Résumé du cours Python

Python - HEIG-VD

Bruno Sutterlet

31.05.2023

Table des matières

Chap 1: Introduction

Control Structures

If-else L'utilisation des conditions if-else est utilisée pour mettre en place des conditions dans le code. Son fonctionnement est globalement le même que dans les autres langages de programmation.

```
if condition:
    # code
elif condition:
    # code
else:
    # code
```

Loops De même que dans la programation en C/C++, Python propose deux types de boucles : la boucle for et la boucle while :

```
for i in range(10):
    # code
    print(i)
```

```
i = 0
while i < 10:
    # code
    print(i)
    i += 1
```

Functions

Les fonctions en Python sont définies par le mot-clé `def` suivi du nom de la fonction et des paramètres entre parenthèses. La fonction peut retourner une valeur avec le mot-clé **`return`**.

```
def function_name(param1, param2):
    # code
    return value
```

File Handling

Open Pour ouvrir un fichier, on utilise la fonction `open()` qui prend en paramètre le chemin du fichier et le mode d'ouverture (lecture, écriture, etc.).

```
file = open("path/to/file", "r")
```

Les modes d'ouverture sont : + `r` : lecture + `w` : écriture + `a` : ajout + `r+` : lecture et écriture + `w+` : lecture et écriture (crée le fichier s'il n'existe pas) + `a+` : ajout et lecture (crée le fichier s'il n'existe pas)

Read Pour lire le contenu d'un fichier, on utilise la méthode `read()` qui prend en paramètre le nombre de caractères à lire. Si aucun paramètre n'est passé, la méthode lit tout le contenu du fichier.

```
file.read()
```

La methode read() retourne une chaîne de caractères qui correspond à la totalité du contenu du fichier.

Pour lire le contenu d'un fichier ligne par ligne, on utilise la méthode `readlines()` qui retourne une liste de chaînes de caractères.

```
file.readlines()
```

Elle avance le curseur à la fin de la ligne lue. Pour revenir au début du fichier, on utilise la méthode `seek()` qui prend en paramètre la position du curseur.

```
file.seek(0)
```

Write Pour écrire dans un fichier, on utilise la méthode `write()` qui prend en paramètre le texte à écrire.

```
file.write("Hello World!")
```

Pour écrire dans un fichier ligne par ligne, on utilise la méthode `writelines()` qui prend en paramètre une liste de chaînes de caractères.

```
file.writelines(["Hello", "World!"])
```

Attention, la méthode `write()` écrase le contenu du fichier. Pour ajouter du contenu à la fin du fichier, on utilise la méthode `append()` qui prend en paramètre le texte à ajouter.

```
file.append("Hello World!")
```

Pour écrire dans un fichier à une position donnée, on utilise la méthode `seek()` qui prend en paramètre la position du curseur.

```
file.seek(0)
```

Une autre méthode est de modifier la chaine de caractères retournée par la méthode `read()` et de réécrire le contenu du fichier avec la méthode `write()`.

```
file.seek(0)
content = file.read()
content = content.replace("Hello", "Bonjour")
file.seek(0)
file.write(content)
```

Attention, si le nouveau contenu est plus court que l'ancien, il faut vide le fichier avant d'écrire le nouveau contenu.

```
file.seek(0)
file.truncate()
file.write(content)
```

Chap 2: Data Structures (Scalars)

Integers

Les entiers sont des nombres entiers positifs ou négatifs. Ils sont définis par le mot-clé `int`. En python, il n'y a pas de limite de taille pour les entiers et ils possèdent des méthodes pour les manipuler comme un objet.

```
a = 1
b = 2
c = a + b
```

Voici plusieurs méthodes pour manipuler les entiers :

```
a = 1
a.bit_length() # 1
a.to_bytes(2, byteorder="big") # b'\x00\x01'
a.to_bytes(2, byteorder="little") # b'\x01\x00'
```

Floats

Les flottants sont des nombres à virgule. Ils sont définis par le mot-clé `float`. En python, il n'y a pas de limite de taille pour les flottants et ils possèdent des méthodes pour les manipuler comme un objet.

```
a = 1.0
b = 2.
c = a + b
```

Voici plusieurs méthodes pour manipuler les flottants :

```
a = 1.0
a.as_integer_ratio() # (1, 1)
a.hex() # '0x1.0000000000000p+0'
a.is_integer() # True
```

Complex Numbers

Les nombres complexes sont des nombres à virgule. Ils sont définis par le mot-clé `complex`. En python, il n'y a pas de limite de taille pour les nombres complexes et ils possèdent des méthodes pour les manipuler comme un objet.

```
a = 1 + 2j
b = 2j
c = a + b
```

Voici plusieurs méthodes pour manipuler les nombres complexes :

```
a = 1 + 2j
a.conjugate() # (1-2j)
a.imag # 2.0
a.real # 1.0
```

Attention, même s'il est possible de gérer des nombres complexes en python, il n'est pas possible de les utiliser dans les fonctions mathématiques de base comme `sin()`, `cos()`, `sqrt()`, etc. Il est souvent préférable d'utiliser des bibliothèques comme `numpy` pour manipuler des nombres complexes.

Booleans

Les booléens sont des variables qui peuvent prendre deux valeurs : `True` ou `False`. Ils sont définis par le mot-clé `bool`.

```
a = True
b = False
c = a and b
```

None

`None` est une valeur spéciale qui représente l'absence de valeur. Elle est définie par le mot-clé `None`.

```
a = None
```

Elle est souvent utilisée pour initialiser des variables qui seront modifiées plus tard dans le code. Elle est aussi utile dans tests conditionnels. En cas de valeur non définie, la valeur `None` est retournée.

```
if a is None:
    # code d'erreur
```

Chap 3: Data Structures (Collections)

Lists

Les listes sont des collections ordonnées d'éléments. Elles sont définies par le mot-clé `list`. Les éléments d'une liste peuvent être de n'importe quel type.

```
a = [1, 2, 3]
b = ["a", "b", "c"]
c = [1, "a", True]
```

Les listes sont des objets mutables et itérble. Il est donc possible de modifier les éléments d'une liste.

```
a = [1, 2, 3]
a[0] = 4
```

Les listes possèdent des méthodes pour les manipuler comme un objet.

```
a = [1, 2, 3]
a.append(4) # [1, 2, 3, 4]
a.extend([5, 6]) # [1, 2, 3, 4, 5, 6]
a.insert(0, 0) # [0, 1, 2, 3, 4, 5, 6]
a.remove(0) # [1, 2, 3, 4, 5, 6]
a.pop(0) # [2, 3, 4, 5, 6] et retourne 1
a.reverse() # [6, 5, 4, 3, 2]
a.sort() # [2, 3, 4, 5, 6]
```

Oppération in L'opération `in` permet de vérifier si un élément est présent dans une liste.

```
a = [1, 2, 3]
1 in a # True
4 in a # False
```

Oppération all L'opération `all` permet de vérifier si tous les éléments d'une liste sont vrais.

```
a = [True, True, True]
all(a) # True
a = [True, False, True]
all(a) # False
```

Oppération any L'opération `any` permet de vérifier si au moins un élément d'une liste est vrai.

```
a = [True, True, True]
any(a) # True
a = [True, False, True]
any(a) # True
a = [False, False, False]
any(a) # False
```

Tuples

Les tuples sont des collections ordonnées d'éléments. Ils sont définis par le mot-clé `tuple`. Les éléments d'un tuple peuvent être de n'importe quel type.

```
a = (1, 2, 3)
b = ("a", "b", "c")
c = (1, "a", True)
```

Les tuples sont des objets immutables et itérble. Il n'est donc pas possible de modifier les éléments d'un tuple. Mais il est possible d'accéder à un élément d'un tuple par son index.

```
a = (1, 2, 3)
a[0] = 4 # TypeError: 'tuple' object does not support item assignment
```

tips Il est possible de créer un tuple avec un seul élément en ajoutant une virgule après l'élément.

```
a = (1) # int
type(a) # <class 'int'>
a = (1,) # tuple
type(a) # <class 'tuple'>
```

Il n'est n'est pas possible de créer un tuple vide. Il faut utiliser la fonction `tuple()`.

```
a = () # SyntaxError: invalid syntax
a = tuple() # ()
```

Il n'est pas possible de modifier un tuple mais il est possible de créer un nouveau tuple à partir d'un tuple existant.

```
a = (1, 2, 3)
b = a + (4, 5, 6) # (1, 2, 3, 4, 5, 6)
```

Dictionaries

Les dictionnaires sont des collections non ordonnées d'éléments. Ils sont définis par le mot-clé `dict`. Les éléments d'un dictionnaire sont des paires clé-valeur. Les clés d'un dictionnaire doivent être uniques et immutables. Les valeurs d'un dictionnaire peuvent être de n'importe quel type.

```
a = {"a": 1, "b": 2, "c": 3}
b = {1: "a", 2: "b", 3: "c"}
c = {"a": 1, "b": "c", "d": True}
```

On peut accéder à une valeur d'un dictionnaire en utilisant sa clé.

```
a = {"a": 1, "b": 2, "c": 3}
a["a"] # 1
```

On peut modifier la valeur d'un dictionnaire en utilisant sa clé.

```
a = {"a": 1, "b": 2, "c": 3}
a["a"] = 4
a # {"a": 4, "b": 2, "c": 3}
```

On peut ajouter une nouvelle paire clé-valeur à un dictionnaire en utilisant une nouvelle clé.

```
a = {"a": 1, "b": 2, "c": 3}
a["d"] = 4
a # {"a": 1, "b": 2, "c": 3, "d": 4}
```

On peut supprimer une paire clé-valeur d'un dictionnaire en utilisant sa clé.

```
a = {"a": 1, "b": 2, "c": 3}
del a["a"]
a # {"b": 2, "c": 3}
```

On peut vérifier si une clé est présente dans un dictionnaire en utilisant l'opération `in`.

```
a = {"a": 1, "b": 2, "c": 3}
"a" in a # True
"d" in a # False
```

On peut obtenir la liste des clés d'un dictionnaire avec la méthode `keys()`.

```
a = {"a": 1, "b": 2, "c": 3}
a.keys() # dict_keys(['a', 'b', 'c'])
```

On peut obtenir la liste des valeurs d'un dictionnaire avec la méthode `values()`.

```
a = {"a": 1, "b": 2, "c": 3}
a.values() # dict_values([1, 2, 3])
```

On peut obtenir la liste des paires clé-valeur d'un dictionnaire avec la méthode `items()`.

```
a = {"a": 1, "b": 2, "c": 3}
a.items() # dict_items([('a', 1), ('b', 2), ('c', 3)])
```

On peut obtenir la taille d'un dictionnaire avec la fonction `len()`.

```
a = {"a": 1, "b": 2, "c": 3}
len(a) # 3
```

On peut créer un dictionnaire vide avec la fonction `dict()`.

```
a = dict()
a # {}
```

tips Les clés d'un dictionnaire doivent être hashable. C'est-à-dire qu'il doit être possible de calculer un hash pour la clé. Les types hashables sont : - les types numériques (int, float, complex, bool) - les chaînes de caractères - les tuples

Les clés d'un dictionnaire doivent être uniques. Si une clé est définie plusieurs fois, seule la dernière valeur sera conservée. (il est intéressant de noter que si on prends une clés avec un hash X et un int égal à X, on obtient pas la même clé)

Il est possible d'obtenis le hash d'un objet avec la fonction `hash()`

```
a = "Hello World"
hash(a) # -5368474246136574527
b = 5613047207200643141
hash(b) # -756788227709186625
```

Sets

Les sets sont des collections non ordonnées d'éléments. Ils sont définis par le mot-clé `set`. Les éléments d'un set doivent être uniques et immutables. Les éléments d'un set peuvent être de n'importe quel type.

```
a = {1, 2, 3}
b = {"a", "b", "c"}
c = {1, "a", True}
```

On peut ajouter un élément à un set avec la méthode `add()`.

```
a = {1, 2, 3}
a.add(4)
a # {1, 2, 3, 4}
```

On peut supprimer un élément d'un set avec la méthode `remove()`.

```
a = {1, 2, 3}
a.remove(1)
a # {2, 3}
```

On peut vérifier si un élément est présent dans un set en utilisant l'opération `in`.

```
a = {1, 2, 3}
1 in a # True
4 in a # False
```

On peut obtenir la taille d'un set avec la fonction `len()`.

```
a = {1, 2, 3}
len(a) # 3
```

On peut créer un set vide avec la fonction `set()`.

```
a = set()
a # set()
```

tips Les éléments d'un set doivent être hashable. C'est-à-dire qu'il doit être possible de calculer un hash pour l'élément. Les types hashables sont : - les types numériques (int, float, complex, bool) - les chaînes de caractères - les tuples

Strings

Les chaînes de caractères sont des séquences de caractères. Elles sont définies par le mot-clé `str`. Les caractères d'une chaîne de caractères doivent être immutables. Les caractères d'une chaîne de caractères peuvent être de n'importe quel type.

```
a = "Hello World"
b = 'Hello World'
c = """Hello World"""
d = '''Hello World'''
```

tips Les chaine de charactères sont itérables :

```
a = "Hello World"
a[0] # H
a[1] # e
a[2] # l
...
a[10] # d


a[-1] # d
a[-2] # l

a[0:5] # Hello

a[0:5:2] # Hlo

a[6:] # World

a[:5] # Hello
```

Regular Expressions Les expressions régulières sont des chaînes de caractères qui permettent de définir des motifs de recherche. Elles sont définies par le module `re`. Les motifs de recherche sont définis par des caractères spéciaux. Les motifs de recherche sont utilisés par les fonctions du module `re` pour rechercher des chaînes de caractères qui correspondent à ces motifs.

```python
import re

re.match("Hello", "Hello World") # <re.Match object; span=(0, 5), match='Hello'>
re.match("World", "Hello World") # None

re.search("Hello", "Hello World") # <re.Match object; span=(0, 5), match='Hello'>
re.search("World", "Hello World") # <re.Match object; span=(6, 11), match='World'>

re.findall("Hello", "Hello World") # ['Hello']
re.findall("World", "Hello World") # ['World']

re.finditer("Hello", "Hello World") # <callable_iterator object at 0x7f9b1c0b5d30>
re.finditer("World", "Hello World") # <callable_iterator object at 0x7f9b1c0b5d30>
```

tips Séparation d'une chaîne de caractères en fonction d'un motif :

```python
a = "Hello World"
list_of_words = a.split(" ") # ["Hello", "World"]
```

Exemple plus complexe :

```python
import re

a = "Hello World, my name is Bruno, I'm 25 years old and in Yverdon.
        here are some important streets in this city :
        - Rue des Moulins
        - Rue du Lac
        - Rue de la Plaine
        - Rue de la Source
        - Rue de la Prairie
    Thanks for your attention (and your time)."

list_of_streets = re.findall("Rue [a-zA-Z ]+", a)
# ['Rue des Moulins', 'Rue du Lac', 'Rue de la Plaine', 'Rue de la Source', '
    Rue de la Prairie']

list_of_words = re.findall("[a-zA-Z]+", a)
# ['Hello', 'World', 'my', 'name', 'is', 'Bruno', 'I', 'm', 'years', 'old', '
    and', 'in', 'Yverdon', 'here', 'are', 'some', 'important', 'streets', 'in',
    'this', 'city', 'Rue', 'des', 'Moulins', 'Rue', 'du', 'Lac', 'Rue', 'de', '
    la', 'Plaine', 'Rue', 'de', 'la', 'Source', 'Rue', 'de', 'la', 'Prairie', '
    Thanks', 'for', 'your', 'attention', 'and', 'your', 'time']
```

Pour plus d'informations sur les expressions régulières, vous pouvez consulter le site :

https://docs.python.org/3/library/re.html.

Chap 4: Data Structures (Advanced)

Namedtuple

Les namedtuples sont des tuples nommés. Ils sont définis par la fonction `namedtuple()` du module `collections`. Les namedtuples sont des tuples immutables. Les éléments d'un namedtuple peuvent être de n'importe quel type.

```python
from collections import namedtuple

Person = namedtuple("Person", ["name", "age", "city"])

p1 = Person("Bruno", 25, "Yverdon")
p2 = Person("John", 30, "Lausanne")
p3 = Person("Jane", 28, "Genève")
```

On peut accéder aux éléments d'un namedtuple en utilisant la notation pointée.

```python
p1.name # Bruno
p1.age # 25
p1.city # Yverdon
```

tips On peut accéder aux éléments d'un namedtuple en utilisant la notation indexée.

```python
from collections import namedtuple

Person = namedtuple("Person", ["name", "age", "city"])

p1 = Person("Bruno", 25, "Yverdon")

p1[0] # Bruno
p1[1] # 25
p1[2] # Yverdon

p1._asdict() # OrderedDict([('name', 'Bruno'), ('age', 25), ('city', 'Yverdon')])

p1._replace(name="John") # Person(name='John', age=25, city='Yverdon')

p1._fields # ('name', 'age', 'city')

p1._tuple # ('Bruno', 25, 'Yverdon')
```

Numpy

Numpy est une librairie qui permet de manipuler des tableaux multidimensionnels. Elle est définie par le module `numpy`. Les tableaux multidimensionnels sont définis par le mot-clé `ndarray`. Les éléments d'un tableau multidimensionnel doivent être de même type. Les tableaux multidimensionnels sont mutables. Les tableaux multidimensionnels sont itérables. Cette bibliothèque est inspirée du langage Matlab.

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])
c = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

Il existe plusieurs fonctions pour créer des tableaux multidimensionnels :

```python
import numpy as np

a = np.zeros((2, 3)) # [[0., 0., 0.], [0., 0., 0.]]
b = np.ones((2, 3)) # [[1., 1., 1.], [1., 1., 1.]]
c = np.full((2, 3), 5) # [[5, 5, 5], [5, 5, 5]]
d = np.eye(3) # [[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]
e = np.random.random((2, 3)) # [[0.37454012, 0.95071431, 0.73199394],
    [0.59865848, 0.15601864, 0.15599452]]
```

Calcules en numpy Les opérations arithmétiques de base sont disponibles en numpy. Ces opérations sont effectuées élément par élément.

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])

a + 1 # [2, 3, 4, 5, 6]
a - 1 # [0, 1, 2, 3, 4]
a * 2 # [2, 4, 6, 8, 10]
```

Broadcasting Le broadcasting est une fonctionnalité de numpy qui permet d'effectuer des opérations entre des tableaux de tailles différentes. Le broadcasting est possible si les dimensions des tableaux sont compatibles. Les dimensions sont compatibles si elles sont égales ou si l'une des dimensions est égale à 1.

```python
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([1, 2, 3])

a + b # [[2, 4, 6], [5, 7, 9]]
```

Indexation L'indexation en numpy est similaire à l'indexation en python. On peut utiliser des indices négatifs. On peut utiliser des slices. On peut utiliser des listes d'indices.

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])

a[0] # 1
a[-1] # 5

b[0] # [1, 2, 3]
b[-1] # [4, 5, 6]
b[0, 0] # 1
b[-1, -1] # 6
```

Slicing Le slicing en numpy est similaire au slicing en python. On peut utiliser des indices négatifs. On peut utiliser des slices. On peut utiliser des listes d'indices.

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])

a[0:2] # [1, 2]

b[0:2] # [[1, 2, 3], [4, 5, 6]]
b[0:2, 0:2] # [[1, 2], [4, 5]]
b[0:2, 0:2] = 0 # [[0, 0, 3], [0, 0, 6]]
```

Indexation avancée L'indexation avancée en numpy permet d'extraire des éléments d'un tableau multidimensionnel en utilisant des listes d'indices. On peut utiliser des listes d'indices pour chaque dimension.

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])

a[[0, 2, 4]] # [1, 3, 5]

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

b[[0, 1], [0, 2]] # [1, 6]
```

Booléens L'indexation booléenne en numpy permet d'extraire des éléments d'un tableau multidimensionnel en utilisant des tableaux de booléens. On peut utiliser des tableaux de booléens pour chaque dimension.

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])

a[[True, False, True, False, True]] # [1, 3, 5]
```

Cette méthode est très utile pour filtrer des données à l'aide de conditions.

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])

a[a > 3] # [4, 5]
```

Tips Numpy est une librarie très complète, pour plus de documentation, allez sur le site officiel de Numpy https://numpy.org/doc/stable/reference

Classes

Les classes sont les fondations de la programmation orientée objet. Une classe est un modèle qui permet de créer des objets. Les objets sont des instances de classe. Les objets ont des attributs et des méthodes. Les attributs sont des variables. Les méthodes sont des fonctions. Les attributs et les méthodes sont accessibles avec le point.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Constructeur Le constructeur est une méthode spéciale qui permet d'initialiser les attributs d'un objet. Le constructeur est appelé lors de la création d'un objet. Le constructeur est une méthode qui a pour nom **init**.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Attributs Les attributs sont des variables qui sont attachées à un objet. Les attributs sont accessibles avec le point, elles sont accessibles en lecture et en écriture car en python, tous les attributs sont publics.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

magic methods and operator overloading Les magic methods sont des méthodes spéciales qui permettent de modifier le comportement d'un objet. Les magic methods sont appelées automatiquement dans des situations spécifiques. Les magic methods sont des méthodes qui ont pour nom `__nom__`.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Person(name={self.name}, age={self.age})"
```

Voici une liste des magic methods les plus utilisées :

- `__init__` : constructeur
- `__str__` : conversion en string
- `__repr__` : conversion en string
- `__add__` : addition
- `__sub__` : soustraction
- `__mul__` : multiplication
- `__truediv__` : division
- `__floordiv__` : division entière
- `__mod__` : modulo
- `__pow__` : puissance
- `__eq__` : égalité
- `__ne__` : inégalité
- `__lt__` : inférieur
- `__le__` : inférieur ou égal
- `__gt__` : supérieur
- `__ge__` : supérieur ou égal
- `__len__` : longueur
- `__getitem__` : indexation
- `__setitem__` : indexation
- `__delitem__` : indexation
- `__contains__` : appartenance

Object-Oriented Programming La programmation orientée objet est un paradigme de programmation qui permet de créer des objets. Les objets sont des instances de classe. Les objets ont des attributs et des méthodes. Les attributs sont des variables. Les méthodes sont des fonctions. Les attributs et les méthodes sont accessibles avec le point.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.valentine = None

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    def __add__(self, other):
        # définir l'addition de deux objets (ici une mise en couple)
        if self.age < 18 and other.age < 18:
            return Exception("You are too young to be in a relationship.")
        else if self.age < 18:
            return Exception(f"{self.name} is too young, {other.name} is {other.age} years old and should go to prison.")
        else if other.age < 18:
            return Exception(f"{other.name} is too young, {self.name} is {self.age} years old and should go to prison.")
        else:
            self.valentine = other
            other.valentine = self
            return f"{self.name} and {other.name} are now in a relationship."
```

Exemple d'utilisation :

```python
from person import Person

p1 = Person("John", 20)
p2 = Person("Jane", 18)

p1.say_hello() # Hello, my name is John and I am 20 years old.

p1 + p2 # John and Jane are now in a relationship.
```

inheritance L'héritage est un mécanisme qui permet de créer une classe à partir d'une autre classe. La classe qui est héritée est appelée classe parente. La classe qui hérite est appelée classe enfant. La classe enfant hérite des attributs et des méthodes de la classe parente. La classe enfant peut modifier les attributs et les méthodes de la classe parente. La classe enfant peut ajouter des attributs et des méthodes.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.valentine = None

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    def __add__(self, other):
        # définir l'addition de deux objets (ici une mise en couple)
        if self.age < 18 and other.age < 18:
            return Exception("You are too young to be in a relationship.")
        else if self.age < 18:
            return Exception(f"{self.name} is too young, {other.name} is {other.age} years old and should go to prison.")
        else if other.age < 18:
            return Exception(f"{other.name} is too young, {self.name} is {self.age} years old and should go to prison.")
        else:
            self.valentine = other
            other.valentine = self
            return f"{self.name} and {other.name} are now in a relationship."

class Student(Person):
    def __init__(self, name, age, school):
        super().__init__(name, age)
        self.school = school

    def say_hello(self):
        super().say_hello()

    def __add__(self, other):
        # définir l'addition de deux objets (ici une mise en couple)
        # on peut utiliser la méthode de la classe parente
        return super().__add__(other)
```

Attention, il y a plsieurs bonnes pratiques à respecter : - la classe enfant doit avoir le même comportement que la classe parente - la classe enfant doit avoir le même type que la classe parente - il faut éviter d'utiliser l'héritage multiple (une classe enfant ne doit hériter que d'une seule classe parente)

polymorphism Le polymorphisme est un mécanisme qui permet de modifier le comportement d'une méthode héritée. Le polymorphisme est utilisé pour créer des méthodes qui ont le même nom mais qui ont un comportement différent. Le polymorphisme est utilisé pour créer des méthodes qui ont le même nom mais qui ont des paramètres différents.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.valentine = None

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    def __add__(self, other):
        # définir l'addition de deux objets (ici une mise en couple)
        if self.age < 18 and other.age < 18:
            return Exception("You are too young to be in a relationship.")
        else if self.age < 18:
            return Exception(f"{self.name} is too young, {other.name} is {other.age} years old and should go to prison.")
        else if other.age < 18:
            return Exception(f"{other.name} is too young, {self.name} is {self.age} years old and should go to prison.")
        else:
            self.valentine = other
            other.valentine = self
            return f"{self.name} and {other.name} are now in a relationship."

class Student(Person):
    def __init__(self, name, age, school):
        super().__init__(name, age)
        self.school = school

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old. I am a student at {self.school}.")

    def __add__(self, other):
        # définir l'addition de deux objets (ici une mise en couple)
        # si les deux objets sont des étudiants, il peuvent se mettre en couple
        if isinstance(other, Student):
            self.valentine = other
            other.valentine = self
            return f"{self.name} and {other.name} are now in a relationship."
        # sinon, il faut 3 ans d'écart
        else if self.age - other.age > 3 or other.age - self.age > 3:
            self.valentine = other
            other.valentine = self
            return f"{self.name} and {other.name} are now in a relationship."
        else:
            return Exception("You are too young to be in this type of relationship.")
```

encapsulation L'encapsulation est un mécanisme qui permet de cacher les attributs et les méthodes d'une classe. L'encapsulation est utilisée pour créer des attributs et des méthodes privés. L'encapsulation est utilisée pour créer des attributs et des méthodes protégés. L'encapsulation est utilisée pour créer des attributs et des méthodes publics.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.__health = 100

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    def get_health(self):
        return self.__health

    def set_health(self, health):
        self.__health = health
```

Une autre bonne pratique est d'utiliser le décorateur `@property` pour créer des attributs et des méthodes privés. Une autre bonne pratique est d'utiliser le décorateur `@property` pour créer des attributs et des méthodes protégés. Une autre bonne pratique est d'utiliser le décorateur `@property` pour créer des attributs et des méthodes publics.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.__health = 100

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    @property
    def health(self):
        return self.__health

    @health.setter
    def health(self, health):
        self.__health = health
```

Chap 5: Functional Programming

Lambda Functions

Les fonctions lambda sont des définitions de fonctions qui peuvent être utilisées comme des fonctions normales. Les fonctions lambda sont utilisées pour créer des fonctions anonymes. Les fonctions lambda sont utilisées pour créer des fonctions qui prennent un nombre variable d'arguments. Les fonctions lambda sont utilisées pour créer des fonctions qui prennent un nombre variable d'arguments et qui retournent une valeur.

Exemple d'utilisation d'une fonction lambda :

```python
# définir une fonction lambda qui prend un argument et qui retourne le carré de cet argument
square = lambda x: x**2
# utiliser la fonction lambda
print(square(2))
```

Exemple d'utilisation d'une fonction lambda avec un nombre variable d'arguments :

```python
# définir une fonction lambda qui prend un nombre variable d'arguments et qui retourne la somme de ces arguments
sum = lambda *args: sum(args)
# utiliser la fonction lambda
print(sum(1, 2, 3, 4, 5))
```

Map Function

La fonction map est utilisée pour appliquer une fonction à une liste. La fonction map est utilisée pour appliquer une fonction à une liste et retourner une nouvelle liste. La fonction map est utilisée pour appliquer une fonction à une liste et retourner une nouvelle liste avec les résultats de la fonction appliquée à chaque élément de la liste.

Exemple d'utilisation de la fonction map :

```python
# définir une fonction qui prend un argument et qui retourne le carré de cet
    argument
def square(x):
    return x**2

# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la fonction map
squared_numbers = map(square, numbers)
print(squared_numbers) # <map object at 0x7f8a4c1b6a90>
print(list(squared_numbers)) # [1, 4, 9, 16, 25]
```

Elle peut aussi être utilisée avec une fonction lambda :

```python
# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la fonction map avec une fonction lambda
print(list(map(lambda x: x**2, numbers))) # [1, 4, 9, 16, 25]
```

Filter Function

La fonction filter est utilisée pour filtrer une liste. La fonction filter est utilisée pour filtrer une liste et retourner une nouvelle liste. La fonction filter est utilisée pour filtrer une liste et retourner une nouvelle liste avec les éléments de la liste qui vérifient une condition.

Exemple d'utilisation de la fonction filter :

```python
# définir une fonction qui prend un argument et qui retourne True si cet
    argument est pair, False sinon
def is_even(x):
    return x % 2 == 0

# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la fonction filter
even_numbers = filter(is_even, numbers)
print(even_numbers) # <filter object at 0x7f8a4c1b6a90>
print(list(even_numbers)) # [2, 4]
```

Elle peut aussi être utilisée avec une fonction lambda :

```python
# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la fonction filter avec une fonction lambda
print(list(filter(lambda x: x % 2 == 0, numbers))) # [2, 4]
```

Zip Function

La fonction zip est utilisée pour combiner deux listes. La fonction zip est utilisée pour combiner deux listes et retourner une nouvelle liste. La fonction zip est utilisée pour combiner deux listes et retourner une nouvelle liste avec les éléments des deux listes combinés.

Exemple d'utilisation de la fonction zip :

```python
# définir deux listes
numbers = [1, 2, 3, 4, 5]
letters = ["a", "b", "c", "d", "e"]

# utiliser la fonction zip
zipped = zip(numbers, letters)
print(zipped) # <zip object at 0x7f8a4c1b6a90>
print(list(zipped)) # [(1, "a"), (2, "b"), (3, "c"), (4, "d"), (5, "e")]
```

Attention, si les listes changent entre l'appel de la fonction zip et la conversion en liste, le résultat sera différent :

```python
# définir deux listes
numbers = [1, 2, 3, 4, 5]
letters = ["a", "b", "c", "d", "e"]

# utiliser la fonction zip
zipped = zip(numbers, letters)

# modifier la liste numbers
numbers.append(6)
letters.append("f")

# convertir le résultat en liste
print(list(zipped)) # [(1, "a"), (2, "b"), (3, "c"), (4, "d"), (5, "e"), (6, "f
    ")]
```

List Comprehension

La compréhension de liste est utilisée pour créer une liste à partir d'une autre liste. La compréhension de liste est utilisée pour créer une liste à partir d'une autre liste en appliquant une fonction à chaque élément de la liste. La compréhension de liste est utilisée pour créer une liste à partir d'une autre liste en appliquant une fonction à chaque élément de la liste et en filtrant les éléments de la liste.

Exemple d'utilisation de la compréhension de liste :

```python
# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la compréhension de liste
squared_numbers = [x**2 for x in numbers]
print(squared_numbers) # [1, 4, 9, 16, 25]
```

Elle peut aussi être utilisée avec des conditions :

```python
# définir une liste
numbers = [1, 2, 3, 4, 5]

# utiliser la compréhension de liste avec une condition
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers) # [2, 4]
```

Note : *La compréhension peut utiliser n'importe quelle fonction (dont celle vues en dessus)*

Exception Handling (Try-Except)

La gestion des exceptions est utilisée pour gérer les erreurs. La gestion des exceptions est utilisée pour gérer les erreurs et éviter que le programme ne s'arrête. La gestion des exceptions est utilisée pour gérer les erreurs et éviter que le programme ne s'arrête en cas d'erreur.

Exemple d'utilisation de la gestion des exceptions :

```
# définir une fonction qui prend un argument et qui retourne le carré de cet
    argument
def square(x):
    return x**2

# utiliser la fonction avec un argument invalide
print(square("a")) # TypeError: unsupported operand type(s) for ** or pow(): '
    str' and 'int'
```

On peut utiliser la gestion des exceptions pour gérer cette erreur :

```
# définir une fonction qui prend un argument et qui retourne le carré de cet
    argument
def square(x):
    try:
        return x**2
    except TypeError:
        return "Invalid argument"

# utiliser la fonction avec un argument invalide
print(square("a")) # Invalid argument
```

Decorators

Les décorateurs sont utilisés pour modifier le comportement d'une fonction. Les décorateurs sont utilisés pour modifier le comportement d'une fonction sans la modifier. Les décorateurs sont utilisés pour modifier le comportement d'une fonction sans la modifier en utilisant une fonction qui prend une fonction en argument et qui retourne une fonction.

Exemple d'utilisation des décorateurs :

```python
# définir une fonction qui prend un argument et qui retourne le carré de cet argument
def square(x):
    return x**2

# utiliser la fonction
print(square(2)) # 4
```

On peut utiliser un décorateur pour modifier le comportement de cette fonction :

```python
# définir un décorateur qui prend une fonction en argument et qui retourne une fonction
def decorator(func):
    def wrapper(x):
        return func(x) + 1
    return wrapper

# utiliser le décorateur
@decorator
def square(x):
    return x**2

# utiliser la fonction
print(square(2)) # 5
```

Chap 6: Modules

Third-Party Libraries

Les bibliothèques tierces sont utilisées pour ajouter des fonctionnalités à Python. Les bibliothèques tierces sont utilisées pour ajouter des fonctionnalités à Python en utilisant des modules. Les bibliothèques tierces sont utilisées pour ajouter des fonctionnalités à Python en utilisant des modules qui peuvent être installés avec pip.

Exemple d'utilisation d'une bibliothèque tierce :

```python
# importer le module math
import math

# utiliser le module math
print(math.sqrt(4)) # 2.0
```

Installer une bibliothèque tierce :

```
pip install <library>
```

Virtual Environments

Il exist des environnements virtuels pour isoler les bibliothèques tierces. Il exist des environnements virtuels pour isoler les bibliothèques tierces en utilisant le module venv. Il exist des environnements virtuels pour isoler les bibliothèques tierces en utilisant le module venv et en créant un environnement virtuel.

Créer un environnement virtuel :

```
python3 -m venv <name>
```

Activer un environnement virtuel :

```
source <name>/bin/activate
```

Désactiver un environnement virtuel :

```
deactivate
```

Attention, il faut installer vent pour pouvoir utiliser cette commande :

```
pip install venv
```

Une bonne pratique est de créer un fichier requirements.txt pour lister les bibliothèques tierces utilisées :

```
pip install -r requirements.txt
```

Exemple de fichier requirements.txt :

```
numpy==1.19.4
pandas==1.1.4
jupyter==1.0.0
matplotlib==3.3.3
```

Click (Command-line Interface)

Click est utilisé pour créer une interface en ligne de commande en utilisant le module click et en créant une fonction qui prend des arguments et qui retourne une valeur.

Exemple d'utilisation de Click :

```python
# importer le module click
import click

# définir une fonction qui prend des arguments et qui retourne une valeur
@click.command()
@click.option("--name", prompt="Your name", help="The person to greet.")
def hello(name):
    return f"Hello, {name}!"

# utiliser la fonction
print(hello()) # Hello, World!
```

Utiliser la fonction en ligne de commande :

```
python3 hello.py --name World
```

Chap 7: Jupyter Notebook

Jupyter Notebook

Jupyter Notebook est utilisé pour créer des notebooks en utilisant le module jupyter et en exécutant la commande jupyter notebook. Ces notebooks peuvent être utilisés pour créer des documents interactifs qui altèrne entre du code et du texte.

Exemple d'utilisation de Jupyter Notebook :

```
1 jupyter notebook
```

Jupyter Lab

Jupyter Lab est utilisé pour créer des notebooks en utilisant le module jupyterlab et en exécutant la commande jupyter lab. Ces notebooks peuvent être utilisés pour créer des documents interactifs qui altèrne entre du code et du texte.

Exemple d'utilisation de Jupyter Lab :

```
1 jupyter lab
```

Intégrantion de Jupyter Notebook dans VSCode

Il est possible d'intégrer Jupyter Notebook dans VSCode en installant l'extension Jupyter. Ces fichiers ont comme extension .ipynb et peuvent être utilisés pour créer des documents interactifs qui altèrne entre du code et du texte. L'intégration des notebooks dans VSCode est bien implémentées et utilisée avec un environnement virtuel, les bibliothèques tierces sont bien prises en compte. Je pense que c'est la meilleure solution pour utiliser des notebooks.

Chap 8: Web Development

Il est possible de faire du développement web avec Python en utilisant des bibliothèques tierces comme Flask. Il est possible de faire du développement web avec Python en utilisant des bibliothèques tierces comme Flask et en créant une application web.

Jinja2 (Template Engine) and Flask (Web Framework)

Jinja2 est utilisé pour créer des templates en utilisant le module jinja2 et en créant un template. Ces templates peuvent être utilisés pour créer des documents HTML dynamiques.

Flask est utilisé pour créer une application web en utilisant le module flask et en créant une application web. Ces applications web peuvent être utilisées pour créer des sites web.

Exemple d'utilisation de Jinja2 :

```python
# importer le module jinja2
from jinja2 import Template

# définir un template
template = Template("Hello, {{ name }}!")

# utiliser le template
print(template.render(name="World")) # Hello, World!
```

Les templates sont généralement stockés dans un dossier templates :

```
templates/
    index.html
```

Exemple de template index.html :

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Flask</title>
    </head>
    <body>
        <h1>Hello, {{ name }}!</h1>
    </body>
</html>
```

Utiliser le template index.html :

```python
# importer le module flask
from flask import Flask, render_template

# définir une application flask
app = Flask(__name__)

# définir une route
@app.route("/")
def index():
    return render_template("index.html", name="World")

# exécuter l'application flask
if __name__ == "__main__":
    app.run(debug=True)
```

Une bonne pratique est de créer un template base.html qui contient le code HTML commun à toutes les pages :

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Flask</title>
    </head>
    <body>
        {% block content %}{% endblock %}
    </body>
</html>
```

Le template index.html peut alors hériter du template base.html :

```jinja
{% extends "base.html" %}

{% block content %}
    <h1>Hello, {{ name }}!</h1>
{% endblock %}
```

Chap 9: Data Analysis

Pandas (Data Analysis Library)

Pandas est utilisé pour analyser des données en utilisant le module pandas et en créant un DataFrame. Ces DataFrame peuvent être utilisés pour analyser des données. Cette bilblithèque est très utilisée pour analyser des données de grande taille.

Pandas comprend beaucoup de fonctionnalités pour analyser des données :

Importer des données Il est possible d'importer des données depuis un fichier CSV, un fichier Excel, une base de données SQL, une URL, etc.

Exemple d'importation de données depuis un fichier CSV :

```
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier CSV
df = pd.read_csv("data.csv")
```

Exemple d'importation de données depuis un fichier Excel :

```
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier Excel
df = pd.read_excel("data.xlsx")
```

DataFrames Les DataFrames sont utilisés pour analyser des données en utilisant le module pandas.

Il est possible de créer des DataFrames en utilisant des listes, des dictionnaires, des tuples, des séries, etc.

Exemple de création d'un DataFrame depuis une liste :

```python
# importer le module pandas
import pandas as pd

# créer un DataFrame depuis une liste
df = pd.DataFrame([1, 2, 3, 4, 5])
```

Exemple de création d'un DataFrame depuis un dictionnaire :

```python
# importer le module pandas
import pandas as pd

# créer un DataFrame depuis un dictionnaire
df = pd.DataFrame({"a": [1, 2, 3, 4, 5], "b": [1, 2, 3, 4, 5]})
```

Séries Les Séries sont utilisées pour analyser des données en utilisant le module pandas.

Il est possible de créer des Séries en utilisant des listes, des dictionnaires, des tuples, des DataFrames, etc.

Exemple de création d'une Série depuis une liste :

```python
# importer le module pandas
import pandas as pd

# créer une Série depuis une liste
s = pd.Series([1, 2, 3, 4, 5])
```

La différence entre un DataFrame et une Série est que les DataFrames sont des tableaux à deux dimensions et les Séries sont des tableaux à une dimension.

Indexation Il est possible d'indexer des données en utilisant le module pandas. L'indexation permet de sélectionner des données dans un DataFrame ou une Série.

Exemple d'indexation:

```python
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier CSV
df = pd.read_csv("data.csv")

# indexer un DataFrame
df["a"] # ici on sélectionne la colonne dont le nom est "a"
```

Visualisation de données Il est possible de visualiser des données en utilisant le module pandas. La visualisation permet de créer des graphiques à partir d'un DataFrame ou d'une Série.

Exemple de visualisation:

```python
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier CSV
df = pd.read_csv("data.csv")

# visualiser un DataFrame
df.plot() # ici on crée un graphique à partir du DataFrame
df.hist() # ici on crée un histogramme à partir du DataFrame
df.boxplot() # ici on crée un boxplot à partir du DataFrame
df.describe() # ici on affiche des statistiques à partir du DataFrame
df.info() # ici on affiche des informations à partir du DataFrame
df.sample() # ici on affiche un échantillon du DataFrame
df.head() # ici on affiche les 5 premières lignes du DataFrame
df.tail() # ici on affiche les 5 dernières lignes du DataFrame
df.shape # ici on affiche la taille du DataFrame et on la retourne
df.size # ici on affiche le nombre de valeurs du DataFrame et on le retourne
```

Manipulation de données Il est possible de manipuler des données en utilisant le module pandas. La manipulation permet de modifier un DataFrame ou une Série.

Exemple de manipulation:

```python
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier CSV
df = pd.read_csv("data.csv")

# manipuler un DataFrame
df["a"] = df["a"] + 1 # ici on ajoute 1 à la colonne "a"
df.drop("a", axis=1) # ici on supprime la colonne "a"
col_a = df.pop("a") # ici on supprime la colonne "a" et on la retourne
df.rename(columns={"a": "b"}) # ici on renomme la colonne "a" en "b"
df.sort_values(by="a") # ici on trie le DataFrame par la colonne "a"
df.sort_index() # ici on trie le DataFrame par l'index
df2 = df.copy() # ici on copie le DataFrame
df.drop_duplicates() # ici on supprime les doublons
df.dropna() # ici on supprime les valeurs manquantes
df.fillna(0) # ici on remplace les valeurs manquantes par 0
df.replace(1, 2) # ici on remplace les valeurs 1 par 2
df.sample() # ici on sélectionne une ligne aléatoire
df.sample(n=2) # ici on sélectionne 2 lignes aléatoires
df.sample(frac=0.5) # ici on sélectionne 50% des lignes aléatoirement
df.columns # ici on affiche les noms des colonnes et on les retourne
df.index # ici on affiche les index et on les retourne
df.values # ici on affiche les valeurs et on les retourne
df.T # ici on transpose le DataFrame
```

Statistiques Il est possible de calculer des statistiques en utilisant le module pandas. Les statistiques permettent de calculer des statistiques sur un DataFrame ou une Série.

Exemple de statistiques:

```python
# importer le module pandas
import pandas as pd

# importer des données depuis un fichier CSV
df = pd.read_csv("data.csv")

# calculer des statistiques sur un DataFrame
df.mean() # ici on calcule la moyenne du DataFrame
df.median() # ici on calcule la médiane du DataFrame
df.mode() # ici on calcule le mode du DataFrame
df.min() # ici on calcule le minimum du DataFrame
df.max() # ici on calcule le maximum du DataFrame
df.std() # ici on calcule l'écart-type du DataFrame
df.var() # ici on calcule la variance du DataFrame
df.sum() # ici on calcule la somme du DataFrame
df.cumsum() # ici on calcule la somme cumulée du DataFrame
df.prod() # ici on calcule le produit du DataFrame
df.cumprod() # ici on calcule le produit cumulé du DataFrame
df.quantile() # ici on calcule le quantile du DataFrame
df.describe() # ici on calcule des statistiques sur le DataFrame
df.info() # ici on calcule des informations sur le DataFrame
```

Lien vers la documentation officielle Pour plus d'informations, vous pouvez consulter la documentation officielle de pandas :

- https://pandas.pydata.org/docs/ documentation officielle de pandas
- https://pandas.pydata.org/docs/getting_started/10min.html tutoriel officiel de pandas

Matplotlib (Data Visualization)

Matplotlib est une bibliothèque de visualisation de données en Python. Elle permet de créer des graphiques à partir de données.

Installation Pour installer Matplotlib, il faut exécuter la commande suivante :

```
pip install matplotlib
```

Importation Pour importer Matplotlib, il faut exécuter la commande suivante :

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

Graphiques Il est possible de créer des graphiques en utilisant Matplotlib. Les graphiques permettent de visualiser des données.

Pour les exemples suivant, nous allons utiliser les données suivantes :

```
# importation des librairies
import matplotlib.pyplot as plt
import numpy as np

# créer des données
time = [i for i in range(1000)]
temperature = [np.random.randint(0, 100) for i in range(1000)]
```

Exemple de graphique scatter:

```
# créer un graphique scatter
plt.scatter(time, temperature)
plt.show()
```

Exemple de graphique line:

```
# créer un graphique line
plt.plot(time, temperature)
plt.show()
```

Exemple de graphique stem:

```
# créer un graphique stem
plt.stem(time, temperature)
plt.show()
```

Exemple de graphique bar:

```python
# créer un graphique bar
plt.bar(time, temperature)
plt.show()
```

Exemple d'histogramme:

```python
# créer un histogramme
plt.hist(temperature)
plt.show()
```

Il existe beaucoup de paramètres pour personnaliser les graphiques. Vous pouvez consulter la documentation officielle pour plus d'informations.

Lien vers la documentation officielle Pour plus d'informations, vous pouvez consulter la documentation officielle de Matplotlib :

- https://matplotlib.org/ documentation officielle de Matplotlib

Notes

1) Pour avoir un résumé parfaitement complait, il manque les modules suivants :

- Turtle
- Machine Learning

J'ai décidé de ne pas les mettre car soit hors propaux dans une utilisation professionnelle, soit faisant partie d'un autre cours.

2) Ce pdf à été généré à partir d'un markdown via pandoc. Le code source est disponible ici :

https://github.com/Sutterlet-Bruno/resume-python-cours.git