In-depth Python (fast, unstructured)

Master IoT

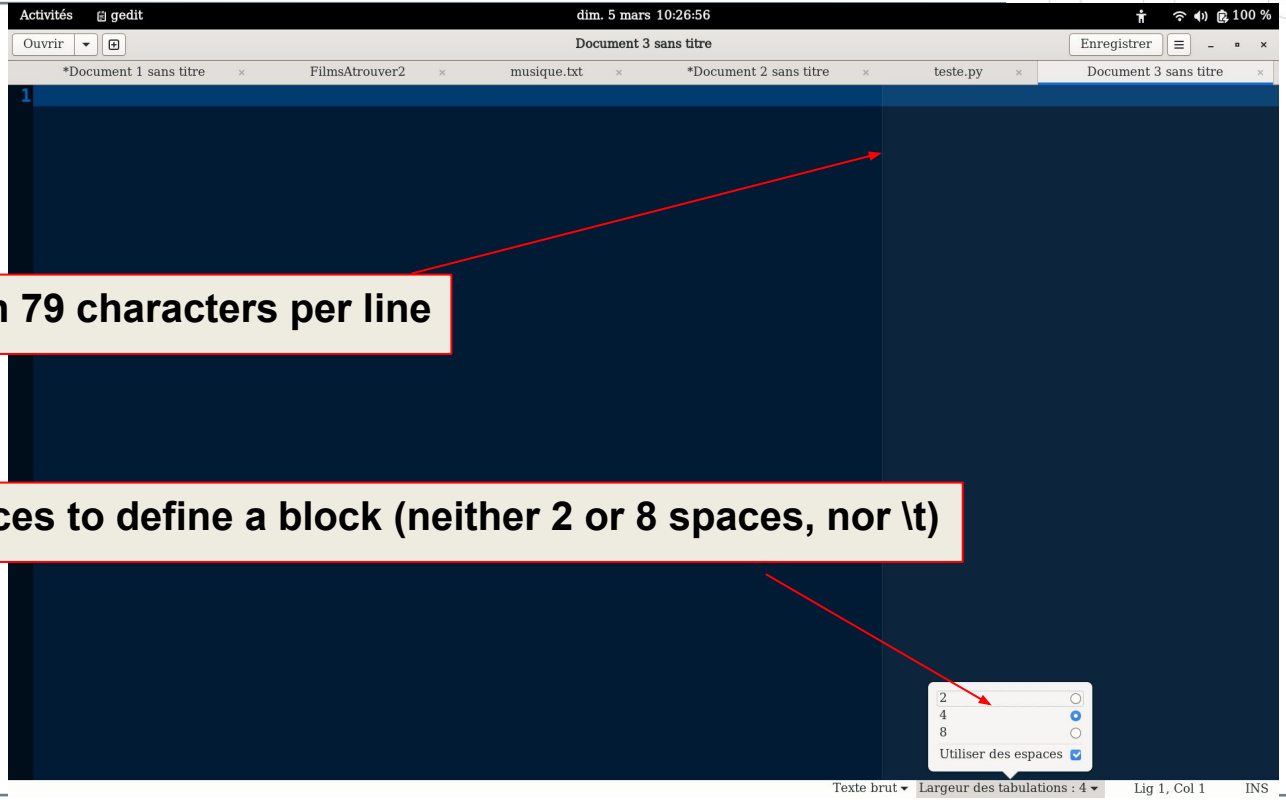
C. Guyeux

```
>>> import types
>>> help(types.CodeType)
...
Help on class code in module builtins:

class code(object)
 |  code(argcount, kwonlyargcount, nlocals, stacksize, flags, codestring,
 |        constants, names, varnames, filename, name, firstlineno,
 |        lnotab[, freevars[, cellvars]])
 |
 |  Create a code object.  Not for the faint of heart.
 |
 |  Methods defined here:
 |
 ...
```

```
>>> import types
>>> help(types.CodeType)
...
Help on class code in module builtins:

class code(object)
 |  code(argcount, kwonlyargcount, nlocals, stacksize, flags, codestring,
 |        constants, names, varnames, filename, name, firstlineno,
 |        lnotab[, freevars[, cellvars]])
 |
 |  Create a code object. Not for the faint of heart.
 |
 |  Methods defined here:
 |
 ...
```

Generalities

pep8



Less than 79 characters per line

4 spaces to define a block (neither 2 or 8 spaces, nor \t)

femto-st SCIENCES & TECHNOLOGIES

pep8: readability

```
if this_happens or that_happens \
   or these_happen:
    print('blah')

filepath = "this/sweet/line/" \
        "looksbetter.xlsx"
```

Parentheses can be used too to divide long lines:

```
    from minibelt import (dmerge, get, iget, normalize,
                          chunks, window, skip_duplicates, flatten)

     ("strings are automatically concatenated "
     "if they are only separated by "
     "white spaces.")
```

pep8

Operators must be surrounded by spaces:

```
>>> variable = 'valeur'
>>> this == that
>>> 1 + 2
```

except when grouping the mathematical operators with the highest priority to distinguish the groups:

```
>>> a = x*2 - 1
>>> b = x*x + y*y
>>> c = (a+b) * (a-b)
```

and for the = sign in the declaration of arguments and the passing of parameters:

```
>>> def my_funct(arg='value'):
...     pass
...
result = my_funct(arg='value')
```

pep8

Modules :

- Import of module > import of module content
- Import standard lib > import third party libs > import your project

Variable names:

- Single letters, in lower case: for loops and indices.
- Lower case letters + underscores (bottom dash _): for modules, variables, functions and methods.
- Upper case letters + underscores : for (pseudo) constants.
- Camel case : class names

pep8

How to check if our code satisfies the pep8?

```
$ pip install pycodestyle
$ pycodestyle script.py
```

or: `pylint`

types

Numbers, strings and tuples are not mutable, lists and dictionaries are.

There are generic operations for sequences such as slicing, which can be applied to any type of sequence: tuple, list, str... A sequence is, in fact, a collection of objects indexed by their position.

Multi-type compatible operations are usually functions or expressions, like len(X) or X[0]. Type-specific operations are methods: string.upper()...

types

```
>>> isinstance([], list)
True

>>> type(True) == int, isinstance(True, int)
(False, True)
```

The isinstance() function is used to check if an object is an instance of the specified class while taking care of inheritance.

type(), on the other hand, only checks for equality of reference types and discards subtypes

Numbers

binary digits

```
>>> 1&1
1
>>> 1&0
0
>>> 0&1
0
>>> 0&0
0
>>> 1|0
1
>>> 1|1
1
>>> 0|1
1
>>> 0|0
0
```

- xor :

```
0^1
```

- shift of bits:

```
>>> 1<<2
4
>>> 1<<3
8
>>> 1<<3+1
16
>>> (1<<3)+1
9
>>> 4<<1
8
```

binary digits

From ascii to bin:

```
>>> bin(int.from_bytes('hello'.encode(), 'big'))
'0b110100001100101011011000110110001101111'
```

and the opposite:

```
>>> n = int('0b110100001100101011011000110110001101111', 2)
>>> n.to_bytes((n.bit_length() + 7) // 8, 'big').decode()
'hello'
```

int, float and decimal

```
>>> 10_000_000, 11_111_111
(10000000, 11111111)

>>> scientific_nb = 1.23e2  # 1.23 * 10^2

>>> from decimal import Decimal
>>> from more_itertools import numeric_range
>>> list(numeric_range(Decimal('1.7'), Decimal('3.5'), Decimal('0.3')))
[Decimal('1.7'), Decimal('2.0'), Decimal('2.3'), Decimal('2.6'),
Decimal('2.9'), Decimal('3.2')]
```

complex numbers

```
>>> 4+3j
(4+3j)

>>> j^2
1
```

Strings

string

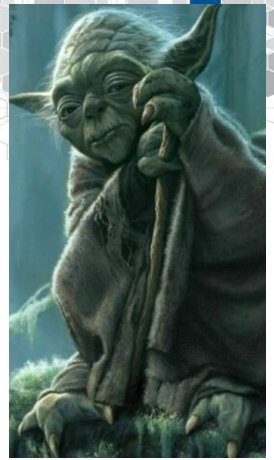
```
>>> dd="dfdfDSQFqsd"

>>> "fDSQFq" in dd
True

>>> dd.lower()
'dfdfdsqfqsd'

>>> dd.upper()
'DFDFDSQFQSD'

>>> dd.islower()
False

>>> dd.isspace()
False
```

```
>>> "12123".isdigit()
True

>>> "sample".endswith(("ple","ple."))
True
```

string

```
>>> 'aze'.rstrip('ze')
'a'
>>> 'dpoip ds    \n'.rstrip()
'dpoip ds'
```

string: the pythonic way



Find the first non-repeated occurrence of a letter

```
for c in my_string:
    if my_string.index(c) == my_string.rindex(c):
        print(c)
        break
```

Remove duplicate words while maintaining the order of the string:

```
sentence = "The sound sounds sound".split()
list(dict.fromkeys(sentence).keys())
```

f-string

```
>>> text = "Data Science Blog"
>>> print(f"{text=}")
text='Data Science Blog'
```

f"{stuff_to_print:format}"

```
>>> f"{1000000:d}"
'1000000'

>>> f"{1000000:,d}"
'1,000,000'

>>> dd='abc'
>>> f"blabla : {dd:>10} fois"
'blabla :        abc fois'
```

f-string: numbers

```
>>> pi_val = 3.141592
>>> print(f"Example 1: {pi_val:f}")
Example 1: 3.141592
>>> print(f"Example 2: {pi_val:.2f}")
Example 2: 3.14
```

f-strings are flexible enough to allow nesting:

```
>>> float_val = 1.5
>>> precision = 3
>>> print(f"{float_val:.{precision}f}")
1.500
```

f-string: percentage and scientific notation

Percentage:

```
>>> val = 0.5
>>> print(f"Example 1: {val:%}")
Example 1: 50.000000%
>>> print(f"Example 2: {val:.0%}")
Example 2: 50%
```

Scientific notation:

```
>>> val = 1.23e3  # 1.23 * 10^3
>>> print(f"Example 1: {val:e}")
Example 1: 1.230000e+03
>>> print(f"Example 2: {val:.3E}")
Example 2: 1.230E+03
```

f-string: dates

```
>>> from datetime import date, datetime
>>> day = date(
...  year=2022,
...  month=9,
...  day=1
... )
>>> print(f"{day}")
2022-09-01
>>> print(f"{day:%Y/%m/%d}")
2022/09/01
>>> print(f"{day:%Y %B %d (%A)}")
2022 September 01 (Thursday)
>>> now = datetime.now()
>>> print(f"{now:%Y-%m-%d %H:%M:%S.%f}")
2023-03-05 13:03:00.824671
```

f-string: padding

```
>>> val = 1
>>> print(f"1: {val:1d}")
1: 1
>>> print(f"2: {val:2d}")
2:  1
>>> print(f"3: {val:3d}")
3:   1
```

Padding with zeros:

```
>>> print(f"3: {val:03d}")
3: 001
```

Containers

list

```
>>> L=[1,2,1,3,2]
>>> L.remove(2)
>>> L
[1, 1, 3, 2]
```

=> (only) the first occurrence is removed

```
>>> L.pop(3)
2
>>> L
[1, 1, 3]
```

=> return and remove the third element

```
>>> [1,2,1,3]+[2,3,1,2,3]
[1, 2, 1, 3, 2, 3, 1, 2, 3]
```

list: accession, deletion…

```
L = ['abc', 9.1, (2012, 12, 21)]
a, _, (year, month, day) = L

a, *middle, c = [1, 2, 3, 4]
a, *_, c = [1, 2, 3, 4]

del L[0]      # remove element at position 0
del L[2:5]
L.remove(x)   # remove x

L.extend((1,2)) # any kind of container
```

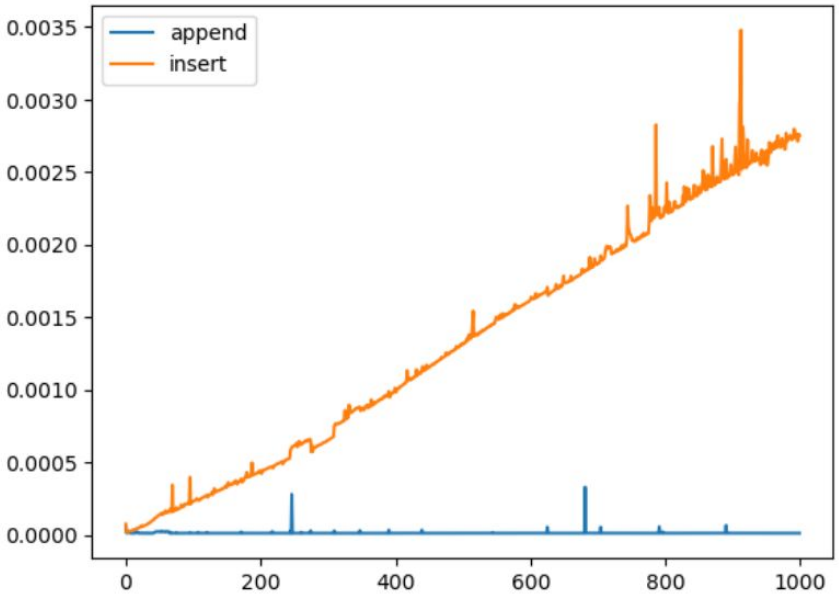
list: append and insert

```
1  liste1, liste2 = [],[]
2  T1, T2 = [], []
3  N=100000
4  t=time()
5  for k in range(N):
6      liste1.append(1)
7      if k%100 == 0:
8          T1.append(time()-t)
9          t=time()
10 t=time()
11 for k in range(N):
12     liste2.insert(0, 1)
13     if k%100 == 0:
14         T2.append(time()-t)
15         t=time()
```

list: append and insert

```
liste1, liste2 = [],[]
T1, T2 = [], []
N=100000
t=time()
for k in range(N):
    liste1.append(1)
    if k%100 == 0:
        T1.append(time()-t)
        t=time()
t=time()
for k in range(N):
    liste2.insert(0, 1)
    if k%100 == 0:
        T2.append(time()-t)
        t=time()
```

```
%matplotlib inline
%pylab inline
plot(T1, label='append')
plot(T2, label='insert')
legend()
```



femto-st SCIENCES & TECHNOLOGIES

list: sort

```
>>> l = [[1, 2], [1, 4, 3], [2, 3]]
>>> sorted(l, key=lambda x:x[1])
[[1, 2], [2, 3], [1, 4, 3]]
>>> l
[[1, 2], [1, 4, 3], [2, 3]]
>>> l.sort(key=lambda x:x[1])
>>> l
[[1, 2], [2, 3], [1, 4, 3]]
```

list: manipulate them as stacks

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack
[3, 4, 5]
```

LIFO: Last in, first out

list: manipulate them as queues

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

FIFO: First in, first out

list: iteration

```
>>> L=[(1,2),(2,3)]
>>> for i,j in L:
...     print(i,j)
...
1 2
2 3

a = [1,2,3,4,5]
b = [10,20,30,40,50]
c = [100,200,300,400,500]
for x,y,z in zip(a,b,c):
    print(x,y,z)
```

list of lists

```
>>> l1, l2 = [1, 3, 8], [2, 4, 9]
>>> list(zip(l1, l2))
[(1, 2), (3, 4), (8, 9)]

>>> l = [(1,2), (3,4), (8,9)]
>>> list(zip(*l))
[(1, 3, 8), (2, 4, 9)]
```

list comprehensive

```
>>> [k for k in container if condition]
```

```
>>> seq1 = 'abc'
>>> seq2 = (1,2,3)
>>> [ (x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3),
('c', 1), ('c', 2), ('c', 3)]
```

=> all possible couples

>>> s = [['Every', 'piece of'], ['software written today is', 'likely'], ['going to'], ['infringe on', "someone else's", 'patent.']]

```
>>> [k for sublist in s for k in sublist]
```

['Every', 'piece of', 'software written today is', 'likely', 'going to', 'infringe on', 'someone else's', 'patent.']

iter

The function to build an iterator from an iterable: string, list, tuple, set...

So, in a for loop, we call :

first to iter(), to build an iterator,

then to next(), until the exception StopIteration is raised, then caught.

set

```
>>> s=set([1,2,3])
>>> s.remove(3)
>>> 1 in s
True
>>> s.add(5)
>>> s
{1, 2, 5}
>>> for k in set([1, 2]):
        print(k)
```

dictionary: keys and values

Test the existence of a key:

```
>>> key in dico
```

Note that the membership test for (keys of) dictionaries is faster than for lists: as for sets, they are hash tables. We go from O(n) to O(1).

Recovering the keys:

```
>>> dd={'a':10, 'b':20, 'c':30}
>>> [*dd]
['a', 'b', 'c']
```

Update multiple values at once:

```
>>> dd.update(c=5, d=3, e=2)
```

Either the keys are created or their values are updated.

dictionary: accession

First, second value of a dictionary:

```
dd = iter(dico)
next(dd)
next(dd)
```

When accessing an element, dd[k] returns an error if k is not a key of the dd dictionary.
An alternative is to use get():

```
>>> dd.get('d')
>>> dd.get('d', 5)
5
>>> dd
{'a': 10, 'b': 20, 'c': 30}
```

dictionary: deletion

Delete the last added item:

```
>>> dd={}
>>> dd[1]=5
>>> dd[2]=6
>>> dd[3]=7
>>> dd.popitem()
(3, 7)
>>> dd
{1: 5, 2: 6}
```

Recover a value while deleting the key:

```
>>> dd.pop(2)
6
>>> dd
{1: 5}
```

dictionary: deletion

Delete the last added item:

```
>>> dd={}
>>> dd[1]=5
>>> dd[2]=6
>>> dd[3]=7
>>> dd.popitem()
(3, 7)
>>> dd
{1: 5, 2: 6}
```

Recover a value while deleting the key:

```
>>> dd.pop(2)
6
>>> dd
{1: 5}
```

this returns dict[key] by deleting it from dict, if the key exists. And which, otherwise, raises the KeyError exception. To return a default value if it doesn't exist:

```
>>> dd.pop(2)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
KeyError: 2
>>> dd.pop(2, 5)
5
>>>
```

dictionary: operations

Copy (deep) a dictionary:

```
>>> dico = {'a':1, 'b':2}
>>> dico2 = dico.copy()
>>> dico2['c'] = 3
>>> dico
{'a': 1, 'b': 2}
```

Making an over-dictionary:

```
>>> {'x':5, **dico}
{'x': 5, 'a': 1, 'b': 2}
```

And to concatenate two dictionaries:

```
>>> dico3 = {'x':7, 'y':8}
>>> dict(dico, **dico3)
{'a': 1, 'b': 2, 'x': 7, 'y': 8}
```

dictionary <-> XML

Save a dictionary in XML:

```
>>> import xmltodict
>>> xmltodict.unparse(dico, output=open('fic.xml','wb'))
```

the reverse:

```
>>> with open('fic.xml', 'rb') as f:
        dico = xmltodict.parse(f)
```

dictionary: max elements

Get the key of the largest value in a dictionary:

```
>>> max(dd, key = dd.get)
```

Obtain the key-value pair corresponding to the largest of the values in a dictionary:

```
>>> max(dd.items(), key = lambda x: x[1])
```

dictionary: defaultdict

```
>>> import collections
>>> dd=collections.defaultdict(lambda: 0)
>>> dd[4]
0
>>> dd
defaultdict(<function <lambda> at 0x7fb87e89b520>, {4: 0})
>>> dd=collections.defaultdict(lambda: {})
>>> dd[4]
{}
>>> dd[6][7]=8
>>> dd
defaultdict(<function <lambda> at 0x7fb87e89b490>, {4: {}, 6: {7: 8}})
```

dictionary: defaultdict

```
>>> import collections
>>> dd=collections.defaultdict(lambda: 0)
>>> dd[4]
0
>>> dd
defaultdict(<function <lambda> at 0x7fb87e89b520>, {4: 0})
>>> dd=collections.defaultdict(lambda: {})
>>> dd[4]
{}
>>> dd[6][7]=8
>>> dd
defaultdict(<function <lambda> at 0x7fb87e89b490>, {4: {}, 6: {7: 8}})
```

see also: `from sortedcontainers import SortedDict`

Structures of control

if

```
>>> x = 5 if a==3 else 0
```

for

else in for :

```
>>> a="123"
>>> for each in a:
...     if each == 0:
...         break
... else:
...     print('break statement did not executed')
...
break statement did not executed
```

enumerate

```
>>> L=[(1,2),(2,3)]
>>> for index, item in enumerate(L):
...     print(index, item)
...
0 (1, 2)
1 (2, 3)
```

while

while and walrus operator:

```
line = f.readLine()
while line:
    print(line)
    line = f.readLine()
```

can be simplified in:

```
while line := f.readLine():
    print(line)
```

try ... except

```
try:
    block-1 ...
except Exception1 as e:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code of block-1 is executed.

If this code raises an exception, the different "except" blocks are tested: if the exception is of class Exception1, handler-1 is executed; otherwise if it is of class Exception2, handler-2 is executed, etc.

If no exception is raised, the else-block is executed.

No matter what has happened before, the final-block is executed once all the above has been completed.

Functions

functions: definition

```
>>> def increment(s):
...     """
...     doctring
...     """
...     y = x + 1
...     return there
...
```

recursive:

```
>>> def facto(n):
...     if n == 1 :
...         return 1
...     else :
...         return n*facto(n-1)
...
```

functions: definition

```
>>> def increment(s):
...     """
...     doctring
...     """
...     y = x + 1
...     return there
...
```

```
>>> from itertools import accumulate
>>> import operator
>>> list(accumulate(range(1, 11),
operator.mul))
```

recursive:

```
>>> def facto(n):
...     if n == 1 :
...         return 1
...     else :
...         return n*facto(n-1)
...
```

functions: special parameters

For a function, there are special parameters capable of intercepting an indefinite number of values. The signature of the function becomes polymorphic.

*args: placed at the end of the parameter list, this variable aggregates in a tuple all the parameters that have not been defined.

```
>>> def toto(x, *args):
...     print args
...
>>> toto(5, 2,'a')
(2, 'a')
```

functions: special parameters

**kw placed just after *args, this variable intercepts all the remaining named parameters. They are stored in a dictionary.

```
>>> def toto(x, *args, **kw):
...     print args
...     print kw
...
>>>
>>> toto(5, 2,'a')
(2, 'a')
{}
>>> toto(5,'a', r = 2)
('a',)
{'r': 2}
```

functions: lambda

The lambdas are there to make disposable functions, usable only once.

```
>>> (lambda x, y: x*y)(2, 6)
12

>>> list(map(lambda a, b: a+b, [1, 2, 3], [2, 4]))
[3, 6]
```

Files and directories

Reading/writing files

Read files:

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

write in files:

```
with open('file_name.txt', 'w') as f:
    f.write(string_data)
```

=> `'a'` to append

Read an ISO-8859-1 file:

```
import codecs
codecs.open('myfile', 'r', 'iso-8859-1').read()
```

files: info

File size:

```
os.stat('somefile.txt').st_size
```

Get the name of the most recent file:

```
files = glob.glob('../SynologyDrive/predictops*.csv')
max_file = max(files, key=os.path.getctime)
```

Knowing if a file is less than an hour old:

```
dt.datetime.fromtimestamp(
os.path.getctime('predictops.2021-08-23-6.csv')) >
dt.datetime.now()-dt.timedelta(hours=1))
```

pickle

To store or restore any object:

```
import pickle
with open('file_name.pkl', 'wb') as f:
    pickle.dump(object, f)

with open('file_name.pkl', 'rb') as f:
    object = pickle.load(f)
```

OS

Test if a directory exists:

```
>>> from os.path import isdir
>>> isdir('my_dir')
True
```

Remove a file:

```
>>> os.remove(‘my_file’)
```

pathlib

To create an object of type path:

```
>>> import pathlib
>>> path = pathlib.Path('foo') / 'bar' / 'foo.txt'
```

We then obtain an object of the PosixPath or WindowsPath class, depending on the platform.

To create a directory:

```
>>> p = pathlib.Path("my_dir")
>>> p.mkdir(exist_ok=True, parents=True)
```

Get the current directory:

```
>>> pathlib.Path.cwd()
```

pathlib

Go to his home :

```
>>> p = pathlib.Path.home()
```

Create an empty file:

```
>>> p = pathlib.Path('my_file.txt')
>>> p.touch()

>>> p.name, p.stem, p.suffix
('my_file.txt', 'my_file', '.txt')
```

The pathlib objects correspond to paths that can be absolute (starting from the root) or relative, which we can test:

```
>>> p.is_absolute()
```

To obtain the absolute path:

```
>>> pathlib.Path('.').resolve()
```

pathlib

Get the parent directory of a pathlib object (directory or file)

```
>>> p.parent
```

Get the current directory of the executed file:

```
>>> pathlib.Path(__file__)
```

Change the extension of a file

```
>>> p.rename(p.with_suffix('.fasta'))
```

Add subdirectories to a directory defined with pathlib:

```
>>> p = p.joinpath(*['results', 'dept'])
```

Iterate on a directory/subdirectory:

```
for path in p.iterdir():
    for subpath in path.iterdir():
```

zip files

```
import zipfile
with zipfile.ZipFile('fichier.zip', 'w') as file:
    file.write('contenu.txt')

with open('fichier.zip', 'rb') as file:
    zipfile.ZipFile(file).extractall('.')
```

xlsx files

Get the list of sheets:

```
from openpyxl import load_workbook
wb = load_workbook(filename='fichier.xlsx',
                   read_only=True)
print(wb.sheetnames)
```

Read a given sheet from a file:

```
from openpyxl import load_workbook
wb = load_workbook(filename='fichier.xlsx',
                   read_only=True)
ws = wb['2012']
for row in ws.iter_rows(min_row=4):
    if row[1].value != None:
        print(row[4].value)
```

xls (old) files

```
from xlrd import open_workbook

wb = open_workbook('fic.xls')

# sheet names
print(wb.sheet_names())

ws = wb.sheet_by_index(0) # or by_names
for row in range(1, ws.nrows):
    print(ws.cell_value(row, 4)) # column 5
```

csv files: read

Reading line by line:

```
with open('file.csv', mode='r') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        print(float(row[1]))
```

Reading line -> dictionary

```
with open('file.csv', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    next(csv_reader, None)  # skip the headers
    for row in csv_reader:
        print(float(row['GC rate']))
```

csv files: write

```
import csv
with open('fic.csv', mode='w') as f:
    fieldnames =
['accession']+list(dico[next(iter(dico))].keys())
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()
    for k in dico:
        writer.writerow({'accession':k, **dico[k]})
```

Modules

logging

config parser

statistics

```
from statistics import mean, stdev
```

datetime

distance: Hamming

Number of different characters:

```
>>> from Levenshtein import hamming
>>> hamming("spam","swim")
2
```

distance: Levenshtein

Measure the similarity between two strings (edit distance) :

```
>>> from Levenshtein import distance, ratio

>>> ratio('Levenshtein', 'Lenvinsten')
0.7619047619047619

>>> distance('Levenshtein', 'Lenvinsten')
4
```

=> It takes at least 4 insertion/deletion/replacement operations to go from the first to the second.

itertools : product

1 2 3 4 5 6 7 8 9 = 2002

Spaces, + and * must be placed so that the equality is fair.

itertools : product

1 2 3 4 5 6 7 8 9 = 2002

Spaces, + and * must be placed so that the equality is fair.

itertools : product

1 2 3 4 5 6 7 8 9 = 2002

Spaces, + and * must be placed so that the equality is fair.

```
>>> from itertools import product
>>> eqn = '%s'.join(list('123456789')) + '==2002'
>>> eqn
'1%s2%s3%s4%s5%s6%s7%s8%s9==2002'
>>> for ops in product(*[['', '+', '*']]*8):
        if eval(eqn % ops):
            print(eqn % ops)


1*23+45*6*7+89==2002
1*2+34*56+7+89==2002
```

more_itertools: map_reduce

```
>>> from more_itertools import map_reduce
>>> data = 'This sentence has words of various lengths in it, both
short ones and long ones'.split()
>>> keyfunc = lambda x: len(x)
>>> map_reduce(data, keyfunc)
defaultdict(None,
{4: ['This', 'both', 'ones', 'long', 'ones'],
8: ['sentence'],
3: ['has', 'it,', 'and'],
5: ['words', 'short'],
2: ['of', 'in'],
7: ['various', 'lengths']})
```

more_itertools: map_reduce

```
>>> valuefunc = lambda x: 1
>>> map_reduce(data, keyfunc, valuefunc)
defaultdict(None,
{4: [1, 1, 1, 1, 1],
8: [1],
3: [1, 1, 1],
5: [1, 1],
2: [1, 1],
7: [1, 1]})

>>> reducefunc = sum
>>> map_reduce(data, keyfunc, valuefunc, reducefunc)
defaultdict(None, {4: 5, 8: 1, 3: 3, 5: 2, 2: 2, 7: 2})
```

more_itertools: split_at

Transform a list into a list of lists split according to the occurrence of an element ?
(works with any iterable)

```
lines = [ "erhgedrgh", "erhgedrghed", "esdrhesdresr", "ktguygkyuk",
"-------------", "srdthsrdt", "waefawef", "ryjrtyfj",
"-------------", "edthedt", "awefawe", ]
list(more_itertools.split_at(lines, lambda x: '-------------' in
x))

 =>  [['erhgedrgh', 'erhgedrghed', 'esdrhesdresr', 'ktguygkyuk'],
['srdthsrdt', 'waefawef', 'ryjrtyfj'], ['edthedt', 'awefawe']]
```

more_itertools: partition

Separates an iterable in two, according to a boolean function:

```
is_old = lambda x: datetime.now() - x < timedelta(days=30)

old, recent = partition(is_old, dates)

list(old)
# [datetime.datetime(2015, 1, 15, 0, 0), datetime.datetime(2019, 2,
1, 0, 0), datetime.datetime(2018, 2, 4, 0, 0)]

list(recent)
# [datetime.datetime(2020, 1, 16, 0, 0), datetime.datetime(2020, 1,
17, 0, 0), datetime.datetime(2020, 2, 2, 0, 0)]
```

Image

convert an image to grayscale:

```
>>> from PIL import Image as im
>>> from PIL import ImageOps
>>> dd=im.open('my_image.png')
>>> ImageOps.grayscale(dd).save(my_new_image.png')
```

IDEs, etc.

ipython

xonsh - Python-powered shell

```
christophe@coriace:~ ×   christophe@coriace: ~ | xonsh ×   christophe
(base) christophe@coriace ~ $ ls
Bureau          Modèles              Untitled.ipynb   deces_naissances.csv
Documents       Musique              Vidéos           go
Downloads       Public               config           igv
Images          Raisons_sortie.txt   deces.csv        interventions.csv
Malaya-Speech   Téléchargements      deces_dates.csv  liste.txt
(base) christophe@coriace ~ $ 2+1
3
(base) christophe@coriace ~ $ a = 5; print(a)
5
(base) christophe@coriace ~ $ from os import listdir
(base) christophe@coriace ~ $ for k in listdir():
.............................     print(k.upper())
.............................     break
.............................
.ASPELL.EN.PREPL
(base) christophe@coriace ~ $
```

spyder

Miscellanous

exec

```
>>> statement = '''a = 10; print(a + 5)'''
>>> exec(statement)
15
```

The zen of python (easter egg)

```
[christophe@coriace ~]$ python
Python 3.10.9 (main, Dec 19 2022, 17:35:49) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

The zen of python (easter egg)

```
[christophe@coriace ~]$ python
Python 3.10.9 (main, Dec 19 2022, 17:35:49) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is bet
Sparse is b
Readability
Special cas                                          rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

```
import antigravity
```

femto-st SCIENCES & TECHNOLOGIES