Welcome to

Python Intermédiaire

Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Transition Python 2 à Python 3

Avertissement

Python 2 n'étant plus supporté, il est dorénavant indispensable d'utiliser exclusivement Python 3.

Si votre code est encore sous Python 2.x, il existe de nombreux outils permettant de **faciliter** la transition vers 3.x (mais pas de la repousser *ad eternam*):

- L'interpréteur Python 2.7 dispose d'une option `-3` mettant en évidence dans un code les parties qui devront être modifiées pour un passage à Python 3.
- Le script 2to3 permet d'automatiser la conversion du code 2.x en 3.x.

La bibliothèque standard `__future__` permet d'introduire des constructions 3.x dans un code 2.x, p.ex.:

```
from __future__ import print_function  # Fonction print()
from __future__ import division        # Division non-euclidienne

print(1/2)                             # Affichera '0.5'
```

- La bibliothèque *non* standard six fournit une couche de compatibilité 2.x-3.x, permettant de produire de façon transparente un code compatible simultanément avec les deux versions.

Python 3.x

Python 3 apporte quelques changements fondamentaux, notamment:

- `print()` n'est plus un mot-clé mais une fonction: `print(…)`;
- l'opérateur `/` ne réalise plus la division euclidienne entre les entiers, mais toujours la division *réelle*;
- la plupart des fonctions qui retournaient des itérables en Python 2 (p.ex. `range()`) retournent maintenant des itérateurs, plus légers en mémoire;
- un support complet (mais encore complexe) des chaînes Unicode;
- un nouveau système de formatage des chaînes de caractères (`f-string` du **PEP 498** à partir de Python 3.6);
- la fonction de comparaison `cmp` (et la méthode spéciale associée `__cmp__`) n'existe plus [4].

⟹ Pour plus de ressources sur Python 2 vs Python 3

Les séquences: chaînes, listes et tuples

- Une séquence est une structure de données linéaire constituée d'une suite d'objets (éléments).

- Les séquences partagent tous le même modèle d'accès (via des index): un élément est accessible par son index. Un index désigne la position de l'élément dans la séquence.

Les séquences: chaînes, listes et tuples

- Les chaînes de caractères: suite d'objets de type homogène (caractères), délimités par de simples ou doubles côtes.
- Les listes: suite d'objets de types hétérogènes, délimités par des crochets.
- Les tuples: suite d'objets de types hétérogènes, délimités par des parenthèses.

Les listes sont des objets mutables (modifiables), tandis que les chaînes de caractères et les tuples sont des objets non mutables (non modifiables).

Les listes

- Une liste en PYTHON est une structure de données mutable, c'est-à-dire qu'on peut modi er son contenu sans changer son adresse mémoire.

- Modifier une liste ⇒ ajouter/remplacer/supprimer un élément ou une suite d'éléments.

Ajout

- Les méthodes append et insert permettent d'ajouter un élément dans une liste.

- Ces méthodes modi ent la liste sur laquelle s'applique la méthode.
  - L.append(obj) ajoute obj à la n de L.
  - L.insert(i , obj) ajoute obj à la position i dans L.

```
In [1]: L=[1,2,3]
In [2]: L.append('cc')
In [3]: L
Out [3]:[1,2,3,'cc']
```

```
In [4]: L.insert(1,('bb',5))
In [5]: L
Out [5]:[1,('bb',5),2,3,'cc']
```

Extension d'une liste

- La méthode extend permet d'étendre le contenu d'une liste.
- L.extend (seq) ajoute le contenu de seq (itérable) à la n de L

```
In [1]: L=[1,2,3]

In [2]: L.extend('Bonjour')

In [3]: L

Out [3]:
[1,2,3,'B','o','n','j','o','u','r']
In [4]:
L.extend(('coucou',4))
```

```
In  [5]: L

Out [5]:
[1,2,3,'B','o','n','j','o','u','r','coucou',4]
In[6]: L.extend(67)

Out[6]:TypeError: 'int' object is not iterable
```

Suppression

- La méthode L.remove(obj) supprime la première occurrence de obj;
- La méthode L.pop() supprime le dernier élément dans L et le retourne;
- La méthode L.pop(i) supprime l'élément à la position i et le retourne;

```
In [1]: L=[1,2,3,4,5,6]
In [2]: L.remove(1)
In [3]: L
Out[3]:[2,3,4,5,6]
In [4]: L.pop()
Out [4]:6
```

```
In [5]: L
Out [5]:[2,3,4,5]
In [6]: L.pop(1)
Out[6]:3
In [7]: L
Out [7]:[2,4,5]
```

Suppression

Suppression d'un élément

```
In [1]: L=[1,2,3,4,5,6]

In [2]: del L[4]# equivalent à L.pop(4)
In [3]: L
Out [3]:[1,2,3,4,6]
```

Suppression de plusieurs éléments

```
In [4]: del L[2:4]

In [5]: L
Out [5]:[1,2,6]
```

Modification

Modification d'un élément

In [1]: L=[1,2,3,4,'a','b','c','d']

In [2]: L[4]='A'

In [3]: L

Out [3]:[1,2,3,4,'A','b','c','d']

Modification de plusieurs éléments

In [4]: L[2:4]=30,40

In [5]: L

Out [5]:[1,2,30,40,'A','b','c','d']

Autres méthodes

- L.sort() trie les éléments de L
- L.reverse() inverse les éléments de L

```
In [1]: L=[1,8,3,4,9,5,2]
In [2]: L.sort()
In [3]: L
Out [3]:[1,2,3,4,5,8,9]
```

```
In [4]: L.reverse()
In [5]: L
Out [5]:[9,8,5,4,3,2,1]
```

Création de liste - par énumération

Par énumération en utilisant la méthode list(iterable)

```
In [1]:  L=list(range(0,11))

In [2]: L
Out[2] [0,1,2,3,4,5,6,7,8,9,10]
In [3]: L=list(range(0,11,2)

In[4]: L
Out [4]:[0,2,4,6,8,10]
```

```
In [5]: L=list('Bonjour')

In [5]: L
Out [5]:['B','o','n','j','o','u','r']
```

Création de liste - par compréhension

La création de liste par compréhension permet de parcourir un objet itérable en renvoyant une liste.

```
In  [1]:  L=[i*2 for  i in range(0,11)]

In [2]: L
Out[2]
[0,2,4,6,8,10,12,14,16,18,20]
In [3]: L=[i for i in 'Bonjour']

In[4]: L
Out [4]:['B','o','n','j','o','u','r']
```

```
In [5]: L=[i+i for i in 'Bonjour']

In [6]: L
Out [6]:['BB','oo','nn','jj','oo','uu','rr']
In [6]: L=[(i,j) for i in range(3) for j in
range(2)]
In  [7]: L
Out [7]:
[(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]
```

Création de liste compréhension

La création de liste par compréhension permet de renvoyer une liste dont le contenu peut-être ltré:

```
In [1]: L=[i for i in range(0,11) if i%2==0]  In

[2]: L

Out[2][0,2,4,6,8,10]
In [3]: L=[i for i in 'Bonjour' if i=='B' or i=='j']  In[4]:

L
```

Out [4]:['B','j']

Parcours de liste

```
L=[1,'a','coucou',4]
foriinrange(len(L)):
    print(L[i])
1
'a'  'coucou'  4
```

```
L=[1,'a','coucou',4]  for i in
L:
    print(i)
1
'a'  'coucou'  4
```

Les chaînes - Rappel

- Les chaînes de caractères: suite d'objets de types homogènes (caractères), délimités par de simples ou doubles côtes.

- Les chaînes sont des objets non mutables (non modi ables).

- Les chaînes sont des séquences. Par conséquent, toutes les

opérations relatives aux séquences sont applicables aux chaînes:

- L'accès à un élément/le découpage;
- La fonction len;
- L'appartenance d'un élément (in, not in);
- Les opérations de concaténation/répétition;
- Les méthodes count/index;
    - PYTHON o re plusieurs méthodes de manipulation de chaînes de caractères.

Les chaînes de caractères

c: une chaîne de caractères.

c.split()	Retourne une liste composée des mots composant la chaîne c. Par défaut, le séparateur de mots est le caractère espace.
c.split(':')	Même définition, sauf que cette fois, le séparateur de mots est le caractère ':'.
c. find(cc)	Détermine si cc est une sous-chaîne de c. Retourne l'indice si cc est trouvée, -1 sinon.
c. find(cc,deb, n)	Même dé nition, sauf que cette fois, la recherche commence à partir de la position deb et nit à la position n − 1.
c.replace(str 1,str 2)	remplace toutes les occurrences de str 1 dans c par str 2.
c.replace(str 1,str 2,n)	remplace les n premières occurrences de str 1 dans c par str 2.

Les chaînes de caractères

Méthode	Description
c.upper()	Convertit toutes les minuscules de c en majuscules.
c.lower()	Convertit toutes les majuscules de c en minuscules.
c.title()	Retourne une version de c où chaque initiale de chaque mot est une majuscule, comme dans un titre en anglais.
c.isupper()	Retourne True si c contient au moins un caractère alphabétique et que tous les caractères soient en majuscules, False, sinon.
c.islower()	Retourne True si c contient au moins un caractère alphabétique et que tous les caractères soient en minuscules, False, sinon.
c.istitle()	Retourne True si toutes les initiales des mots de c sont en majuscules, False, sinon.

Les tuples - Rappel

- Les tuples: suite d'objets de types hétérogènes, délimités par des parenthèses.

- Les tuples sont des objets non mutables (non modi ables).
- Les tuples sont des séquences. Par conséquent, toutes les opérations relatives aux séquences sont applicables aux tuples:
  - L'accès à un élément/le découpage;
  - La fonction len;
  - L'appartenance d'un élément (in, not in);
  - Les opérations de concaténation/répétition;
  - Les méthodes count/index;

Les tuples

```
In [1]: T=tuple();T# ou bien T=() Tuple vide  Out
[1]:()

In [2]:type(T)
Out [2]:<class 'tuple'>

In [3]: T=('Bonjour', 14, ['coucou', 15], (67, 'salut'));T# Tuple
composé de 4 éléments
Out [3]:('Bonjour', 14, ['coucou', 15], (67, 'salut'))
```

Les tuples

```
In [4]: T=('Bonjour');type(T)# Attention, ceci n'est pas tuple
Out [4]:<class 'str'>

In [5]: T=('Bonjour',);type(T)# Tuple composé d'un seul élément, remarquez la virgule à la n
Out [5]:<class 'tuple'>
```

Les tuples

- On peut également créer un tuple par énumération, en utilisant la méthode tuple avec en paramètre un objet itérable : T=tuple(iterable).
- La création de tuple par compréhension est également possible.

```
In [6]: T=tuple('Bonjour');T
Out [6]:('B', 'o', 'n', 'j', 'o', 'u', 'r')

In [7]: T=tuple(range(11));T
Out [7]:(0,1,2,3,4,5,6,7,8,9,10)

In [8]: T=tuple([1,6, ['Bonjour', 65]]);T
Out [8]:(1,6,['Bonjour',65])

In [9]: T=tuple((i,i) for i inrange(11) if i%2==0);T  Out [9]:((0,0), (2,2),
(4,4), (6,6), (8,8), (10,10))
```

Copie d'un objet mutable

- Une affection x = y n'implique pas la création d'un nouvel objet, mais plutôt la création d'une nouvelle étiquette y (référence) vers l'objet déjà créé.

- Ceci ne pose pas de problème pour les objets non mutables. Par contre, ceci peut poser un problème pour les objets mutables, tels que les listes, où la modi cation d'une liste implique la modi cation de l'autre liste.

```
In [1]: L1=[iforiinrange(0,6)]

In [2]: L1
Out[2][0,1,2,3,4,5]
In [3]: L2=L1
```

```
In[4]: L1[0]='A'

In [5]: L1
Out[5]:['A',1,2,3,4,5]
In [6]: L2
Out [6]:['A',1,2,3,4,5]
```

Copie d'un objet mutable

- Solution: Créer une copie de l'objet mutable indépendante de l'originale.
- Différentes méthodes :

```
In [7]: L3=L1[:]
In [8]: L4=[iforiinL1]
In [9]: L5=list()  In[10]: for i in L1:
                       L5.append(i)
In [11]:importcopyasc  In[12]:
L6=c.copy(L1)
```

⇒ vérifier les id des objets crées.

Copie superficielle/profonde

- Si une liste contient des objets eux-mêmes mutables, la copie que nous venons d'effectuer est insuffisante ⇒ copie superficielle

```
In [1]: L1=[[1,2,3],[4,5,6]]
In [2]: L2=L1[:]
In [3]: L1.append(56)
In[4]: L1,L2
Out [4]:[[1,2,3],[4,5,6],56],[[1,2,3],[4,5,6]]
In[5]: L1[0].append(66)
In[6]: L1,L2
Out[6]:[[1,2,3,66],[4,5,6],56],[[1,2,3,66],[4,5,6]]
```

Copie superficielle/profonde

- Solution: Effectuer une copie profonde, en utilisant la fonction deepcopy du module copy.

```
In [6]: L3=c.deepcopy(L1)  In
[7]:id(L1[0])==id(L3[0])
Out [7]:False
In[8]: L1[0].append(80)
In[9]: L1,L3
Out[10]:[[1,2,3,66,80],[4,5,6],56],[[1,2,3,66],[4,5,6],56]
```

Les ensembles

- Un ensemble est une collection d'objets distincts et de types hétérogènes.
- Un ensemble est une structure non ordonnée: il n'est pas possible d'utiliser un indice (rang) ni de sélectionner un sous-ensemble.
- Un ensemble en PYTHON est un objet mutable.

Comment initialiser/créer un ensemble?

- Initialisation: Il n'existe pas de syntaxe particulière pour les ensembles comme il en existe par exemple pour les listes et les tuples (crochets vides ou parenthèses vides). Les accolades vides sont relatives aux dictionnaires.

- La seule méthode possible consiste à utiliser la méthode set() pour l'initialisation.

- La méthode set(iterable) permet de créer un ensemble avec comme éléments, les éléments de l'itérable.

Comment initialiser/créer un ensemble?

```
In [1]: E=set(); E# ensemble vide  Out
[1]:set()

In [2]: type(E);
Out [2]:<class 'set'>

In [3]: E={}; type(E)# Attention, ceci n'est pas un ensemble vide  Out
[3]:<class 'dict'>

In [4]: E={4,7,8};E
Out [4]:{4,7,8}

In [5]: E={'Bonjour', 56, 56};E# éléments distincts  Out
[5]:{'Bonjour', 56}
```

Comment initialiser/créer un ensemble?

```
In [6]: E=set('Bonjour'); E# L'itérable est une chaîne  Out
[6]:{'B','j','o','n' 'r','u'}

In [7]: E=set([1,4,8,8]);E# L'itérable est une liste  Out
[7]:{1,4,8}

In [8]: E=set((4,7,8,8));E# L'itérable est un tuple  Out
[8]:{4,7,8}

In [9]: E=set(range(11));E# L'itérable est un range  Out [9]:{0, 1,
2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
In [10]: E=set((i,i) for i in range(11) if i%2==0);E# par compréhension  Out [10]:{(0,0),
(6,6), (4,4), (10,10), (8,8), (2,2)}
```

Opérations sur les ensembles

```
In [1]: E=set(range(11))
In [2]: len(E);
Out [2]:11

In [3]: 5 in E  In
[3]:True

In [4]: 6 not in E
Out [4]:False
```

```
In [5]:fori inE:  print(i)
0
1
2
3
4
5
6
7
8
9
10
```

Opérations sur les ensembles

Intersection - Union

```
In [1]: {1,2,3} | {4,5,6}# le symoble | pour l'union
Out[1]:{1,2,3,4,5,6}

In [2]: {1,2,3}.union({4,5,6})# la méthode union
Out [2]:{1,2,3,4,5,6}

In [3]: {4,5} & {4,5,6}# le symoble & pour l'intersection
Out[3]:{4,5}

In [4]: {4,5}.intersection({4,5,6})# la méthode intersection  Out
[4]:{4,5}
```

Opérations sur les ensembles

Différence - Différence symétrique

```
In [1]: {4, 5, 6, 7, 8} - {4, 5, 6, 10}# le symoble - pour la différence
Out[1]:{8, 7}

In [2]: {4,5,6,7,8}.difference({4, 5, 6, 10})# la méthode difference
Out [2]:{8, 7}


In [4]: {4,5,6,7,8}.symmetric_difference({4, 5, 6, 9,10})# la méthode
symmetric_difference
Out [4]:{7,8,9,10}
```

Opérations sur les ensembles

```
In [1]: {4, 5}.issubset({4, 5, 6, 10})# teste si un ensemble est un  sous-ensemble
d'un autre ensemble
Out[1]:True

In [2]: {4,5,6,10}.issuperset({4, 5})# teste si un ensemble est un
super-ensemble d'un autre ensemble
Out [2]:True

In [3]: {4,5,6}.isdisjoint({4, 5})# teste si deux ensembles sont disjoints  Out [3]:False
```

Opérations sur les ensembles

Ajout - suppression

```
In [1]: E={1, 2, 3}
In [2]: E.add(4);E
Out [2]:{1,2,3,4}

In [3]: E.remove(3);E
Out [3]:{1,2,4}
```

Les dictionnaires

- Les dictionnaires sont le seul type associatif en PYTHON. Ces objets associatifs permettent de réaliser une correspondance entre valeurs

  de hashage (clés) et objets associés (valeurs). On parle de couples

  <clef, valeur>

- Le dictionnaire n'est pas une structure ordonnée ⇒ Pas de position (rang) d'une valeur dans le dictionnaire. Par contre, l'accès à une valeur se fait à travers la clef.

- Un dictionnaire est un objet mutable

Comment initialiser/créer un dictionnaire

```
In [1]: D=dict(); D# dictionnaire vide  In
[1]:{}

In [2]: type(D)
Out [2]:<class 'dict'>

In [3]: D={}; type(D)# dictionnaire vide  In
[3]:<class 'dict'>

In [4]: D={1:'A',2:'B',3:'C',4:'D',5:'E'};D# ensemble de clef:valeur
Out [4]:{1:'A',2:'B',3:'C',4:'D',5:'E'}

In [5]: D=dict((i,i) for i in range(1,6))# par compréhension  Out
[5]:{1:1,2:2,3:3,4:4,5:5}
```

Opérations sur les dictionnaires

```
In [1]: D={1:'A',2:'B',3:'C',4:'D',5:'E'};D
Out [1]:{1:'A',2:'B',3:'C',4:'D',5:'E'}

In [2]: len(D)
In [2]:5

In [3]: 5 in D#Attention la recherche se fait par clef  Out [3]:True

In [4]: 'A' in D
Out [4]:False

In [5]: D.get(1)# ou bien D[1], attention 1 est la clef et non pas la  position
Out [5]:'A'
```

Les méthodes keys(), values() et items()

```
In [1]: D.keys()
Out [1]:dict_keys([1,2,3,4,5])

In [2]: D.values()
In [2]:dict_values(['A','B','C','D','E'])

In [3]: D.items()
Out [3]:dict_items([(1,'A'),(2,'B'),(3,'C'),(4,'D'),(5,'E')])
```

Les méthodes keys(), values() et items()

```
foriinD.keys():
    print(i)
1
2
3
4
```

```
foriinD.values():  print(i)
A  B  C  D  E
```

```
foriinD.items():
    print(i)
(1,'A')
(2,'B')
(3,'C')
(4,'D')
(5,'E')
```

```
fori,jinD.items():  print(i,j)
1 A
2 B
3 C
4 D
5 E
```

Les dictionnaires

Ajout/Modification/suppression

```
In [1]:
D={'MP6-1':'Ala','MP6-2':'Azhar','MP6-3':'Youssef','MP6-4':'Aymen',
'MP6-5':'Iheb','MP6-6':'Khaoula','MP6-7':'Nesrine','MP6-8':'Ali',
'MP6-9':'Haykel','MP6-10':'Fatma'}

In [2]: D['MP6-11']='Leila'; D# si la clef n'existe pas, la paire va être  ajoutée
Out [2]:{'MP6-1':'Ala','MP6-2':'Azhar','MP6-3':'Youssef',
'MP6-4':'Aymen','MP6-5':'Iheb','MP6-6':'Khaoula','MP6-
7':'Nesrine','MP6-8':'Ali'
,'MP6-9':'Haykel','MP6-10':'Fatma','MP6-11':'Leila'}
```

Ajout/Modification/suppression

```
In [3]: D['MP6-10']='Fattouma'; D# si la clef existe, la valeur va être  modifiée
Out [3]:{'MP6-1':'Ala','MP6-2':'Azhar','MP6-3':'Youssef',
'MP6-4':'Aymen','MP6-5':'Iheb','MP6-6':'Khaoula','MP6-
7':'Nesrine','MP6-8':'Ali'
,'MP6-9':'Haykel','MP6-10':'Fattouma','MP6-11':'Leila'}

In [4]: D.pop('MP6-11');D# suppression de la valeur dont la clef est  'MP6-11'
Out [4]:{'MP6-1':'Ala','MP6-2':'Azhar','MP6-3':'Youssef',
'MP6-4':'Aymen','MP6-5':'Iheb','MP6-6':'Khaoula','MP6-
7':'Nesrine','MP6-8':'Ali'
,'MP6-9':'Haykel','MP6-10':'Fattouma'}
```

Fonction PYTHON

Une fonction en PYTHON:

- est un bloc d'instructions;
- prend (éventuellement) des paramètres non typés;
- retourne un objet, ou plusieurs objets en même temps.
- peut ne rien retourner (procédure).

Définition d'une fonction

En PYTHON, une fonction est définie à l'aide du mot clé def:

def nom_fonction(paramètres formels):

```
.....
.....<instructions>.....
.....
return(objet)
```

- N'oubliez pas les ":" c'est pour l'indentation.
- L'instruction return renvoie l'objet à retourner.
- L'instruction return provoque la sortie de la fonction.
- Procédure = Fonction sans return.

Exemples de fonctions

```
def somme(a,b):
        return(a+b)
def quotient_reste(a,b):
return(a//b,a%b)
def a che(a):
        print('La valeur est ',a)
```

Appel - Fonction renvoyant un ou plusieurs objets

Une fonction renvoyant un ou plusieurs objets est appelée comme suit:

Récupération du résultat de la fonction dans une variable

```
res=nom_fonction(paramètres effectifs)
```

A chage du résultat de la fonction

```
print('le résultat est ', nom_fonction(paramètres effectifs))
```

Remarque: Lorsque la fonction retourne plusieurs objets en même temps, il s'agit d'un tuple.

Appel - Fonction ne renvoyant pas d'objet

Une fonction ne renvoyant pas d'objet est appelée comme suit:

```
nom_fonction(paramètres effectifs)
```

Remarque: Attention, si vous faites appel à une fonction ne renvoyant pas d'objet avec la syntaxe res=nom_fonction(paramètres effectifs). La variable res contiendra la valeur None.

Exemple

```
#définition des fonctions

#fonction qui retourne un objet (int)
def somme(a,b):
    return(a+b)

#fonction qui retourne deux objets en même temps, c'est donc un tuple
def quotient_reste(a,b):
    return(a//b,a%b)

#fonction qui ne retourne rien
def afficher(a):
    print('La valeur est ',a)

#Script principal

x=int(input('donner un entier'))
y=int(input('donner un entier'))
s=somme(x,y)
afficher(s)
qr=quotient_reste(x,y)
#qr est un tuple
afficher(qr)
```

Exemple - suite

```
#On peut manipuler le tuple qr retourné par la fonction quotient_reste
q=qr[0]
print("quotient= ",q)
r=qr[1]
print("reste = ", r)

#On peut parcourir les élèments du tuple comme ceci
print('premier parcours')
for i in qr:
    afficher(i)

#On peut également parcourir les élèments du tuple comme celà
print('deuxième parcours')
for i in range(len(qr)):
    afficher(qr[i])
```

Passage de paramètres par position

- Le passage de paramètres par position est le mode de passage où l'ordre (la position) des paramètres est important.
- Chaque paramètre formel sera substitué par le paramètre effectif correspondant selon sa position lors de l'appel.

Exemple:

- quotient_reste(25,5) ⇒ renvoie (5,0)
- quotient_reste(5,25) ⇒ renvoie (0,5)

Passage de paramètres par nom

- Le passage de paramètre par nom consiste à faire appel à la fonction en précisant paramètre formel=paramètre effectif

Exemple:

- quotient_reste(a=25,b=5) ⇒ renvoie (5,0)
- quotient_reste(b=5,a=25) ⇒ renvoie (5,0)

Paramètres par défaut (1)

- Le principe de paramètres par défaut consiste à a ecter des valeurs aux paramètres lors de la dé nition de la fonction.

- Si le paramètre n'est pas précisé lors de l'appel, c'est la valeur par défaut qui sera utilisée.

- Si la valeur du paramètre est précisée lors de l'appel, c'est bien cette dernière qui sera utilisée.

- Les paramètres avec valeur par défaut doivent être regroupés à la n dans la liste des paramètres.

Paramètres par défaut (2)

Définition de la fonction

```
# ici, le paramètre b possède une valeur par défaut égale à 5  def
quotient_reste(a,b=5):
    return(a//b,a%b)
```

Appel de la fonction

```
# quand le deuxième paramètre n'est pas spéci é, PYTHON utilise la
valeur par défaut
(1)    quotient_reste(25) ⇒ renvoie (5,0)
# quand le deuxième paramètre est spéci é, PYTHON utilise ce dernier
(2)    quotient_reste(25,2) ⇒ renvoie (12,1)
```

Paramètres par défaut (3)

```
# Attention, cette dé nition n'est pas correcte car les paramètres par défaut doivent être placés après les paramètres obligatoires
def quotient_reste(a=25,b):
    return(a//b,a%b)
```

```
# Cette définition est correcte  def quotient_reste(b,a=25):
    return(a//b,a%b)
```

Mode de passage des paramètres

- Les paramètres sont toujours passés par référence, mais ils sont modi ables selon qu'ils sont mutables (listes, ensembles, dictionnaires) ou non mutables (types simples, chaînes de caractères et les tuples).

Exemple 1: on va essayer de modi er un objet non mutable (int)

```
def modi er_non_mutable(a):  a=99
    print('La valeur locale est',a)


x=int(input('Donner un entier'))  modi
er_non_mutable(x)
print('La valeur de x est',x)
```

Testez et interprétez

Variables locales/globales

- Une variable locale est une variable dé nie au niveau d'une fonction. Elle n'est visible que dans cette fonction.

- Par défaut, toute variable utilisée dans une fonction est une variable locale à cette fonction.

- Une variable gloable est une variable dé nie au niveau du programme principal. Elle est donc visible par n'importe quelle fonction dé nie au niveau de ce programme.

- Pour qu'une fonction utilise une variable globale dé nie au niveau du programme principal, il faut employer le mot clé global.

Variables locales/globales

Testez et interprétez

```
def f1(v):
    x=v
#appel
x=10
f1(5)
print(x)
```

```
def f2(v):
    x=x+v
#appel
x=10
f2(5)
```

```
def f3(v):
    global x
    x=x+v
#appel
x=10
f3(5)
print(x)
```

Fonction locale/globale

- Il est possible de dé nir une fonction dans une autre fonction. Dans ce cas, on parle de fonction locale. Elle n'est visible que par la fonction dans laquelle elle a été dé nie.

```
def max3(a,b,c):
    def max2(a,b):
        if a>b:
            return a
        else:
            return b
    m=max2(a,b)
    return(max2(m,c))

x=24
y=8
z=12
maxim=max3(x,y,z)
print('le max est',maxim)
```

L'appel à la fonction max2 dans le script principal provoquera une erreur.

Fonction lambda

- La fonction lambda est utilisée lorsque la fonction est une expression mathématique.
- Une fonction lambda est dé nie à l'aide du mot clef lambda.

Syntaxe de dé nition d'une fonction lambda:

```
nom_fonction=lambdax:expression
nom_fonction=lambdax,y:expression
nom_fonction=lambdax: expression1 if condition else expression2
nom_fonction=lambdax,y: expression1 if condition else expression2
```

Fonction lambda

```
In [1]: f=lambda x:x**2

In [2]: f(2)
Out [2]: 4
In [3]: somme=lambda x,y:x+y  In [4]: somme(5,4)

Out [4]: 9
In [5]: valeur_absolue=lambda x:x if x>0 else -x  In [6]: valeur_absolue(5)

Out [6]: 5

In [7]: valeur_absolue(-5)
Out [7]: 5
```

Les fonctions récursives

Factorielle

$$0! = 1$$

$$n! = n \times (n - 1) \times .... \times 1$$

```
def fact(n):
    res=1
        for i in range(2, n+1):
            res=res*i
        return(res)
```

Eya Smati ⇒ version itérative de la fonction factorielle

Les fonctions récursives

Une autre façon de formaliser la fonction factorielle

Factorielle

$$0! = 1$$
$$n! = n \times (n - 1)!$$

```
def fact(n):
    if n==0:
        return(1)
    else:
        return(n*fact(n-1))
```


⇒ version récursive de la fonction factorielle

Les fonctions récursives

Qu'est-ce qu'une fonction récursive?

- Une fonction récursive est une fonction qui contient un appel à elle-même.

Comment écrire une fonction récursive?

- Identifier le cas de base
- Identifier le cas général qui effectue la récursion (les appels récursifs)

Les fonctions récursives

Pourquoi un cas de base?

- Le cas de base est le cas qui précise la condition d'arrêt des appels récursifs.
- Dans la fonction factorielle, il faut arrêter les appels récursifs lorsque n=0 car dans ce cas on connaît le résultat.
- Attention: Absence de cas de base ⇒ suite in nie d'appels récursifs!

Les fonctions récursives

Comment ça marche?

- Les appels récursifs se déroulent selon le principe LIFO (Last-In-First-Out ou dernier arrivé premier servi).
- On peut représenter ces appels avec une structure en pile.

Les fonctions récursives

Le calcul de fact(4)

Appel	n	Appel récursif	Résultat
fact(0)	0	arrêt	1
fact(1)	1	*1* × fact(0)	1 × 1=1
fact(2)	2	*2* × fact(1)	2 × 1=2
fact(3)	3	*3* × fact(2)	3 × 2=6
fact(4)	4	*4* × fact(3)	4 × 6=24

Fichier

Motivation

- Les données manipulées jusqu'ici sont introduites par l'utilisateur au moment de l'exécution.

- Le résultat est perdu dès que le programme s'arrête.

  ⟹ Les fichiers sont un excellent moyen pour la sauvegarde des données.

Contexte

Programmes qui récupèrent les données à partir d'un chier et qui sauvegardent le résultat dans un chier.

Fichier en PYTHON

- PYTHON dispose d'un objet de type le qui va permettre de sauvegarder et de récupérer les données à partir d'un chier.

- Les opérations sur un chier:
  - ouverture/création
  - fermeture
  - lecture
  - écriture

Ouverture/Création (1)

Syntaxe

f=open(nom_fichier , mode)

- nom_fichier: une chaîne de caractères indiquant le nom du fichier à ouvrir (exemple: 'test.txt')
- mode: le mode d'ouverture du chier est donné sous forme de chaîne de caractères. Voici les principaux modes d'ouverture:

  'r': ouverture en lecture seule (read). Si le fichier n'existe pas, une erreur IOError est levée.

  'w': ouverture en écriture (write). Si le chier existe, alors il sera écrasé, sinon il sera crée.

  'a': ouverture en écriture en mode ajout (append). Si le chier existe, l'écriture sera faite à la n du chier. Sinon, le fichier est créé.

- On peut parler de mode 'rb', 'wb' et 'ab' pour ouvrir le fichier en mode binaire. Nous verrons un peu plus loin ce mode particulier avec le module pickle.

Ouverture/Création (2)

- Pour ouvrir en lecture, le fichier 'test.txt' du répertoire courant:

```
f=open('test.txt','r')
```

- Si le fichier existe dans un autre emplacement, le nom du fichier doit être précédé par son chemin d'emplacement.
  Exemple: f=open('/home/leila/Bureau/test.txt','r')
  NB: Le répertoire courant peut-être changé: voir Section module os.

Fermeture

Syntaxe

```
f.close()
```

NB: N'oubliez pas de fermer un fichier après l'avoir lu!

Lecture

Pour lire le contenu d'un fichier, on utilise l'une des méthodes suivantes:

- f.read(): permet de lire tout le contenu du chier. Cette méthode retourne une chaîne de caractères représentant tout le contenu du chier.
- f.read(n): permet de lire n caractères.
- f.readlines(): permet de lire tout le contenu du chier. Cette méthode retourne une liste composée de chaînes de caractères. Chaque chaîne représente une ligne du fichier.
- f.readline(): permet de lire une ligne à la fois. Cette méthode retourne une chaîne de caractère représentant la ligne courante lue.

Écriture

Pour écrire dans un chier, on utilise l'une des méthodes suivantes:

- f.write('texte'): permet d'écrire la chaîne de caractères ('texte') passée en paramètre.
- f.writelines(L): permet d'écrire une liste L, où chaque élément de L est une chaîne de caractères.

NB: pour constituer des lignes dans le chier, on doit ajouter le symbole "\n" (symbole retour chariot).

Le module os

Par défaut, l'interprète PYTHON effectue les opérations d'ouverture/lecture/écriture à partir du répertoire courant. Il se peut que le chier avec lequel vous travaillez soit placé dans un autre répertoire.
Pour cela, on peut changer de répertoire de travail ⟹ utiliser le module os.

- L'import du module os

import os as os

- Pour connaître le répertoire courant os.getcwd()

'/home/leila'

- Pour changer de répertoire courant

os.chdir('/home/leila/Bureau/Cours_LBO_2eme')

Le module pickle

Le module pickle est le module qui permet de faire la sérialisation de données (en binaire). La sérialisation est le concept de conversion de données structurées dans un format qui lui permet d'être stocké de manière à ce que sa structure d'origine puisse être récupérée.

- Pour importer le module pickle

```
import pickle as p
```

- Pour enregistrer un objet (ici un dictionnaire) f=open(' lename', 'wb')

```
obj = {1 : 'a' , 2 : 'b'}
p.dump(obj, f)
```

- Pour récupérer l'objet f=open(' lename', 'rb')

```
obj = p.load(f)
```

Exemple

```
import p i c k l e as p
# c r e a t i o n d ' o b j e t s de d i f f e r e n t s ty p e s
 e t u d i a n t s = { 1 : ' A l i ' , 2 : ' Khaoula ' , 3 : ' N e s r i n e ' }  L=[ '
Bonjour ' , 123 , ' abc ' ]
T=(L , ' s o l e i l ' )
 a=12
b=12.5
# e n r e g i s t r e m e n t des o b j e t s
 f = open ( ' mes_donnees . b i n ' , ' wb ' )
 p . dump( e tu d i a n t s , f )
p . dump( L , f )
p . dump(T, f )
p . dump( a , f )
p . dump( b , f )
 f . c l o s e ()
```

Exemple

```
#recuperation des objets
 f = open ('mes_donnees.bin','rb')
 o1=p.load(f)
o2=p.load(f)
o3=p.load(f)
o4=p.load(f)
o5=p.load(f)
print(o1)
print(o2)
print(o3)
 print(o4)
 print(o5)
 f.close()
```

Exemple

On peut récupérer les objets autrement:

```
# r e c u p e r a t i o n des o b j e t s
 f = open ( ' mes_donnees . b i n ' , ' rb ' )
  while True :
        try :
              o b j = p . l o ad ( f )
              p r i n t ( obj , type ( o b j ) )
        except EOFError :
              break
 f . c l o s e ()
```

NumPy

- NumPy est une bibliothèque python parmi les plus importantes pour faire du calcul scienti que, du machine learning et du data science. Tous les autres modules tels que scipy et matplotlib se base sur le type ndarray de numpy.
- Le module numpy est un module PYTHON spécialisé dans la manipulation des tableaux à n dimensions (ndarray), qui est un objet extrêmement puissant. Dans le cadre de notre cours, on s'intéressera aux tableaux à 1 et à 2 dimensions.
- Les tableaux numpy sont homogènes, c'est-à-dire constitués d'éléments de même type.
- Site officiel de numpy: http://www.numpy.org/

Le type ndarray - N-dimensional array

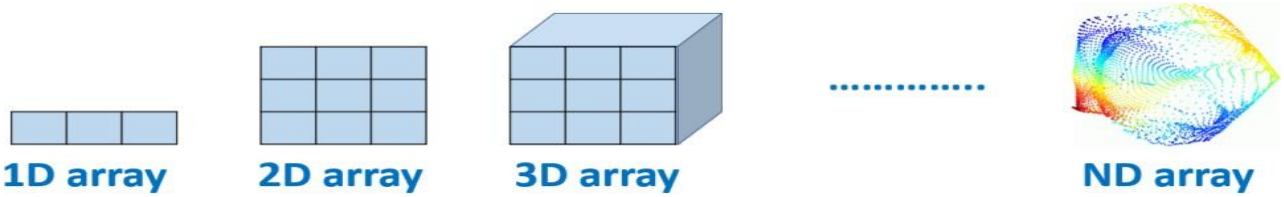
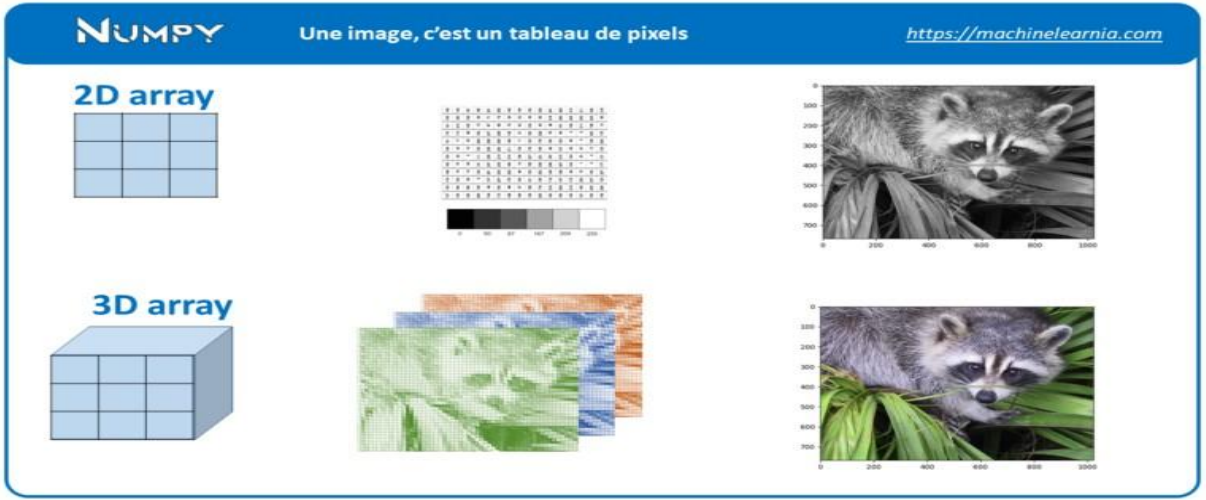


Image en niveaux de gris (2D array) - Image couleur (3D array)



Exemple de tableaux ndarray

- Matrice:

$$\begin{bmatrix} 0 & 0 & 0.003 & 0.868 & 0 \\ 0 & 100 & 0.002 & 0.812 & 1 \\ 30.223 & 100 & 0.004 & 0.832 & 1 \\ 29.104 & 100 & 0.004 & 0.814 & 1 \end{bmatrix}$$

- Vecteur colonne:

$$\begin{bmatrix} 9.515 \\ 8.234 \\ 30.225 \\ 29.104 \end{bmatrix}$$

- Vecteur ligne:

$$27.340$$

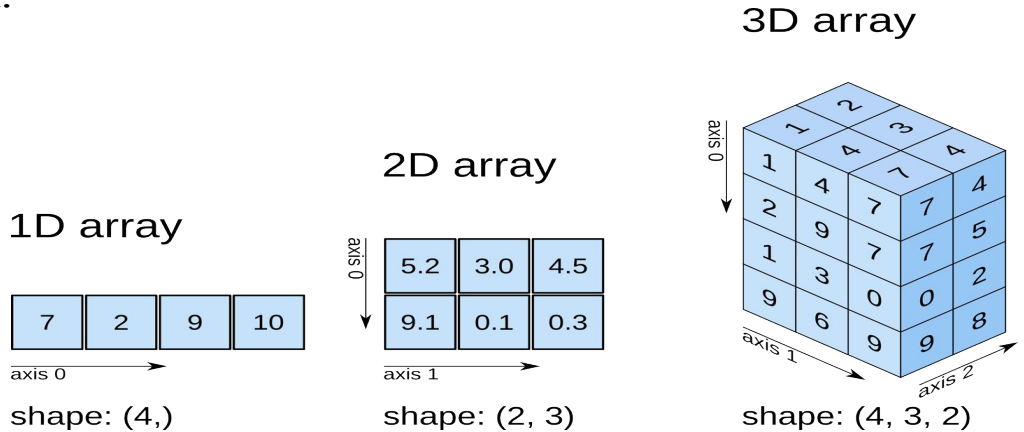
$$\cdot \; 9.515 \; 8.234 \; 30.225 \; 29.104 \; 27.340 \; \Sigma$$

Le type ndarray - ndim/shape

Soit a un tableau (ndarray)

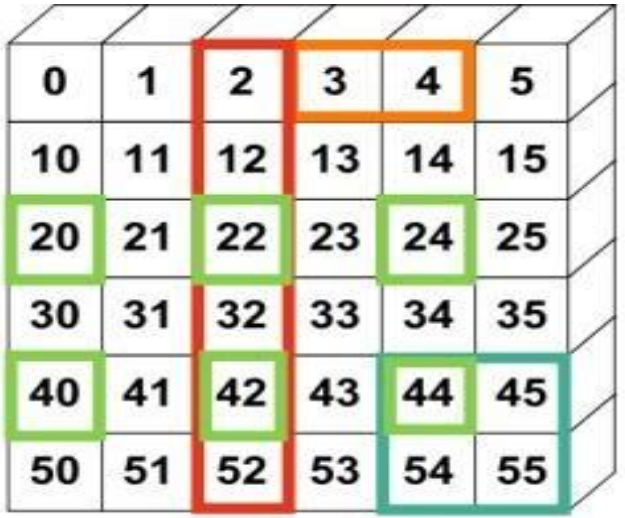
- a.ndim ==⇒ le nombre de dimensions du tableau.

- a.shape ==⇒ un tuple composé du nombre d'éléments sur chaque dimension.



NumPy - Slicing

```
>>> a[0,3:5]
array([3,4])
>>> a[4:,4:]
array([[44,45],[54,55]])
>>> a[:,2]
array([2,12,22,32,42,52])
>>> a[2::2,::2]
array([[20,22,24],
       [40,42,44]])
```



Création d'un tableau (1)

- Import du module numpy

```
import numpy as np
```

- Création d'un tableau à une dimension avec la fonction array (Typage implicite)

```
> T=np.array([1,2,3])
>  T
array([1, 2, 3])
```

- L'objet crée par la fonction array est de type ndarray.

```
>  type(T)
<class 'numpy.ndarray'>
```

Création d'un tableau (2)

- Tous les éléments d'un tableau sont de même type. L'attribut dtype (data type) permet de voir le type des éléments.

```
> T.dtype
dtype('int32')
```

- Création d'un tableau avec un typage explicite en utilisant dtype

```
> T=np.array([1,2,3],dtype=float)
> T
    array([ 1., 2., 3.])
> T.dtype
dtype(' oat64')
```

- Les principaux types sont : bool, int, float, complex

Taille - Accès à un élément du tableau

- On calcule la taille d'un tableau T (le nombre d'éléments) à l'aide la fonction len (len(T));

- On accède à chaque élément du tableau par son index:
  - T [0]: premier élément de T
  - T [len(T) − 1]: dernier élément de T

- Lorsque l'index est négatif, le parcours est effectué à rebours à partir de la n:
  - T [−1]: premier élément de T (de droite vers la gauche).
  - T [−len(T)]: dernier élément de T (de droite vers la gauche).

Attributs d'un tableau

- Le nombre de dimensions

```
>T.ndim  1
```

- Le nombre d'éléments

```
>T.size  3
```

- Le nombre d'éléments sur chaque dimension

```
>   T.shape
(3,) # tuple: 3 éléments sur la première dimension
```

Attributs d'un tableau

- Attention: Les attributs ndim, size et shape peuvent être invoqués à la fois comme des attributs (caractéristiques) d'un tableau et comme des fonctions du module numpy prenant un tableau en paramètre.

- Par exemple pour l'attribut shape, on peut donc écrire aussi bien T.shape que np.shape(T).

Découpage ou Slicing sur un tableau

- Comme pour les séquences, le module numpy permet le découpage du tableau en tranches avec la syntaxe suivante T[debut : n]
- T[debut : n] désigne la tranche de T constituée des éléments dont les index sont compris entre debut et n − 1.
- Quand debut n'est spéci é, alors la tranche commencera du début de T
- Quand n n'est spéci é, alors la tranche nira à la n de T
- La syntaxe T[debut : n : pas] possède un troisième paramètre (par défaut, égal à 1), indiquant le pas de sélection.

Création d'une matrice (1)

- Création d'un tableau à deux dimensions (matrice) avec la fonction array

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

- L'objet crée par la fonction array est toujours de type ndarray.

```
>  type(M)
<class 'numpy.ndarray'>
```

Taille - Accès à un élément/une ligne d'une matrice

- La fonction len(M) retourne le nombre de lignes de M.

- On accède à chaque élément de la matrice par son index ligne et index colonne:
  - M[i,j]: élément qui se trouve à la ligne i et à la colonne j.

- On peut récupérer toute la ligne i avec la syntaxe M[i]. Le résultat est un tableau à une dimension.
  - M[0]: première ligne de M.
  - M[len(M)-1]: dernière ligne de M.

- Lorsque l'index est négatif, le parcours est e ectué à rebours à partir de la n:
  - M[-1]: dernière ligne de M.
  - M[-len(M)]: première ligne de M.

Taille - Accès à un élément/une ligne d'une matrice

```
>len(M)  4

> M[0,2]  3

> M[2]
array([7, 8, 9])
> M[0]
array([1, 2, 3])
```

```
>  M[-len(M)]
array([1, 2, 3])
> M[-1]
array([10, 11, 12])
> M[len(M)-1]
array([10, 11, 12])
> M[-1,-1]  12
```

Attributs de la matrice

- Le nombre de dimensions

```
>M.ndim  2
```

- Le nombre d'éléments

```
>M.size  6
```

- Le nombre d'éléments sur chaque dimension: nombre de lignes, nombres de colonnes

```
>  M.shape
(2,3) # tuple: 2 lignes et 3 colonnes
```

Découpage ou Slicing sur une matrice

```
>M=np.array([[1,2,3,4,5,6,7,8],[9,10,11,12,14,15,16,17],
            [18,19,20,21,22,23,24,25],[26,27,28,29,30,31,32,33]])
>  M
array([[ 1, 2, 3, 4, 5, 6, 7, 8],
       [ 9, 10, 11, 12, 14, 15, 16, 17],
       [18,  19, 20, 21, 22, 23, 24,  25],
       [26, 27, 28, 29, 30, 31, 32, 33]])
```

Lignes 1 à 3, colonnes 2 à 5

```
> M[1:4,2:6]
array([[11, 12, 14, 15],
       [20, 21, 22, 23],
       [28, 29, 30, 31]])
```

Lignes et colonnes paires

```
> M[::2,::2]
array([[ 1, 3, 5, 7],
       [18, 20, 22, 24]])
```

Découpage ou Slicing sur une matrice

Lignes et colonnes impaires

```
> M[1::2,1::2]
array([[10, 12, 15, 17],
       [27, 29, 31, 33]])
```

Trois premières lignes, deux premières colonnes

```
> M[:3,:2]
array([[ 1, 2],
       [ 9, 10],
       [18, 19]])
```

Découpage ou Slicing sur une matrice

à partir de la ligne 2, à partir de la colonne 4

```
> M[2:,4:]
array([[22, 23, 24, 25],
       [30, 31, 32, 33]])
```

on inverse l'ordre des colonnes

```
> M[:,::-1]
array([[ 8, 7, 6, 5, 4, 3, 2, 1],
       [17, 16, 15, 14, 12, 11, 10, 9],
       [25, 24, 23, 22, 21, 20, 19, 18],
       [33, 32, 31, 30, 29, 28, 27, 26]])
```

Découpage ou Slicing sur une matrice

on inverse l'ordre des lignes

```
>M[::-1,::]
array([[26, 27, 28, 29, 30, 31, 32, 33],
       [18,  19, 20, 21, 22, 23, 24,  25],
       [ 9, 10, 11, 12, 14, 15, 16, 17],
       [ 1, 2, 3, 4, 5, 6, 7, 8]])
```

on inverse l'ordre des lignes et l'ordre des colonnes

```
> M[::-1,::-1]
array([[33, 32, 31, 30, 29, 28, 27, 26],
       [25, 24, 23, 22, 21, 20, 19, 18],
       [17, 16, 15, 14, 12, 11, 10, 9],
       [ 8, 7, 6, 5, 4, 3, 2, 1]])
```

Accès à une colonne

- La syntaxe M[:,b] permet d'accéder à la colonne d'indice b. Le résultat est un tableau à une dimension.
- De même, la syntaxe M[:,b:b+1] permet d'accéder à la colonne d'indice b. Le résultat est un tableau à deux dimensions.

Accès à la colonne d'indice 3

```
> M[:,3]
array([ 4, 12, 21, 29])
```

Accès à la colonne d'indice 3

```
> M[:,3:4]
array([[ 4],
       [12],
       [21],
       [29]])
```

Création de tableaux particuliers

- Tableau d'une dimension de 5 valeurs aléatoires (par défaut, des réels)

```
> np.empty(5)
array([ 2.79206344e+178, 2.25315555e-316, 0.00000000e+000,
2.80858511e+178, 6.35862486e-321])
```

Création de tableaux particuliers

- Tableau d'une dimension de 5 valeurs égales à 1.

```
> np.ones(5)
array([ 1., 1., 1., 1., 1.])
> np.ones(5,int)
array([1, 1, 1, 1, 1])
```

- Tableau d'une dimension de 5 valeurs égales à 0.

```
> np.zeros(5)
array([ 0., 0., 0., 0., 0.])
```

Création de tableaux particuliers

- La fonction arange(d , f , h) permet de créer un tableau de valeurs de l'intervalle [d, f [avec un pas de h.
- Les paramètres d et h sont optionnels.

```
>  np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>  np.arange(1,10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>np.arange(1,10,2)
 array([1,3, 5, 7, 9])
```

Création de tableaux particuliers

- La fonction linspace(a, b, n) permet de créer un tableau de n valeurs régulièrement échelonnées de l'intervalle [a,b] (les bornes de l'intervalle sont incluses). Par défaut, n est égale à 50.

```
> np.linspace(0,100,4)
array([ 0. , 33.33333333, 66.66666667, 100. ])
```

pour exclure la borne supérieure de l'intervalle:

```
> np.linspace(0,100,4,endpoint=False) array([ 0., 25., 50., 75.])
```

pour connaître la valeur du pas utilisé pour générer les éléments:

```
> np.linspace(0,100,4,retstep=True) (array([ 0. , 33.33333333,
66.66666667, 100. ]), 33.333333333333336)
```

Création de matrices particulières

- Tableau de deux dimensions de valeurs aléatoires (2 lignes, 2 colonnes).

```
>  np.empty((2,2))
array([[ 9.61395921e-293, 9.26242852e-293],
       [9.08574574e-293, 6.35862486e-321]])
```

Création de matrices particulières

- Tableau de deux dimensions dont les valeurs sont égales à 1 (2 lignes, 2 colonnes).

```
> np.ones((2,2))
array([[ 1., 1.],
       [ 1., 1.]])
```

- Tableau de deux dimensions dont les valeurs sont égales à 0 (2 lignes, 2 colonnes).

```
> np.zeros((2,2))
array([[ 0., 0.],
       [ 0., 0.]])
```

Création de matrices particulières

- La fonction identity(n) permet de créer la matrice identité d'ordre n.

```
> np.identity(4)
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])
```

Création de matrices particulières

- La fonction diag, permet de créer une matrice diagonale formée par les coefficients passés en paramètres (sous forme de liste, de tuple ou même de tableau).

```
> np.diag([1,2,3,4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])

> np.diag((1,2,3,4))
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Opérations arithmétiques

Soient A et B deux tableaux numpy, les opérations mathématiques suivantes sont disponibles:

- Addition: A + B additionne terme à terme les éléments de A et B
- Soustraction: A − B soustrait terme à terme les éléments de B à ceux de A
- Multiplication: A ∗ B multiplie terme à terme les éléments de A et B
- Division: A⁄B quotients terme à terme des éléments de A par ceux de B
- Puissance: A ∗ ∗B élève les éléments de A à la puissance les éléments de B

NB: Toutes ces opérations sont également applicables avec un scalaire: addition des éléments d'un tableau et un scalaire, multiplication des éléments par un scalaire, etc.

Fonctions mathématiques

- Toutes les fonctions mathématiques usuelles sont disponibles dans le module numpy et restent applicables sur des tableaux à une ou deux dimensions. Citons par exemple: les fonctions sin, cos, tan, log, sqrt, exp, etc.

```
> T=np.linspace(-3*np.pi,3*np.pi,5)
> T
array([-9.42477796, -4.71238898, 0. , 4.71238898, 9.42477796])

> np.sin(T)
array([ -3.67394040e-16, 1.00000000e+00,
0.00000000e+00,-1.00000000e+00, 3.67394040e-16])
```

Méthodes usuelles

- M.sum() renvoie la somme des éléments de M.
- M.min() renvoie le minimum des éléments de M.
- M.max() renvoie le maximum es éléments de M.

NB:

- Quand le paramètre axis=0 est spéci é, la méthode M.sum(axis=0) (resp M.min(axis=0), M.max(axis=0)) retourne un tableau composé respectivement de la somme, du min et du max des éléments de chaque colonne.
- Quand le paramètre axis=1 est spéci é, la méthode M.sum(axis=1) (resp M.min(axis=1), M.max(axis=1)) retourne un tableau composé respectivement de la somme, du min et du max des éléments de chaque ligne.

Méthodes usuelles

```
>  M=np.array([[1,2,3],[4,5,6]])
>  M
     array([[1, 2, 3],
            [4, 5, 6]])

>M.sum()  21

>  M.sum(axis=0)
array([5, 7, 9], dtype=int32)

>  M.sum(axis=1)
array([ 6, 15], dtype=int32)
```

Produit matriciel et transposé

- La fonction produit matriciel est disponible directement sous numpy sous le nom dot. De même pour la transposition de matrice (transpose).

```
> A = np.array([[1,2],[3,4]])
> B = np.array([[2,0],[3,4]])
>np.dot(A,B)
>array([[ 8, 8],[18, 16]])

> A=np.array([[1, 2],[3, 4]])
>np.transpose(A)
>array([[1, 3],[2, 4]])
```

Le module linalg

- A n d'utiliser d'autres méthode d'algèbre linéaire, il faut utiliser le sous module linalg de numpy qu'il faudra importer avec la commande: import numpy.linalg as lng

Déterminent lng.det(M)

Inverse lng.inv(M)

Vecteurs propres lng.eig(M)

Valeurs propres lng.eigvals(M)

Résolution d'un système linéaire lng.solve(A,B) : retourne le vecteur X solution du système A*X=B

La bibliothèque matplotlib

- Le module matplotlib est une bibliothèque du langage Python destinée à tracer et visualiser des données sous formes de graphiques. Elle peut être combinée avec les bibliothèques python de calcul scienti que NumPy et SciPy. Matplotlib est distribuée librement et gratuitement.

- Site officiel: https://matplotlib.org/

- Pour tracer des courbes avec le module matplotlib, il suffit d'importer le sous-module pyplot : import matplotlib.pyplot as plt

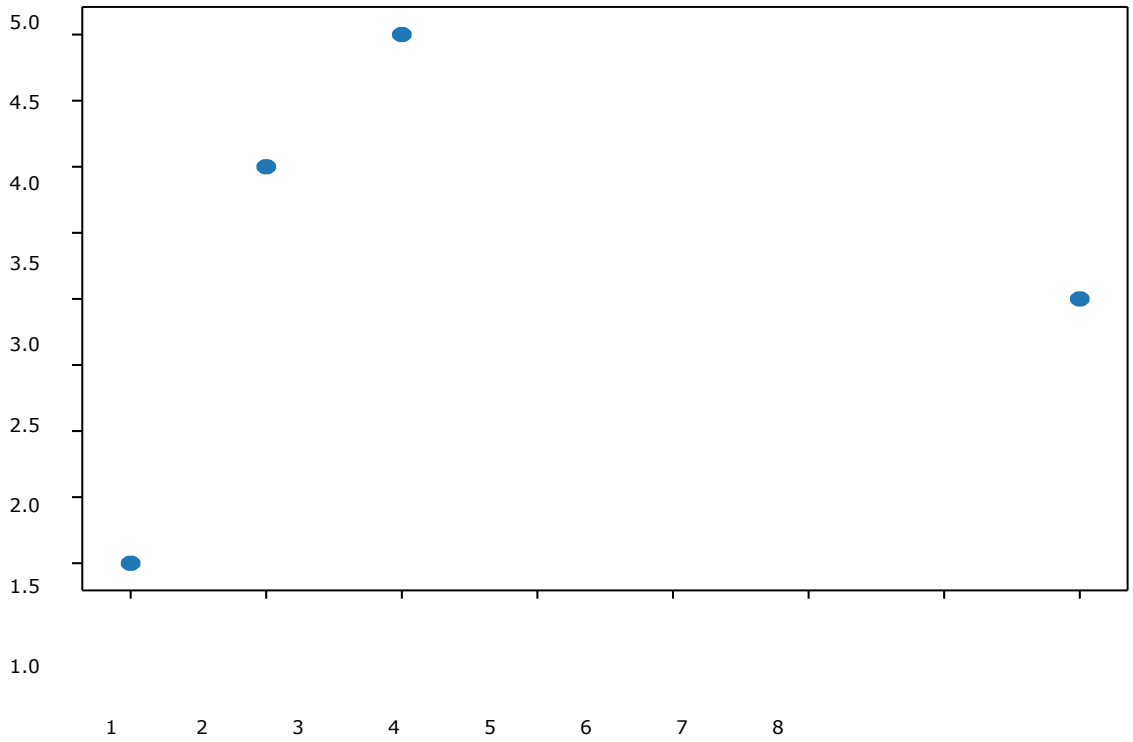
Exemple 1

- Pour représenter un nuage de points avec matplotlib, il su t de créer une liste d'abscisses x et une liste d'ordonnées y (même taille) et d'appliquer la fonction scatter(x,y).

```
import matplotlib.pyplot as plt
x =[ 1 , 2 , 3 , 8 ]
y =[ 1 , 4 , 5 , 3 ]
plt.scatter( x , y )
plt.savefig('figure1.pdf') #pour sauvegarder la figure
 plt.show ( )
```

Exemple 1

Le résultat est la figure suivante:



Exemple 2

- Pour tracer une courbe avec matplotlib, il su t de créer une liste d'abscisses et une liste d'ordonnées (de même taille). Les points seront reliés par des segments.

```
import matplotlib.pyplot as plt
x =[1,2,3,8]
y =[0,4,5,3]
plt.plot(x,y)
plt.savefig('figure2.pdf')
plt.show()
```

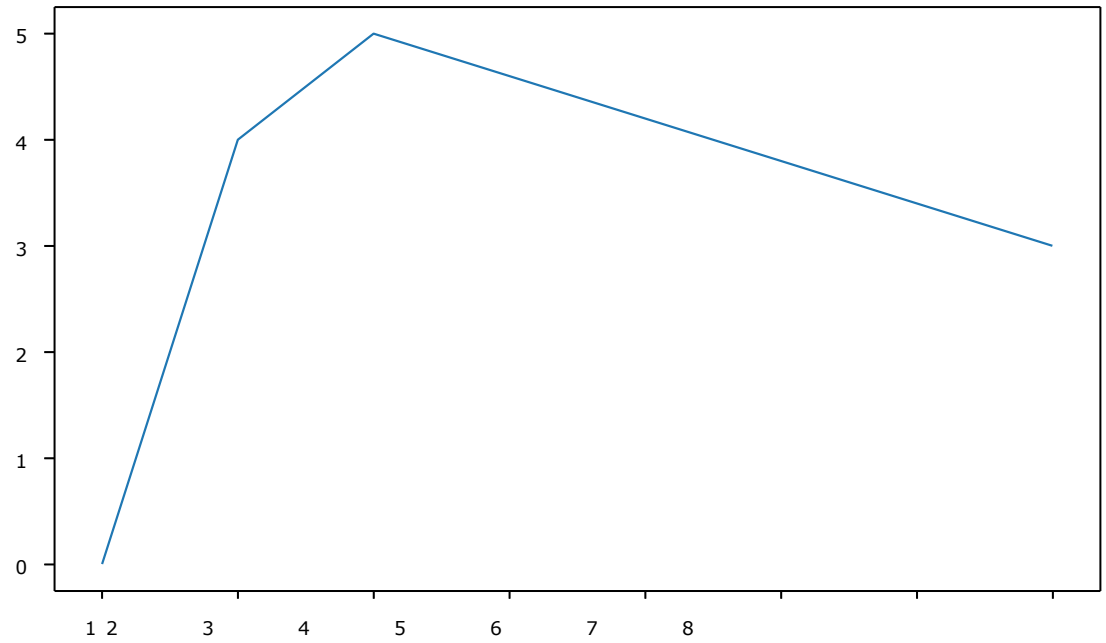
Exemple 2

- On peut également utiliser les tableaux numpy à une dimension (au lieu des listes) pour tracer une courbe.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.array([1,2,3,8])
y=np.array([0,4,5,3])
plt.plot(x,y)
plt.savefig('figure2.pdf')
plt.show()
```

Exemple 2

Dans les deux cas, le résultat est la figure suivante:

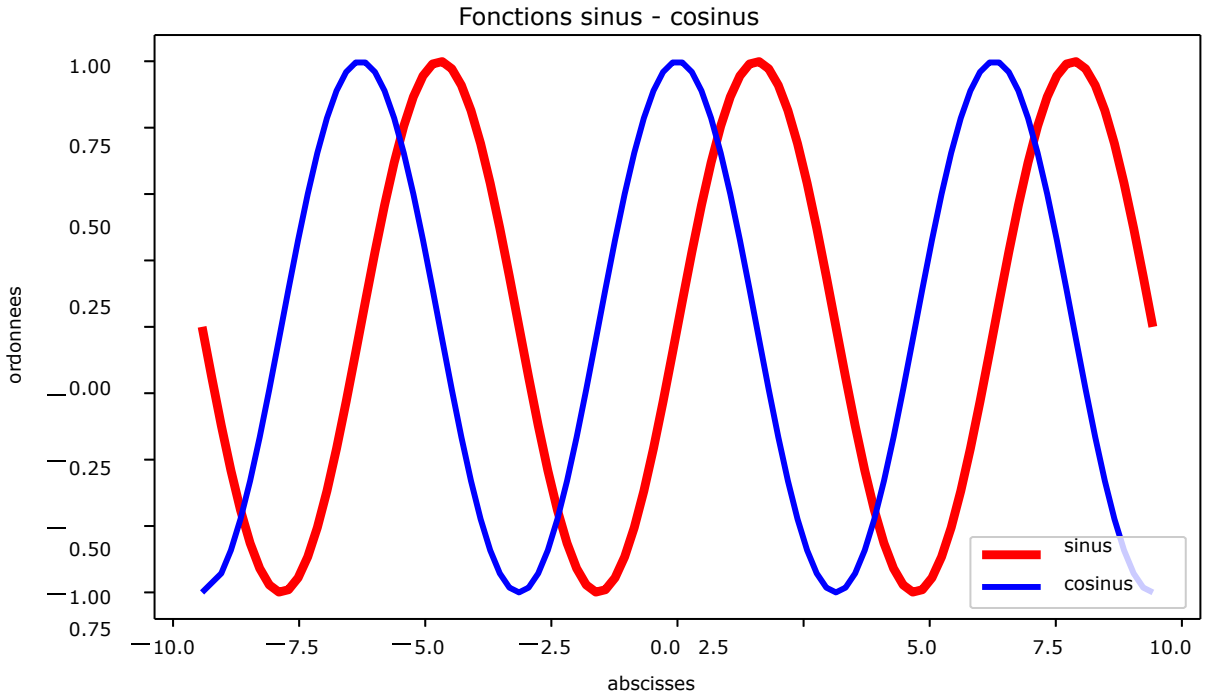


Exemple 3

- On peut tracer les courbes des fonctions sinus et cosinus.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-3*np.pi,3*np.pi,100)
y=np.sin(x)
z=np.cos(x)
plt.plot(x,y,color='r',linewidth=6)
plt.plot(x,z,color='b',linewidth=4)
plt.xlabel('abscisses')
plt.ylabel('ordonnees')
plt.title('Fonctionssinus-cosinus')
plt.legend(['sinus','cosinus'])
plt.savefig('figure3.pdf')
plt.show()
```

Exemple 3

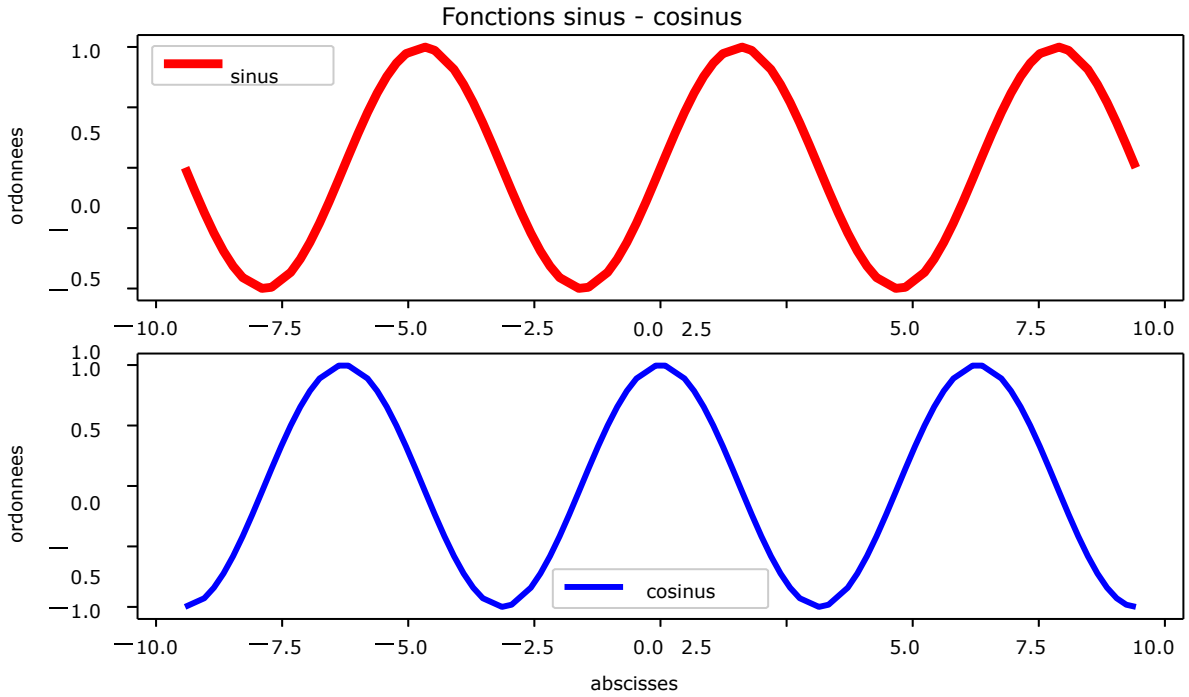

Fonctions sinus - cosinus
ordonnees
abscisses
1.00
0.75
0.50
0.25
0.00
0.25
0.50
1.00
0.75
10.0
7.5
5.0
2.5
0.0
2.5
5.0
7.5
10.0
sinus
cosinus

Exemple 4

On peut a cher plusieurs graphiques dans même gure avec subplot:

```
x=np.linspace(-3*np.pi,3*np.pi,100) y=np.sin(x)
z=np.cos(x)
plt.subplot(2,1,1)
plt.plot(x,y,color='r',linewidth=6)
plt.xlabel('abscisses')
plt.ylabel('ordonnees')
plt.legend(['sinus'])
plt.title('Fonctionssinus-cosinus')
plt.subplot(2,1,2)
plt.plot(x,z,color='b',linewidth=4)
plt.xlabel('abscisses')
 plt.ylabel('ordonnees')
plt.legend(['cosinus'])
plt.savefig('figure4.pdf')
```

Exemple 4


Fonctions sinus - cosinus
sinus
cosinus
ordonnees
ordonnees
abscisses
1.0
0.5
0.0
0.5
1.0
10.0
7.5
5.0
2.5
0.0
2.5
5.0
7.5
10.0

Récapitulatif

Fonction	Rôle
plot(x,y)	trace une courbe en reliant les points se trouvant dans x et y.
xlabel('etiquette')	associe une étiquette à l'axe des abscisses.
ylabel('etiquette')	associe une étiquette à l'axe des ordonnées.
title('titre')	associe un titre à la gure.
legend(['legende1','legende2'])	associe une légende aux coubres.
xlim(a,b)	xe l'intervalle [a,b] pour l'axe des abscisses.
ylim(a,b)	xe l'intervalle [a,b] pour l'axe des ordonnées.
subplot(l,c,n)	divise la figure en l lignes et c colonnes où n est le numéro de chaque graphique compté par ligne.
save(' gure.pdf')	sauvegarde la figure avec le nom donné en pramètre.
show()	Afficher le graphique à l'écran.

Couleur du tracé - option color

- La fonction plot() possède plusieurs options.
- La couleur par défaut est le bleu. Cependant, on peut décrire de nombreuses couleurs. Certaines sont disponibles avec les raccourcis suivants (sous formes de chaînes de caractères).

Mot clef	couleur
'b'	bleu
'g'	vert
'r'	rouge
'c'	cyan
'm'	magenta
'y'	jaune
'k'	noir
'w'	blanc

- Par exemple, pour tracer une courbe en rouge, on écrira la commande suivante: plt.plot(x,y,color = 'r')

Style du trait - option linestyle

- On peut préciser le style du trait avec l'option linestyle:

Mot clef	Style du trait
'-'	ligne continue
'- -'	ligne en tirets
':'	ligne pointillée
'-.'	ligne en tirets et points

- Par exemple, pour tracer une courbe en pointillés, on écrira la commande suivante: plt.plot(x,y,linestyle = ':')

Épaisseur des lignes - option linewidth

- On peut régler l'épaisseur des lignes reliant les points. L'argument donné est un ottant, qui vaut 1 par défaut.

- Par exemple, pour tracer une courbe dont le trait est deux fois plus épais que la normale, on écrira la commande suivante: plt.plot(x,y,linewidth = 2)

Type de point - option marker

On peut changer la manière dont les points sont représentés, avec l'option marker:

Mot clef	Type de point
'.'	point
'*'	étoile
'+'	forme de '+'
'×'	forme de '×'
'o'	cercle
's'	carré

Programmation Orientée Objet avancée

Variables de classe

Il s'agit d'attributs fondamentaux communs à toutes les instances de la classe, contrairement aux attributs d'instance (définis à l'initialisation).

```
class MyClass:
    version = 1.2           # Variable de classe (commun à toutes les instances)
    def __init__(self, x):
        self.x = x          # Attribut d'instance (spécifique à chaque instance)
```

Méthodes statiques

Ce sont des méthodes qui ne travaillent pas sur une instance (le `self` en premier argument). Elles sont définies à l'aide de la fonction **`staticmethod()`** généralement utilisée en décorateur (Un décorateur est une fonction qui modifie le comportement d'autres fonctions) .Elles sont souvent utilisées pour héberger dans le code d'une classe des méthodes génériques qui y sont liées, mais qui pourrait être utilisées indépendamment (p.ex. des outils de vérification ou de conversion).

```
class MyClass:

    def __init__(self, speed):

        self.speed = speed  # [m/s]

    @staticmethod
    def ms_to_kmh(speed):
        "Conversion m/s → km/h."

        return speed * 3.6  # [m/s] → [km/h]
```

Une méthode statique peut être invoquée directement via la classe en dehors de toute instanciation (p.ex. `MyClass.ms_to_kmh()`), ou via une instance (p.ex. `self.ms_to_kmh()`).

Méthodes de classe

Ce sont des méthodes qui ne travaillent pas sur une instance (`self` en premier argument) mais directement sur la classe elle-même (`cls` en premier argument). Elles sont définies à l'aide de la fonction **`classmethod()`** généralement utilisée en décorateur.(Un décorateur est une fonction qui modifie le comportement d'autres fonctions)

Elles sont souvent utilisées pour fournir des méthodes d'instanciation alternatives.

```
class Date:
    "Source: https://stackoverflow.com/questions/12179271"

    def   init  (self, day=0, month=0, year=0):
        """Initialize from day, month and year values (no verification)."""

        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from string(cls, astring):
        """Initialize from (verified) 'day-month-year' string."""

        if cls.is valid date(astring):
            day, month, year= map(int, astring.split('-'))

            return cls(day, month, year)
        else:
            raise IOError(f"{astring!r} is not a valid date string.")

    @staticmethod
    def is valid date(astring):
        """Check validity of 'day-month-year' string."""

        try:
            day, month, year= map(int, astring.split('-'))
        except ValueError:
            return False
        else:
            return (0 < day <= 31) and (0 < month <= 12) and (0 < year <= 2999)
```

Attributs et méthodes privées

Contrairement p.ex. au C++, Python n'offre *pas* de mécanisme de *privatisation* des attributs ou méthodes [2]:

Les attributs/méthodes standards (qui ne commencent pas par `_`) sont publiques, librement accessibles et modifiables (ce qui n'est pas une raison pour faire n'importe quoi):

```
>>> youki = Animal(10.); youki.masse
10.0
>>> youki.masse = -5; youki.masse  # C'est vous qui voyez...
-5.0
```

- Les attributs/méthodes qui commencent par un simple `_` sont *réputées* privées (mais sont en fait parfaitement publiques): une interface est généralement prévue (*setter* et *getter*), même si vous pouvez y accéder directement *à vos risques et périls*.

```python
class AnimalPrive:

    def __init__(self, mass):

        self.set_mass(mass)

    def set_mass(self, mass):
        """Setter de l'attribut privé `mass`."""

        if float(mass) < 0:
            raise ValueError("Mass should be a positive float.")

        self._mass = float(mass)

    def get_mass(self):
        """Getter de l'attribut privé `mass`."""

        return self._mass
```

```
>>> youki = AnimalPrive(10); youki.get_mass()
10.0
>>> youki.set_mass(-5)
ValueError: Mass should be a positive float.
>>> youki._mass = -5; youki.get_mass()  # C'est vous qui voyez...
-5.0
```

Les attributs/méthodes qui commencent par un double __ (*dunder*) sont « cachées » sous un nom complexe mais prévisible (cf. **PEP 8**).

```python
class AnimalTresPrive:

    def __init__(self, mass):

        self.set_mass(mass)

    def set_mass(self, mass):
        """Setter de l'attribut privé `mass`."""

        if float(mass) < 0:
            raise ValueError("Mass should be a positive float.")

        self.__mass = float(mass)

    def get_mass(self):
        """Getter de l'attribut privé `mass`."""

        return self.__mass
```

```
>>> youki = AnimalTresPrive(10);
youki.get_mass()
10.0
>>> youki.  mass = -5; youki.get_mass()  # L'attribut __mass n'existe pas sous ce nom...
10.0
>>> c. AnimalTresPrive  mass = -5;
youki.get_mass()  # ... mais sous un alias compliqué.
-5.0
```

Propriétés

Compte tenu de la nature foncièrement publique des attributs, le mécanisme des *getters* et *setters* n'est pas considéré comme très pythonique. Il est préférable d'utiliser la notion de **`property`** (utilisée en décorateur).

```
class AnimalProperty:

    def __init__(self, mass):

        self.mass = mass          # Appelle le setter de la propriété

    @property
    def mass(self):               # Propriété mass (= getter)

        return self._mass

    @mass.setter
    def mass(self, mass):         # Setter de la propriété mass

        if float(mass) < 0:
            raise ValueError("Mass should be a positive float.")

        self._mass = float(mass)
```

```
>>> youki = AnimalProperty(10); youki.mass
10.0
>>> youki.mass = -5
ValueError: Mass should be a positive floa
>>> youki._mass = -5; youki.mass
-5.0
```

Les propriétés sont également utilisées pour accéder à des quantités calculées à la volée à partir d'attributs intrinsèques.

```
class Interval:

    def   init  (self, minmax):
        """Initialisation à partir d'un 2-tuple."""

        self._range = _, _ = minmax  # Test à la volée

    @property
    def min(self):
        """La propriété min est simplement _range[0]. Elle n'a
pas de setter."""

        return self._range[0]

    @property
    def max(self):
        """La propriété max est simplement _range[1]. Elle n'a
pas de setter."""

        return self._range[1]

    @property
    def middle(self):
        """La propriété middle est calculée à la volée. Elle n'a
pas de setter."""

        return (self.min + self.max) / 2
```

```
>>> i = Interval((0, 10));
i.min, i.middle, i.max
(0, 5, 10)
>>> i.max = 100

    AttributeError: can't
    set attribute
```

Now

Let's Code our Project

Projet : Gestionnaire de To-Do List avec Python

Description du Projet

Dans ce projet, vous allez créer un gestionnaire de To-Do List en utilisant le langage de programmation Python. Ce programme permettra aux utilisateurs d'ajouter des tâches, de les marquer comme terminées, d'afficher les tâches actuelles et de générer des statistiques.

Étapes du Projet

Étape 1: Mise en Place des Modules

- **Importez les Bibliothèques Nécessaires :**
- Utilisez les bibliothèques pickle, os, datetime, numpy et matplotlib.pyplot pour gérer les fichiers, les dates, les tableaux et les graphiques.

Étape 2: Les Fonctions en Python

- **Fonction de Filtrage par État :**
- Créez une fonction lambda pour filtrer les tâches par état.
- **Sauvegarde et Chargement de Tâches :**
- Définissez des fonctions pour sauvegarder et charger les tâches depuis un fichier.

Étape 3: Les Tableaux en NumPy

- **Calcul des Statistiques des Tâches :**
- Créez une fonction utilisant NumPy pour calculer le nombre total de tâches, le nombre de tâches terminées, et le pourcentage de tâches terminées.

Étape 4: Les Graphiques avec Matplotlib

- **Affichage des Statistiques sous Forme de Graphique :**
- Utilisez Matplotlib pour créer une fonction qui affiche les statistiques sous forme de graphique.

Étape 5: Programmation Orientée Objet Avancée

- **Implémentation de la Classe ToDoList :**
  - Ajoutez une classe ToDoList avec les méthodes suivantes :
  - ajouter_tache(self, description) : Ajoute une nouvelle tâche à la liste.
  - afficher_taches(self) : Affiche toutes les tâches de l'utilisateur.
  - marquer_comme_terminee(self, indice_tache) : Marque une tâche comme terminée.
  - filtrer_taches_terminees(self) : Retourne la liste des tâches terminées.

Testez vos Méthodes !

Conclusion

BE PROUD!

You Succeeded in finishing the course