Pense-bête Python

— affectations, appels de fonctions, arithmétique (entier, flottant, ...)

```
x = "hello world"
print(x)
y = 2+2
y += 2 # adds 2 to y, equivalent to y = y+2
y = 2*2
y *= 2 # multiplies y by 2
3/2 # float : 1.5
3//2 # euclidean division : 1
3 % 2 # the remainder in the euclidean division
2**5 # 2 to the power 5
```

— conditions

```
x == y # is equal to
x < y # stricly smaller
x <= y # smaller or equal
x != y # not equal to
x and y
x or y
not x
x in l # tests if x is an element of l
```

— if/then/else

```
if x :
    # block if x is True
elif y :
    # block if x is False and y is True
else :
    # block if x and y are False
```

— définition de fonction

```
def function(arg1, arg2, ...):
    '''
    doc between 3 apostrophes
    '''
    # function body
    return x # optional, the result of the function

def function(arg1, arg2=<default value>, ...):
    '''
    doc between 3 apostrophes
    '''
    # an argument can be given a default value using 'arg=<value>' in the parameters

def function(arg1, arg2, *args):
    '''
    doc between 3 apostrophes
    '''
    # we can define a function with an unknown number of arguments using *args
    # args is then a tuple in the scope of the function

# One-line functions and anonymous functions:
f = lambda x,y : x*y + 2*x # f : x,y |-> xy+2x
                           # "lambda x,y :" serves to specify that x and y are parameters
{0 : (lambda : print(0)), 1 : (lambda : print(1))}[x]() # prints 0 if x=0, 1 if x=1 (or raises an error)
```

— boucle conditionnelle

```
while x:
    # block to do while x is True
```

— boucle pour

```
for x in range(n):
    # x = 0, 1, ..., n-1
for x in range(n0, n):
    # x = n0, n0+1, ..., n-1
for x in range(n0, n, t):
    # x = n0, n0+t, ...

for x in l:
    # x goes through the objects in l
for x in range(len(l)):
    # x goes through the indices of l
for (x,y) in [(0,1), (2,5), (-6,4)]:
    # x and y unpack the pairs in the previous list
```

— listes

```
l = []
l = [1, 6, 4, 2, 3]
l = [0]*10 # a list of 0s of size 10
len(l) # the number of elements
l[i] # the ith element
l[-i] # the ith element from the right
l.append(x) # adds x to the end
x = l.pop() # removes the last element, and stores it in x
l.remove(x) # removes element x from the list, throws an error if not (x in l)
l.copy()
l[:j] # the first j elements
l[i:] # the rest from index i
l[i:j] # elements between i and j
l1 + l2 # concatenation
a, b = [1,2] # unpacks the list a<-1 and b<-2
list(x) # tries to convert x in a list e.g. list(range(5))
```

— chaînes de caractère (partage beaucoup avec les listes)

```
s = "hello world"
s[3] # 'l'
s + " !" # "hello world !"
s[:4] # "hell"
...
a, b = "hi" # unpacking
str(x) # tries to convert x to a string
", ".join(["hell", "o", "world"]) # "hell, o, world"
```

— tuples

```
t = (0,) # a tuple with one element
t = (1, 6, 4, 2, 3)
t = (0,)*10 # a tuple of 0s of size 10
...
a, b = (2,5) # unpacking
tuple(x) # tries to convert x to a tuple
```

— dictionnaires

```
d = {"a":0, "b":1, "c":2}
d["a"] # the value associated to "a"
d["d"] # error : "d" not in dictionnary
d["d"] = 3 # adds the key/val pair "d":3
d.get("d", -1) # d["d"] if "d" in dictionnary, -1 otherwise
d.keys() # gets the keys
d.values() # gets the values
x = d.pop("a") # removes "a" from the keys, and stores the associated value in x
```

```python
for k in d: # k goes through the keys
    pass
```

— compréhensions

```python
[x+3 for x in range(10) if x%2==0] # [3, 5, 7, 9, 11]
{x+3 : x for x in l if x!= 3}
```

— classes

```python
class Point:
    # in the following methods, self refers to the instance of the class
    def __init__(self, x, y):
        '''
        how a Point instance is initialised
        '''
        self.x = x
        self.y = y

    def __str__(self):
        '''
        defines what str(.) returns
        '''
        return '('+ str(x) + ', ' + str(y) + ')

    def __repr__(self):
        '''
        defines what print(.) prints
        '''
        return str(self)

    def __eq__(self, p2):
        '''
        how to test equality
        '''
        return(self.x==p2.x and self.y==p2.y)

    def __add__(self, p2):
        '''
        what .+. returns
        '''
        return Point(self.x+p2.x, self.y+p2.y)
    ...

    @classmethod
    def origin(cls):     # here the default argument becomes 'cls', which stands for the name of the class
        return cls(0,0)  # in this context, this is equivalent to doing 'Point(0,0)'

    def distance_from_origin(self):
        '''
        computes the 1-distance from origin
        '''
        return abs(self.x)+abs(self.y)

p1 = Point(5,3)
p2 = Point(-2,2)
print(p1) # (5, 3)
p1==p2 # False
p1+p2 # Point(3,5)
p1.distance_from_origin() # 8


class Interval(Point): # a subclass of Point

    def __init__(self, x, y):  # not mandatory. If not defined, will use the __init__ of the superclass
        if x<y:
            super().__init__(x, y)  # we use the __init__ method of the superclass
        else:
            super().__init__(y, x)

    def distance_from_origin(self): # overwriting of the method
        if self.x >= 0:
            return self.x
        if self.y <= 0:
            return -self.y
        return 0

i1 = Interval(2,3)
```

```
i2 = Interval(4,5)
i1==i2 # False
i1+i2 # Interval(6,8)

# When calling a method of Interval, Python will first check for the method in "Interval",
# if not found, it will check in "Point", etc...
# The method resolution order can be found with "Interval.__mro__"
```

— utilisation de librairies

```
import math
math.sqrt(2)

from math import * # imports all the functions from math without giving
sqrt(2)            # them a separate namespace. Beware of overlap!

from math import sqrt # imports a single function from math
sqrt(2)

import math as mth
mth.sqrt(2)
```

— obtention d'informations

```
type(x) # gets the type of x
dir(x) # gets the methods of x or its class

import inspect
import random
print(inspect.getsource(random.randint)) # prints the source code of randint
                                         # does not work on built-in objects
                                         # nor on interactively defined ones
print(inspect.getdoc(random.randint)) # prints the doc
print(inspect.getfile(random.randint)) # prints the path to the file
                                       # containing the object
```

— tests unitaires avec `unittest`

```
import unittest
from <tested module> import *

class <class test name>(unittest.TestCase):

    def <method/function test name>(self):
        <assignments>
        # possible assertions and what they test:
        self.assertEqual(a, b) # a == b
        self.assertNotEqual(a, b) # a != b
        self.assertTrue(x) # bool(x) is True
        self.assertFalse(x) # bool(x) is False
        self.assertIs(a, b) # a is b
        self.assertIsNot(a, b) # a is not b
        self.assertIsNone(x) # x is None
        self.assertIsNotNone(x) # x is not None
        self.assertIn(a, b) # a in b
        self.assertNotIn(a, b) # a not in b
        self.assertIsInstance(a, b) # isinstance(a, b)
        self.assertNotIsInstance(a, b) # not isinstance(a, b)

    def setUp(self):
        <code to be run before every test of the class>
        # example:
        self.x = ... # self.x can then be used in other test methods

    def tearDown(self):
        <code to be run after every test of the class>
```

— gestion des erreurs

```
try:
    <block where there could be an error>
except <exception_type>: # exception type is optional (if none given, all exceptions are caught)
```

```
                        # exception_type ::= ValueError, TypeError, SyntaxError, NameError, ...
                        # exception_type can be left blank, all errors are then caught
    <what to do if an error is caught>
else: # optional
    <what to do if no error is caught>
finally: # optional
    <what to do at the end anyway>

raise <error_type>("<error message>") # raises an error of type <error_type>,
                                      # and prints an explanatory message
```
