meystre_rapport_python

June 11, 2023

1 Rapport - Python par la pratique - Samuel Meystre

1.1 Table des matières

1.2 Introduction

Lors du 2ème semestre de 3ème année, le cours à choix de python par la pratique a été choisi pour pouvoir approfondir dans ce langage de programmation.

Car nous avons utilisé ce langage dans le cadre du cours TSA (traitement du signal appliqué), mais sans aller dans les détails (utilisation uniquement de numpy, matplotlib, scipy ou un autre module python pour le traitement du signal).

Le cours Python par la pratique permet de combler ce que nous avions pas vu lors du cours TSA. Notamment avec les règles de programmation (zen de Python), les structures de données, Pandas, Click ou encore l'utilisation de templates HTML avec Jinja2 et Flask.

1.3 Structures de données

1.3.1 Scalaires

Nous avons vu l'utilisation des scalaires dans python et également vu comment s'en servir.

[1378]:
```
i = 9           # integer
f = 5.674       # float
c = 2 + 3j      # complex
s = 'Coucou'    # string
b = True        # boolean
n = None
```

Le type string ne peut pas être modifié comme montré ci-dessous:

[1379]:
```
# s[1] = 'i'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[40], line 1
----> 1 s[1] = 'i'

TypeError: 'str' object does not support item assignment
```

1.3.2 Conteneurs

En plus des scalaires, nous avons vu également les différents conteneurs présents en Python:

Listes [] Les listes peuvent s'apparenter à des tableaux en C. Elles ne sont pas hashable et elles possèdent un itérateur.

[1380]:
```
l = []      # Création d'une liste vide
l = ['Chien', 'Chat', 'Souris', 'Chat', 'Chat', 'Hamster']
print(l)
```

```
['Chien', 'Chat', 'Souris', 'Chat', 'Chat', 'Hamster']
```

[1381]:
```
l[-1]
```

[1381]:
```
'Hamster'
```

[1382]:
```
l[2:-2]
```

[1382]:
```
['Souris', 'Chat']
```

[1383]:
```
l[::2]
```

[1383]:
```
['Chien', 'Souris', 'Chat']
```

[1384]:
```
# ajout d'un élément dans une liste
l.append('Cochon')
print(l)
```

```
['Chien', 'Chat', 'Souris', 'Chat', 'Chat', 'Hamster', 'Cochon']
```

[1385]:
```
# comptage des éléments
l.count('Chat')
```

[1385]: 3

[1386]:
```
# itérateur
iterator = iter(l)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

```
Chien
Chat
Souris
```

[1387]:
```
# hash d'une liste (ERROR)
# print(hash(l))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[29], line 2
      1 # hash d'une liste (ERROR)
----> 2 print(hash(l))

TypeError: unhashable type: 'list'
```

Tuples () Le tuple est un conteneur ressemblant à une liste, mais qui est hashable. Le tuple ne peut pas être modifié et il est itérable.

[1388]:
```
t = (1,2,3,4)
t
```

[1388]: (1, 2, 3, 4)

[1389]:
```
# hash d'un tuple
print(hash(t))
```

```
5908993871830677792
```

[1390]:
```
# itérateur
iterator = iter(t)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

```
1
2
3
```

[1391]:
```
# Modification d'un tuple (ERROR)
# t[2] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[55], line 2
      1 # Modification d'un tuple (ERROR)
----> 2 t[2] = 0

TypeError: 'tuple' object does not support item assignment
```

Dictionnaires Le dictionnaire est conteneur ayant une clé et une valeur associée à la clé. Les clées sont hashable.

[1392]:
```
d = {23: 'vingt-trois', 42: 'La réponse', 95: 'neuf-cinq'}
d
```

[1392]:
```
{23: 'vingt-trois', 42: 'La réponse', 95: 'neuf-cinq'}
```

[1393]:
```
d.items()
```

[1393]:
```
dict_items([(23, 'vingt-trois'), (42, 'La réponse'), (95, 'neuf-cinq')])
```

[1394]:
```
d.values()
```

[1394]:
```
dict_values(['vingt-trois', 'La réponse', 'neuf-cinq'])
```

[1395]:
```
d.keys()
```

[1395]:
```
dict_keys([23, 42, 95])
```

[1396]:
```
d[42]
```

[1396]:
```
'La réponse'
```

Set Le set est un dictionnaire avec que des clés.

[1397]:
```
st = {23, 42, 95}
st
```

[1397]:
```
{23, 42, 95}
```

1.4 Fonctions

Le langage Python permet de créer ses fonctions.

[1398]:
```
def my_func(a, b):
    return a**b
```

[1399]:
```
c = my_func(4, 2)
c
```

[1399]: 16

1.5 Classes

Le langage Python est orienté objet. Il permet donc de créer des classes.

[1400]:
```
from unidecode import unidecode
```

[1401]:
```
class my_class:
    def __init__(self, words):
        self.words = words
        self.counter = len(words)

    def __iter__(self):
        return self

    def __next__(self):
        if self.counter <= 0:
            raise ValueError('Plus de mots')
        self.counter -= 1
        word = self.words[self.counter]
        word = unidecode(word)
        word = word.upper()
        return word
```

Ouvrir des fichiers

[1402]:
```
filename = 'mots.txt'

fp = open(filename, 'r') # 'w', 'a'
fp.read() # Lis tout le fichier
fp.readline() # Lis la prochaine ligne
fp.readlines() # Lis toutes les lignes
fp.close()
```

Ou:

[1403]:
```
words_table = []
with open(filename,'r') as fd:
```

```
    while line := fd.readline():
        word = line.split('\n')[0]
        words_table.append(word)
```

Instanciation d'une classe

[1404]:
```
mots = my_class(words_table)

for k in range(len(words_table)):
    print(next(mots))
```

```
CHENE
FURIEUX
LOOPING
RUISSEAU
BRUTE
TRUITE
```

1.6 Tests, destructuration, zip, déréférencement et surcharge

1.6.1 Tests

in Ce test permet de vérifier si une valeur souhaitée se trouve dans un conteneur

[1405]:
```
l = [1,2,3,4,5,6]
5 in l
```

[1405]: `True`

[1406]:
```
0 in l
```

[1406]: `False`

all Retourne `True` si tous les éléments sont vrai

[1407]:
```
la = ['True', True, 1, 'Oiseau']
all(la)
```

[1407]: `True`

[1408]:
```
lna = [78, 'Truite', True, 'Fraise', False]
all(lna)
```

[1408]: `False`

any Retourn `True` si au moins un élément est vrai

[1409]:
```
any(lna)
```

[1409]: True

[1410]:
```
lnaa = [0,0,0,0,0,0,None,False]
any(lnaa)
```

[1410]: False

1.6.2 Destructuration

La destructuration en Python permet, par exemple, d'échanger deux valeurs ou de récupérer la valeur de chaque élément d'une liste, d'un tuple ou d'un dictionnaire.

[1411]:
```
a = 9
b = 5
a, b
```

[1411]: (9, 5)

[1412]:
```
a, b = b, a
a, b    # les valeurs ont été échangée
```

[1412]: (5, 9)

[1413]:
```
l = [1,2,3,4]
a,b,c,d = l
a,b,c,d
```

[1413]: (1, 2, 3, 4)

[1414]:
```
d = {51: 'Apéro', 42: 'Réponse'}
a, b = d
a, b
```

[1414]: (51, 42)

Faire `a, b = d` ne prend que les clés du dictionnaire. Pour avoir les valeurs, voici ce qu'il faut faire:

[1415]:
```
a, b = d.values()
a, b
```

[1415]: ('Apéro', 'Réponse')

La déstructuration d'une liste fonctionne aussi avec l'opérateur `*`.
Suivant sa position, cela permet de récupérer, par exemple, la première valeur de la liste puis tout le reste dans une nouvelle variable/liste.

[1416]:
```
head, *reste = l
print(head)
```

```
print(reste)
```

```
1
[2, 3, 4]
```

[1417]:
```
*head, reste = l
print(head)
print(reste)
```

```
[1, 2, 3]
4
```

En plus de tout cela, la desctructuration permet également d'extraire l'indice d'une valeur dans une boucle `for`.

[1418]:
```
for i, v in enumerate(l):
    print(f'Indices: {i}, Valeurs: {v}')
```

```
Indices: 0, Valeurs: 1
Indices: 1, Valeurs: 2
Indices: 2, Valeurs: 3
Indices: 3, Valeurs: 4
```

1.6.3 Zip

Cet opérateur permet de "fusionner" deux listes ensemble.

[1419]:
```
firstnames = ['John', 'Emmet', 'Luke']
lastnames = ['Doe', 'Brown', 'Skywalker']

list(zip(firstnames, lastnames))
```

[1419]:
```
[('John', 'Doe'), ('Emmet', 'Brown'), ('Luke', 'Skywalker')]
```

Pour que ce soit plus joli à l'affichage:

[1420]:
```
[' '.join(x) for x in list(zip(firstnames, lastnames))]
```

[1420]:
```
['John Doe', 'Emmet Brown', 'Luke Skywalker']
```

1.6.4 Déréférencement

Le déréférencement se fait à l'aide de `*args` ou `**kwargs` dans une fonction.
Cela permet de récupérer tous les arguments entrés dans la fonction.

[1421]:
```
def foo(*args, **kwargs):
    print(args)
    print(kwargs)
```

[1422]:
```
foo(8,2,3,'pamplemousse',42)
```

```
(8, 2, 3, 'pamplemousse', 42)
{}
```

Une fonction utilisant le `**kwargs`:

[1423]:
```
def operate(a, b, **kwargs):
    if 'add' in kwargs:
        print(f'{a}+{b} = {a+b}')
    if 'sub' in kwargs:
        print(f'{a}-{b} = {a-b}')
```

[1424]:
```
operate(5, 4, add=True)
operate(5, 4, sub=True)
```

```
5+4 = 9
5-4 = 1
```

1.6.5 Surcharge

La surcharge d'un opérateur permet de se passer, dans le cadre d'une classe, de l'appel de la fonction addition, par exemple:

[1425]:
```
class Number:
    def __init__(self, number):
        self.number = number
    def __add__(self, number):
        self.number += number
```

[1426]:
```
n = Number(42)
n.number
```

[1426]:
```
42
```

[1427]:
```
n + 9
n.number
```

[1427]:
```
51
```

1.7 Compréhension, lambda, map et filter

1.7.1 Compréhension

La compréhension permet de travailler avec une liste sans devoir passer par une boucle `for` et donc, d'optimiser le code.

[1428]:
```
l = ['Chat','Souris','Chien','Souris','Chat']
l
```

[1428]:
```
['Chat', 'Souris', 'Chien', 'Souris', 'Chat']
```

Admettons qu'il faut changer "Chat" par "Tom". En compréhension cela donne:

[1429]:
```
nl = ['Tom' if x=='Chat' else x for x in l]
nl
```

[1429]:
```
['Tom', 'Souris', 'Chien', 'Souris', 'Tom']
```

1.7.2 Lambda

`lambda` est une fonction Python contenant qu'une seule expression. Elle permet de faire des fonctions simple si la compréhension deviet trop complexe.

[1430]:
```
x = lambda a, b : a**b
res = x(2, 3)
res
```

[1430]:
```
8
```

1.7.3 Map

`map` est une fonction qui retourne un itérable du résultat.

[1431]:
```
def double_num(n):
    return n + n

t = (1, 2, 3, 4)

res = map(double_num, t)
print(list(res))
```

```
[2, 4, 6, 8]
```

La fonction `map`peut être utilisée avec `lambda` pour afficher, par exemple, directement le résultat sans passer par une fonction.

[1432]:
```
t = (5, 6, 7, 8)
res = map(lambda x: x + x, t)
print(list(res))
```

```
[10, 12, 14, 16]
```

1.7.4 Filter

La fonction `filter` permet de filtrer des éléments d'un conteneur selon une condition donnée. Comme pour `map`, `filter` peut être directement utilisé avec `lambda`, ce qui est le cas usuellement dans la programmation en Python

[1433]:
```
mots = ['souris', 'chat', 'chien', 'singe', 'oiseau', 'ornythorynque']
thr = 6
mots_tri = list(filter(lambda x: len(x) >= thr, mots))
mots_tri
```

[1433]:
```
['souris', 'oiseau', 'ornythorynque']
```

1.8 Namedtuple

Le module `namedtuple` contribue à de faire des `tuple` qui permettent d'accéder aux valeurs avec noms descriptif et le `t.` au lieu des indices.

[1434]:
```
from collections import namedtuple

Cat = namedtuple('Cat',('name','genre'))
c = Cat('Felix','MALE')
c
```

[1434]:
```
Cat(name='Felix', genre='MALE')
```

[1435]:
```
c.name
```

[1435]:
```
'Felix'
```

[1436]:
```
c.genre
```

[1436]:
```
'MALE'
```

[1437]:
```
#c.name = 'oops'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[980], line 1
----> 1 c.name = 'oops'

AttributeError: can't set attribute
```

1.9 Numpy

Numpy est un module Python ayant comme particularité de faire des calculs matriciels (à l'image de MatLab).
Car, en Python, il n'est pas possible de faire des opérations mathématiques avec deux listes, cela donnera un résultat totalement différent à ce qui est attendu.
Exemple:

[1438]:
```
l1 = [1,2,3]
l2 = [4,5,6]
```

```
l3 = l1+l2
l3
```

[1438]: `[1, 2, 3, 4, 5, 6]`

Faire une addition entre deux listes équivaut à les concaténeer.
Pour faire une addition entre deux listes (vecteurs) il faut utiliser le module Numpy.

[1439]:
```
import numpy as np

v1 = np.array([1,2,3])
v2 = np.array([4,5,6])

v3 = v1+v2
v3
```

[1439]: `array([5, 7, 9])`

Avec Numpy, c'est le monde de MatLab qui s'ouvre sur Python. Il est possible d'avoir toutes les fonctions mathématiques.
Les matrices ou vecteurs créés avec Numpy sont également compatibles avec la notation des indices de Python.

[1440]:
```
v4 = v1*v2
v4
```

[1440]: `array([ 4, 10, 18])`

[1441]:
```
v = np.arange(1,11)
v
```

[1441]: `array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])`

[1442]:
```
v[1]
```

[1442]: `2`

[1443]:
```
v[-1]
```

[1443]: `10`

[1444]:
```
v[3:6]
```

[1444]: `array([4, 5, 6])`

[1445]:
```
s = np.sin(v)
s
```

[1445]:
```
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892427,
       -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111])
```

[1446]:
```
pi = np.pi
pi
```

[1446]:
```
3.141592653589793
```

1.10 Click

Click est un module Python assurant l'entrée d'arguments avec le clavier de l'ordinateur.
Cela fonctionne avec des décorateurs placés avant la fonction souhaitée.

[1447]:
```
"""
import click

@click.command()
@click.argument('a', type=float)
@click.argument('b', type=float)
@click.argument('c', type=float)
def resolve_quad(a, b, c):
    delta = b**2 - 4*a*c

    x1 = (-b + delta**0.5)/(2*a)
    x2 = (-b - delta**0.5)/(2*a)

    click.echo(f'X1 = {x1}')
    click.echo(f'X2 = {x2}')


if __name__ == '__main__':
    resolve_quad()
"""
```

[1447]:
```
"\nimport click\n\n@click.command()\n@click.argument('a',
type=float)\n@click.argument('b', type=float)\n@click.argument('c',
type=float)\ndef resolve_quad(a, b, c):\n    delta = b**2 - 4*a*c\n\n    x1 =
(-b + delta**0.5)/(2*a)\n    x2 = (-b - delta**0.5)/(2*a)\n\n    click.echo(f'X1
= {x1}')\n    click.echo(f'X2 = {x2}')\n\n\nif __name__ == '__main__':\n
resolve_quad()\n"
```

1.11 Pandas

Pandas est un module Python utilisé pour le traitement de données.
Ce module est ainsi fait pour pouvoir traiter des centaines de milliers de données stockées dans un fichier csv par exemple.

[1448]:
```
import pandas as pd

decimals = 2
file = "foo.csv"

df = pd.read_csv(file, sep=",")
df.head()
```

[1448]:

	Unnamed: 0	A	B	C	D
0	2000-01-01	-0.543391	-0.709364	-0.155659	-0.657926
1	2000-01-02	-2.314978	-1.822206	-0.807061	-0.784460
2	2000-01-03	-4.001682	-1.717682	-1.443619	-0.260029
3	2000-01-04	-5.380479	-1.623373	-0.792619	-1.176698
4	2000-01-05	-7.003829	-1.557467	0.171087	-1.083572

Renommage de la colonne Unnamed par date:

[1449]:
```
df = df.rename(columns = {'Unnamed: 0': 'Date'})
df.head()
```

[1449]:

	Date	A	B	C	D
0	2000-01-01	-0.543391	-0.709364	-0.155659	-0.657926
1	2000-01-02	-2.314978	-1.822206	-0.807061	-0.784460
2	2000-01-03	-4.001682	-1.717682	-1.443619	-0.260029
3	2000-01-04	-5.380479	-1.623373	-0.792619	-1.176698
4	2000-01-05	-7.003829	-1.557467	0.171087	-1.083572

[1450]:
```
df.dtypes
```

[1450]:
```
Date     object
A       float64
B       float64
C       float64
D       float64
dtype: object
```

Suppression d'une colonne

[1451]:
```
df = df.drop(['D'], axis = 1)
df.head()
```

[1451]:

	Date	A	B	C
0	2000-01-01	-0.543391	-0.709364	-0.155659
1	2000-01-02	-2.314978	-1.822206	-0.807061
2	2000-01-03	-4.001682	-1.717682	-1.443619
3	2000-01-04	-5.380479	-1.623373	-0.792619
4	2000-01-05	-7.003829	-1.557467	0.171087

Ici, Pandas n'est présenté que de manière succinte, pour plus de détails voir labo03 du cours ou suivre le tutoriel de Pandas.

1.12 Jinja2 et Flask

Les modules Jinja2 et Flask sont utilisés pour faire du HTML avec Python.
Pour cela, il faut avoir un template HTML et ces modules travailleront avec ledit template.

[1452]:
```
from jinja2 import Environment, FileSystemLoader, select_autoescape
env = Environment(
    loader=FileSystemLoader('templates/'),
    autoescape=select_autoescape()
)

data = [
    {
        'name': 'Ballons',
        'variants': [
            {'name': 'Taille 5', 'price': 23},
            {'name': 'Taille 4', 'price': 21}
        ]
    },
    {
        'name': 'Chaussures',
        'variants': [
            {'name': 'Taille 42', 'price': 120},
            {'name': 'Taille 43', 'price': 130}
        ]
    }
]

template = env.get_template('template_articles.html')

with open('BonSportif.html', 'w') as fp:
    fp.write(template.render(articles=data))
```

1.13 Functools

Dans le module Functools, il y a un outils nommé `cache` qui permet de réduire le temps d'exécution d'une fonction ayant une longue exécution.

[1453]:
```
import functools

@functools.cache
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
```

```
fib(100)
```

[1453]: 354224848179261915075

1.14 Singleton

Un Singleton est un patron de conception (design pattern) qui permet de créer une classe qui n'a qu'une seule instance (voir diary 12 cours heig-python-a).

[1454]:
```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,␣
 ↪**kwargs)
        return cls._instances[cls]

class Logger(object):
    __metaclass__ = Singleton


def foo():
    logger = Logger()
    logger.error('This is an error message')
    logger.info('This is an info message')
    logger.debug('This is a debug message')

def bar():
    logger = Logger()
    logger.error('This is an error message')
```

1.15 Exceptions

Les exceptions sont les erreurs qui s'affichent lors de l'exécution d'une fonction quelconque.

[1455]:
```
codes = [4, 8, 15, 16, 23, 42]
def getCode(i):
    try:
        return codes[i]
    except IndexError:
        raise ValueError("Mauvaise valeur de i")

c = getCode(3)
c
```

[1455]: 16

[1456]:
```
#c = getCode(8)
#c
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[919], line 4, in getCode(i)
      3 try:
----> 4     return codes[i]
      5 except IndexError:

IndexError: list index out of range

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
Cell In[920], line 1
----> 1 c = getCode(8)
      2 c

Cell In[919], line 6, in getCode(i)
      4     return codes[i]
      5 except IndexError:
----> 6     raise ValueError("Mauvaise valeur de i")

ValueError: Mauvaise valeur de i
```

1.16 Conclusion

Le cours Python par la pratique a été bénéfique car il a approfondi les connaissances et en a amené de nouvelles concernant ce langage de programmation.
Même si le langage était utilisé pour le cours de traitement de signal, seul une minorité de fonctionnalités a été explorée. Les programmes écrits étaient faits avec des fonctions utiles pour ce genre de cours. Le concept de listes, tuples, dictionnaires et plein d'autres vus dans ce rapport étaient pour la plupart inconnus (à part Numpy).
Ce présent rapport résume tout ce qui a été vu lors de ce semestre.

Yverdon, le 11 juin 2023
Samuel Meystre