run some commands at the end of a loop, . . . For that purposes, we have at our disposal `continue`, `break` and `else`:

```
# Skip some iteration
for i in range(5):
    if i == 2:
        continue # 2 is skipped
    print(i)

# Breaking loops
for i in range(5):
    if i == 2:
        break # Stop at 2
    print(i)

s = ''
while True: # Infinite loop ...
    if len(s) > 10:
        break # ... that ends here
    s += '*'
    print(s)

# Concluding a loop
s = ''
while len(s) < 7: # Modify this value to see how the run changes
    if len(s) == 8:
        break # Forced to break, skip 'else' block
    s += '~'; print(s)
else:
    # This command is run only if the loop ends without 'break'
    print('###')

names = ['Benny', 'Cindy', 'Teddy'] # Add 'Bobby' here
for name in names:
    if name == 'Bobby':
        print("Oh no, Bobby's here!")
        break # Forced to break, skip 'else' block
    print('Hello {}!'.format(name))
else:
    # This command is run only if the loop ends without 'break'
    print('Welcome to you!')
```

The `for` statement can also be used to create lists. This technique is known as *list comprehension* and is often useful:

```
[17 for i in range(5)]
[3 * i for i in range(5)]
['#' * i for i in range(5)]

names = ['Bobby', 'Billy', 'Teddy']
['Hello ' + name for name in names]
```

Questions

- Write some commands to print the following picture:

```
*
**
***
****
*****
```

- Write some commands to print the following picture:

```
*
**
***
****
*****
#
```

- Write some commands to print the following picture:

```
    *
   ***
  *****
 *******
*********
    #
```

- Write some commands to print the following picture:

```
# # # # #
 # # # #
# # # # #
```

- Write some commands to print the following picture:

```
+++++
++++
+++
++
+
++
+++
++++
+++++
```

- Write some commands to print the following picture:

```
+       +
++     ++
+++   +++
++++ ++++
+++++++++
```

2.5 Functions

There are some built-in functions that are always available with in a Python interpreter. We have already seen some of them and you can read the documentation to get a full list of these functions. Such a function has a type like other objects, has a name and may have some arguments in the pair of parenthesis. So, a name for a function is like a name for a variable and a function object can be assigned to a variable! This is a common source of bugs and strange behaviors in Python scripts.

```
# Absolute value of the argument
abs(-8)
type(abs) # Built-in function

# Convert argument into a floating number
float(-8)

# Be careful with function names
abs = float # Yes, you can do it!
abs(-8) # Ouch!
```

Of course, we can define our own functions with Python. To give a first example, we define a function `f` that takes no arguments and that does nothing:

```
def f():
    pass

type(f) # Function type
f() # Nothing happens...
f; id(f); hex(id(f))
g = f; id(g) # A function can be assigned
g is f # Same object pointed by f and g
g() # Nothing happens...
```

Note the use of the statement `pass` that corresponds to an empty block of code. Indeed, a function must possess its own block of commands, called its *body*, and you can not omit it. Thus, statement `pass` allowed us to define a function with an empty body. As explained above, the created function has a type and can be assigned.

Let us now define a function without argument but that does something:

```
def say_hello():
    lang = input("Lang ['e' for english, 'f' for french] : ")
    if lang == 'e':
        print('Hello!')
    elif lang == 'f':
        print('Salut!')
    else:
        print('Error: unknown language')

say_hello()
```

There is nothing extraordinary here, calling the function `say_hello` amounts to run its body. Nevertheless, it is important to understant the *scope* of a variable, *i.e.* where the content of a variable makes sense:

```
v1 = 17 # Global scope
dir() # Names in global scope

def f1():
  print(dir()) # Names in f1 scope
  # Variable v1 does not exist in current scope
  print(v1) # So v1 is taken from global scope

f1()
```

```
def f2():
  v1 = 8; v2 = 42 # Define v1 and v2 in current scope
  print(dir()) # Names in f2 scope
  print(v1) # Content of v1 taken in local scope
  print('v2 = {}'.format(v2)) # Here, v2 exists ...

f2()
print(v2) # ... but not in global scope
```

Scopes can be seen as nested lists of names in the sense that when a variable name is called, Python first looks in the current scope, if the name is defined then this scope is used, otherwise, Python does the same thing in the parent scope till it reaches the global scope. Because there is nothing above global scope, if the name is still not defined, an error `NameError` is *raised* (see further for explanations about errors in Python).

Now we know how to define a function and deal with the scopes, let us give some arguments to our function:

```
def say_hello(name, lang):
    if lang == 'e':
        greeting = 'Hello'
    elif lang == 'f':
        greeting = 'Salut'
    else:
        greeting = '###' # Unknown language

    print(greeting + ' ' + name + '!')

say_hello() # Error, arguments are missing
say_hello('Bobby') # Error, an argument is missing
say_hello('Bobby', 'f') # It works!
say_hello('Bobby', 'f', 42) # Error, too much arguments

say_hello('e', 'Bobby') # Order of arguments is important
say_hello(lang='e', name='Bobby') # But they can be explicitly named
```

Our function `say_hello` prints some greeting string but this string can not be assigned to a variable (for being used elsewhere, for instance). To return a value, we use the word `return`:

```
s = say_hello('Bobby', 'f') # Greeting is displayed...
print(s) # ... but s is None
```

```python
# New definition of say_hello
def say_hello(name, lang='e'):
    if lang == 'e':
        greeting = 'Hello'
    elif lang == 'f':
        greeting = 'Salut'
    else:
        greeting = '###' # Unknown language

    return greeting + ' ' + name + '!' # Return the string

s = say_hello('Bobby', 'f') # Greeting is no more displayed
print(s) # Greeting is in s now!
```

A function can return only one object but a tuple is an object. The tuples are commonly used to return more than one value and allow a nice code syntax by omitting the parentheses:

```python
def tell_me_more_about(x):
    return id(x), type(x), repr(x) # Return three values in tuple

# Get the three values in one line
a, b, c = tell_me_more_about(17+8j)
print(a); print(b); print(c)
```

As you noticed, if an argument is expected for some function, it has to be passed. Sometimes, this point can be disturbing and default value can be given. An argument with a default value can then be omitted when calling the function:

```python
# By default, argument lang is 'e'
def say_hello(name, lang='e'):
    if lang == 'e':
        greeting = 'Hello'
    elif lang == 'f':
        greeting = 'Salut'
    else:
        greeting = '###' # Unknown language

    print(greeting + ' ' + name + '!')

say_hello('Bobby') # Now, it works
say_hello('Bobby', 'f') # Argument can still be used, of course
```

It must be paid attention to mutable default value. Actually, the default value is evaluated only once during a run of a script. This does not matter for immutable objects because their content won't change but the one of mutable objects can be modified. This can be useful sometimes but it can also lead to unexpected effects:

```python
# Default value for l is mutable
def list_builder(x, l=[]):
    l.append(x)
    return l

# Notice the special behavior
list_builder(1)
list_builder(2)
list_builder(3)

# To avoid such a feature, use None and a test for the default
def list_builder(x, l=None):
    if l is None:
        l = [] # Default value
    l.append(x)
    return l

# More standard behavior
list_builder(1)
list_builder(2)
list_builder(3)
```

Finally, a good habit when writing functions in Python is to document them. We have already met documentation string, called *docstring*, which are used to give informations about a function. Such strings are surrounded with triple quotes `"""` and may span multiple lines. Commonly, to document a function, the docstring comes directly after the `def` statement, a first line is used to give a brief summary of what the function does and, after an empty line, we put some additional specifications (arguments, default values, ...). Lot of details about docstring conventions can be found in the PEP 257.

```python
def say_hello(name, lang='e'):
    """Return a greeting string.

    Keyword arguments:
    name -- string for the name to be greeted
    lang -- language, 'e' for english or 'f' for french (default'e')

    Return value:
```

```
    a string containing a greeting or '###' if unknown language
    """

    if lang == 'e':
        greeting = 'Hello'
    elif lang == 'f':
        greeting = 'Salut'
    else:
        greeting = '###' # Unknown language

    print(greeting + ' ' + name + '!')

# The docstring belongs to the function
print(say_hello.__doc__)

# Python uses the docstrings to build the documentation
help(say_hello)
```

Questions

- Read the documentation about the function `input` we use in the first implementation of `say_hello`.
- With the help of the documentation, find a built-in function to compute the sum of the numeric items of a list.
- Write a function `draw_pine` that take one integer argument `n` and that draw a pine with `n` levels of characters `*` and a character `#` for the trunk similar to what you did in exercises about loops. See below for an example:

```
>>> draw_pine(4)
   *
  ***
 *****
*******
   #
>>>
```

- Do the following improvements for the function `draw_pine`:
  - Properly manage negative value of `n` or non integer value.
  - Give a default value of `5` to `n`.
  - Add two arguments `leaf` and `trunk` with default values that are supposed to be the characters used to draw the pine. Manage non character values for these arguments.

- Modify the function to return a string containing the pine **in addition to** drawing it.
- Add a boolean argument `verbose` to print or not the pine with a default value and correct behavior if a non boolean value is given.
- Document your function.

2.6 More about function arguments

We introduce here some advanced concepts related to Python functions. They may be skipped in first reading but these features will be useful in the sequel.

2.6.1 Unpacking arguments

It is sometimes useful to pass arguments to a function through a list or a dictionary rather than explicitely. Although this is not considered as a good practice because it does not really improve code readability, these tricks are not deprecated because they make easier passing arguments to a subfunction, for instance.

```
# Unpacking a list with *
args = [17, 42, 3]
list(range(*args))

# Unpacking a dictionary with **
args={'name': 'Teddy', 'lang': 'f'}
say_hello(**args)
```

2.6.2 Arbitrary argument lists

An other practical mechanism with function arguments is to allow them to be arbitrary and to properly manage what the user pass to a function. To introduce that, we have to distinguish two kinds of arguments: *positional argument* and *keyword argument*. A positional argument is not explicitely named when it is passed to a function whereas a keyword argument has a name. For example, in `f(42, msg='Hi!')`, the number `42` is a positional argument and `msg` is the key of a keyword argument. A set of positional argument can be seen as a tuple and a set of keyword arguments as a dictionary. As mentioned previously, using keyword arguments, you can modify the order of the arguments when calling the function but positional arguments always have to be placed before keyword arguments.

To handle arbitrary positional arguments in a function, you must add a specific arguments of the form `*args` that receives a tuple containing the values of positional arguments:

```
# A function with arbitrary positional arguments
def posarg(*args):
    if len(args) == 0:
        print('No argument')
    else:
        for arg in args:
            print('Argument: {}'.format(arg))

posarg()
posarg('Tobby', 42)
```

For keyword arguments, the same principle is used but with a special argument `**keyargs` that receives a dictionary:

```
# A function with arbitrary keyword arguments
def keyarg(**keyargs):
    if len(keyargs) == 0:
        print('No argument')
    else:
        for key, val in keyargs.items():
            print("Argument '{}': {}".format(key, val))

keyarg()
keyarg(name='Tobby', number=42)
```

Of course, this is feasible to mix positional arguments and keyword arguments as long as positional ones occur before keyword ones.

```
def add_movie(title, *comments, **data):
    print('Title: {}'.format(title))
    for key, val in data.items():
        print('{}: {}'.format(key, val))
    print('-' * 42)
    for comment in comments:
        print('"{}"'.format(comment))

add_movie() # Argument title is mandatory

add_movie('Back to the Future') # Without optional argument

add_movie('Back to the Future',
    'Amazing!',
    'One of the greatest movies ever made'
) # Positional only
```

```
add_movie('Back to the Future',
    Year=1985,
    Runtime='1h56'
) # Keyword only

add_movie('Back to the Future',
    'Amazing!',
    'One of the greatest movies ever made',
    Year=1985,
    Runtime='1h56'
) # Both argument types
```

2.7 Exceptions

An error detected during execution of some Python code is called an *exception*. We have seen some examples previously without discussing about them. If an exception is not handled, it commonly leads to end the current script and to output some error message. Typically, such a message contains the name of the exception and some details about what happened:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> print(undefined_variable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'undefined_variable' is not defined
>>> 1 + '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In these examples, the names of the exceptions are `ZeroDivisionError`, `NameError` and `TypeError`. These exception names are quite explicit and you can get the whole list of built-in exceptions in the documentation.

Although an *unhandled exception* stops the current execution, exceptions are not unconditionally fatal and we can handle them. To do it, we use a `try ... except` statement:

```
a = 0
try:
    inv = 1 / a
```

```
except ZeroDivisionError:
    print('Error, division by zero!')
```

First, the block of code between the `try` and `except` keywords is executed. If an exception occurs during this execution, the remainder of the code is skipped and, if the exception type matches the name after `except`, the associated clause is executed. Otherwise, execution of the `try` statement is finished and the block of code after `except` is skipped. Such a statement can have more than one `except` clause:

```
try:
    # Run some commands which can produce exceptions
    some_function()
    some_other_function()
except ZeroDivisionError:
    print('Wow, division by zero!')
except IndexError:
    print('Argh, out of range!')
except KeyboardInterrupt:
    print('Why did you stop me?')

try:
    some_function()
except (ZeroDivisionError, IndexError, KeyboardInterrupt):
    # More than one type can be handled in one clause
    print('Oops, something happened!')
```

When an `except` statement is executed, we say that the exception has been *catched*. If an exception is not catched by none `except` statement, it remains unhandled. Note that a catched exception can be handled as any other object with the help of `as`:

```
try:
    1 / 0
except ZeroDivisionError as e:
    print(type(e)) # Exception type
    print(e) # Message about what happened
```

We have seen how to catch exceptions, let now see how we can *raise* them. The `raise` statement is used to force a specified exception to occur:

```
>>> raise NameError('Bobby')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Bobby
```

You can raise built-in exception types to deal with errors in your scripts. Usually, an exception takes only one argument and this is its associated message. For instance, the exception type `TypeError` is commonly used to indicate an inappropriate type:

```
def i_want_integer(n):
    if not isinstance(n, int):
        raise TypeError('n has to be an integer')
    print('I got an integer! This is {}'.format(n))

i_want_integer('Hi!')
i_want_integer(42)
```

For now, we cannot build custom exception types because we need to introduce the concept of classes before (see further). However, we can introduce the basic type for any custom exception, the aptly named `Exception`. Creating such an object, we can pass an arbitrary arguments list and, handling it, these arguments are available as a tuple:

```
try:
    # Force an Exception to occur
    raise Exception('Cindy', 81)
except Exception as e:
    print(type(e))
    print(e.args) # Arguments stored in .args
    print(e) # Print arguments
```

2.8 Input and Output

Allowing the user to interact with a script is a very common feature. In order to get values from the user, we have already seen the built-in function `input`. In the other way, to display some informations, we can use the function `print` and the method `format` of the strings. The interested reader is encouraged to look for more details about string methods (`str.ljust`, `str.center`, `str.rjust`, `str.zfill`, . . .).

We now focus on handling files with Python. Before using it, a file has to be opened with `open` which return a file object. This function takes two arguments, the name of the file to open and the *mode* to use. The mode involves announcing what we want to do with the file: `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), `'a'` opens the file for appending to the end and `'r+'` opens the file for both reading and writing. By default, a file is opened in read only mode. At the end, the file object can be closed with `close`.

```
# Try to open some nonexistent file
f = open('my_text_file') # FileNotFoundError

# Create and open a text file
# A new file appears in the working directory
f = open('my_text_file', 'w')
type(f)
f.close() # Close the file

# Reopen the file in read/write mode
f = open('my_text_file', 'r+')
f.close()
```

We do not mention binary files here. Indeed, opening files like the above example implicitly considers that the file is a text file. This mainly matters for dealing with line endings but it can be tragic if you want to handle binary files like JPEG files, for instance. We do not give more details about that in this document but you can refer to the documentation about the function `open`.

The method `read` reads the whole file and returns a string with its content. To write a string to a file, the method `write` is used. Be careful that `write` can only write strings, so any non string data has to be converted to a string before being written.

```
f = open('my_text_file', 'r+')
content = f.read()
print(content) # An empty string for an empty file

f.write('Hi Pythonista!\n')
f.write('How are you?\n')
f.write('Bye\n')

f.write(42) # Error, write only string!
f.write(str(42)) # OK

f.close() # Open the file with a text editor to see its content

f = open('my_text_file', 'r') # Reopen in read only
content = f.read()
print(content)
```

To read a file line by line, there is the method `readline` but you can also use the file as an iterable object and directly plug it into a `for` loop!

```
f = open('my_text_file', 'r')
f.readline()
f.readline()
f.readline()
f.close()

f = open('my_text_file', 'r')
i = 1
for line in f:
    print('Line {}: {}'.format(i, line))
    i += 1
f.close()
```

When you deal with a file, you silently use an object that contain the current position in the file, *i.e.* the position where the next read will occur. You do not really need to take care of that in a first time but, if you want to get this position, you can ask with the method `tell` and modify it with `seek`.

```
f = open('my_text_file', 'r')
f.tell()
f.readline()
f.tell()
f.seek(0) # Back to the beginning
f.readline() # First line again
f.seek(10) # Next read starts at position 10
f.readline()
f.close()
```

As you can see, a file has to be closed after use. If you forget it, the file will remain open and it can be problematic. To automate the closing and make it easier to handle files, the file object offers a mechanism called a *context*. We will see later how to create such a mechanism for our objects but this is a very useful feature that you can use with a `with ... as` statement.

```
with open('my_text_file', 'r') as f:
    content = f.read()

print(content)
f.closed # File has been silently and properly closed
```

Questions

- What is returned by the method `write`?

- Understand how the line endings are handled.
- Read the documentation about `seek`. How can you set a position from the end of the file?
- Add an option to your function `draw_pine` to draw into a file.
- Write a function `draw_pine_from_file` to draw a pine with options read from a file.

2.9 Classes

This is beyond the scope of this document to provide a complete introduction to object oriented programming. In this section, we simply introduce the basics to define classes and create custom objects in Python.

To say it simply, a *class* is a structure which can contain variables, called *attributes*, and functions, called *methods*, which can be accessed with a dot `.` symbol. A class is used as template to contruct objects, we say that an object *instantiates* a class. A class can be documented with the help of a docstring in the same way we described previously.

```python
class MyClass:
    """Documentation goes here."""
    n = 42 # An attribute
    def f(self): # A method, see further for self
        return 'I am a method'

# Two instances our class
obj1 = MyClass()
obj2 = MyClass()

# Object type
type(obj1)

# Use the attribute
print(obj1.n)
obj1.n = 17
print(obj1.n)
# Value is in the scope of the instance, not the one of the class
print(obj2.n)

# Call the method
obj1.f()
```

The role of a class is to *encapsulate* values and mechanisms to provide an interface to the user for some specific purpose. Before giving more details about classes, we have to note some special syntactic features in the class definition. Such a definition is introduced by the word `class` followed by the name of the class. Then, you can define attributes and methods. As illustrated in the example, an attribute is a variable defined in the scope of the instance. This point is important and actually not well illustrated by our class `MyClass`. Indeed, defining an attribute in the class definition like we did for `n` provide a reference to the class that can be overwritten in an instance. This can be problematic for mutable types as we explained previously.

```
class ClassWithAttributes:
    n = 42
    v = [1, 2, 3] # Mutable attribute

# Class-level attributes
print(ClassWithAttributes.n)
print(ClassWithAttributes.v)

# Instance-level attributes
x = ClassWithAttributes()
print(x.n)
print(x.v)

# Attributes can be overwritten ...
x.n = 17
print(x.n) # Instance-level
print(ClassWithAttributes.n) # Class-level

# ... but be careful with mutable types
x.v[1] = 17
print(x.v) # Instance-level
print(ClassWithAttributes.v) # Class-level
```

Class-level attributes can be useful for specific role but to avoid problems like the one illustrated in the previous example, the attributes should be defined at instance-level. To do it, we need a reference to the instance itself which has not already been created ... it is a vicious circle! The solution is based on the statement usually denoted by `self` (this word is a simple convention) which is used to precisely designate this reference to an instance. The common way to define instance-level attributes is to do it in a special method `__init__`, called *constructor*, that receives `self` as **first** argument, potentially followed by other arguments, and that is silently called when the object is created.

```
class ClassWithConstructor:
    def __init__(self):
        self.n = 42 # Instance-level attribute

# No more class-level attribute
```

```
print(ClassWithConstructor.n)

x = ClassWithConstructor() # Silent call to __init__(self)
print(x.n)
x.n = 17
print(x.n)

# Constructor can have arguments
class ClassWithArgs:
    def __init__(self, a, b='Hello'): # First self, then arguments
        self.a = a
        self.message = b

y = ClassWithArgs(42)
print(y.a)
print(y.message)

z = ClassWithArgs(42, 'Oh yeah!')
print(z.a)
print(z.message)
```

In our first class example, you can note that we have already use `self` as argument for the method `f`. Similarly, if you omit it, the method is defined at class-level but, with it, the method is defined at instance-level which is usually what we want.

```
class ClassWithMethods:
    # Constructor
    def __init__(self, x):
        self.value = x

    # Instance-level method
    def f1(self):
        # Here, self exists because instance exists
        print('Value: {}'.format(self.value))

    # Class-level method
    def f2():
        # Here, self does not exist
        print('I am at class-level')

x = ClassWithMethods(42)
x.f1()
ClassWithMethods.f1() # Error
x.f2() # Error
```

```
ClassWithMethods.f2()
```

For vocabulary, class-level attributes or methods are said to be *static*.

We have seen that an exception is raised when an error occurs. What happens when an unhandled exception is raised in a constructor? The question is not naive because it is related to the existence of an object. Actually, if an unhandled exception is raised in a constructor, the object is not created and the statement is simply ignored. Then, if the variable already exists, it is not modified.

```
class Inverse:
    def __init__(self, x):
        self.value = 1 / x # Ready to raise exception

x = Inverse(2) # No problem
id(x); print(x.value) # Object x exists

y = Inverse(0) # Exception is raised
id(y) # Object y has not been created

x = Inverse(0) # Try to replace x but exception is raised
id(x); print(x.value) # Previous line is ignored, x does not change
```

The power of object oriented programming mainly comes from the *inheritance* principle. Indeed, we can derive a class from some existing base class in order to adapt it or to enhance it for a specific purpose, for instance. To use this mechanism, you just have to mention the name of the base class between parentheses when you define the derived class.

```
# Base class definition
class Base:
    def __init__(self):
        print('Base constructor')
        self.value = 42

    def print_value(self):
        print('My value is {}'.format(self.value))

    def say_hello(self):
        print('Hello !')

base = Base()
```

```python
base.print_value()
base.say_hello()

# A derived class
class Derived1(Base): # Derived inherits from Base
    def __init__(self, value):
        print('Derived1 constructor')
        self.value = value

derived = Derived1(123) # Note that parent constructor is not called
# Attributes and methods from Base are available
print(derived.value)
derived.print_value()
derived.say_hello()

# An other derived class
class Derived2(Base):
    def __init__(self, value):
        print('Derived2 constructor')
        self.value = value

    def print_value(self): # Methods can be overwritten
        print('### Value : {} ###'.format(self.value))

derived = Derived2(456)
derived.print_value() # Overwritten method
derived.say_hello() # Original method

# Again a derived class
class Derived3(Base):
    def __init__(self):
        # Parent constructor has to be explicitly called, if needed
        super().__init__()
        print('Derived3 constructor')

derived = Derived3()
derived.print_value()
derived.say_hello()
```

This is quite common to check if some object has a given type or if a class inherits from an other given class. The built-in functions `isinstance` and `issubclass` are available for that.

```
isinstance(base, Base)
isinstance(derived, Base) # Derived inherits from Base

issubclass(Derived1, Base)
# Note that a class is considered a subclass of itself
issubclass(Base, Base)
```

Of course, we do not have to always inherit from custom classes and we can derive from built-in classes. To illustrate that, let us give a useful example to derive custom exception classes. We have already introduced the base class `Exception` from which any custom exception class should be derived.

```
class MyException(Exception):
    def __init__(self, message, other_arg):
        # Calling base constructor allows standard behavior
        super().__init__('I am a custom exception\n'
            + '*** Message : {}\n'.format(message)
            + '*** Argument: {}\n'.format(other_arg)
        )

        # Some attributes
        self.arg = other_arg
        self.message = message

def f():
    # Custom exception can be raised as other ones
    raise MyException('in a bottle', 42)

try:
    f()
except MyException as e:
    print('MyException catched!')
    print('Argument is {}'.format(e.arg))
    print(e) # Inheritance power!
```

We have seen the special method `__init__` which defines the class constructor. Actually, there are a lot of other special methods that can be defined in a class to give it some specific behaviors. These data models are described in a devoted documentation page and it would be beyond the scope of this document to make an exhaustive review. The special methods are all surrounded by double underscores `__` and are listed in Section 3.3 of the documentation page. Hereafter, we give some examples of such methods.

```python
# A class that can be converted as a string
class A:
    def __str__(self):
        return 'A object ({})'.format(id(self))

a = A()
str(a) # Explicit conversion
print(a) # Silent call to a.__str__()

# A (very) minimal and stupid sequence-like class
class B:
    def __len__(self):
        # Fixed length of 42 ...
        return 42

    def __getitem__(self, key):
        # Implement read-only self[key]
        if isinstance(key, int):
            # Dummy content
            return key % 2
        else:
            # Integer key only
            raise TypeError('B class allows only integers')

b = B()
print(len(b))
print(b['Ni'])
print(b[17])
```

Questions

- Create a class `Pet` which contains two attributes `age` and `name` and a method `get_info` that returns a string with informations about the pet. You should also provide a proper constructor.
- Derive two classes `Cat` and `Dog` from `Pet` and improve the method `get_info` in these specific cases.
- Create a custom exception `BadPetAge` to be raised when an error about the age occurs.
- Enhance the constructor of `Pet` to raise a `BadPetAge` if `age` is negative.
- Add comparison methods to the class `Pet` to order them according to their age.

```
# Example of comparisons
felix = Cat('Felix', 5)
puppy = Dog('Puppy', 3)

felix > puppy # Should return True
puppy > puppy # Should return False
felix <= puppy # Should return False
```

- Find in the documentation how to allow your class to be used through a context `with ... as`.

3 Extending Python with modules

3.1 What is a Python module?

If we restrict ourselves to built-in types and functions, we will have to define a lot of things and, most importantly, to do it in all our scripts. Of course, we do not do it and, when a program gets longer, we split it into several files. This practice makes code maintenance easier and helps us to organize our code. Then, to use some handy function or class located in a file, we just have to tell Python to look for the definition in this file. Such a file containing definitions is called a *module* and we say that we *import* it.

A module is nothing else than a Python file and the file name is the module name with the suffix `.py` appended. Within a module, the module's name is available as a string through the global variable `__name__`. To serve as an example, consider the module `my_sequences` given by a file `my_sequences.py` as follows:

```
# Fibonacci sequence
def fibonacci(n):
    s = [0]
    a, b = 0, 1
    while len(s) < n:
        s.append(b)
        a, b = b, a+b
    return s

# Syracuse sequence
def syracuse(n):
    s = []
    u = n
    while u != 1:
        s.append(u)
        u = u // 2 if u % 2 == 0 else 3*u + 1
    s.append(1)
    return s
```

3.2 Importing from a module

The easiest way to import all the definitions contained in a module is to use `import` with the name of the module (the files have to be in the same directory). Then, objects defined in the module are available but in a *namespace* named like the module:

```
import my_sequences

fibonacci(10) # Error, fibonacci is not in global scope
```

```
my_sequences.fibonacci(10) # Call fibonacci in my_sequences namespace
my_sequences.syracuse(10) # Idem with syracuse

# You can get the module name with __name__
my_sequences.__name__
```

Sometimes, a module name (or a package name, see further) can be long to repeat again and again in the content of the code. Typing it is boring and does not make the code easier to read. For that, Python provides an alias mechanism based on statement `as`:

```
import my_sequences as ms
ms.syracuse(42) # Shorter, isn't it?
```

If you need it, you can also import elements from a modules in the global scope (*i.e.* without namespace) by naming them explicitly with `from ... import ...`. However, this practice is not encouraged because it makes code harder to read because of the presence of not standard functions.

```
from my_sequences import fibonacci,syracuse

my_sequences.syracuse(10) # Error, no more namespace
syracuse(10) # Function is now global
```

You can also import everything from a module in the global scope with the wildcard `*` but this is definitively not recommended because it can silently overwrite definitions (*e.g.* function `open` with standard module `os`). Such a practice should only be used for testing purposes.

```
from my_sequences import *
fibonacci(10) # All is global
```

3.3 Packages

When a project gets larger, we naturally split the code into several modules, each one being adpated for a specific purpose (getting the data, computing statistics, producing outputs, ...). A *package* is a collection of such modules simply based on directory hierarchy.

Let start with a basic package called `my_package`. Create a directory named `my_package` an put the module `my_sequences.py` inside it. Moreover, create a file `__init__.py` in `my_package`. Your directory should be as follows:

```
my_package/
    __init__.py
    my_sequences.py
```

The special file `__init__.py` is required to treat the directory as a package. This file can be left empty but it can contain code to initialize the package. We are not going into details about this file (see the documentation) and we simply use it to load the module `my_sequences` on initialization. So, putting `from . import my_sequences` in `__init__.py` is sufficient for now. The statement `from . ...` stands for importing directly from current directory.

```python
# Import a package like a module
import my_package

# Functions are in module's namespace, nested in package's namespace
# (automated by __init__.py)
my_package.my_sequences.fibonacci(10)

# Import just a module (with a saving alias)
import my_package.my_sequences as mpseq
mpseq.syracuse(17)

# Importing in global scope from a package
from my_package.my_sequences import syracuse
syracuse(10)
```

With only one module, a package is useless but it becomes more interesting when you collect various modules and *subpackages*. Consider the follwing file hierarchy:

```
my_package/
    __init__.py
    tools/
        __init__.py
        one_amazing_module.py
        an_other_module.py
    sequences/
        __init__.py
        fibonacci.py
        syracuse.py
    some_stuff.py
    other_stuff.py
```

Filling the modules and `__init__.py` files, you will learn how to properly organize a Python project for your future developments.

3.4 The Python standard library

Python comes with a lot of standard modules which form the Python standard library. This is definitively one of the great strengths of Python to offer a so rich standard library. It would not be possible to present all these modules and packages but the clear and well organized documentation is here to help you in finding the ones you need. A good habit before trying to reinvent the wheel is to have first a look in these documentation pages. Hereafter, we introduce some parts of the standard library (with links to the documentation pages for further readings) which we consider useful but with no claim of exhaustiveness.

3.4.1 A bit of math

More advanced mathematical packages are presented further but, for basic mathematics, the standard module `math` already offers several useful definitions. Among them, we can start with the constants π and e:

```
import math

print(math.pi)
print(math.e)
```

Common mathematical functions are also provided:

```
x = 4.2
print(math.fabs(x)) # Absolute value
print(math.exp(x)) # Exponential
print(math.log(x)) # Logarithm
print(math.sqrt(x)) # Square root
print(math.cos(x), math.sin(x), math.tan(x)) # Trigonometry
# Et cetera...
```

It is also better to keep in mind that the functions provided by `math` are usually more accurate than standard functions.

```
>>> x = [0.1]*10
>>> sum(x) # Standard sum
0.9999999999999999
>>> math.fsum(x) # From math
1.0
```

An other useful mathematical tool is a (pseudo-)random number generator. The module `random` offers generators for various distributions.

```
import random

# Random integer between 4 and 9
print(random.randint(4, 9))
# Uniform distribution in [0, 1)
print(random.random())
# Uniform distribution in [1.7, 4.2)
print(random.uniform(1.7, 4.2))
# Gaussian distribution with mean -8.1 and standard deviation 4.2
print(random.gauss(-8.1, 4.2))
# Et cetera...

# Some useful tools on set (subsampling, for instance)
x = [i for i in range(10)]
random.shuffle(x); print(x)
print(random.sample(x, 4))
```

To conclude this brief review of mathematical tools in standard library, we mention the module `statistics` which provides (very) basic statistical functions.

```
import statistics

x = [random.gauss(0, 1) for i in range(50)]
print(statistics.mean(x))
print(statistics.median(x))
print(statistics.variance(x))
```

Documentation pages :

- https://docs.python.org/3.4/library/math.html
- https://docs.python.org/3.4/library/random.html
- https://docs.python.org/3.4/library/statistics.html

3.4.2 System interfaces

Some standard modules provide portable ways to interact with the underlying system. Maybe not the easiest to use, the module `os` offers a lot of low-level functions wherein some allow to deal with the current working directory.

```
import os

```

```
# Get the working directory
print(os.getcwd())
# List the content of the working directory
print(os.listdir())
# Create a directory in the working directory
os.mkdir('test_dir')
# Change the working directory
os.chdir('test_dir')
print(os.getcwd())
os.chdir('..')
print(os.getcwd())
# Remove a directory
os.rmdir('test_dir')
# Et cetera...
```

When you need to handle path names, the module `os.path` is your best friend. It allows you to extract the base name of a file with `os.path.basename`, to test if a path exists with `os.path.exists`, . . . and especially to manage path names with `os.path.join` in a portable way:

```
import os.path

# Content of shire varies from an operating system to an other
shire = os.path.join('hobbit', 'home', 'land')
print(shire)
```

Finally, the module `sys` provides system tools which are not dependent of the operating system.

```
import sys

# Get a platform identifier
print(sys.platform)

# A quick way to exit a program
sys.exit('Before I die...')
```

Documentation pages :

- https://docs.python.org/3.4/library/os.html
- https://docs.python.org/3.4/library/os.path.html
- https://docs.python.org/3.4/library/sys.html

3.4.3 Data formats

We now browse some usual data formats and ways to import and export data with Python. The module `datetime` supplies classes for manipulating dates and times. This module offers advanced tools to handle dates and times but we restrict our introduction to the simplest one, namely `datetime.datetime` class (read the documentation to get more informations).

```
import datetime

now = datetime.datetime.now()
print(str(now)) # Date and time as a string
print(now.weekday()) # 0 is Monday, 1 is Tuesday, ...

# Direct access to the attributes
print(now.year)
print(now.month)
print(now.day)
print(now.hour)
print(now.minute)
print(now.second)
print(now.microsecond)

# Time arithmetic
delta = datetime.timedelta(days=17,
            seconds=8,
            microseconds=81,
            milliseconds=1,
            minutes=3,
            hours=15,
            weeks=42)
print(str(now + delta))
print(str(now - 2*delta))
```

Standard library offers several ways to import and export data. Such mechanisms are useful to save data into files and load them some time after. Module `pickle` supplies `dump` and `load` to these ends. Pay attention to the fact that `pickle` uses binary files (`'wb'` and `'rb'` in the following examples), this is similar to what we have seen previously but with bytes instead of characters.

```
import pickle

amazing_object = {
    'Baggins': ('Bilbo', 'Frodo'),
    'Gamgee': ('Samwise',),
    'Brandybuck': ('Meriadoc',)
}

# Write an object to a binary file
with open('amazing.data', 'wb') as f:
    pickle.dump(amazing_object, f)

# Delete the object to simulate a new session (or quit)
del amazing_object

# Load an object from a binary file
with open('amazing.data', 'rb') as f:
    amazing_object = pickle.load(f)
print(amazing_object)
```

An other common way to serialize data is the JSON format. The module `json` offers a handsome interface to deal with files in such a format.

```
import json

amazing_object = {
    'Baggins': ('Bilbo', 'Frodo'),
    'Gamgee': ('Samwise',),
    'Brandybuck': ('Meriadoc',)
}

# See it in JSON format
print(json.dumps(amazing_object))
print(json.dumps(amazing_object, indent=4)) # More pretty

# Load from a string
my_object = json.loads('["gandalf", {"is":["amazing", null, 1.7]}]')
print(my_object); type(my_object)

# Export to a file
with open('amazing.json', 'w') as f:
    f.write(json.dumps(amazing_object))

del amazing_object # Simulate again

# Load from a file
with open('amazing.json', 'r') as f:
    amazing_object = json.loads(f.read())
print(amazing_object)
```

Documentation pages :

- https://docs.python.org/3.4/library/datetime.html
- https://docs.python.org/3.4/library/pickle.html
- https://docs.python.org/3.4/library/json.html

4 Graphics with Matplotlib

4.1 Introduction

Matplotlib is a plotting library originally wrote by John D. Hunter and distributed under a BSD compatible licence (Matplotlib is a free software).

Hereafter, we propose an introduction to basic concepts underlying the Matplotlib package. To experiment the following examples, we recommend to work in a Python console rather than a IPython one. Indeed, this latter integrates graphics in its own way and is not handy for an introduction.

4.2 Simple graphs

To create figures, Matplotlib supplies the module `pyplot`. To avoid long commands, we use the common alias `plt` for this module.

```
import matplotlib.pyplot as plt
```

The most basic graph consists in plotting some points considering their x and y coordinates with `plot`. If you work with Spyder, a special window containing the graph appears after the call to the `plot` function. However, if you work with a raw Python console, you will have to call explicitly the function `show` to see this window which allows you to handle some graphics parameters. To adjust thes axes, you can also deal with the function `axis`.

```
plt.plot([15, 3, 1, 5], [42, 17, 8, 5])
plt.show() # If you are not working with Spyder

# Modify the axes
plt.axis([-5, 20, 0, 50]) # [xmin, xmax, ymin, ymax]
```

If you call again the function `plot`, you see that the new points are plotted in the same figure. This is one way for plotting several graphs together but you can also call `plot` with all the points to plot in one time. To distinguish the plots, use format strings after each pair of points sequences (see the documentation for details). When you want to restart the figure, you can clean it with `clf` or simply call `close` and start a new one. Alternatively, if you want to keep the existing one for further manipulations, call `figure` to create a new device.

```
# Clean the current figure
plt.clf()

# Cyan star markers
plt.plot([15, 3, 1, 5], [42, 17, 8, 5], 'c*')
```

```python
# End the previous figure
plt.close()

# Red square markers
plt.plot([15, 3, 1, 5], [42, 17, 8, 5], 'rs')
# Additional plot with green dash-dot line and circle markers
plt.plot([0, 5, 10, 15], [42, 17, 8, 5], 'go-.')

# Open a new figure
plt.figure()

# Same graph as above created in one command
plt.plot(
    [15, 3, 1, 5], [42, 17, 8, 5], 'rs',
    [0, 5, 10, 15], [42, 17, 8, 5], 'go-.'
)
```

The function `plot` produces an `Line2D` object which permits to control line properties. These properties can be set at the creation or later with the help of `setp` (again, see the documentation).

```python
# With properties
my_line = plt.plot([0, 5, 10, 15], [17, 8, 42, 5],
    color='green',
    linestyle='dashed',
    marker='o',
    markerfacecolor='blue',
    markersize=12
)

# Change some properties
plt.setp(my_line, 'color', 'red', 'linewidth', 2.0)

# With more than one line, a list is returned
my_lines = plt.plot(
    [15, 3, 1, 5], [42, 17, 8, 5], 'rs',
    [0, 5, 10, 15], [42, 17, 8, 5], 'go-.'
)
plt.setp(my_lines[0], 'color', 'cyan')
plt.setp(my_lines[1], 'marker', 'D')
```

4.3 Tweaking a figure

Now that we know how to plot a simple graph, let see how arrange it. First, we introduce a way to put several graphics in the same figure with `subplot`. This function takes three arguments which correspond to a plotting grid: number of rows, number of columns and the position for next plot in this grid.

```
import math

# Some data
n = 256
x = [5 * i / (n - 1) for i in range(n)]
y = [math.exp(-t) * math.cos(2 * math.pi * t) for t in x]
z = [math.cos(2 * math.pi * t) for t in x]

# 2 rows, 1 column and position 1
plt.subplot(2, 1, 1)
plt.plot(x, y, 'bo')
# 2 rows, 1 column and position 2
plt.subplot(2, 1, 2)
plt.plot(x, z, 'r--')
```

An other common decoration of a graph is text with title, labels and comments. The functions `title`, `xlabel`, `ylabel` and `text` are supplied to this end.

```
import math

# Some parameters
mu = -2.5
sigma = 3

# Gaussian density
def gaussian_density(t, m=0, s=1):
    return math.exp(-(t-m)**2 / (2*s**2)) / math.sqrt(2*math.pi*s**2)

# Some data
n = 256
x = [mu + 4 * sigma * i / n for i in range(-n, n+1)]
y = [gaussian_density(t, mu, sigma) for t in x]

plt.plot(x, y, 'r--')

# Add a title
plt.title('Gaussian density')
```

```python
# Add labels for the axes
plt.xlabel('Abscissa')
plt.ylabel('Ordinate')

# Add some text
plt.text(mu, 1 / math.sqrt(8 * math.pi * sigma**2), 'Hello Gauss!')
```

If you need to include mathematical notations in the text, you can write a TeX expression surrounded by dollar signs,

```python
plt.title(
    'Gaussian density with $\mu$={} and $\sigma$={}'.format(mu, sigma)
)
```

4.4 Adding elements

We had just a brief look on all the options offered by Matplotlib to produce figures. Reading the documentation, you see that adding common elements is quite straightforward. Hereafter, we give some examples and we encourage you to browse the Matplotlib beginner's guide.

```python
z = [gaussian_density(t, mu, sigma/2) for t in x]

# Lines with labels
plt.plot(x, y, 'r--', label='$\sigma$={}'.format(sigma))
plt.plot(x, z, 'g-.', label='$\sigma$={}'.format(sigma / 2))

# Add a legend based on labels
plt.legend()

# Add a grid
plt.grid()
```

4.5 Other graphs

Of course, Matplotlib is not restricted to plot simple sequences of points and to the module `pyplot`. The package supplies a lot of graphical representations and tools and it would be beyond the scope of this document to make an exhaustive review. We give below an insight about the possibilities. If you are interested in data visualization, you should have a look to the Matplotlib examples.

```python
import math
import random

import matplotlib.pyplot as plt

# Scatter plot
n = 256
x = [random.gauss(0, 1) for i in range(n)]
y = [random.gauss(0, 1) for i in range(n)]
color = [random.randint(0, 50) for i in range(n)]
size = [100 * random.random() for i in range(n)]
plt.scatter(x, y, c=color, s=size, alpha=0.5)

plt.figure()

# Histogram
n = 2048
x = [random.gauss(0, 1) for i in range(n)]
u = [4 * i / n for i in range(-n, n + 1)]
v = [math.exp(-t**2/2) / math.sqrt(2*math.pi) for t in u]
plt.hist(x, bins=50, normed=True, color='orange')
plt.plot(u, v, 'r-', linewidth=2)

plt.figure()

# Pie chart
ore = ['Gold', 'Silver', 'Ore', 'Mithril']
count = [31, 58, 93, 21]
color = ['gold', 'silver', 'brown', 'lightskyblue']
explode = (0, 0.1, 0, 0)
plt.pie(count, labels=ore, colors=color, explode=explode,
    shadow=True, startangle=90)

plt.figure()

# Boxplot
n = 256
x = [random.gauss(0, 1) for i in range(n)]
y = [random.gauss(0, 1) for i in range(n)]
plt.boxplot([x, y])

plt.figure()

# Violin plot
plt.violinplot([x, y])
```

5 Mathematics with NumPy

5.1 What is NumPy?

NumPy is a Python package which provides tools to work with multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. NumPy is distributed under the terms of a revised BSD license and is thus a free software.

To shorten the commands of this section, we use the common alias `np` for `numpy` as follows:

```
import numpy as np

# NumPy supplies usual constants
print(np.pi)
print(np.e)
```

5.2 The array class

The `array` class plays a central role in NumPy. Such an object can be instantiated with the function `array` from a list for a vector or a list of lists for a matrix.

```
# Vector
v = np.array([17.0, 42.0, 8.1, 19.0])
print(v, type(v))

# Matrix (list of rows)
m = np.array([[1.0, 2.0, 3.0],
              [4.0, 5.0, 6.0]])
print(m, type(m))

# Function print behaves nicely with large arrays
print(np.array(range(10000)))
np.array([range(500) for i in range(500)])
```

To access elements in an array, we use naturally a syntax based on square brackets `[]` and slicing.

```
print(v[0])
print(v[1:3])

# Modify items
v[2] = 15.0
```

```
print(v)
v[1:3] = 0.0
print(v)

# Idem for matrices
print(m[0,1])
print(m[:,2])

# Et cetera ...
```

To get informations about the array dimensions and the number of rows, columns, ... the `array` class supplies an attribute `shape` which is a tuple. Moreover, the method `reshape` allows you to change the dimensions of an array.

```
print(v.shape)
print(m.shape)

m1 = v.reshape(2, 2)
print(m1)

m2 = m.reshape(3, 2)
print(m2)

v1 = m.reshape(6)
print(v1)
```

The `array` class offers several useful function to do basic computations with arrays. There are a lot of tools defined for arrays as you can see in the index of the documentation. All these tools can be called as methods or as functions from the NumPy namespace.

```
# Sum of the elements
s = v.sum()
s = np.sum(v) # Equivalent to the previous one
print(s)

# Cumulative sum of the elements
cs = v.cumsum()
cs = np.cumsum(v) # Equivalent to the previous one
print(cs)

# Maximum and minimum
print(m.min(), m.max())
print(np.min(m), np.max(m)) # Equivalent to the previous one

# Mean and variance
print(v.mean()) # Or np.mean(v)
print(m.var()) # Or np.var(m)

# Et cetera ...
```

An important advantage offered by NumPy arrays and functions is the capacity to apply a function to all the items of an array without a loop. Such a syntax makes definitively the code easier to read and is often used in the sequel.

```
n = 6
x = np.array([np.pi * i / (n - 1) for i in range(n)])
print(np.cos(x))

# It also works with common operations
print(x + 1)
print(2 * x)
print(x / 5)
# Et cetera ...

import math
print(math.cos(x)) # Error, only available with NumPy functions
```

As we mentioned before, we have to pay attention about copy operation when we manipulate arrays. Indeed, as mutable objects, the content of an array is sometimes copied and sometimes only passed as a reference.

```
a = np.array([1, 2, 3, 4])

# Simple assignment does not copy
b = a
print(b is a) # True

# Use copy method to copy arrays
c = a.copy()
print(c is a) # False
```

5.3 Creating arrays

Fortunately, we do not have to create explicitly all our arrays and NumPy provides some functions to help us for the common situations.

```
# Create arrays full of zeros
v = np.zeros(17)
m = np.zeros((3, 4)) # Note the tuple argument
print(v); print(m)

# Create arrays full of ones
v = np.ones(17)
m = np.ones((3, 4))
print(v); print(m)

# Create arrays with uninitialized entries
v = np.empty(17)
m = np.empty((3, 4))
print(v); print(m)
```

We have also at our disposal such tools for matrices:

```
# Identity matrix
m = np.identity(5)
print(m)

# Diagonal matrix from a vector
v = np.array([1.0, 2.0, 3.0])
m = np.diag(v)
print(m)

# Vector from the matrix diagonal
m = np.array([[1.0, 2.0],
              [3.0, 4.0]])
v = m.diagonal() # Equivalent to n = np.diag(m)
print(v)
```

To avoid difficult commands based on `range` when creating arrays, NumPy provides two useful functions. The first one, `arange` is similar to `range` but the returned object is an array. The second function, `linspace`, returns an array of evenly spaced numbers over a specified interval. This second function is very useful to plot a function, for instance.

```
# Create arrays in range style
v = np.arange(17); print(v)
v = np.arange(8, 17); print(v)
v = np.arange(8, 17, 2); print(v)
```

```
# Useful for matrices too!
m = np.arange(8, 17).reshape(3, 3)
print(m)

# Evenly spaced numbers
x = np.linspace(2, 3, 10); print(x)

# Useful to plot a function
import matplotlib.pyplot as plt
x = np.linspace(0, 1, 256)
plt.plot(x, np.sin(2 * np.pi * x), 'r-')
plt.grid()
```

5.4 Indexing with boolean arrays

NumPy offers a syntax based on boolean arrays to access some items in an array. This way can be very useful when we handle data sets and it is available to get values but also to modify them.

```
# Note the concatenation
a = np.append(np.arange(10), np.arange(8, -1, -1))

# Explicit boolean array ...
b = [i % 3 != 0 for i in range(len(a))]
print(a[b])

# ... or determined by conditions
b = a >= 7
print(a[b])

# Useful to modify values
import matplotlib.pyplot as plt
x = np.linspace(-2 * np.pi, 2 * np.pi, 256)
y = np.sin(x)
y[y > 0.5] = 0.5 # Nice syntax, isn't it?
plt.plot(x, y)

# With matrices
a = np.arange(12).reshape(3,4)
b1 = np.array([False,True,True])
b2 = np.array([True,False,True,False])

a[b1, :] # Selecting rows
a[:, b2] # Selecting columns
```

```
a[b1, b2] # Selecting both
```

5.5 More with matrices

When we are working with matrices, more specific operations and tools are needed. First, there are common matrices operations.

```
m1 = np.array([[1,1],
               [0,1]])
m2 = np.array([[2,0],
               [3,4]])

# Addition works as usual
print(m1 + m2)

# Be careful with multiplication!
print(m1 * m2) # Item by item
print(m1.dot(m2)) # Right matrix multiplication
print(np.dot(m1, m2)) # Idem from np namespace

# Many NumPy functions can be applied on axis (sum, mean, min, ...)
print(m1.sum()) # Sum of all items
print(m1.sum(axis=0)) # Sum of each column
print(m1.sum(axis=1)) # Sum of each row
```

Of course, NumPy also supplies tools to handle matrices as `array` objects in Python scripts.

```
# Stacking rows and columns
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
m1 = np.vstack((v1, v2)) # Pay attention to tuple
print(m1)
m2 = np.hstack((v1, v2)) # One line
print(m2)
v1 = np.array([[1], [2], [3]])
v2 = np.array([[4], [5], [6]])
m3 = np.hstack((v1, v2)) # Three lines
print(m3)

# Iterate on matrix rows
for row in m3:
    print('--->', row)
```

```
# Iterate on matrix elements
for item in m3.flat: # Operator flat
    print('--->', item)
```

Matrix calculation with NumPy is not restricted to such elementary operations and a large set of linear alegra tools are supplied. Note that some functions have to be taken from the submodule `numpy.linalg` (see module documentation for details).

```
m1 = np.arange(12).reshape(3, 4)
m2 = np.array([[3, 1, 2],
               [2, 0, 5],
               [1, 2, 3]])

# Transpose operator
print(m1.T)
print(m1.transpose())
print(np.transpose(m1))

# Determinant and trace operators
print(np.linalg.det(m2))
print(np.trace(m2)) # Or m2.trace()

# Inverse matrix
np.linalg.inv(m1) # Raise a LinAlgError exception
m2_inv = np.linalg.inv(m2)
print(np.dot(m2, m2_inv))

# Eigenvalues and eigenvectors
np.linalg.eigh(np.dot(m1, m1.T)) # For Hermitian matrix only
np.linalg.eig(m2) # Otherwise, right eigenvectors are returned
```

More advanced procedures are at your disposal and we encourage you to browse the NumPy documentation to discover all the possibilities. As an example, we give below the commands to perform a least squares polynomial fit.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
z = np.polyfit(x, y, 3) # Polynomial of degree 3

```

```
print(z) # Array z contains the coefficients of the polynomial

plt.plot(x, y, 'bo')
plt.axis([-1, 6, -2, 2])

# Create a function for polynomial given by z
pol_z = np.poly1d(z)
plt.plot(x, pol_z(x), 'r--')
```

5.6 A bit of randomness

Among a lot of other things, NumPy supplies a large set of random generators in the submodule `numpy.random`. Some distribution are already available in the standard library but, with `numpy.random` functions, you are able to directly get array objects.

```
import numpy.random as rnd

# Random arrays
rnd.rand(3) # Uniform distribution in [0, 1)
rnd.randn(2, 3) # Standard normal distribution

# Random sample
rnd.choice(np.array([17, 8, 19, 1, 3, 15]), 4) # With replace
rnd.choice(np.array([17, 8, 19, 1, 3, 15]), 4, False) # No replace

# Distributions
rnd.standard_exponential(5) # Standard exponential
rnd.exponential(2, 5) # Exponential with parameter 2
rnd.poisson(np.pi, 10) # Poisson with parameter pi
# Et cetera ...
```

As an illustration, let us run a Brownian motion simulation in the plane:

```
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt

n = 1e5
u = rnd.choice([-1, 1], n).cumsum() / np.sqrt(n)
v = rnd.choice([-1, 1], n).cumsum() / np.sqrt(n)

plt.plot(u, v, 'k-')
```