Python For Finance

M1 - Economie Finance

Université Paris Dauphine

October 2025

Outline

Utilisation de Python

- Directement depuis le CMD.exe (après avoir installé Python) :
- Avec un IDE (Integrated Development Environment)
  - PyCharm
  - Spyder
  - VScode
  - ...
- Avec Anaconda. Anaconda est une distribution de python et R et non un IDE. Il propose une structure de gestion des packages et des environnements.
- Avec Jupyter Notebook (ou JupyLab). Des applications web permettant de développer en Python, Julia, R, Ruby, Scala... Permet de coder dans des Notebooks : Ceux-ci combinent des passages de code et des passages de Markdown.
  NB : Jupyter utilise le noyau IPython qui permet d'autres syntaxes, de l'auto-complétion de code, des "magics commands" ...

Tout est objet (1/2)

En Python tout est **objet**, les objets, les fonctions, les modules, les méthodes. La dimension objet de python est sa structure de base c'est ainsi que le language doit être abordé.
Dès qu'une variable est créée, elle référence un objet, que ce soit un nombre entier (int), un nombre flottant (float), un tableau (list) ou toute autre structure.

Tout est objet (1/2)

En Python tout est **objet**, les objets, les fonctions, les modules, les méthodes. La dimension objet de python est sa structure de base c'est ainsi que le language doit être abordé.
Dès qu'une variable est créée, elle référence un objet, que ce soit un nombre entier (int), un nombre flottant (float), un tableau (list) ou toute autre structure.

Définition

En informatique, un objet est une modélisation d'un élément du monde réel. Il s'agit en fait de données et de fonctions qui agissent sur ces données réunies dans une même entité. Ces données sont appelées les **arguments** de l'objet. Les fonctions sont les **méthodes**. On peut se représenter chaque objet comme une capsule qui regroupe des attributs et des méthodes.

objet = attributs + méthodes

Tout est objet (2/2)

Chaque objet appartient au moins à une **classe**. Une classe permet de créer des objets. C'est une famille d'objets équivalents qui représentent un type de données. Un objet, aussi appelé une **instance**, est un exemplaire particulier d'une classe.

Tout est objet (2/2)

Chaque objet appartient au moins à une **classe**. Une classe permet de créer des objets. C'est une famille d'objets équivalents qui représentent un type de données. Un objet, aussi appelé une **instance**, est un exemplaire particulier d'une classe.

Exemple

Une classe peut être la classe "Voiture", quelque-uns de ses **attributs** seront :

- Roues
- Cylindrée
- Couleur

Quelque-unes de ses **méthodes** seront :

- Rouler
- Démarrer
- Mettre le clignotant

Une **instance** de cette classe sera : une unique voiture avec ses caractéristiques.

Les mots-clés (1/2)

Définition

Les **mots-clés** sont des mots prédéfinis et réservés. Ils ont une signification particulière et immuable pour l'interpréteur.

Il existe 2 types de mots-clés :

- **Les mots clés réservés** : vous n'avez pas le droit d'utiliser un tel mot clé pour définir un identifiant (nom de variable, de fonction, de classes, d'attribut de méthode...). Sa seule utilisation autorisée est l'instruction à laquelle il est rattaché.
- **Les mots clés contextuels** : un tel mot clé prend un sens dans un contexte bien particulier. En dehors de ce contexte, le mot reste utilisable pour nommer une variable, une fonction... Cette possibilité est assez récente et s'explique par le fait qu'il est nécessaire de faire évoluer le langage (et notamment d'y ajouter de nouvelles instructions) sans produire de rupture de compatibilité avec les programmes écrits avant l'apparition de la nouvelle fonctionnalité.

Les mots-clés (2/2)

Pour obtenir la liste des mots-clés **réservés** :

```
help("keywords")
#help(any_key_word) - permet d'obtenir de l'aide sur un mot-clé
```

```
Entrée [4]: help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Il y a 35 mots-clés réservés en python 3.10 et uniquement 2 mots clés contextuels apparus avec Python 3.10 (**match** et **case**).

Outline

Téléchargement

Télécharger **Anaconda** :

- Cliquer ici

Télécharger **Pycharm** (version gratuite) :

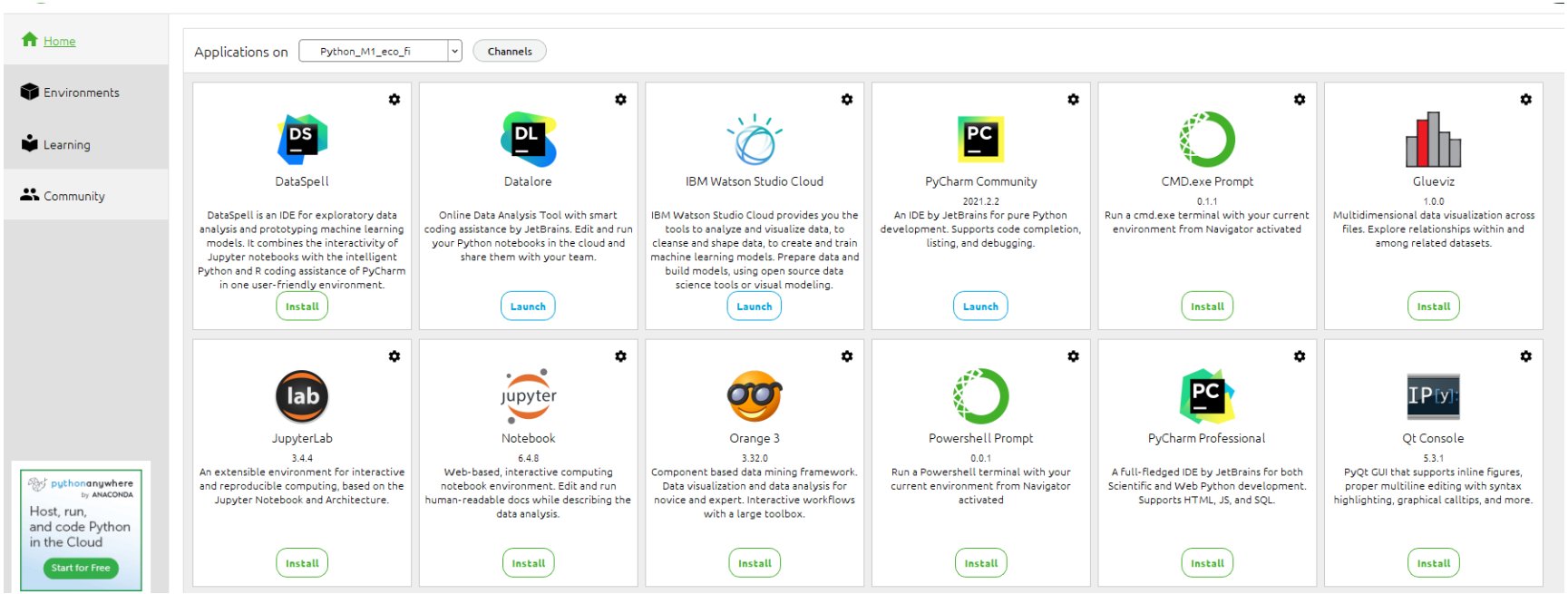
- Cliquer ici

Optionnel : Télécharger un **noyau python** (Version 3.8 et +) :

- Cliquer ici

Anaconda

Ouvrir d'Anaconda Navigator et installer Jupyter Notebook, Spyder et CMD.exe Prompt :



Jupyter

Créer un premier Notebook :

Outline

Vue d'ensemble des différents types de variables

Les variables constantes

En python nous retrouvons les types basiques des autres langages tels que les entiers **(int)**, les decimaux **(float)**, les chaînes de caractères **(str)** et les booléens **(bool)**.

Vue d'ensemble des différents types de variables

Les variables constantes

En python nous retrouvons les types basiques des autres langages tels que les entiers **(int)**, les decimaux **(float)**, les chaînes de caractères **(str)** et les booléens **(bool)**.

Les "Built-in" variables

En python nous retrouvons aussi les types plus complexes tels que les listes **(list)**, les tuples **(tuple)**, les dictionnaires **(dict)** et les ensembles **(set)**.

Outline

String

Le type String est noté **str** en Python.
Il permet de représenter des lettres et des mots, il n'existe pas en python de différenciation des lettres et des chaînes de caractères comme dans certains autres langages.

```
my_str = "Dauphine"
print(my_str)
==> Dauphine

my_str2 = 'dauphine'
print(my_str2)
==> dauphine

my_str == my_str2
==> False
```

String

Le type String est noté **str** en Python.
Il permet de représenter des lettres et des mots, il n'existe pas en python de différenciation des lettres et des chaînes de caractères comme dans certains autres langages.

```
my_str = "Dauphine"
print(my_str)
==> Dauphine

my_str2 = 'dauphine'
print(my_str2)
==> dauphine

my_str == my_str2
==> False
```

Caractéristique

- Les strings se déclarent avec des simples ou doubles quotes (' ou ").
- Elles sont sensibles à la casse.
- Les espaces sont des caractères.

Méthode de la classe string et fonctions utiles

```
my_str = "Université Paris Dauphine"
print(len(my_str))
==> ?
print(my_str.find("i"))
==> ?
print(my_str.capitalize(),my_str.upper(),my_str.lower())
==> ?
print(my_str.count("i"))
==> ?
print(my_str.replace("Dauphine","9"))
==> ?
print(my_str.isdigit(),my_str.isalpha())
==> ?
print(my_str.split())
==> ?
print(my_str[slice(11,-9)])
==> ?
```

Question

Quels seront les outputs de ces méthodes de la classe str ?

Méthode de la classe string et fonctions utiles

```
my_str = "Université Paris Dauphine"
print(len(my_str))
==> 25
print(my_str.find("i"))
==> 2
print(my_str.capitalize(),my_str.upper(),my_str.lower())
Université Paris Dauphine UNIVERSITÉ PARIS DAUPHINE université paris dauphine
print(my_str.count("i"))
==> 4
print(my_str.replace("Dauphine","9"))
==> Université Paris 9
print(my_str.isdigit(),my_str.isalpha())
==> False False
print(my_str.split())
==> ['Université','Paris','Dauphine']
print(my_str[slice(11,-9)])
==> Paris
```

Nombres entiers et décimaux

Les nombres peuvent être de type entiers ("Integer" noté **int**) ou décimaux ("Float" noté **float**).

```
my_int = 7
my_int2 = int(7.7)
my_float = 7.7
my_float2 = float(7)
print(my_int,my_int2,my_float,my_float2)
==> 7 7 7.7 7.0

type(my_int + my_float)
==> <class 'float'>

print("7.3+7.7 =",7.3+7.7, "de type :",type(7.7+7.3))
==> 7.3+7.7 = 15.0 de type : <class 'float'>
```

Nombres entiers et décimaux

Les nombres peuvent être de type entiers ("Integer" noté **int**) ou décimaux ("Float" noté **float**).

```
my_int = 7
my_int2 = int(7.7)
my_float = 7.7
my_float2 = float(7)
print(my_int,my_int2,my_float,my_float2)
==> 7 7 7.7 7.0

type(my_int + my_float)
==> <class 'float'>

print("7.3+7.7 =",7.3+7.7, "de type :",type(7.7+7.3))
==> 7.3+7.7 = 15.0 de type : <class 'float'>
```

NB : L'application de la fonction int() sur un float supprime toute partie décimale de ce dernier. Ce n'est pas un arrondi, une partie entière ou autre méthode de discrétisation d'une variable continue en variable discrète.

Manipulation de nombres

La manipulation des nombres en python est possible avec toutes les opérations usuelles. Il existe aussi des *built-in function* et des fonctions du module "math" (module natif) très utiles.

Manipulation de nombres

La manipulation des nombres en python est possible avec toutes les opérations usuelles. Il existe aussi des *built-in function* et des fonctions du module "math" (module natif) très utiles.

Built-in functions :

```
pow(base = 7, exp = 3)
round(7.8)
min(1,5,2.8)
max(1,5,2.8)
abs(-8)
```

Quelques fonctions du module math :

```
import math
print(math.ceil(7.7),math.ceil(-7.7))
==> 8 -7
print(math.floor(7.7),math.floor(-7.7))
==> 7 -8
math.sqrt(64)
==> 8
```

NB : Le module math possède une grande quantité de fonctions, vous pouvez les retrouver ici .

Outline

Listes

Une liste est un objet de type container, les containers contiennent d'autres objets de tout type et ils sont ordonnés.
Déclaration d'une liste et affichage de cette dernière :

```
my_list = list()
my_list = []
print(my_list)
```

Caractéristiques

Une liste est :

- **Ordonnée/indexée**
- **Mutable** (donc copie par référence, non-hashable)
- **Accessible**
- **Imbriquable**
- **Extensible**
- **Non-Hashable (car mutable)**

Manipuler des listes

Les listes étant ordonées on les manipulent essentiellement par leurs indices mais on peut aussi les fusionner.

```
my_stocks = ['AMZN','TSLA','AAPL','META']
print(my_stocks[1],my_stocks[1:3],my_stocks[-2:],my_stocks[::2])
==> TSLA ['TSLA', 'AAPL'] ['AAPL', 'META'] ['AMZN', 'AAPL']

my_stocks2 = ['KO','P911']
my_stocks2 = my_stocks2 + my_stocks
print(my_stocks2)
==> ['KO', 'P911', 'AMZN', 'TSLA', 'AAPL', 'META']

copy_stocks = my_stocks
copy_stocks.pop()
print(copy_stocks,my_stocks)
['AMZN', 'TSLA', 'AAPL'] ['AMZN', 'TSLA', 'AAPL']
```

On remarque le caractère **non-hashable** des listes (car elles sont **mutables**). Ici la copie de la liste a été faite en **référence** et non en **valeur**.

Listes : Méthodes et fonctions

```
my_stocks = ['AMZN','TSLA','AAPL','META']
print(len(my_stocks))
==> ?
print(my_stocks.append('MSFT'),my_stocks)
==> ? ?
print(my_stocks.insert(1,'FTKE'),my_stocks)
==> ? ?
print(my_stocks.count('MSFT'))
==> ?
print(my_stocks.pop(),my_stocks)
==> ? ?
print(my_stocks.remove('TSLA'),my_stocks)
==> ? ?
print(my_stocks.reverse(),my_stocks)
==> ? ?
print(my_stocks.extend(['MC','TTE']),my_stocks)
==> ? ?
print(my_stocks.sort(reverse=True), my_stocks)
==> ? ?
Print(my_stocks.index('MC'))
==> ?
```

Listes : Méthodes et fonctions

```
my_stocks = ['AMZN','TSLA','AAPL','META']
print(len(my_stocks))
==> 4
print(my_stocks.append('MSFT'),my_stocks)
==> ['AMZN','TSLA','AAPL','META','MSFT']
print(my_stocks.insert(1,'FTKE'),my_stocks)
==> None ['AMZN','FTKE','TSLA','AAPL','META','MSFT']
print(my_stocks.count('MSFT'))
==> 1
print(my_stocks.pop(),my_stocks)
==> 'MSFT' ['AMZN','FTKE','TSLA','AAPL','META']
print(my_stocks.remove('TSLA'),my_stocks)
==> None ['AMZN','FTKE','AAPL','META']
print(my_stocks.reverse(),my_stocks)
==> None ['META','AAPL','FTKE','AMZN']
print(my_stocks.extend(['MC','TTE']),my_stocks)
==> None ['META','AAPL','FTKE','AMZN','MC','TTE']
print(my_stocks.sort(reverse=True), my_stocks)
==> None ['TTE', 'META', 'MC', 'FTKE', 'AMZN', 'AAPL']
Print(my_stocks.index('MC'))
==> 2
```

Listes : Approfondissement sur le tri

Il est possible d'effectuer des tris sur des conditions autres que par ordre alphabétique (resp anti-alphabétique) ou croissant (resp décroissant). Ceci s'effectue grâce à l'argument '**key**'. Il est même possible de donner plusieurs critères de tris successifs.

```
my_Stocks = ['AMZN', 'TTE', 'META', 'AAPL', 'FTKE']
my_Stocks.sort(key = lambda x : len(x), reverse = False)
print(my_Stocks)
==> ?

my_Stocks.sort(key = lambda x : (len(x),x), reverse = False)
print(my_Stocks)
==> ?
```

Commentaires

Pour cela il suffit de renseigner une fonction pour l'argument **key** de la méthode **sort** de la classe liste. Il est conseillé d'utiliser une fonction lambda si cette fonction n'a pas vocation à être utilisée ailleurs.
NB : Cette méthode est naturellement utilisable sur des listes de nombres.

Listes : Approfondissement sur le tri

```
my_Stocks = ['AMZN', 'TTE', 'META', 'AAPL', 'FTKE']
my_Stocks.sort(key = lambda x : len(x), reverse = False)
print(my_Stocks)
==> ['TTE', 'AMZN', 'META', 'AAPL', 'FTKE']
```

Tri simple

Ici la fonction lambda désigne la longueur de la chaîne de caractère comme critère de tri.

```
my_Stocks.sort(key = lambda x : (len(x),x), reverse = False)
print(my_Stocks)
==> ['TTE', 'AAPL', 'AMZN', 'FTKE', 'META']
```

Multi-tri

Ici un second critère de tri est utilisé. Attention, Il est dominé par le premier critère. Le code tri d'abord par taille, puis par ordre alphabétique une fois le tri par taille réalisé.

Tuples

Un tuple est un objet de type **container**, les containers contiennent d'autres objets de tous types et ils sont ordonnés.
Déclaration d'un tuple et affichage de ce dernier :

```
my_tuple = tuple()
my_tuple = ()
print(my_tuple)
```

Caractéristiques

Un tuple est :

- **Ordonné/indexé**
- **Non-Mutable**
- **Accessible**
- **Imbriquable**
- **Extensible**
- **Hashable** (car non-mutable)

Manipuler des tuples

```
my_tuple = ('TTE', 'AAPL', 'AMZN', 'FTKE', 'META')
extra_tickers = ('KO','P911')
print(my_tuple[1],my_tuple[1:3],my_tuple[:-1])
==> AAPL ('AAPL', 'AMZN') ('TTE', 'AAPL', 'AMZN', 'FTKE')

print(hash(extra_tickers))
==> -6671021492883455079 # valeur variable

extra_tickers = extra_tickers + my_tuple
print(extra_tickers)
==> ('KO', 'P911', 'TTE', 'AAPL', 'AMZN', 'FTKE', 'META')
print(hash(extra_tickers))
==> --7572457658930527OO # changement du hash de la variable
```

Un tuple étant **non-mutable** (donc **hashable**), extra_tickers n'est plus le même objet après qu'il ait été "modifié", en réalité il a été **ré-instancié**. On le remarque facilement en regardent le **hash** de l'objet. Si le hash d'un objet change entre 2 opérations alors l'objet n'est plus le même, c'est une nouvelle instance.

Tuples : méthodes et fonctions

Les tuples sont une classe d'objets bien plus simples que les listes. Les tuples sont moins flexibles et comportent beaucoup moins de méthodes.

```
my_tuple = ('TTE', 'AAPL', 'AMZN', 'FTKE', 'META')
print(my_tuple.count('TTE'),my_tuple)
==> 1 ('TTE', 'AAPL', 'AMZN', 'FTKE', 'META')

print(my_tuple.index('AMZN'),my_tuple)
==> 2 ('TTE', 'AAPL', 'AMZN', 'FTKE', 'META')
```

Les fonctions usuelles telles que

- len()
- max()
- min()
- ...

fonctionnent sur les tuples.

Exemple

Exemples :

Listes et tuples

Dictionnaires

Un dictionnaire est une collection d'objet non-ordonée. Chaque élement d'un dictionnaire est un couple clé **unique**/valeur. Il peut prendre tous les objets en valeur mais seuls les objets hashables peuvent être passés en clé. Déclaration d'un dictionnaire et affichage de ce dernier :

```
my_dict = dict([('key1', V1), ('key2', V2), ('key3', V3)])
my_dict = {"key1":"Value1","key2":"Value2","key3":"Value3"}
print(my_dict)
```

Caractéristiques

Un dictionnaire est :

- **Non-ordonné**
- **Mutable** (donc copie par référence, non-hashable)
- **Accessible (clés)**
- **Imbriquable (en valeurs uniquement)**
- **Extensible**
- **Non-hashable**

Dictionnaires : méthodes et fonctions

```
quotes = {'AMZN':133.62,'TSLA':889.36,'AAPL':167.23}

print(my_dict["TSLA"], my_dict.get("TSLA"))
==> 889.36 889.36

print(my_dict.values())
==> dict_values([133.62, 889.36, 167.23])

print(my_dict.keys())
==> dict_keys(['AMZN', 'TSLA', 'AAPL'])

print(quotes.items())
==> dict_items([('AMZN', 133.62), ('TSLA', 889.36), ('AAPL', 167.23)])
```

Les 4 méthodes ci-dessus sont les moyens d'entrée dans un dictionnaire. On les manipule via ses clés (objet itérable sur ses clés), on peut le parcourir via les méthodes **.keys()**, **.values()** et **.items()**.

Dictionnaires : Suppression d'éléments

```
quotes = {'AMZN':133.62,'TSLA':889.36,'AAPL':167.23}

print(quotes.pop("AAPL","Stocks not found"), quotes)
==> AAPL  {'AMZN': 133.62, 'TSLA': 889.36}

print(quotes.pop("AAPL","Stocks not found"), quotes)
==> Stocks not found  {'AMZN': 133.62, 'TSLA': 889.36}

quotes["TTE"] = 52.60

print(quotes.popitem(), quotes)
==> ('TTE', 52.6) {'AMZN': 133.62, 'TSLA': 889.36}
```

Pour supprimer des éléments d'un dictionnaire la méthode **.pop()** permet de choisir la clé de l'élément à retirer et la valeur à retourner en cas d'échec. La méthode **.popitem()** retire le dernier élément ajouté au dictionnaire.

Dictionnaires : MaJ et fusions

```
stocks, prices = ['AMZN', 'TSLA', 'AAPL'], [133.62, 889.36, 167.23]
quotes = dict(zip(stocks,prices)) # tickers prix
quotes_bis = {"KO":54.51} # tickers et prix supplémentaires

all_quotes = dict(**quotes,**quotes_bis) # Python 3.5 et +
#or
all_quotes = quotes | quotes_bis # | est l'opérateur logique "ou"
print(all_quotes)
==> {'AMZN':133.62, 'TSLA':889.36, 'AAPL':167.23, "KO":54.51}

quotes_bis.update(quotes)
print(quotes_bis.update(quotes))
==> {'AMZN':133.62, 'TSLA':889.36, 'AAPL':167.23, "KO":54.51}
```

La méthode **.update()** permet de mettre à jour un dictionnaire de nouveaux couples clé/valeur mais aussi de mettre à jour la valeur de clés existantes. L'argument de la méthode est le dictionnaire prioritaire en cas de doublons.

Set

Un ensemble est une collection d'éléments non ordonnée, non indexée, sans doublons, mutable d'éléments non-mutables.
Déclaration d'un set et affichage de ce dernier :

```
my_set = set(['a', 'b', 'c', 'd', 'e'])
my_set = {'a', 'b', 'c', 'd', 'e'}
print(my_set)
```

Caractéristiques

Un Set est :

- **Non-ordonné**
- **Mutable** (donc copie par référence, non-hashable)
- **Accessible**
- **Non-imbriquable**
- **Extensible**

Set : méthodes et fonctions

```
my_set ={'KO', 'P911', 'FTKE', 'META'}
my_set2 = {'KO', 'MSFT', 'MC'}
print(my_set.add("F"), my_set)
==> None {'F', 'META', 'KO', 'P911', 'FTKE'}

print(my_set.add("F"),my_set)
==> None {'F', 'META', 'KO', 'P911', 'FTKE'}

print(my_set.add(("MC.PA","MC.VI")),my_set)
==> {('MC.PA','MC.VI'), 'F', 'META', 'KO', 'P911', 'FTKE'}
```

Les sets ne peuvent pas présenter de doublons, ils peuvent comporter des objets de types différents mais uniquement des objets hashables (immuables).

NB : Ils peuvent se montrer très utiles pour les sujets sensibles aux doublons et pour les comparaisons de dictionnaires par exemple.

Set : opérateurs logiques mathématiques

Les set permettent d'exécuter toutes les opérations usuelles sur les ensembles (intersection, union, inclusion ...).
Ci-dessous plusieurs exemples :

```
my_set2 & my_set
# or
my_set2.intersection(my_set)

my_set2 | my_set2
# or
my_set2.union(my_set)

my_set.difference(my_set2) :
my_set.isdisjoint(my_set2) :
my_set2.issubset(my_set) :
my_set.issuperset(my_set2) :
my_set2 = {'KO', 'MSFT', 'MC'}
```

Outline

Boucles for

Exemple de boucles for :

```
my_stocks = ['AMZN','TSLA','AAPL','META']
for stock in my_stocks:
    print (stock)

for i in range(1,10):
    print (i)

my_prices = [133.62,889.36,167.23,161.11]
my_ptf = dict(zip(my_stocks,my_prices))
for stock,price in my_ptf.items():
    print (f" Le prix d\'{stock} est {price}")
```

Question

Quel sera l'output de ces 3 boucles ?

Outline

Boucles while

Exemple de boucle While :

```
portfolio1 = ['AMZN','TSLA','AAPL','META']
portfolio2 = ['V','TTE','FTKE','F']
while True :
    portfolio1.append(portfolio2[-1])
    portfolio2.pop()
    if not bool(portfolio2):
        break
print(portfolio1)
```

Boucles while

Exemple de boucle While :

```
portfolio1 = ['AMZN','TSLA','AAPL','META']
portfolio2 = ['V','TTE','FTKE','F']
while True :
    portfolio1.append(portfolio2[-1])
    portfolio2.pop()
    if not bool(portfolio2):
        break
print(portfolio1)
```

Question

*Pourquoi le code commence par **"while True :"** ?*
Quel sera l'output ?

Boucles while

Variante plus efficiente :

```
portfolio1 = ['AMZN','TSLA','AAPL','META']
portfolio2 = ['V','TTE','FTKE','F']
while True :
    portfolio1.append(portfolio2.pop())
    if not bool(portfolio2):
        break
print(portfolio1)
```

Question

Pourquoi cette variante fonctionne-t-elle ?

Pourquoi est-elle plus efficiente/parcimonieuse ?

Outline

Fonctions

Une fonction est une suite d'instructions que l'on peut appeler depuis l'extérieur et l'intérieur (récursivité - voir exercice factorielle TD) de la fonction.
Déclaration d'une fonction et appel de cette dernière :

```
def addition_function(a : float ,b : float = 2 ) -> float:
    return a+b

print(addition_function (3,4))
==> 7
print(addition_function (3))
==> 5
```

Commentaires

Il est impératif **d'indenter** au début de la fonction.
Une bonne pratique est de préciser le type de l'input et de l'output.
Il est possible de passer un argument par défaut s'il n'est pas stipulé par l'utilisateur.

Fonctions Lambda

En cas d'un besoin ponctuel de fonction il est possible de déclarer une fonction qui ne restera pas en mémoire et ne sera plus disponible à l'appel dans la suite du code.
On appelle ce genre de fonction les **lambda functions**.
Déclaration de la fonction addition avec une fonction lambda :

```
print((lambda x,y: x+y)(1,3))
==> 4
```

Commentaires

Ce type de fonction permet d'optimiser l'utilisation de la mémoire.
Les fonctions lambda sont particulièrement utiles lorsque l'on souhaite appliquer une fonction à toute une liste ou à un dataframe (voir section pandas).

Outline

If

If sert à vérifier une ou plusieurs conditions avant d'exécuter des commandes :

```
my_stocks = ['AMZN','TSLA','AAPL','META']
stocks2 = ['V','TTE','AAPL','MS','AMZN']
for stocks in stocks2 :
    if stocks in my_stocks:
        print(f'{stocks} is all ready in the first portfolio')
    else :
        my_stocks.append(stocks)
print(my_stocks)
```

Question

Quel sera l'output de ces instructions ?

Outline

Créer une première classe

Définitions

"Les classes sont un moyen de réunir des données et des fonctionnalités. Créer une nouvelle classe crée un nouveau type d'objet et ainsi de nouvelles instances de ce type peuvent être construites. Chaque instance peut avoir ses propres attributs, ce qui définit son état." - Python.org - classes

Création d'une classe simple avec une variable/attribut de classe "marque" :

```
class Voiture :
""" Un exemple de classe """
  marque = "Renault"

my_car = Voiture()
print(a)
```

Créer une première classe

Définitions

"Les classes sont un moyen de réunir des données et des fonctionnalités. Créer une nouvelle classe crée un nouveau type d'objet et ainsi de nouvelles instances de ce type peuvent être construites. Chaque instance peut avoir ses propres attributs, ce qui définit son état." - Python.org - classes

Création d'une classe simple avec une variable/attribut de classe "marque" :

```
class Voiture :
""" Un exemple de classe """
  marque = "Renault"

my_car = Voiture()
print(a)
== > <__main__.Voiture object at 0x0000011940965730>
print(a.marque)
==> Renault
```

Init

Il serait plus utile de créer une classe permettant de choisir nous même le modèle de la voiture. Pour cela on utilise "init()" (built-in function) une fonction systématiquement exécutée lorsqu'un objet est instancié.

```
class Voiture :
  def __init__(self, marque : str, couleur : str):
    self.marque = marque
    self.couleur = couleur

my_car = Voiture()
```

Init

Il serait plus utile de créer une classe permettant de choisir nous même le modèle de la voiture. Pour cela on utilise "init()" (built-in function) une fonction systématiquement exécutée lorsqu'un objet est instancié.

```
class Voiture :
  def __init__(self, marque : str, couleur : str):
    self.marque = marque
    self.couleur = couleur

my_car = Voiture()
==> TypeError: __init__() missing 2 required positional arguments

my_car = Voiture("Renault","noire")
print(my_car.marque, my_car.couleur)
```

Init

Il serait plus utile de créer une classe permettant de choisir nous même le modèle de la voiture. Pour cela on utilise "init()" (built-in function) une fonction systématiquement exécutée lorsqu'un objet est instancié.

```
class Voiture :
  def __init__(self, marque : str, couleur : str):
    self.marque = marque
    self.couleur = couleur

my_car = Voiture()
==> TypeError: __init__() missing 2 required positional arguments

my_car = Voiture("Renault","noire")
print(my_car.marque, my_car.couleur)
==> Renault noire
```

On peut désormais instancier plusieurs objets Voiture de marque différente et de couleur différente avec la même classe.

Méthodes de classe (1/3)

Les classes sont dotés d'attributs (ici marque et couleur) mais peuvent aussi être dotées de méthodes (fonction propre à une classe).

```
class Voiture :
    def __init__(self, marque : str, couleur : str):
        self.marque = marque
        self.couleur = couleur

    def repaint(self, color : str):
        self.couleur = color

my_car = Voiture("Renault","noire")
my_car.repaint("vert")
print(my_car.couleur)
```

Méthodes de classe (1/3)

Les classes sont dotés d'attributs (ici marque et couleur) mais peuvent aussi être dotées de méthodes (fonction propre à une classe).

```
class Voiture :
    def __init__(self, marque : str, couleur : str):
        self.marque = marque
        self.couleur = couleur

    def repaint(self, color : str):
        self.couleur = color

my_car = Voiture("Renault","noire")
my_car.repaint("vert")
print(my_car.couleur)
```

```
==> vert
```

Méthodes de classe (1/3)

Les classes sont dotés d'attributs (ici marque et couleur) mais peuvent aussi être dotées de méthodes (fonction propre à une classe).

```
class Voiture :
    def __init__(self, marque : str, couleur : str):
        self.marque = marque
        self.couleur = couleur

    def repaint(self, color : str):
        self.couleur = color

my_car = Voiture("Renault","noire")
my_car.repaint("vert")
print(my_car.couleur)
```

```
==> vert
```

Ici repaint() est une méthode avec un argument : "color".
NB : Le "self" en premier argument de la méthode n'est pas un argument à renseigner par l'utilisateur. Le self sert à préciser que cette méthode s'applique à l'instance.

Méthodes de classe (3/3)

```
class Voiture :
    def __init__(self, marque : str, couleur : str):
        self.marque = marque
        self.couleur = couleur

    def open_door(self):
        return "Door opened"

my_car = Voiture("Renault","noire")
method_open = my_car.open_door
print(method_open())
==> Door opened

print(type(method_open), type(method_open())
```

Méthodes de classe (3/3)

```
class Voiture :
    def __init__(self, marque : str, couleur : str):
        self.marque = marque
        self.couleur = couleur

    def open_door(self):
        return "Door opened"

my_car = Voiture("Renault","noire")
method_open = my_car.open_door
print(method_open())
==> Door opened

print(type(method_open), type(method_open())
==> <class'method'> <class'str'>
```

On remarque que la méthode de classe doit être munie des "()" pour être appelée, elle peut être instanciée dans une autre variable et exécutée à tout moment et autant de fois que nécessaire par la suite.

Attribut de classe VS attribut d'instance (1/3)

L'utilisation d'attributs de classe à la place d'attributs d'instance est dangereuse, en effet une modification d'un attribut de classe peut se propager à toutes les instances de la classe sous certaines conditions !

```
class Voiture :
    puissance = 100 # partagé par toutes les instances de la classe
    def __init__(self, marque : str, couleur : str):
        self.marque = marque # propres à chaques instances
        self.couleur = couleur

    def add_power(self, extra_power : int):
        self.puissance += extra_power

my_black_car = Voiture("Renault","noire")
my_red_car = Voiture("Renault","rouge")
my_black_car.add_power(50)
print(my_black_car.puissance,my_red_car.puissance)

==> 150 100
```

Attribut de classe VS attribut d'instance (2/3)

Ci-dessous le même type d'exemple qu'à la slide précédente mais l'attribut de classe est une liste :

```
class Voiture :
    broken_parts = [] # partagé par toutes les instances de la classe
    def __init__(self, marque : str, couleur : str):
        self.marque = marque # propres à chaques instances
        self.couleur = couleur

    def append_broken_parts(self, new_broken_part : str):
        self.broken_parts.append(new_broken_part)

my_black_car = Voiture("Renault","noire")
my_red_car = Voiture("Renault","rouge")
my_black_car.append_broken_parts("breaks")
print(my_black_car.broken_parts,my_red_car.broken_parts)
```

Attribut de classe VS attribut d'instance (2/3)

Ci-dessous le même type d'exemple qu'à la slide précédente mais l'attribut de classe est une liste :

```
class Voiture :
    broken_parts = [] # partagé par toutes les instances de la classe
    def __init__(self, marque : str, couleur : str):
        self.marque = marque # propres à chaques instances
        self.couleur = couleur

    def append_broken_parts(self, new_broken_part : str):
        self.broken_parts.append(new_broken_part)

my_black_car = Voiture("Renault","noire")
my_red_car = Voiture("Renault","rouge")
my_black_car.append_broken_parts("breaks")
print(my_black_car.broken_parts,my_red_car.broken_parts)

==> ['breaks'] ['breaks']
```

Que remarque-t-on ?

Attribut de classe VS attribut d'instance (3/3)

Explications

Lorsqu'un attribut de classe est un objet de type "non-hashable" alors une modification de ce dernier se propage à toutes les instances existantes de cette classe. En effet, un objet non hashable est copié en référence, une modification de l'un entraîne modification de tous.

Outline

Pandas

Pandas est un package python pour la manipulation et l'analyse de données. Il organise les données dans des objets appelés *DataFrame*. Le package propose une multitude d'outils destinés à manipuler les données.

Installation de pandas :

```
# à ecrire dans la console python
pip install pandas

# import du package (à mettre au début du script python)
import pandas as pd
```

Pandas

Pandas est un package python pour la manipulation et l'analyse de données. Il organise les données dans des objets appelés *DataFrame*. Le package propose une multitude d'outils destinés à manipuler les données.

Installation de pandas :

```
# à ecrire dans la console python
pip install pandas

# import du package (à mettre au début du script python)
import pandas as pd
```

Pandas est un package utilisé en conjonction avec **numpy** et **matplotlib** qui permet de *visualiser*, *nettoyer*, *transformer* et *analyser* des bases de données.

Les Series et les DataFrames

Il existe 2 types d'objets principaux au sein du package pandas :

- **Les Series** : Structure de données unidimensionnelle hétérogène indexées en lignes (base 0).
- **Les DataFrame** : Structure de données multidimensionnelle hétérogène indexées en lignes et en colonnes (base 0).

Déclaration d'une pd.Séries pandas :

```
data = range(1,101)
pd.Series(data = data)
# ou
data = {'a': 1, 'b': 2, 'c': 3}
pd.Series(data = data)
# ou
from numpy.random import randint
data= randint(1,7,10000)
pd.Series(data=data)
```

A partir d'un dictionnaire

Création d'un premier DataFrame :

```
data = {'stocks':['AMZN','TSLA','AAPL','Meta'],
'prices':[133.62,889.36,167.23,161.11]}

df = pd.DataFrame(data)

df
```

Output :

	stocks	prices
0	AMZN	133.62
1	TSLA	889.36
2	AAPL	167.23
3	META	161.11

A partir d'un CSV, Excel ...

Import de données stockées sous d'autres formats :

```
df = pd.read_csv(my_csv.csv, sep = ',', header = True)
# argument possibles : use_cols, index_col, squeeze (to get a pd.series)
# prefix, na_values, true_values, false_values etc
df = pd.read_excel('my_csv.xlsx')
```

Output :

	stocks	prices
0	AMZN	133.62
1	TSLA	889.36
2	AAPL	167.23
3	META	161.11

Outline

Ajout d'une colonne

Ajout de la colonne market cap

```
df = df.assign(market_cap=[1363, 2692, 931])
# or
df.insert(2,'market_cap_bis',[1363, 2692, 931])

df
```

Output :

stocks	prices	market_cap	market_cap_bis
AMZN	133.62	1363	1363
TSLA	889.36	2692	2692
AAPL	167.23	931	931

Ajout d'une colonne

Ajout de la colonne quantité d'actions via une fonction lambda :

```
df = df.assign(number_of_share=lambda x : x.market_cap*10**9/x.prices)
```

Output :

stocks	prices	market_cap	market_cap_bis	number_of_share
AMZN	133.62	1363	1363	1.020057e+10
TSLA	889.36	2692	2692	3.026896e+09
AAPL	167.23	931	931	5.567183e+09

Ajout d'une colonne

Ajout de la colonne market_cap via la méthode insert :

```
try :
    df = df.insert(2,'market_cap',[1363, 2692, 931])
    print('Column insérée')
except ValueError :
    print('Colonne déjà présente')
```

Output :

```
Colonne déjà présente
```

Remarque

La gestion d'erreur permet, grâce à une structure "*try : ... except*", d'anticiper si la colonne existe déjà et de ne pas déclencher une erreur. Il peut être assorti de plusieurs *except* et d'un *else*.

.loc[] et .iloc[]

Les propriétés *.loc* et *.iloc* permettent d'accéder à des lignes et colonnes spécifiques par le biais de leur nom (*loc*) ou de leur index (*iloc*).

Ci-dessous 2 façons de récupérer le prix et la market cap d'Amazon :

```
df.loc["AMZN",["prices","market_cap"]]
df.iloc[0,[0,1]]
```

.loc[] et .iloc[]

Les propriétés *.loc* et *.iloc* permettent d'accéder à des lignes et colonnes spécifiques par le biais de leur nom (*loc*) ou de leur index (*iloc*).

Ci-dessous 2 façons de récupérer le prix et la market cap d'Amazon :

```
df.loc["AMZN",["prices","market_cap"]]
df.iloc[0,[0,1]]
```

Remarque

- *loc* et *iloc* s'utilisent avec des **crochets**.
- Pour passer plusieurs arguments il faut les passer sous forme de liste (cf exemple ci-dessus)

Accéder aux colonnes

L'accès aux colonnes se fait de 2 façons principales :

- df.ColumnName
- df["ColumnName"]

Les 2 commandes ci-dessous retournent la colonne prices du DataFrame :

```
df.prices
df["prices"]
```

Accéder aux colonnes

L'accès aux colonnes se fait de 2 façons principales :

- df.ColumnName
- df["ColumnName"]

Les 2 commandes ci-dessous retournent la colonne prices du DataFrame :

```
df.prices
df["prices"]
```

```
stocks
AMZN    133.62
TSLA    889.36
AAPL    167.23
Name: prices, dtype: float64
```

Liste des indices et des colonnes

Les attributs *index* et *columns* permettent de récupérer tous les indices et les colonnes du DataFrame.

```
df.index.tolist()
==> ?
==> ['AMZN', 'TSLA', 'AAPL']
df.columns.tolist()
==> ?
==> ['prices', 'market_cap', 'market_cap_bis', 'number_of_share']
```

Remarque

Les attributs *index* et *columns* sont de type **"Index"**. Ce type d'objet à une méthode *.tolist()* qui permet de le convertir en liste.
NB : Le type **"Index"** est tout de même itérable. Il n'est donc pas nécessaire de passer en liste dès que l'on utilise *index* ou *columns*.

apply

La méthode *.apply()* permet d'appliquer une fonction (lambda ou pré-définie) à tout ou partie du DataFrame.

```
currency_col = ["prices","market_cap","market_cap_bis"]
df[currency_col].apply(lambda x : x/1.0768)
```

La commande ci-dessus applique un *lambda function* aux colonnes : prices, market_cap et market_cap_bis. **NB :** Ce n'est pas une modification du DataFrame, pour cela il faut réasigner les colonnes modifiée avec la fonction lambda.

stocks	prices	market_cap	market_cap_bis
AMZN	136.458333	1391.952614	1391.952614
TSLA	908.251634	2749.183007	2749.183007
AAPL	170.782271	950.776144	950.776144

head et tail

Les méthodes *.head()* et *.tail()* permettent de visualiser les premières et dernières lignes du DataFrame (respectivement).

```
df.head(2)
df.tail(2)
```

Output :

stocks	prices	market_cap	market_cap_bis	number_of_share
AMZN	133.62	1363	1363	1.020057e+10
TSLA	889.36	2692	2692	3.026896e+09

stocks	prices	market_cap	market_cap_bis	number_of_share
TSLA	889.36	2692	2692	3.026896e+09
AAPL	167.23	931	931	5.567183e+09

shape et dtypes

Les objets de type DataFrame et Series ont un attribut *shape* qui contient les dimensions de l'objet :

```
df.shape
==> (3, 4)
df.prices.shape
==> (3,)
```

L'attribut *dtypes* contient les types de chacune des colonnes du DataFrame :

```
df.dtypes
```

Output :

```
prices           float64
market_cap         int64
market_cap_bis     int64
number_of_share  float64
dtype: object
```

set_index

Ré-indexation du DataFrame avec les tickers des actions :

```
df.set_index('stocks', inplace = True)

df
```

Output :

	prices
stocks	
AMZN	133.62
TSLA	889.36
AAPL	167.23
META	161.11

rename

La méthode *.rename()* permet de renommer les colonnes d'un DataFrame. Il prend en argument un dictionnaire où les **clés** sont les noms actuels et les **valeurs** sont les nouveaux noms de colonnes.

```
df.rename(columns = {"number_of_share":"shares"}, inplace = False)
```

NB : *Inplace = False* (par défaut à False) sert à **ne pas modifier** l'objet "df", c'est donc une copie de df qui est retournée.

Output :

stocks	prices	market_cap	market_cap_bis	shares
AMZN	133.62	1363	1363	1.020057e+10
TSLA	889.36	2692	2692	3.026896e+09
AAPL	167.23	931	931	5.567183e+09

drop

La méthode *.drop()* permet de supprimer des lignes et/ou des colonnes. Il possède aussi l'argument *inplace* (booléen) qui précise si on modifie l'objet ou si on crée une copie. Il faut préciser l'axe sur lequel on intervient, *axis = 1 ou 0 (colonnes ou lignes)*.

```
df.drop(["market_cap","number_of_share"], axis = 1)
```

Output :

stocks	prices	market_cap_bis
AMZN	133.62	1363
TSLA	889.36	2692
AAPL	167.23	931

info

La méthode *.info()* permet d'obtenir les informations de base sur le DataFrame telles que les types de variables, la mémoire utilisée, le nombre de données par colonnes et leurs noms.

```
df.info()
```

Output :

```
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, AMZN to AAPL
Data columns (total 4 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   prices           3 non-null      float64
 1   market_cap       3 non-null      int64
 2   market_cap_bis   3 non-null      int64
 3   number_of_share  3 non-null      float64
dtypes: float64(2), int64(2)
memory usage: 120.0+ bytes
```

describe

La méthode *.describe()* permet d'obtenir les statistiques descriptives basiques d'un DataFrame. Cela permet de s'approprier la base de données en amont du data engineering.

```
df.describe()
```

Output :

	prices	market_cap	market_cap_bis	number_of_share
count	3.000000	3.000000	3.000000	3.000000e+00
mean	396.736667	1662.000000	1662.000000	6.264882e+09
std	426.955173	917.785923	917.785923	3.637373e+09
min	133.620000	931.000000	931.000000	3.026896e+09
25%	150.425000	1147.000000	1147.000000	4.297039e+09
50%	167.230000	1363.000000	1363.000000	5.567183e+09
75%	528.295000	2027.500000	2027.500000	7.883876e+09
max	889.360000	2692.000000	2692.000000	1.020057e+10

Méthodes statistiques usuelles

Les méthodes de statistiques usuelles sont toutes implémentées et accessible facilement. Ci-dessous une liste non-exhaustive :

- *.mean()*
- *.median()*
- *.std()*
- *.var()*
- *.corr()*
- *.cov()*
- *.min()*
- *.max()*
- *.count()*
- *.pct_change()*
- ...