Présentation rapide du langage Python

E. DUBOIS 10/01/2025

La documentation complète du langage est également disponible en français ici.

1 - Particularités de Python

Des blocs d'instruction définis par l'indentation

Le regoupement des instructions en blocs est visuellement défini par une indentation commune suivant un multiple de 4 espaces : toutes les lignes d'un même bloc doivent avoir la même indentation.

Expression `:`
`<retrait de 4 espaces>` *bloc d'*
`<retrait de 4 espaces>` *instructions*
`<retrait de 4 espaces>` *se rapportant*
`<retrait de 4 espaces>` *à l'expression*
`<retrait de 4 espaces>` *qui précède le* `:`

Les commandes python peuvent contenir des commentaires précédés par `#`. Le moyen le plus commun pour afficher un objet est la commande **print(** *identifiant de l'objet ou valeur* **)**

In [ ]:
```python
def bienvenue():
    print('Bonjour')
    print('depuis')
    print('Python')
    if (1>0):
        print('.')
        print('.')
        print('.')
    else:
        print('?')
        print('?')
        print('?')

bienvenue()
```

```
Bonjour
depuis
Python
.
.
.
```

Hormis quelques instructions composées pour des tests, des boucles ou la gestion des exceptions et erreurs d'exécution, la plupart des instructions Python sont conçues pour tenir sur une seule ligne. En cas de contenu particulièrement long, une instruction peut être proposée sur plusieurs lignes en terminant chaque ligne sauf la dernière par le caractère `\`. De même il est possible de sauter des lignes dès que l'on a affaire à des énumérations (de paramètres ou arguments de fonctions ou d'éléments dans des listes) entre parenthèses `()`, crochets `[]` ou accolades `{}`.

In [ ]:
```python
# saut de ligne rendu possible dans le \ en fin de ligne après el print ci-dessous
print \
    ( "aaa", # sauts de lignes autorisés dans les énumérations
      "bbb",
      "ccc", [ "ddd",
               "eee",
               "fff"
             ],
             {
               "ggg",
               "hhh"
             }
    )
```

```
aaa bbb ccc ['ddd', 'eee', 'fff'] {'hhh', 'ggg'}
```

2 - Types de données Python

Type de données	Classe Python	Exemple de valeur	Notation de création	Description
Entier	`int`	`42`, `-7`, `0`	`x = 42`	Représente des nombres entiers (positifs, négatifs ou 0).
Flottant	`float`	`3.14`, `-0.5`, `2.0`	`x = 3.14`	Représente des nombres réels (décimaux).
Booléen	`bool`	`True`, `False`	`x = True`	Représente les valeurs logiques.
Chaîne de caractères	`str`	`"Bonjour"`, `'Python'`	`x = "Bonjour"` ou `x = 'Python'`	Représente du texte, délimité par des guillemets simples ou doubles.
Liste	`list`	`[1, 2, 3]`, `['a', 'b']`	`x = [1, 2, 3]`	Séquence ordonnée d'éléments, modifiable. Délimité par des crochets `[]`.
Tuple	`tuple`	`(1, 2, 3)`, `('x', 'y')`	`x = (1, 2, 3)`	Séquence ordonnée d'éléments, non modifiable. Délimité par des parenthèses `()`.
Dictionnaire	`dict`	`{"clé": "valeur"}`	`x = {"clé": "valeur"}`	Collection d'éléments sous forme de paires clé-valeur. Délimité par des accolades `{}`.
Ensemble	`set`	`{1, 2, 3}`, `{'a', 'b'}`	`x = {1, 2, 3}`	Collection non ordonnée d'éléments uniques.
Ensemble immuable	`frozenset`	`frozenset({1, 2, 3})`	`x = frozenset({1, 2, 3})`	Version immuable d'un `set`.
Complexe	`complex`	`3+4j`, `1-2j`	`x = 3+4j`	Représente les nombres complexes.
Plage	`range`	`range(5)`	`x = range(5)`	Représente une séquence immuable de nombres (ici : `0, 1, 2, 3, 4`).
Bytes	`bytes`	`b"Hello"`	`x = b"Hello"`	Représente une séquence immuable d'octets (valeurs de 0 à 255).
Bytearray	`bytearray`	`bytearray(b"Hello")`	`x = bytearray(b"Hello")`	Représente une séquence mutable d'octets.
Mémoire vue	`memoryview`	`memoryview(b"Hello")`	`x = memoryview(b"Hello")`	Permet d'accéder à la mémoire d'autres objets sans copier les données.
Aucun	`NoneType`	`None`	`x = None`	Représente l'absence de valeur ou un objet nul.

Les types de base utilisés le plus couramment, mémorisant une seule valeur, sont :

Type	Exemples de valeurs	Propriétés
`int`	`123` `0` `1_000_00`	Nombre entier sans limitation de taille

Type	Exemples de valeurs	Propriétés
`float`	`1.23` `1.5e9` `1_000.000_001`	Nombre décimal
`bool`	`True` `False`	Valeur logique ou booléene
`str`	"Bonjour" 'Python'	Chaîne de caractères
`bytes`	b"octets"	Suite d'octets
`None`	*None*	Absence de valeur ou valeur nulle

La valeur *None* est utilisée fréquemment pour représenter l'absence de valeur, comme lorsque des arguments par défaut ne sont pas passés à une fonction.

Outre les chaines de caractères assimilables à des conteneurs de caractères, les principaux *iterables* (car ils peuvent être parcourus élément par élement) pouvant recevoir des valeurs multiples (et de types distincts) sont :

Type conteneur	Exemples de valeurs	Propriétés
`list`	`[1,2,3]` `['a','b','c']` `[]`	accès aux élements par position, possibilité de doublons, triable, modifiable
`tuple`	`(1,2)` `('élément',)` `()`	accès aux éléments par position, non modifiable (immuable)
`set`	`{1,2}` `{"clé1","clé2"}` `set()`	accès aux élements par valeur unique, impossibilité de doublons
`dict`	`{1:'un',3:'trois'}` `{}`	couples clés uniques / valeurs

Un type *immuable* (`int` , `float` , `bool` , `str` , `tuple` , `frozenset`) est un type de données dont la valeur ne peut pas être modifiée après sa création. Si une opération semble "*modifier*" un objet immuable, elle crée en réalité une nouvelle instance plutôt que de changer l'objet d'origine. Ainsi les type immuables ne peuvent pas être modifiés accidentellement et économisent la mémoire car plusieurs variables peuvent partager la même valeur. Toutefois, les opérations de modifications, si elle sont possibles sur les types immuables, nécessitent de recréer une nouvelle instance de l'obje en mémoire et peuvent être plus longues sur de grands objets.

Détermination du type d'un objet:
on utilise la fonction **type(** *valeur ou variable* **)**

In [ ]:
```
type(None)
```

Out[ ]:
```
NoneType
```

Tester si un objet est dans un type donné :

on utilise la fonction **isinstance(** *valeur ou objet* **,** *type ou typle de type à vérifier* **)**

In [ ]:
```
isinstance(2.0, int)
```

Out[ ]:
```
False
```

In [ ]:
```
isinstance(2.0, (int, float))
```

Out[ ]:
```
True
```

Convertir un objet vers un type donné :

On utilise la syntaxe *nom du type de destination* **(** *valeur à convertir* **)**

In [ ]:
```
float(2)
```

Out[ ]:
```
2.0
```

In [ ]:
```
int(2.9)
```

Out[ ]:
```
2
```

In [ ]:
```
str(2.9)
```

Out[ ]:
```
'2.9'
```

Exercice A

Quel serait le type des valeurs suivantes : demandez à Python de vous l'indiquer...

- Un montant monétaire de 10,50 € (ne mais mentionner l'unité)?

In [ ]:
```

```

- Un nombre d'actions acquises correspondant à 1000 titres?

In [ ]:
```

```

- Un taux d'actualisation de 0.5% (attention, la notation en % n'est pas acceptée par python?

In [ ]:
```

```

- Le fait pour une option d'être un call (valeur True) ou un put (valeur False)?

In [ ]:
```

```

- Le code Ticker TSLA de Tesla Motors?

In [ ]:
```

```

3 - Affectation de valeur à une ou plusieurs variables en Python

3.a - Contraintes de nommage

- Les identificateurs d'objets, de variables ou de fonctions débutent toujours par une lettre ou un `_`. Les lettres peuvent être de n'importe quel alphabet du moment mais l'alphabet anglais est parfois recommandé (pour éviter d'éventuels problèmes d'encodage des fichiers sources)
- Ils peuvent ensuite contenir des lettres, chiffres ou `_`
- Il est recommandé de séparer les mots dans un identificateur par des `_`
- Les identificateurs ne peuvent ressembler à un mot-clé du langage
- Il y a distinction des majuscules/minuscules : abc et Abc ne sont pas le même objet.

3.b - Modalités d'affectation de valeurs à une variable

Type d'affectation	Syntaxe/Exemple	Description
Affectation simple	x = 10 print(x) # Résultat : 10	Assigne une valeur unique à une variable.
Affectation multiple	a, b, c = 1, 2, 3 print(a, b, c) # Résultat : 1, 2, 3	Permet d'assigner plusieurs variables en une seule ligne.
Affectation à la même valeur	x = y = z = 0 print(x, y, z) # Résultat : 0, 0, 0	Assigne une même valeur à plusieurs variables.
Affectation déstructurée	a, b = [1, 2] print(a, b) # Résultat : 1, 2	Extrait des valeurs d'une séquence dans des variables individuelles.

Affectation avec opérateurs	x = 5
	x += 3 # Équivaut à x = x + 3
	print(x)
	# Résultat : 8
		Met à jour une variable en appliquant une opération.

Échange de valeurs	a, b = 5, 10 a, b = b, a print(a, b) # Résultat : 10, 5	Échange les valeurs entre deux variables.
Affectation de liste ou tuple	a, *b, c = [1, 2, 3, 4] print(a, b, c) # Résultat : 1, [2, 3], 4	Assigne des valeurs à des variables avec une liste ou un tuple, en utilisant l'opérateur `*`.
Affectation conditionnelle (ternaire)	x = 10 status = "positif" if x > 0 else "négatif" print(status) # Résultat : positif	Assigne une valeur en fonction d'une condition.

Affectation de type booléen	x = bool(1) # Équivaut à True
	y = bool(0) # Équivaut à False
	print(x, y)
	# Résultat : True, False
		Assigne une valeur booléenne en fonction de l'évaluation logique.

3.c - Vidage et suppression d'une variable

Le vidage d'une variable s'effectue en lui affectant la valeur *None* : elle devient alors non définie.

La suppression d'une variable de la mémoire s'effectue par la commande **del** *variable*.

In [ ]:
```
a = None # a est mis à non défini (sans valeur)
a is None # test si a est non définie
```

Out[ ]:
```
True
```

In [ ]:
```
del a # suppression de a
```

4 - Calculs en Python

4.a - Opérateurs Python

Catégorie	Opérateur	Exemple	Description
Affectation	`=`	`x = 5`	Assigne une valeur à une variable.
⚠ **Affectation composée**	`+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`	`x += 1` (équivalent à `x = x + 1`)	Met à jour une variable en appliquant une opération.
Arithmétique	`+`	`3 + 2`	Additionne deux nombres.
	`-`	`5 - 2`	Soustrait deux nombres.
	`*`	`4 * 3`	Multiplie deux nombres.
	`/`	`10 / 2`	Divise deux nombres (résultat flottant).
⚠	`//`	`10 // 3`	Division entière (quotient sans reste).
⚠	`%`	`10 % 3`	Modulo (reste de la division entière).
⚠	`**`	`2 ** 3`	Exponentiation (puissance).
⚠ **Comparaison**	`==`	`x == y`	Vérifie l'égalité entre deux valeurs.
⚠	`!=`	`x != y`	Vérifie si deux valeurs sont différentes.
	`<`	`x < y`	Vérifie si une valeur est inférieure à une autre.
	`>`	`x > y`	Vérifie si une valeur est supérieure à une autre.
	`<=`	`x <= y`	Vérifie si une valeur est inférieure ou égale à une autre.
	`>=`	`x >= y`	Vérifie si une valeur est supérieure ou égale à une autre.
Logique	`and`	`True and False`	Renvoie `True` si les deux conditions sont vraies.
	`or`	`True or False`	Renvoie `True` si au moins une condition est vraie.
	`not`	`not True`	Renverse la valeur logique.
⚠ **Bit à bit**	`&`	`5 & 3`	Effectue un ET bit à bit.
	`|`	`5 | 3`	Effectue un OU bit à bit.
	`^`	`5 ^ 3`	Effectue un OU exclusif (XOR) bit à bit.

Catégorie	Opérateur	Exemple	Description
	`~`	`~5`	Renverse les bits (complément à 1).
	`<<`	`5 << 1`	Décale les bits vers la gauche.
	`>>`	`5 >> 1`	Décale les bits vers la droite.
Appartenance	`in`	`"a" in "abc"`	Vérifie si un élément appartient à une séquence.
	`not in`	`"d" not in "abc"`	Vérifie si un élément n'appartient pas à une séquence.
Identité	`is`	`x is y`	Vérifie si deux objets sont identiques (même référence en mémoire).
	`is not`	`x is not y`	Vérifie si deux objets ne sont pas identiques.
⚠**Indexation**	`[]`	`lst[0]`	Accède à l'élément d'une séquence par son index.
Slicing	`[:]`	`lst[1:3]`	Extrait une sous-séquence.

Exercice B

1. Affectez à une variable, de nom de vottre choix mais lié au concept de *valeur actuelle*, la valeur actuelle d'un montant de 1000 € à recevoir dans 3 ans sachant un taux d'actualisation de 3% et vérifiez en affichant cette variable que vous obtenez 915,14 €.

In [ ]:

2. Affectez en une seule ligne à 3 variables dont vous choisirez le nom les caractéristiques suivantes d'une obligation :

> - valeur nominale de 1000 €
> - maturité de 3 ans
> - taux de coupon de 5%

In [ ]:

3. Calculez et stockez dans une variable ayant un nom relatif à la notion de *cash-flow* le montant d'un coupon calculé à l'aide des variables précédentes ?

In [ ]:

4. Majorez ce cash-flow de la valeur nominale pour calculer le dernier cash-flow et affichez sa valeur

In [ ]:

5. Calculez puis affichez la valeur à l'émission de l'obligation en supposant un taux d'actualisation du marché à 3% et vérifiez qu'elle est voisine de 1145.673

In [ ]:

Out[ ]: `3.0`

4.b - Fonctions intégrées de Python

Fonction	Description	Exemple
`print()`	Affiche des données dans la console.	print("Bonjour, Python!") # Résultat : Bonjour, Python!
`len()`	Retourne la longueur d'une séquence ou d'une collection.	lst = [1, 2, 3] print(len(lst)) # Résultat : 3
`type()`	Retourne le type d'un objet.	x = 42 print(type(x)) # Résultat :
`input()`	Lit une chaîne de caractères saisie par l'utilisateur.	name = input("Votre nom : ") print("Bonjour,", name) # Entrée : Alice # Résultat : Bonjour, Alice
`int()`, `float()`, `str()`	Convertit un objet en entier, nombre flottant ou chaîne.	x = "42" print(int(x)) # Résultat : 42 print(float(x)) # Résultat : 42.0 print(str(42)) # Résultat : '42'
`round()`	Arrondit un nombre à un nombre donné de décimales.	x = 3.14159 print(round(x, 2)) # Résultat : 3.14
`abs()`	Retourne la valeur absolue d'un nombre.	x = -5 print(abs(x)) # Résultat : 5
`sum()`	Calcule la somme des éléments d'un itérable.	lst = [1, 2, 3] print(sum(lst)) # Résultat : 6
`min()`, `max()`	Retourne la valeur minimale ou maximale d'un itérable.	lst = [1, 2, 3] print(min(lst)) # Résultat : 1 print(max(lst)) # Résultat : 3
`range()`	Génère une séquence de nombres.	for i in range(1, 5): print(i) # Résultat : 1, 2, 3, 4
`sorted()`	Retourne une liste triée.	lst = [3, 1, 2] print(sorted(lst)) # Résultat : [1, 2, 3]
`reversed()`	Retourne un itérable inversé.	lst = [1, 2, 3] print(list(reversed(lst))) # Résultat : [3, 2, 1]
`zip()`	Combine plusieurs itérables en tuples.	a = [1, 2] b = ["x", "y"] print(list(zip(a, b))) # Résultat : [(1, 'x'), (2, 'y')]
`enumerate()`	Retourne un itérable avec des index et valeurs.	lst = ["a", "b", "c"] for i, val in enumerate(lst): print(i, val) # Résultat : # 0 a # 1 b # 2 c
`any()`, `all()`	`any()` retourne `True` si au moins un élément est vrai. `all()` retourne `True` si tous les éléments sont vrais.	lst = [0, 1, 2] print(any(lst)) # Résultat : True print(all(lst)) # Résultat : False
`map()`	Applique une fonction à tous les éléments d'un itérable.	lst = [1, 2, 3] print(list(map(lambda x: x * 2, lst))) # Résultat : [2, 4, 6]
`filter()`	Filtre les éléments d'un itérable selon une condition.	lst = [1, 2, 3, 4] print(list(filter(lambda x: x % 2 == 0, lst))) # Résultat : [2, 4]
`help()`	Affiche l'aide ou la documentation d'un objet.	help(len) # Résultat : Documentation de la fonction len

4.c - Utilisation des modules pour obtenir des calculs complémentaires

L'importation de module est une activité incontournable tant les fonctions de base sont limitées et les services déportés dans les modules. Les instructions intégrées au langage Python permettent d'importer des modules (livrés systématiquement avec le langage ou installés en complément avec les utilitaires d'installation — **pip** , **conda**, **apt-get**, ... — de la distribution Python courante).

Commande	Description	Exemple
`import module`	Importe un module entier.	import math print(math.sqrt(16)) # Résultat : 4.0
`from module import fonction`	Importe une fonction spécifique depuis un module.	from math import sqrt print(sqrt(16)) # Résultat : 4.0
`from module import *`	Importe toutes les fonctions et classes d'un module (non recommandé).	from math import * print(sin(0)) # Résultat : 0.0
`import module as alias`	Importe un module avec un alias.	import numpy as np arr = np.array([1, 2, 3]) print(arr) # Résultat : [1 2 3]
`dir(module)`	Affiche une liste des attributs et fonctions disponibles dans un module.	import math print(dir(math)) # Résultat : ['acos', 'acosh', ..., 'sqrt', ...]
`help(module)`	Affiche la documentation du module.	import math help(math) # Résultat : Affiche les fonctions et leur description
`module.__doc__`	Affiche la chaîne de documentation (docstring) d'un module.	import math print(math.__doc__) # Résultat : Documentation du module math
`reload(module)`	Recharge un module déjà importé (nécessite `importlib`).	import importlib import math importlib.reload(math) # Recharge le module math

Les modules d'usage le plus courant sont :

Module	Utilisation principale	Exemple
`math`	Opérations mathématiques avancées (racines carrées, trigonométrie, logarithmes).	import math print(math.sqrt(16)) # Résultat : 4.0

`random`	Génération de nombres aléatoires (entiers ou réels) et simulation de lois statistiques.	import random
		print(random.randint(1, 10)) # Résultat : Nombre entier entre 1 et 10
		print(random.uniform(1.5, 5.5)) # Nombre aléatoire réel entre 1.5 et 5.5
		# Génération d'un nombre aléatoire suivant une loi normale
		x = random.normalvariate(mu=0, sigma=1)
		print(x) # Résultat : Nombre réel aléatoire avec moyenne 0 et écart type 1

`statistics`	Calcul de statistiques (moyenne, médiane, variance, etc.) et évaluation de distributions.	import statistics
		data = [1, 2, 3, 4, 5]
		mean = statistics.mean(data)
		stdev = statistics.stdev(data)
		print(mean) # Résultat : 3.0
		print(stdev) # Résultat : 1.58 (approx.)

`os`	Gestion des interactions avec le système d'exploitation (fichiers, répertoires).	import os print(os.getcwd()) # Résultat : Répertoire de travail actuel
`sys`	Manipulation des arguments de la ligne de commande et du système.	import sys print(sys.version) # Résultat : Version de Python

`time`	Gestion du temps et des délais.	import time
		time.sleep(1) # Pause de 1 seconde
		print("Pause terminée")

`datetime`	Manipulation des dates et heures.	import datetime print(datetime.datetime.now()) # Résultat : Date et heure actuelles
`json`	Lecture et écriture de données JSON.	import json data = {"nom": "Alice", "age": 25} json_data = json.dumps(data) print(json_data) # Résultat : {"nom": "Alice", "age": 25}
`re`	Traitement des expressions régulières.	import re result = re.search(r'\d+', 'abc123') print(result.group()) # Résultat : 123
`collections`	Structures de données avancées (déques, compteurs, etc.).	from collections import Counter print(Counter("abracadabra")) # Résultat : Compte les occurrences des lettres
`itertools`	Itérateurs avancés et outils de manipulation d'itérables.	import itertools for comb in itertools.combinations([1, 2, 3], 2): print(comb) # Résultat : Toutes les combinaisons de 2 éléments
`pandas`	Analyse et manipulation de données structurées (DataFrames).	import pandas as pd data = {"Nom": ["Alice", "Bob"], "Age": [25, 30]} df = pd.DataFrame(data) print(df)
`matplotlib`	Visualisation de données sous forme de graphiques.	import matplotlib.pyplot as plt plt.plot([1, 2, 3], [4, 5, 6]) plt.show()

Exercice C

1. Importez la fonction nommée *linear_regression* présente dans le module nommé *statistics*.

In [ ]:

2. Affichez l'aide de cette fonction *linear_regression*

In [ ]:

3. Importez le module `statistics` avec l'alias `st`

In [ ]:

Sachant m une fréquence annuelle de composition des intérêts ($m = 1 \longrightarrow$ 1 interêt annuel, $m = 2 \longrightarrow$ 2 interets semestriels, $m = 4 \longrightarrow$ 4 intérêts trimestriels, ...) un taux continu r_c (correspondant à $m \to \infty$) s'exprime en fonction d'un taux discret r_m exprimé selon cette fréquence m par la formule :

$$r_c = m \times ln\left(1 + \frac{r_m}{m}\right)$$

et qu'en retour un taux r_m exprimé dans une fréquence m peut être déruit du taux continu r_c par la formule :

$$r_m = m \times \left(\exp\left(\frac{r_c}{m}\right) - 1\right)$$

4. Importez le module *math*

In [ ]:

5. Affichez l'aide de ce module *math* pour obtenir les fonctions permettant de calculer un logarithme ou une exponentielle.

In [ ]:

6. Calculez la valeur d'un taux continu équivalent à un taux trimestriel de 10% et vérifiez que vosu obtenez 9,877%

In [ ]:

7. Identifiez 2 moyens de calculer le taux semestriel correspondant à un taux continu de 5% en utilisant 2 fonctions différentes du module *math* liées au calcul de l'exponentielle et vérifiez que vous obtenez 5,063%.

- formule n°1 :

In [ ]:

- formule n°2 :

In [ ]:

In [ ]:

4.c - Gestion des informations calendaires

In [6]:
```
import datetime as dt
aujourdhui=dt.date.today()
aujourdhui
```

Out[6]: datetime.date(2025, 1, 17)

In [7]:
```
maintenant=dt.datetime.now()
maintenant
```

Out[7]: datetime.datetime(2025, 1, 17, 9, 3, 28, 16590)

Les fonctions convertissant les données calendaires depuis du texte ou vers du texte utilisent les notations suivantes pour décrire les composants d'une date/heure :

codification	Signification	Exemple
`%a`	Nom abrégé du jour de la semaine.	Sun, Mon, ... (dépend de la langue locale)
`%A`	Nom complet du jour de la semaine.	Sunday, Monday, ...(dépend de la langue locale)
`%w`	Jour de la semaine sous forme de nombre décimal.	0, 1, ..., 6
`%d`	Jour du mois sous la forme d'un nombre décimal éventuellement complété d'un zéro.	01, 02, ..., 31
`%-d`	Jour du mois sous forme de nombre décimal.	1, 2, ..., 30
`%b`	Nom du mois abrégé.	Jan, Feb, ..., Dec (dépend de la langue locale)
`%B`	Nom complet du mois.	January, February, ... (dépend de la langue locale)
`%m`	Mois sous la forme d'un nombre décimal éventuellement complété d'un zéro.	01, 02, ..., 12
`%-m`	Mois sous forme de nombre décimal.	1, 2, ..., 12
`%y`	Année sans siècle sous la forme d'un nombre décimal éventuellement complété d'un zéro.	00, 01, ..., 99
`%-y`	Année sans siècle en nombre décimal.	0, 1, ..., 99
`%Y`	Année avec siècle en nombre décimal.	2013, 2019 etc.
`%H`	Heure (horloge de 24 heures) sous forme de nombre décimal éventuellement complété d'un zéro.	00, 01, ..., 23
`%-H`	Heure (horloge de 24 heures) en nombre décimal.	0, 1, ..., 23
`%I`	Heure (horloge de 12 heures) sous forme de nombre décimal éventuellement complété d'un zéro.	01, 02, ..., 12
`%-I`	Heure (horloge de 12 heures) sous forme de nombre décimal.	1, 2, ... 12
`%p`	Locale's AM ou PM.	AM, PM
`%M`	Minute sous la forme d'un nombre décimal éventuellement complété d'un zéro.	00, 01, ..., 59
`%-M`	Minute sous forme de nombre décimal.	0, 1, ..., 59
`%S`	Seconde sous la forme d'un nombre décimal éventuellement complété d'un zéro.	00, 01, ..., 59
`%-S`	Seconde sous forme de nombre décimal.	0, 1, ..., 59
`%f`	Microseconde sous la forme d'un nombre décimal, éventuellement complété de zéros.	000000 - 999999
`%z`	Décalage UTC sous la forme +HHMM ou -HHMM.	
`%Z`	Nom du fuseau horaire.	
`%j`	Jour de l'année sous la forme d'un nombre décimal éventuellement complété de zéros.	001, 002, ..., 366
`%-j`	Jour de l'année sous forme de nombre décimal.	1, 2, ..., 366
`%U`	Numéro de la semaine de l'année (le dimanche étant le premier jour de la semaine). Tous les jours d'une nouvelle année précédant le premier dimanche sont considérés comme faisant partie de la semaine 0.	00, 01, ..., 53
`%W`	Numéro de semaine de l'année (lundi comme premier jour de la semaine). Tous les jours d'une nouvelle année précédant le premier lundi sont considérés comme faisant partie de la semaine 0.	00, 01, ..., 53
`%c`	Représentation appropriée de la date et de l'heure dans la langue locale.	Mon Sep 30 07:06:05 2013 (dépend de la langue locale)
`%x`	Représentation appropriée de la date dans la langue locale.	09/30/13 (dépend de la langue locale)
`%X`	Représentation de l'heure appropriée au système local.	0,29589120370 3704
`%%`	Un caractère "%" littéral.	%
`%G`	Année complète ISO 8601 représentant l'année contenant la plus grande partie de la semaine ISO (%V).	0001, 0002, ..., 2023, 2024, ..., 9998, 9999
`%u`	Jour de la semaine ISO 8601 où 1 correspond au lundi.	1, 2, ..., 7
`%V`	Numéro de la semaine ISO 8601, avec lundi étant le premier jour de la semaine. La semaine 01 est la semaine contenant le 4 janvier.	01, 02, ..., 53
`%:z`	Décalage temporel UTC de la forme ±HH:MM[:SS[.ffffff]] vide si la date ne comprend pas d'information de créneau horaire	(vide), +00:00, -04:00, +10:30

In [8]:
```
print(aujourdhui.strftime('%d/%m/%Y'))
```

```
17/01/2025
```

In [9]:
```
noël_2024=dt.date(2024,12,25) # Création d'une date à partir de ses composants
noël_2024
```

Out[9]:
```
datetime.date(2024, 12, 25)
```

In [10]:
```
noël_2024_midi_trente=dt.datetime(2024,12,25,12,30) # Création d'un instant à partir de ses composants
```

In [13]:
```
fête_nationale_2024=dt.datetime.strptime('14/07/2024', "%d/%m/%Y").date() # Création d'une date à partir d'une représentation textuelle
fête_nationale_2024
```

Out[13]:
```
datetime.date(2024, 7, 14)
```

In [16]:
```
print(noël_2024-fête_nationale_2024) # Expression d'un décalage temporel
```

```
164 days, 0:00:00
```

In [18]:
```
print((noël_2024-fête_nationale_2024).days) # En nombre entier de jours
```

```
164
```

Affichage de données temporelles en fonction des caractéristiques de langue disponibles dans le système sous-jacent :

In [ ]:
```
%%capture locales
!locale -a
```

In [ ]:
```
# Affichage en fonction des caractéristiques de langue du système sous jacent
import locale
for l in str(locales).splitlines():
    locale.setlocale(locale.LC_ALL,l)
    print({"locale":l,"rendu":maintenant.strftime("%c")})
```

```
{'locale': 'C', 'rendu': 'Tue Jan 14 13:00:22 2025'}
{'locale': 'C.utf8', 'rendu': 'Tue Jan 14 13:00:22 2025'}
{'locale': 'en_US.utf8', 'rendu': 'Tue 14 Jan 2025 01:00:22 PM '}
{'locale': 'POSIX', 'rendu': 'Tue Jan 14 13:00:22 2025'}
```

Extraction de composant d'une date-heure :

Composant	Description	Exemple
Jour (`.day`)	Extrait le jour du mois.	from datetime import datetime date = datetime(2023, 1, 11) print(date.day) # Résultat : 11
Mois (`.month`)	Extrait le mois de la date.	from datetime import datetime date = datetime(2023, 1, 11) print(date.month) # Résultat : 1
Année (`.year`)	Extrait l'année de la date.	from datetime import datetime date = datetime(2023, 1, 11) print(date.year) # Résultat : 2023
Jour depuis le 1er janvier (`timetuple().tm_yday`)	Retourne le numéro du jour dans l'année (depuis le 1er janvier).	from datetime import datetime date = datetime(2023, 1, 11) print(date.timetuple().tm_yday) # Résultat : 11
Heure (`.hour`)	Extrait l'heure d'une date-heure.	from datetime import datetime date = datetime(2023, 1, 11, 15, 30) print(date.hour) # Résultat : 15
Minute (`.minute`)	Extrait la minute d'une date-heure.	from datetime import datetime date = datetime(2023, 1, 11, 15, 30) print(date.minute) # Résultat : 30
Seconde (`.second`)	Extrait la seconde d'une date-heure.	from datetime import datetime date = datetime(2023, 1, 11, 15, 30, 45) print(date.second) # Résultat : 45
Microseconde (`.microsecond`)	Extrait la microseconde d'une date-heure.	from datetime import datetime date = datetime(2023, 1, 11, 15, 30, 45, 123456) print(date.microsecond) # Résultat : 123456
Jour de la semaine (`.weekday()`)	Retourne le jour de la semaine (0 = lundi, 6 = dimanche).	from datetime import datetime date = datetime(2023, 1, 11) print(date.weekday()) # Résultat : 2 (mercredi)
Timestamp (`.timestamp()`)	Retourne le timestamp (nombre de secondes depuis le 1er janvier 1970).	from datetime import datetime date = datetime(2023, 1, 11) print(date.timestamp()) # Résultat : Exemple, 1673395200.0
Heure formatée (`.strftime()`)	Formate une date ou une heure selon un format donné.	from datetime import datetime date = datetime(2023, 1, 11, 15, 30) print(date.strftime("%d/%m/%Y %H:%M")) # Résultat : 11/01/2023 15:30

Exercice D

1. Convertissez le texte ci-dessous en une donnée de type date
   `2025-01-30` (en format AAAA-MM-JJ)

In [ ]:
```
```

2. Affichez cette date en présentant (en anglais ou en français selon la configuration de votre système) les jours dans la semaine et mois en toutes lettres : *Vendredi 31 janvier 2025* ou *Thursday 31 January 2025*

In [ ]:
```
```

3. La base calendaire 30E/360 ou *Eurobond basis* exprime en fraction d'année la durée d'un placement par la formule ci-dessous :

- A_1 Année de la date de début
- M_1 Mois de la date de début
- J_1 Jour de la date de début (limité à 30 : utiliser la fonction intégrée *min* pour cela).
- A_2 Année de la date de fin
- M_2 Mois de la date de fin
- J_2 Jour de la date de fin (limité à 30 : utiliser la fonction intégrée *min* pour cela)

$$durée = \frac{360 \times (A_2 - A_1) + 30 \times (M_2 - M_1) + (J_2 - J_1)}{360}$$

Calculez en Python et suivant cette base calendaire la durée en fraction d'année d'un placement émis ce jour et remboursé le 1er janvier de l'année prochaine.

In [ ]:

4. La base ACT/ACT ICMA ou Réel/Réel ICMA calcule la durée par la formule :

Soient :

- d_1 expression numérique de la date de début
- d_2 expression numérique de la date de fin

$$durée = \frac{d_1 - d_2}{\text{nombre de jours sur la période } [d_2 - 1 \text{ an} \rightarrow d_2]}$$

Calculez en Python la durée d'un placement émis ce jour et remboursé le 1er janvier de l'année prochaine.

In [ ]:

5 - Utilisation des conteneurs Python

Notation	Conteneur
`[ ... ]`	Liste
`" ... "` ou `' ... '`	Chaine de caractères
`{ ... : ... }`	Dictionnaire
`{ ... }`	Ensemble
`( ... , ... )`	Tuple

5.1 - Listes

Les listes sont les tableaux à une dimension ayant la possibilité de contenir simultanément des valeurs de plusieurs types de données. Chaque élement d'une liste peut être accessible au travers d'un indice positif à partir du début (position 0 pour le premier élément) ou un indice négatif compris à partir de la fin (-1 pour le dernier élément).

Propriétés des listes : peuvent être triées, peuvent contenir plusieurs types de de données, peuvent contenir des valeurs en double, itérables (peuvent être parcourues de manière itérative). Leurs éléments sont accessibles par position et peuvent changer de valeur.

Type de création	Syntaxe/Exemple	Description
Liste vide	`lst = []` `print(lst)  # Résultat : []`	Crée une liste vide à l'aide de crochets.
Liste avec des éléments	`lst = [1, 2, 3]` `print(lst)  # Résultat : [1, 2, 3]`	Crée une liste avec des éléments définis.
À partir d'une chaîne	`lst = list("abc")` `print(lst)  # Résultat : ['a', 'b', 'c']`	Convertit une chaîne de caractères en liste.

À partir d'un tuple	`tup = (1, 2, 3)`
	`lst = list(tup)`
	`print(lst)  # Résultat : [1, 2, 3]`
		Convertit un tuple en liste.

Par compréhension de liste	`lst = [x**2 for x in range(5)]` `print(lst)  # Résultat : [0, 1, 4, 9, 16]`	Crée une liste à partir d'une expression ou d'une condition.
À partir de la méthode `range()`	`lst = list(range(5))` `print(lst)  # Résultat : [0, 1, 2, 3, 4]`	Crée une liste contenant une séquence de nombres.
Liste avec des éléments répétés	`lst = [0] * 5` `print(lst)  # Résultat : [0, 0, 0, 0, 0]`	Crée une liste contenant un élément répété plusieurs fois.

Copie d'une liste	`lst1 = [1, 2, 3]`
	`lst2 = list(lst1)`
	`print(lst2)  # Résultat : [1, 2, 3]`
		Crée une nouvelle liste à partir d'une liste existante.

À partir d'un itérable	`lst = list(x for x in range(5) if x % 2 == 0)` `print(lst)  # Résultat : [0, 2, 4]`	Crée une liste en parcourant un itérable avec une condition.
Liste imbriquée	`lst = [[1, 2], [3, 4]]` `print(lst)  # Résultat : [[1, 2], [3, 4]]`	Crée une liste contenant d'autres listes.

Commande/Méthode	Description	Exemple
`append()`	Ajoute un élément à la fin de la liste.	lst = [1, 2, 3] lst.append(4) print(lst) # Résultat : [1, 2, 3, 4]
`extend()`	Ajoute les éléments d'une autre séquence à la liste.	lst = [1, 2] lst.extend([3, 4]) print(lst) # Résultat : [1, 2, 3, 4]
`insert()`	Insère un élément à une position donnée.	lst = [1, 3] lst.insert(1, 2) print(lst) # Résultat : [1, 2, 3]
`remove()`	Supprime la première occurrence d'un élément.	lst = [1, 2, 3, 2] lst.remove(2) print(lst) # Résultat : [1, 3, 2]
`pop()`	Supprime et retourne un élément à l'index spécifié (par défaut le dernier).	lst = [1, 2, 3] x = lst.pop() print(x) # Résultat : 3 print(lst) # Résultat : [1, 2]
`clear()`	Supprime tous les éléments de la liste.	lst = [1, 2, 3] lst.clear() print(lst) # Résultat : []
`index()`	Retourne l'index de la première occurrence d'un élément.	lst = [1, 2, 3] idx = lst.index(2) print(idx) # Résultat : 1
`count()`	Compte le nombre d'occurrences d'un élément dans la liste.	lst = [1, 2, 2, 3] cnt = lst.count(2) print(cnt) # Résultat : 2
`sort()`	Trie les éléments de la liste en place (par ordre croissant par défaut).	lst = [3, 1, 2] lst.sort() print(lst) # Résultat : [1, 2, 3]
`reverse()`	Inverse l'ordre des éléments de la liste.	lst = [1, 2, 3] lst.reverse() print(lst) # Résultat : [3, 2, 1]
`copy()`	Crée une copie superficielle de la liste.	lst = [1, 2, 3] lst_copy = lst.copy() print(lst_copy) # Résultat : [1, 2, 3]
`len()`	Retourne le nombre d'éléments dans la liste.	lst = [1, 2, 3] print(len(lst)) # Résultat : 3
`sum()`	Retourne la somme des éléments numériques de la liste.	lst = [1, 2, 3] print(sum(lst)) # Résultat : 6
`min()` et `max()`	Retourne respectivement le plus petit et le plus grand élément de la liste.	lst = [1, 2, 3] print(min(lst)) # Résultat : 1 print(max(lst)) # Résultat : 3

Extraction d'élément et de de sous-ensemble (*tranche* ou *slicing*) :

- **elément** : *liste* **[** *position >=0 à partir du début et position <0 à partir de la fin (-1 = dernier élément)* **]**
- **Sous-ensemble** : *liste* **[** *position de début (0 si omise)* **:** *position de fin (exclue, jusqu'à la fin si omise)* **:** *pas (1 si omis)* **]** ou fonction ***slice****(position de début incluse, position de fin exclue), pas facultatif)*

Type d'extraction	Syntaxe/Exemple	Description
Extraction d'un élément	lst = [10, 20, 30, 40] x = lst[2] print(x) # Résultat : 30	Extrait l'élément situé à un index donné (ici l'index 2).
Extraction avec index négatif	lst = [10, 20, 30, 40] x = lst[-1] print(x) # Résultat : 40	Extrait l'élément en comptant depuis la fin (ici le dernier élément).
Tranche de début	lst = [10, 20, 30, 40] subset = lst[:2] print(subset) # Résultat : [10, 20]	Extrait les premiers éléments jusqu'à l'index 2 (exclus).
Tranche de fin	lst = [10, 20, 30, 40] subset = lst[2:] print(subset) # Résultat : [30, 40]	Extrait les éléments à partir de l'index 2 jusqu'à la fin.
Tranche spécifique	lst = [10, 20, 30, 40] subset = lst[1:3] print(subset) # Résultat : [20, 30]	Extrait les éléments entre les index 1 (inclus) et 3 (exclus).
Tranche avec pas	lst = [10, 20, 30, 40, 50] subset = lst[::2] print(subset) # Résultat : [10, 30, 50]	Extrait des éléments avec un pas (ici 2).
Tranche inversée	lst = [10, 20, 30, 40] subset = lst[::-1] print(subset) # Résultat : [40, 30, 20, 10]	Extrait les éléments dans l'ordre inverse.
Tranche inversée avec pas	lst = [10, 20, 30, 40, 50] subset = lst[4:1:-1] print(subset) # Résultat : [50, 40, 30]	Extrait les éléments dans l'ordre inverse entre les index 4 et 1.
Filtrage avec condition	lst = [10, 20, 30, 40] subset = [x for x in lst if x > 20] print(subset) # Résultat : [30, 40]	Extrait les éléments répondant à une condition.
Extraction avec `enumerate()`	lst = [10, 20, 30, 40] subset = [x for i, x in enumerate(lst) if i % 2 == 0] print(subset) # Résultat : [10, 30]	Extrait des éléments en utilisant les index et une condition.
Extraction par index spécifique	lst = [10, 20, 30, 40] subset = [lst[i] for i in [0, 2]] print(subset) # Résultat : [10, 30]	Extrait les éléments situés aux index spécifiés.

Exercice E

1. Créez une liste de cash-flows périodiques enchainant les valeurs suivantes :

- 1000 en période 0
- 1400 en période 1
- 1300 en période 2
- 1200 en période 3
- 1100 en période 4
- 1400 en période 5

In [ ]:

2. Affichez la première valeur de la liste

In [ ]:

3. Affichez la dernière valeur de la liste

In [ ]:

4. Affichez les 3 premières valeurs de la liste

In [ ]:

5. Affichez les 3 dernières valeurs de la liste

In [ ]:

6. Affichez les valeurs de la liste en ordre inverse des périodes

In [ ]:

7. Affichez les cash-flow par ordre croissant

In [ ]:

8. Ajoutez un cash-flow de 1500 en fin de liste

In [ ]:

9. Retirez le premier élement de la liste puis affichez le contenu de la liste.

In [ ]:

10. Combien de cash-flows restent-ils dans la liste ?

In [ ]:

11. En quelle période sera versé le premier cash-flow de 1200 euros ?

In [ ]:

5.2 - Dictionnaires

Les dictionnaires permettent d'associer une clé à une valeur.

- **Propriétés des dictionnaires:** ordre quelconque des élements, itérables (peuvent être parcourus de manière itérative élement par élement), clés immuables (ne peuvent changer de valeur sans destruction puis création d'une nouvelle association clé/valeur) et valeurs modifiables (leurs élements peuvent changer de valeur), peuvent contenir des valeurs de types différents.
- construits sur un mode d'association clé/valeur.
- leurs clés sont uniques et peuvent être des chaines de caractères, nombres ou tuples.
- leurs valeurs peuvent être de n'importe quel type.

Opération	Description	Exemple
Créer un dictionnaire vide	Initialise un dictionnaire vide.	`d = {}` `print(d)  # Résultat : {}`
Créer un dictionnaire avec des valeurs	Initialise un dictionnaire avec des paires clé-valeur.	`d = {"nom": "Alice", "age": 25}` `print(d)  # Résultat : {'nom': 'Alice', 'age': 25}`
Accéder à une valeur	Récupère la valeur associée à une clé.	`d = {"nom": "Alice", "age": 25}` `print(d["nom"])  # Résultat : Alice`

Ajouter ou mettre à jour une clé	Ajoute une nouvelle clé ou met à jour une clé existante.	`d = {"nom": "Alice"}`
		`d["age"] = 25`
		`print(d)  # Résultat : {'nom': 'Alice', 'age': 25}`

Supprimer une clé	Supprime une clé et sa valeur.	`d = {"nom": "Alice", "age": 25}`
		`del d["age"]`
		`print(d)  # Résultat : {'nom': 'Alice'}`

Obtenir une valeur avec `get()`	Récupère la valeur associée à une clé, avec une valeur par défaut si la clé n'existe pas.	`d = {"nom": "Alice"}` `print(d.get("age", "Non spécifié"))  # Résultat : Non spécifié`
Vérifier si une clé existe	Teste si une clé est présente dans le dictionnaire.	`d = {"nom": "Alice"}` `print("age" in d)  # Résultat : False`
Obtenir les clés	Retourne une vue des clés du dictionnaire.	`d = {"nom": "Alice", "age": 25}` `print(d.keys())  # Résultat : dict_keys(['nom', 'age'])`
Obtenir les valeurs	Retourne une vue des valeurs du dictionnaire.	`d = {"nom": "Alice", "age": 25}` `print(d.values())  # Résultat : dict_values(['Alice', 25])`
Obtenir les paires clé-valeur	Retourne une vue des paires clé-valeur.	`d = {"nom": "Alice", "age": 25}` `print(d.items())  # Résultat : dict_items([('nom', 'Alice'), ('age', 25)])`

Fusionner deux dictionnaires	Combine les clés et les valeurs de deux dictionnaires.	`d1 = {"nom": "Alice"}`
		`d2 = {"age": 25}`
		`d1.update(d2)`
		`print(d1)  # Résultat : {'nom': 'Alice', 'age': 25}`

Supprimer toutes les clés	Vide complètement le dictionnaire.	`d = {"nom": "Alice", "age": 25}`
		`d.clear()`
		`print(d)  # Résultat : {}`

Copier un dictionnaire	Crée une copie superficielle d'un dictionnaire.	`d = {"nom": "Alice", "age": 25}`
		`copy_d = d.copy()`
		`print(copy_d)  # Résultat : {'nom': 'Alice', 'age': 25}`

Parcourir un dictionnaire	Itère sur les clés ou les paires clé-valeur.	`d = {"nom": "Alice", "age": 25}`
		`for key, value in d.items():`
		`  print(key, value)`
		`# Résultat :`
		`# nom Alice`
		`# age 25`

Exercice F

1. Créez un dictionnaire qui permet de mémoriser les correspondances des notations de *Moody's* chez *Standard & Poor's* d'après le tableau suivant :

Moody's	S&P
Aaa	AAA
Aa1	AA+
Aa2	AA
Aa3	AA−
A1	A+
A2	A
A3	A−
Baa1	BBB+
Baa2	BBB
Baa3	BBB−
Ba1	BB+
Ba2	BB
Ba3	BB−
B1	B+
B2	B
B3	B−
Caa1	CCC+
Caa2	CCC
Caa3	CCC−
Ca	CC
C	D

In [ ]:

2. Utilisez ce dictionnaire pour obtenir l'équivalent chez *S&P* d'une note A3 chez *Moody's*

In [ ]:

3. Listez toutes les notes de *Moody's* à partir de ce dictionnaire

In [ ]:

4. Listez toutes les notes de *S&P* à partir de ce dictionnaire

In [ ]:

5. Combien d'équivalences compte ce dictionnaire ?

In [ ]:

6. Proposez une manière de tester l'inexistence de la note A4 chez *Moody's*

In [ ]:

7. A l'aide de dictionnaires imbriqués (dictionnaires dans des dictionnaires) construisez une arborescence des liens financiers entre les sociétés décrites ci-après.
   Soent 5 sociétés A, B, C, D et E.
   La société A a des participations dans les sociétés B (40%) et C (60%).
   La société B a des participations dans les sociétés D (50%) et E (50%).

In [ ]:

8. Indiquez l'instruction qui permettra d'obtenir les participations de la société A.

In [ ]:

9. Sachant la valeur connue du capital de certaines sociétés (B: 100, C: 150, D: 70, E: 80), indiquez l'instruction Python qui permettra de calculer la valeur totale en capital des participations de B.

In [ ]:
```
capital={'B': 100, 'C': 150, 'D': 70, 'E': 80}
```

5.3 - Chaines de caractères

Propriétés des chaînes de caractères : itérables (peuvent être parcourues caractère par caractère de de manière itérative), immuables (on ne peut en modifier un caractère et en cas de modification une chaine est remplacée par une autre)

Les chaines de caractères littérales sont :

- les chaines classiques sont spécifiées entre apostrophes `'` ou guillemets `"`. Elles acceptent des séquences d'échappement précédées par `\` pour inclure
  - des apostrophes ', des guillemets `\"` ou des caractères d'échappement `\\`
  - des caractères non imprimables saut de lignes (`\n`), tabulations (`\t`), ...
- Les chaines de caractères multilignes sont encadrées par des triples apostrophes `'''` ou guillemets `"""`
- les chaines brutes précédéres par `r` pour *raw* traitent tous les caractères de manière directe
- les chaines binaires sont précédées par un `b` et sont assimilées à des listes d'octets et non des listes de caractères
- chaines interpolées précédéres par un `f` qui peuvent contenir une expression python à évaluer entre des accolades, le résultat de l'évaluation se substituant aux accolades

In [ ]:
```
# chaine avec caractères d'échappement
s = "César est réputé être l'auteur de la phrase \n\t\"veni, vidi, vici\""
print(s)
```

```
César est réputé être l'auteur de la phrase
        "veni, vidi, vici"
```

In [ ]:
```
# chaine brute retenant tous les caractères pour eux-mêmes
s= r"\n\t\""
print(s)
```

```
\n\t\"
```

In [ ]:
```
# chaine interpolée
a,b=1,2
s=f"le contenu de s est {a+b}"
print(s)
```

```
le contenu de s est 3
```

In [ ]:
```
# Conversion d'une valeur vers une chaine de caractères
s = str(42)
s
```

Out[ ]:
```
'42'
```

In [ ]:
```
len(s) # taille d'une chaine de caractères
```

Out[ ]:
```
2
```

L'extraction de sous-ensembles d'une chaine (ou *slicing*) s'effectue comme pour les listes :

- **caractère** : *chaine* **[** *position >=0 à partir du début et position <0 à partir de la fin (-1 = dernier élément)* **]**
- **Sous-chaine** : *chaine* **[** *position de début (0 si omise)* **:** *position de fin (exclue, jusqu'à la fin si omise)* **:** *pas (1 si omis)* **]**

In [ ]:
```
# création directe d'une chaine de caractères
s = 'Initiation à Python'
```

Caractère de `s`	`I`	`n`	`i`	`t`	`i`	`a`	`t`	`i`	`o`	`n`		`à`		`P`	`y`	`t`	`h`	`o`	`n`
Position à partir du début	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Position à partir de la fin	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Consultation d'une chaine de caractères :

In [ ]:
```
s[0] # extraction d'un caractère, le premier caractère est à la position 0
```

Out[ ]:
```
'I'
```

In [ ]:
```
s[:10]
```

Out[ ]:
```
'Initiation'
```

In [ ]:
```
s[-6:]
```

Out[ ]:
```
'Python'
```

Instruction	Description	Exemple
Créer une chaîne	Définit une chaîne avec des guillemets simples, doubles, ou triples.	s1 = 'Bonjour' s2 = "Python" s3 = '''Chaîne pouvant être multi-lignes''' print(s1, s2, s3) # Résultat : Bonjour Python Chaîne multi-lignes
Concaténer (mettre bout à bout) des chaînes	Combine plusieurs chaînes en une seule.	s1 = "Bonjour" s2 = "Python" s3 = s1 + " " + s2 print(s3) # Résultat : Bonjour Python
Répéter une chaîne	Multiplie une chaîne pour répéter son contenu.	s = "Python! " print(s * 3) # Résultat : Python! Python! Python!
Accéder à un caractère	Récupère un caractère en fonction de son index.	s = "Python" print(s[0]) # Résultat : P
Substituer une chaîne (f-string)	Insère des valeurs dans une chaîne avec une f-string.	nom = "Alice" age = 25 s = f"Bonjour {nom}, tu as {age} ans." print(s) # Résultat : Bonjour Alice, tu as 25 ans.
Vérifier si une sous-chaîne existe	Teste si une sous-chaîne est présente.	s = "Bonjour Python" print("Python" in s) # Résultat : True
Diviser une chaîne (`split()`)	Scinde une chaîne en une liste selon un séparateur.	s = "Alice,Bob,Charlie" names = s.split(",") print(names) # Résultat : ['Alice', 'Bob', 'Charlie']
Joindre une liste en chaîne (`join()`)	Combine les éléments d'une liste en une chaîne.	names = ['Alice', 'Bob', 'Charlie'] s = ", ".join(names) print(s) # Résultat : Alice, Bob, Charlie
Modifier la casse (`upper()` , `lower()` , etc.)	Convertit une chaîne en majuscules, minuscules, etc.	s = "Python" print(s.upper()) # Résultat : PYTHON print(s.lower()) # Résultat : python
Supprimer les espaces (`strip()`)	Supprime les espaces au début et à la fin.	s = " Python " print(s.strip()) # Résultat : Python
Remplacer des caractères (`replace()`)	Remplace un sous-ensemble par un autre.	s = "Bonjour Python" print(s.replace("Python", "le monde")) # Résultat : Bonjour le monde
Formater une chaîne (`format()`)	Insère des valeurs dans une chaîne.	s = "Bonjour {}, tu as {} ans.".format("Alice", 25) print(s) # Résultat : Bonjour Alice, tu as 25 ans.
Vérifier une condition (`startswith()` , `endswith()`)	Teste si une chaîne commence ou termine par une sous-chaîne.	s = "Bonjour Python" print(s.startswith("Bonjour")) # Résultat : True print(s.endswith("Python")) # Résultat : True
Longueur d'une chaîne (`len()`)	Retourne le nombre de caractères.	s = "Python" print(len(s)) # Résultat : 6
Inverser une chaîne	Retourne une chaîne dans l'ordre inverse.	s = "Python" print(s[::-1]) # Résultat : nohtyP
Vérifier le contenu (`isalpha()` , `isdigit()` , etc.)	Teste si une chaîne contient uniquement des lettres, des chiffres, etc.	s = "Python3" print(s.isalpha()) # Résultat : False print(s.isdigit()) # Résultat : False

Opérateurs Python agissant sur les chaines de caractères

Opérateur	Description	Exemple	Résultat
`+`	Concaténation de deux chaînes.	s1 = "Bonjour" s2 = "Python" result = s1 + " " + s2	Bonjour Python
`*`	Répétition de la chaîne.	s = "Hello " result = s * 3	Hello Hello Hello
`[]`	Indexation : Accès à un caractère par son index.	s = "Python" result = s[0]	P
`[start:stop]`	Tranchage : Extraction d'une sous-chaîne.	s = "Python" result = s[1:4]	yth
`in`	Vérifie si une sous-chaîne existe dans une chaîne.	s = "Bonjour Python" result = "Python" in s	True
`not in`	Vérifie si une sous-chaîne n'existe pas dans une chaîne.	s = "Bonjour Python" result = "Java" not in s	True
`==`	Vérifie si deux chaînes sont identiques.	s1 = "Python" s2 = "Python" result = s1 == s2	True
`!=`	Vérifie si deux chaînes sont différentes.	s1 = "Python" s2 = "Java" result = s1 != s2	True
`>` , `<`	Compare les chaînes en ordre lexicographique.	s1 = "apple" s2 = "banana" result = s1 < s2	True
`len()`	Retourne la longueur de la chaîne.	s = "Python" result = len(s)	6
`min()` , `max()`	Retourne le caractère minimum ou maximum (ordre ASCII).	s = "Python" min_char = min(s) max_char = max(s)	min_char = P, max_char = y

Exercice G

Décomposez le texte ci-dessous, présentant une imputation comptable en une liste de composants puis, à partir de ces élements, créez la phrase suivante dans laquelle les mentions en gras sont extraites du texte

*Le compte **512 Banque** a recu une imputation au **débit** d'un montant de **10000** euros le **15/01/2024***

In [ ]:
```
texte="15/01/2025|débit|512 Banque|10000"
```

5.4 - Ensembles (set)

- **Propriétés des ensembles :** sans ordre à priori, itérables (peuvent être parcourus de manière itérative élement par élement), mutables (les élements peuvent changer de valeur), peuvent contenir des types de données multiples
- composés d'éléments uniques, sans doublons (chaines de caractères, nombres ou tuples)

- similaires aux dictionnaires mais uniquement composés de clés sans valeur associées

Instruction	Description	Exemple
Créer un ensemble vide	Initialise un ensemble vide (en utilisant `set()`).	s = set() print(s) # Résultat : set()
Créer un ensemble avec des valeurs	Initialise un ensemble avec des éléments uniques.	s = {1, 2, 3} print(s) # Résultat : {1, 2, 3}
Ajouter un élément (`add()`)	Ajoute un élément à l'ensemble.	s = {1, 2} s.add(3) print(s) # Résultat : {1, 2, 3}
Supprimer un élément (`remove()`)	Supprime un élément existant (lève une erreur si l'élément n'existe pas).	s = {1, 2, 3} s.remove(2) print(s) # Résultat : {1, 3}
Supprimer un élément en toute sécurité (`discard()`)	Supprime un élément si présent, sans erreur sinon.	s = {1, 2, 3} s.discard(4) # Pas d'erreur print(s) # Résultat : {1, 2, 3}
Vider un ensemble (`clear()`)	Supprime tous les éléments de l'ensemble.	s = {1, 2, 3} s.clear() print(s) # Résultat : set()
Union (`union()` ou `|`)	Retourne un nouvel ensemble contenant tous les éléments de deux ensembles.	s1 = {1, 2} s2 = {2, 3} print(s1 | s2) # Résultat : {1, 2, 3}
Intersection (`intersection()` ou `&`)	Retourne les éléments communs entre deux ensembles.	s1 = {1, 2} s2 = {2, 3} print(s1 & s2) # Résultat : {2}
Différence (`difference()` ou `-`)	Retourne les éléments présents dans le premier ensemble mais pas dans le second.	s1 = {1, 2} s2 = {2, 3} print(s1 - s2) # Résultat : {1}
Différence symétrique (`symmetric_difference()` ou `^`)	Retourne les éléments présents dans un seul des ensembles.	s1 = {1, 2} s2 = {2, 3} print(s1 ^ s2) # Résultat : {1, 3}
Tester une inclusion (`issubset()`)	Vérifie si tous les éléments d'un ensemble sont dans un autre.	s1 = {1, 2} s2 = {1, 2, 3} print(s1.issubset(s2)) # Résultat : True
Tester une sur-inclusion (`issuperset()`)	Vérifie si un ensemble contient tous les éléments d'un autre.	s1 = {1, 2, 3} s2 = {2, 3} print(s1.issuperset(s2)) # Résultat : True
Tester une disjonction (`isdisjoint()`)	Vérifie si deux ensembles n'ont aucun élément en commun.	s1 = {1, 2} s2 = {3, 4} print(s1.isdisjoint(s2)) # Résultat : True
Copier un ensemble (`copy()`)	Crée une copie superficielle d'un ensemble.	s = {1, 2, 3} s_copy = s.copy() print(s_copy) # Résultat : {1, 2, 3}
Itérer sur un ensemble	Parcourt les éléments d'un ensemble.	s = {1, 2, 3} for item in s: print(item) # Résultat : 1, 2, 3 (ordre non garanti)

Exercice H

Enregistrez les composants d'indices par exécution du code ci-dessous puis répondez aux questions suivantes.

In [ ]:
```
cac_next_20={'FR0000120404-ACCOR','FR0010313833-ARKEMA','FR0013280286-BIOMERIEUX','FR0006174348-BUREAU VERITAS','FR0010908533-EDENRED','FR0010242511-EDF','FR0000130452-EIFFAGE',
cac_40={'FR0000120073-AIR LIQUIDE','NL0000235190-AIRBUS','FR0010220475-ALSTOM','LU1598757687-ARCELORMITTAL SA','FR0000051732-ATOS','FR0000120628-AXA','FR0000131104-BNP PARIBAS A
sbf_120={'FR0000120404-ACCOR','FR0010340141-ADP','FR0000031122-AIR FRANCE -KLM','FR0000120073-AIR LIQUIDE','NL0000235190-AIRBUS','FR0000060402-ALBIOMA','FR0013258662-ALD','FR001
```

1. Vérifiez que les indices CAC 40 et CAC NEXT 20 ne se recouvrent pas (aucune société n'appartient à la fois aux 2 indices)

In [ ]:
```

```

2. Affichez les sociétés de l'indice SBF 120 qui ne sont pas dans l'indice CAC 40

In [ ]:
```

```

3. Affichez la liste des sociétés présentes à la fois dans l'indice SBF 120 et dans au moins l'un des deux indices CAC (CAC 40 et CAC NEXT 20)

In [ ]:
```

```

5.5 - Tuples

Ils sont similaires aux listes, mais son immuables : on ne peut changer leur contenu (un élement ne peut changer de valeur et on ne peut ajouter ou retirer un élément) sans recréer un nouveau tuple.

Propriétés des tuples : peuvent être triés, itérables (peuvent être parcourus de manière itérative, c'est à dire élement par élément), immuables (leurs élements ne peuvent pas changer de valeur et toute modificaiton entraine la création d'une nouvelle instance), peuvent contenir des types de données différents. Comme les listes, mais leur contenu ne peut changer après une unique affectation

Opération	Description	Exemple
Créer un tuple	Initialise un tuple avec des valeurs.	t = (1, 2, 3) print(t) # Résultat : (1, 2, 3)
Créer un tuple vide	Initialise un tuple sans éléments.	t = () print(t) # Résultat : ()
Créer un tuple à un seul élément	Ajoute une virgule après l'élément unique.	t = (1,) print(t) # Résultat : (1,)
Accéder à un élément	Utilise l'index pour récupérer un élément.	t = (1, 2, 3) print(t[1]) # Résultat : 2
Accéder à un élément avec index négatif	Utilise des index négatifs pour compter depuis la fin.	t = (1, 2, 3) print(t[-1]) # Résultat : 3
Concaténer des tuples	Combine deux tuples en un seul.	t1 = (1, 2) t2 = (3, 4) t3 = t1 + t2 print(t3) # Résultat : (1, 2, 3, 4)
Répéter un tuple	Répète les éléments d'un tuple un certain nombre de fois.	t = (1, 2) print(t * 3) # Résultat : (1, 2, 1, 2, 1, 2)
Vérifier la présence d'un élément	Teste si un élément est présent dans un tuple.	t = (1, 2, 3) print(2 in t) # Résultat : True
Longueur d'un tuple (`len()`)	Retourne le nombre d'éléments dans un tuple.	t = (1, 2, 3) print(len(t)) # Résultat : 3
Index d'un élément (`index()`)	Retourne l'index de la première occurrence d'un élément.	t = (1, 2, 3, 2) print(t.index(2)) # Résultat : 1
Compter les occurrences (`count()`)	Retourne le nombre d'occurrences d'un élément dans un tuple.	t = (1, 2, 3, 2) print(t.count(2)) # Résultat : 2
Trancher un tuple	Extrait une sous-partie d'un tuple.	t = (1, 2, 3, 4) print(t[1:3]) # Résultat : (2, 3)
Déballer un tuple	Affecte les éléments du tuple à des variables distinctes.	t = (1, 2, 3) a, b, c = t print(a, b, c) # Résultat : 1 2 3
Itérer sur un tuple	Parcourt les éléments d'un tuple avec une boucle.	t = (1, 2, 3) for x in t: print(x) # Résultat : # 1 # 2 # 3
Tuple imbriqué	Crée un tuple contenant d'autres tuples.	t = ((1, 2), (3, 4)) print(t[1][0]) # Résultat : 3
Convertir une liste en tuple	Transforme une liste en un tuple avec `tuple()` .	lst = [1, 2, 3] t = tuple(lst) print(t) # Résultat : (1, 2, 3)
Convertir un tuple en liste	Transforme un tuple en une liste avec `list()` .	t = (1, 2, 3) lst = list(t) print(lst) # Résultat : [1, 2, 3]
Comparer des tuples	Compare deux tuples lexicographiquement.	t1 = (1, 2, 3) t2 = (1, 2, 4) print(t1 < t2) # Résultat : True

Exercice I

1. Créez un tuple nommé *option* contenant dans l'ordre les caractéristiques de l'option ci-dessous :

- Call
- Prix d'exercice de 100
- Echéance dans 0.5 année
- Modalité d'exercice américaine

In [ ]:

2. Transférez dans des variables nommées *sens*, *prix_exercice*, *echeance*, *modalite* les valeurs de la variable *option*

In [ ]:

6 - Structures de contrôle en Python

6.a - Tests en Python

Type de test	Syntaxe/Exemple	Description
Égalité	`x == y`	Vérifie si `x` est égal à `y` .
Différence	`x != y`	Vérifie si `x` est différent de `y` .
Inférieur	`x < y`	Vérifie si `x` est strictement inférieur à `y` .
Supérieur	`x > y`	Vérifie si `x` est strictement supérieur à `y` .
Inférieur ou égal	`x <= y`	Vérifie si `x` est inférieur ou égal à `y` .
Supérieur ou égal	`x >= y`	Vérifie si `x` est supérieur ou égal à `y` .
Appartenance	`x in seq`	Vérifie si `x` est présent dans la séquence `seq` .
Non-appartenance	`x not in seq`	Vérifie si `x` n'est pas présent dans la séquence `seq` .
Identité	`x is y`	Vérifie si `x` et `y` font référence au même objet.
Non-identité	`x is not y`	Vérifie si `x` et `y` ne font pas référence au même objet.
Type	`isinstance(x, type)`	Vérifie si `x` est une instance du type spécifié.
Négation logique	`not condition`	Inverse le résultat d'une condition.
Combinaison logique	`condition1 and condition2`	Vérifie si les deux conditions sont vraies.
	`condition1 or condition2`	Vérifie si au moins une des deux conditions est vraie.
Valeur nulle	`x is None`	Vérifie si `x` est `None` .
Vérification d'une liste vide	`not lst`	Vérifie si la liste `lst` est vide.

Type de test	Syntaxe/Exemple	Description
Type numérique	`x.isdigit()` (pour `str`)	Vérifie si `x` contient uniquement des chiffres.
Vérification d'une clé	`'clé' in dict`	Vérifie si une clé est présente dans un dictionnaire.
Vérification des booléens	`bool(x)`	Renvoie `True` si `x` est évalué comme vrai, sinon `False`.
Tests de plage	`a <= x <= b`	Vérifie si `x` est dans une plage (entre `a` et `b`, inclus).
Vérification d'une chaîne	`"substr" in string`	Vérifie si une sous-chaîne est présente dans une chaîne.
Tests personnalisés	`lambda x: x > 5`	Fonction anonyme pour effectuer un test spécifique.
Tests multiples	`all([cond1, cond2])`	Vérifie si toutes les conditions dans une liste sont vraies.
	`any([cond1, cond2])`	Vérifie si au moins une condition dans une liste est vraie.

6.b - Structures conditionnelles en python

Instruction	Syntaxe/Exemple	Description
`if`	x = 10 if x > 5: print("x est supérieur à 5") # Résultat : x est supérieur à 5	Exécute un bloc de code si la condition est vraie.

`if ... else`	x = 3
	if x > 5:
	  print("x est supérieur à 5")
	else:
	  print("x est inférieur ou égal à 5")
	# Résultat : x est inférieur ou égal à 5
		Exécute un bloc alternatif si la condition est fausse.

`if ... elif ... else`	x = 0
	if x > 0:
	  print("x est positif")
	elif x == 0:
	  print("x est nul")
	# Résultat : x est nul
	else:
	  print("x est négatif")
		Permet de tester plusieurs conditions de manière séquentielle.

Conditions imbriquées	x = 10
	if x > 5:
	  if x % 2 == 0:
	  print("x est supérieur à 5 et pair")
	# Résultat : x est supérieur à 5 et pair
	  else:
	  print("x est supérieur à 5 et impair")
		Permet de vérifier des sous-conditions dans une structure conditionnelle.

Opérateur ternaire	x = 5 message = "positif" if x > 0 else "négatif ou nul" print(message) # Résultat : positif	Permet une condition en une seule ligne avec une syntaxe concise.
Conditions combinées	x, y = 5, 10 if x > 0 and y > 0: print("x et y sont positifs") # Résultat : x et y sont positifs	Utilise `and`, `or`, ou `not` pour combiner plusieurs conditions.
Test d'appartenance	lettre = "a" if lettre in "abc": print("La lettre est dans la chaîne") # Résultat : La lettre est dans la chaîne	Vérifie si un élément appartient à une séquence avec `in` ou `not in`.
Test d'identité	x = None if x is None: print("x est None") # Résultat : x est None	Vérifie si deux objets sont identiques (même référence en mémoire) avec `is` ou `is not`.

7 - Eléments de procédures et fonctions en Python

7.a - Fonctions et procédure nommées

Définition élémentaire d'une procédure sans argument (ou paramètre) ni valeur retournée :

def *nom de la procedure* **() :**
`<retrait de 4 espaces>` **"""** *commentaire éventuel sur une ou plusieurs lignes décrivant la procédure et repris par la fonction* **help(***nom de la procédure***) """**
`<retrait de 4 espaces>` *instructions de la procédure*

Notez le symbole **:** en fin de la ligne du `def` est l'indentation (retrait des instructions) des lignes formant le corps de la procédure permettant de les rapporter à ce même `def`.

La syntaxe de définition d'une fonction est :

def *nom de la fonction* **(** *liste des éventuels paramètres ou arguments séparés par des virgules* **) :**
`<retrait de 4 espaces>` **"""** *commentaire éventuel sur une ou plusieurs lignes décrivant la fonction et repris par la fonction* **help(***nom de la procédure***) """**
`<retrait de 4 espaces>` *instructions de la fonction*
`<retrait de 4 espaces>` **return** *éventuelle valeur retournée par la fonction*

Les arguments de la liste peuvent être :

- un *identificateur* qui permettra de recevoir la valeur transmise et créera une variable locale permettant d'utiliser cette valeur transmise.
- un *identificateur* assorti d'une valeur par défaut de la forme *identificateur* **=** *valeur par défaut* . Dans ce cas la valeur par défaut sera utilisée si la valeur n'est pas fournie lors de l'appel de la fonction. Si la valeur par défaut est une expression, elle est uniquement évaluée lors e l'appel de la fonction.
- une liste variable d'arguments de la forme `*` *args* (vous pouvez remplacer *args* par le nom que vous désirez) qui seront traités comme des élements de *tuple* (dont devront être transmis par position)
- une liste variable d'arguments nommés de la forme `**` *kwargs* (vous pouvez remplacer *kwargs* par le nom que vous désirez) qui seront traités comme un des élements d'un *dictionnaire*.

Pour les fonctions à écrire, vous pouvez utiliser `pass` comme substitut au corps de la la fonction :

In [ ]:
```
def neFaitRien():
    pass
```

Mot-clé	Syntaxe/Exemple	Description
`def`	def addition(a, b): return a + b print(addition(2, 3)) # Résultat : 5	Utilisé pour définir une fonction.
`return`	def double(x): return x * 2 print(double(4)) # Résultat : 8	Permet de renvoyer une valeur depuis une fonction.
`lambda`	doubler = lambda x: x * 2 print(doubler(5)) # Résultat : 10	Définit une fonction anonyme en une seule ligne.

`global`	x = 10
	def modifier_global():
	  global x
	  x = 20
	modifier_global()
	print(x)
	# Résultat : 20
		Permet de modifier une variable définie à l'échelle globale depuis une fonction.

`nonlocal`	def fonction_externe():
	  x = 5
	  def fonction_interne():
	  nonlocal x
	  x = 10
	  fonction_interne()
	  print(x)
	fonction_externe()
	# Résultat : 10
		Permet de modifier une variable dans une fonction englobante (mais pas globale).

`*args` et `**kwargs`	def afficher(*args, **kwargs):
	  print("Args:", args)
	  print("Kwargs:", kwargs)
	afficher(1, 2, 3, nom="Python", version=3.10)
	# Résultat :
	# Args: (1, 2, 3)
	# Kwargs: {'nom': 'Python', 'version': 3.10}
		Permet de passer un nombre variable d'arguments positionnels (`args`) et nommés (`kwargs`).

Valeurs par défaut	def saluer(nom="inconnu"):
	  print(f"Bonjour, {nom} !")
	saluer() # Résultat : Bonjour, inconnu !
	saluer("Alice")
	# Résultat : Bonjour, Alice !
		Définit des valeurs par défaut pour les paramètres d'une fonction.

Annotations	def addition(a: int, b: int) -> int: return a + b print(addition(3, 4)) # Résultat : 7	Ajoute des annotations de type aux paramètres et au retour.

Exercice J

1. Créez une fonction qui reçoit un *montant*, un *nombre d'années* au terme desquelles ce montant sera placé ainsi qu'un *taux d'intérêt annuel* et retourne la valeur capitalisée correspondante. Vérifiez que la fonction retourne le résultat de 1 051.01 pour un montant de 1 000, un placement sur 5 ans et un taux d'intérêt de 1%.

In [ ]:

2. Créez une fonction qui recoit les mêmes élements que précédemment mais également une indication booléenne indiquant lorsqu'elle est à *True* que le taux d'intérêt est en composition continue. Cette indication aura la valeur *False* par défaut. Lorsque le taux est en composition continue, elle utilise la fonction *exp()* importée du package **math** pour calculer la valeur actuelle et sinon délègue le calcul à la fonction réalisée lors de la question préédente. Vérifiez que la fonction retourne pour un montant de 1 000, un placement sur 5 ans et un taux d'intérêt de 1% en composition continue le résultat de 1 051.27 (contre 1 051.01 en composition annuelle des intérêts).

In [ ]:

3. Créez une fonction qui demande les mêmes paramètres que la question 2 mais, en s'appuyant sur la fonction précédente, retourne à la fois la valeur capitalisée mais également la valeur des seuls intérêts calculés. Réceptionnez dans 2 variables les résultats de l'appel de la fonction pour un montant de 1 000, un placement sur 5 ans et un taux d'intérêt de 1% en composition continue.

In [ ]:

7.b - Fonctions anonymes ou fonctions Lambdas

Ces fonctions anonymes ou lambdas sont d'abord utilisées pour passer des fonctions en paramètre à d'autres fonctions, mais peuvent également être associées à un identifiant pour être appelées comme des fonctions. Les lambdas peuvent avoir des paramètres multiples. Une lambda peut également recevoir des paramètres de types conteneurs (listes,...) et peuvent retourner des valeurs de type conteneur(listes)

Leur syntaxe est :

lambda *liste de paramètres séparés par une vigule et sans parenthèses* **:** *expression unique retournant une valeur*

Différence entre les lambdas et les fonctions classiques :

- Une lambda n'est sont formée que d'une seule expression. Les fonctions traditionnelles sont donc mieux à même de présenter des problèmes complexes.
- Une lambda n'a pas d'instruction **return** car sa valeur retournée est toujours le résultat de l'expression qui la compose.

In [ ]:
```
# définition classique de fonction
def carré(x):
    return x**2
```

In [ ]:
```
# définition de fonction en lambda calcul
carré = lambda x: x**2
```

In [ ]:
```
# lambda avec 2 paramètres
(lambda x,y: x+y)(2,3)
```

Out[ ]: 5

Exemple : Tri d'une liste d'après une fonction classique et d'après une fonction lambda

In [ ]:
```
# utilisation classique d'une fonction transmise en paramètre d'une autre fonction
options = ['Option Européenne', 'Option Américaine', 'Option des Bermudes', 'Option Asiatique', 'Option à barrière']
def derniere_lettre(nom):
    return nom[-1]
sorted(options, key=derniere_lettre)
```

Out[ ]:
```
['Option Européenne',
 'Option Américaine',
 'Option Asiatique',
 'Option à barrière',
 'Option des Bermudes']
```

In [ ]:
```
# recours à une fonction anonyme ou lambda
sorted(options, key=lambda nom: nom[-1])
```

Out[ ]:
```
['Option Européenne',
 'Option Américaine',
 'Option Asiatique',
 'Option à barrière',
 'Option des Bermudes']
```

Exercice K

1. Créez une fonction lambda retournant un facteur d'actualisation $\frac{1}{(1+i)^d}$ à partir d'un taux d'intérêt *i* et d'une durée *d*. Vérifiez que cette fonction retourne 0.8638 pour *i*=5% et *d*=3.

In [ ]:
```

```

2. Créez une fonction lambda qui, recevant une chaine de caractère représentant titre boursier de la forme FR0000120404-ACCOR, retourne un tuple contenant successivement le code ISIN à 12 lettres ou chiffres présent au début, le nom de la société qui débute à partir du 13^e caractère, puis les 2 lettres identifiant le pays de cotation présentes au début de l'identifiant ISIN. Ainsi, pour `'FR0000120404-ACCOR'` elle doit retourner `('FR0000120404', 'ACCOR', 'FR')` et pour `'FR0000125007-SAINT GOBAIN'` doit retourner `('FR0000125007', 'SAINT GOBAIN', 'FR')`.

In [ ]:
```

```

3. Créez une fonction lambda qui calcule la valeur intrinsèque d'un call à partir de la valeur S_T à l'échéance du sous-jacent et de la valeur X son prix d'exercice. Vous utilisez pour cela un *Si ... Alors ... Sinon ...* sous la forme d'une opération ternaire (3 opérandes). Associez-la au nom *call* et testez-la pour des valeurs différentes de S_t et X.

$$valeur\ intrinsèque\ call = Max(0; S_T - X)$$

In [ ]:
```

```

8 - Instructions de boucles et structures itératives de programmation procédurale en python

En Python, la fonction `range` permet de générer une *séquence* de valeurs entières pouvant servir de support à une boucle contrôlée par un compteur :

Syntaxe :

- **range(** *fin* **)** retourne des valeurs de 0 à (*fin*-1)
- **range(** *début* **,** *fin* **)** retourne des valeurs de *début* à (*fin*-1)
- **range(** *début* **,** *fin* **,** *pas* **)** retourne des valeurs de *début* à (*fin*-1) espacées de *pas*

Boucle `for` :

Ce type de boucle sera utilisé pour exécuter des opérations un certain nombre de fois (connu au démarrage de la boucle) ou itérer élément par élément à partir d'un ensemble de valeurs.
Sa forme générale est :

for *variable_de_contrôle* **in** *liste* **:**
`retrait de 4 espace` *Instruction(s) à répéter pour chacune de valeurs de la liste* (la valeur courante étant dans lisible dans *variable_de_contrôle*)

In [ ]:
```
# boucle itérant sur les indices (positions)
actifs = ['action', 'obligation', 'option']
for i in range(len(actifs)):
    print(actifs[i].upper())
```

```
ACTION
OBLIGATION
OPTION
```

In [ ]:
```
# boucle itérant sur les valeurs
for actif in actifs:
    print(actif.upper())
```

```
ACTION
OBLIGATION
OPTION
```

In [ ]:
```
# itération sur des couples de valeurs (avec recours à la décomposition ou déballage de tuples)
obligation = {'nominal': 1000, 'maturité': 10, 'remboursement': 'in fine', 'taux de coupon': 0.05}
for clé, valeur in obligation.items():
    print(clé, valeur)
```

```
nominal 1000
maturité 10
remboursement in fine
taux de coupon 0.05
```

In [ ]:
```
# itération sur les valeurs mais utilisation de enumerate pour obtenir également les indices (positions)
for index, actif in enumerate(actifs):
    print(index, actif)
```

```
0 action
1 obligation
2 option
```

In [ ]:
```
# En python de nombreuses boucles for utilisent traditionnellement l'identificateur _ comme variable de contrôle.
# En cas de boucles imbriquées, chaque identificateur _ est associé à la variable de contrôle de la boucle courante.
for _ in range(3):
    print('itération',_,'de la première boucle :')
    for _ in range(3):
        print(_)
```

```
itération 0 de la première boucle :
0
1
2
itération 1 de la première boucle :
0
1
2
itération 2 de la première boucle :
0
1
2
```

Utilisation de `break` et `continue` :

`break` interrompt une boucle lorsqu'elle est exécutée

In [ ]:
```
for actif in actifs:
    if actif == 'option':
        break
    print (actif)
```

```
action
obligation
```

`continue` interrompt seulement l'itération courante pour passer à l'itération suivante

In [ ]:
```
for actif in actifs:
    if actif == 'obligation':
        continue
    print (actif)
```

```
action
option
```

Boucle `for` / `else` :

In [ ]:
```
for actif in actifs:
    if actif == 'obligation':
        print('obligation trouvée !')
        break    # quitte la boucle et évite la clause else
else:
    # ceci ne s'exécutera que si aucun break n'est exécuté dans la boucle for
    print("obligation introuvable")
```

```
obligation trouvée !
```

Boucle `while` : Ce type de boucle sera utilisé chaque fois que l'on ne connait pas le nombre d'itérations à réaliser au départ de la boucle. Dans ce cas on précisera une condition contrôlant le début et la continuation (ou pas) des itérations. Les instructions dans la boucle doivent avoir une incidence sur la condition pour éviter une boucle infinie.

Sa forme générale est :

while *condition_de_contrôle* **:**
`retrait de 4 espace` *Instruction(s) à répéter pour tant que la condition de contrôle est vérifiée.*

In [ ]:
```
compteur = 0
while compteur < 5:
    print('Ceci s\'affichera 5 fois')
    compteur += 1    # équivalent à  'compteur = compteur + 1'
```

```
Ceci s'affichera 5 fois
Ceci s'affichera 5 fois
Ceci s'affichera 5 fois
Ceci s'affichera 5 fois
Ceci s'affichera 5 fois
```

Exemples de boucles

Instruction	Syntaxe/Exemple	Description
`for`	for i in range(5): print(i) # Résultat : 0, 1, 2, 3, 4	Parcourt une séquence ou un itérable.

`while`	i = 0
	while i < 5:
	  print(i)
	# Résultat : 0, 1, 2, 3, 4
	  i += 1
		Exécute un bloc tant qu'une condition est vraie.

`break`	for i in range(5):
	  if i == 3:
	  break
	  print(i)
	# Résultat : 0, 1, 2
		Interrompt la boucle immédiatement.

`continue`	for i in range(5):
	  if i == 2:
	  continue
	  print(i)
	# Résultat : 0, 1, 3, 4
		Saute le reste du bloc de la boucle pour cette itération et passe à la suivante.

`else` avec `for` ou `while`	for i in range(5):
	  print(i)
	else:
	  print("Boucle terminée")
	# Résultat : 0, 1, 2, 3, 4, Boucle terminée
		S'exécute si la boucle se termine sans interruption (`break`).

Boucles imbriquées	for i in range(2):
	  for j in range(3):
	  print(f"i={i}, j={j}")
	# Résultat :
	# i=0, j=0
	# i=0, j=1
	# i=0, j=2
	# i=1, j=0
	# i=1, j=1
	# i=1, j=2
		Boucles à l'intérieur d'autres boucles, utilisées pour parcourir des structures complexes.

Itérations personnalisées	for char in "Python": print(char) # Résultat : P, y, t, h, o, n	Parcours d'une séquence non numérique, comme une chaîne ou une liste personnalisée.

Exercice L

1. Créez une fonction VAN qui reçoit un taux d'actualisation et une liste de tuples (définis dans la variable cashflows ci-dessous) contenant chaque fois un cash-flow et sa maturité puis retourne la somme des cash-flows actualités. Vérifiez à l'aide de cette fonction que des cashflows ci-dessous résulte une valeur actuelle nette 29

In [ ]:
```
cashflows=[(-1000,0),(350,1),(350,2),(350,3)]
```

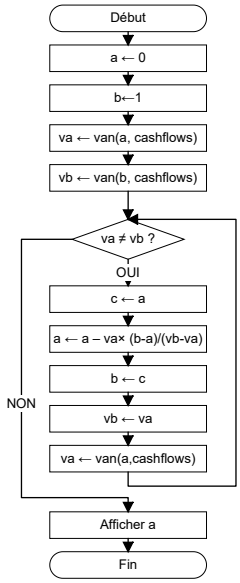
2. Affichez les différentes valeurs de la VAN pour les mêmes cashflows et en faisant successivement varier le taux de 1% dans l'intervalle de 0% à 15%. La boucle peut utiliser une séquence entière sachant que les éléments de cette séquence peuvent être ensuite divisés par 100 pour former des pourcentages.

In [ ]:
```

```

3. En partant d'un taux de 2%; augmentez progressivement ce taux d'un point de base (un centième de pourcentage) jusqu'à ce que la valeur actuelle nette des cashflows devienne nulle ou négative. Affichez la valeur du taux correspondant à la première valeur actuelle nette nulle ou négative. Vous devriez obtenir un taux de 2,48%.

In [ ]:
```

```

4. En raison des propriétés algébriques de la fonction VAN, il est possible de trouver son zéro, c'est à dire le taux actuariel, par la méthode des sécantes.
   Déterminez le taux actuariel, compris entre 0 et 100% en appliquant l'algorithme suivant dans lequel *a* et *b* sont les taux qui encadrent le taux actuariel et vont progressivement se rapprocher de celui-ci jusqu'à se rejoindre (pile sur ce taux actuariel). Vous devriez obtenir la valeur 0,024797547619221167 soit 2,479%.



In [ ]:

9 - Instructions de calcul itératif issus de la programmation fonctionnelle en Python

9.a - Conteneurs définis en compréhension

Ce sont des conteneurs dont le contenu est défini par filtrage du contenu d'une autre conteneur (ou séquence générée par `range`) selon un principe analogue à celui de la définition en compréhension de la théorie des ensembles. Cette construction syntaxique très compacte se distingue de la construction la plus courante dans les langages de programmation qui est de définir un conteneurs par énumération de ses éléments.

Listes définies en comprehension :

Méthode traditionnelle : une boucle **for** génére une liste de valeurs au cube

In [ ]:
```
nombres = [1, 2, 3, 4, 5]
cubes = []
for n in nombres:
    cubes.append(n**3)
cubes
```

Out[ ]: [1, 8, 27, 64, 125]

Liste équivalente par définition en compréhension

Syntaxe : **[** *expression* **for** *variable* **in** *conteneur* **]**

In [ ]:
```
cubes = [n**3 for n in nombres]
cubes
```

Out[ ]: [1, 8, 27, 64, 125]

Méthode traditionnelle : une boucle **for** génère les cubes de nombre pairs

In [ ]:
```
cubes_de_valeurs_paires = []
for n in nombres:
    if n % 2 == 0:
        cubes_de_valeurs_paires.append(n**3)
cubes_de_valeurs_paires
```

Out[ ]: [8, 64]

Liste équivalente par définition en compréhension

> Syntaxe: **[** *expression* **for** *variable* **in** *conteneur* **if** *condition* **]**

In [ ]:
```
cubes_de_valeurs_paires = [n**3 for n in nombres if n % 2 == 0]
cubes_de_valeurs_paires
```

Out[ ]: [8, 64]

Méthode traditionnelle : une boucle for elève au cube les nombres pairs et au carré les nombres impairs

In [ ]:
```
cubes_and_carrés = []
for n in nombres:
    if n % 2 == 0:
        cubes_and_carrés.append(n**3)
    else:
        cubes_and_carrés.append(n**2)
cubes_and_carrés
```

Out[ ]: [1, 8, 9, 64, 25]

Liste équivalente par définition en compréhension

> Syntaxe : **[** *calcul en cas de condition vérifiée* **if** *condition* **else** *calcul si condition non vérifiée* **for** *variable* **in** *conteneur* **]**

In [ ]:
```
cubes_and_carrés = [n**3 if n % 2 == 0 else n**2 for n in nombres]
cubes_and_carrés
```

Out[ ]: [1, 8, 9, 64, 25]

Méthodes traditionnelle : boucles **for** imbriquées pour applatir une matrice exprimée en 2 dimension

In [ ]:
```
matrice = [[1, 2], [3, 4]]
elements = []
for ligne in matrice:
    for valeur in ligne:
        elements.append(valeur)
elements
```

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

Liste équivalente par définition en compréhension

> Syntaxe : **[** *expression utilisant élément* **for** *sous-conteneur* **in** *conteneur* **for** *élément* **in** *sous-conteneur* **]**

In [ ]:
```
elements = [valeur for ligne in matrice for valeur in ligne]
elements
```

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

Des listes par compréhesion peuvent être imbriquée, ici pour retourner le double de toutes les valeurs contenues dans la matrice

In [ ]:
```
elements= [[valeur*2 for valeur in ligne] for ligne in matrice]
elements
```

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

Des listes en compréhensions de valeurs logiques peuvent être transmises aux fonctions *any()* et *all()* pour tester globalement des propriétés concernant les éléments d'une liste.

In [ ]:
```
any( valeur%2==0 for valeur in range(1,4)) # la liste [1,2,3] contient-elle une valeur paire ?
```

Out[ ]:
```
True
```

In [ ]:
```
all( valeur%2!=0 for valeur in range(1,4)) # la liste [1,2,3] contient-elle uniquement des valeurs impaires ?
```

Out[ ]:
```
False
```

Ensembles définis en compréhension :

In [ ]:
```
actifs = ['action', 'obligation', 'option']
tailles_uniques_noms_actifs = {len(actif) for actif in actifs}
tailles_uniques_noms_actifs
```

Out[ ]:
```
{6, 10}
```

Dictionaires définis en compréhension :

In [ ]:
```
tailles_noms_actifs = {actif:len(actif) for actif in actifs}
tailles_noms_actifs
```

Out[ ]:
```
{'action': 6, 'obligation': 10, 'option': 6}
```

In [ ]:
```
indices_actifs = {actif:index for index, actif in enumerate(actifs)}
indices_actifs
```

Out[ ]:
```
{'action': 0, 'obligation': 1, 'option': 2}
```

Exercice M

1. Vérifiez en une instruction incluant une liste en compréhension le fait que $\sum_{i=1}^{9} \frac{i}{10^i} = 0,123456789$

In [ ]:
```

```

2. Créez une fonction qui recevant un taux de coupon et une maturité exprimée sous la forme d'un nombre entier d'années retourne la liste des flux annuels que produirait une obligation à coupons annuels et à remboursement *in fine* au pair. Cette liste sera définie en compréhension.

> Pour un taux de coupon de 3.5% et une maturité de 10 ans, votre fonction retournera les flux : `[0.035, 0.035, 0.035, 0.035, 0.035, 0.035, 0.035, 0.035, 0.035, 1.035]`

In [ ]:
```
def flux_obligataires(taux_coupon,maturite):
   pass # replacez pass par un return suivi de votre liste définie en compréhension
flux_obligataires(0.035,10)
```

3. Créez une fonction plus complète qui reçoit en outre un taux d'actualisation fixe et retourne les flux actualisés à la place des flux bruts.

> Pour un taux de coupon de 3.5%, une maturité de 10 ans et un taux d'actualisation de 1%, votre fonction retournera les flux : `[0.034653465346534656, 0.034310361729242234, 0.03397065517746756, 0.033634312056898576, 0.033301299066236204, 0.032971583233897234, 0.03264513191474974, 0.03232191278688093, 0.03200189384839696, 0.9369719981072376]`

In [ ]:
```

```

4. En modifiant la liste de la question 2, générez un dictionnaire dont les clés sont les maturités des cash-flows et les valeurs les montants associés.

> Pour un taux de coupon de 3.5%, une maturité de 10 ans et un taux d'actualisation de 1%, votre fonction retournera les associations : `{1: 0.035, 2: 0.035, 3: 0.035, 4: 0.035, 5: 0.035, 6: 0.035, 7: 0.035, 8: 0.035, 9: 0.035, 10: 1.035}`

In [ ]:
```

```

5. En utilisant tant les clés (maturités) que les valeurs (cash flows actualisés) du dictionnaire précédant, calculez la duration des flux qui n'est autre que le moyenne des maturités (clés du dictionnaire) pondérée par la valeur actuelle des cashflows correspondants (valeurs du dictionnaire).

> Soit n le nombre de cashflows à recevoir dans le futur :
>
> $$duration = \frac{\sum_{i=1}^{n} maturités_i \times valeurActuelle(cashflow_i)}{\sum_{i=1}^{n} valeurActuelle(cashflows_i)}$$

In [ ]:
```

```

5. A partir de la liste sbf_120 ci-dessous, retournez la liste (type *list*) des seules actions cotées à Paris (dont l'ISIN débute par `FR`). A l'opposé, générez l'ensemble (type *set*) des actions non cotées à Paris.

In [ ]:
```
sbf_120=['FR0000120404-ACCOR','FR0010340141-ADP','FR0000031122-AIR FRANCE -KLM','FR0000120073-AIR LIQUIDE','NL0000235190-AIRBUS','FR0000060402-ALBIOMA','FR0013258662-ALD','FR001
```

6. A partir de liste *sbf_120*, créez une dictionnaire défini en compréhension qui associe à chaque n°ISIN le nom de la société correspondante.

In [ ]:
```
sbf_120=['FR0000120404-ACCOR','FR0010340141-ADP','FR0000031122-AIR FRANCE -KLM','FR0000120073-AIR LIQUIDE','NL0000235190-AIRBUS','FR0000060402-ALBIOMA','FR0013258662-ALD','FR001
```

9.b - Application de fonctions de transformation par `map`, de conditions par `filter`, de synthèse par `reduce`

Il s'agit d'une autre forme de structure itérative, inspirée de la programmation fonctionnelle.

`map` applique une fonction à chaque élément d'un conteneur ou d'une séquence et retourne un itérateur, lequel contrôle une boucle ou génère une liste par conversion.

In [ ]:
```
actifs = ['Actions', 'Obligations', 'Options']
list(map(len, actifs))
```

Out[ ]:
```
[7, 11, 7]
```

In [ ]:
```
# Liste équivalente par définition en compréhension
[len(nom) for nom in actifs]
```

Out[ ]:
```
[7, 11, 7]
```

In [ ]:
```
list(map(lambda nom: nom[-1], actifs))
```

Out[ ]:
```
['s', 's', 's']
```

In [ ]:
```
# Liste équivalente par définition en compréhension
[nom[-1] for nom in actifs]
```

Out[ ]:
```
['s', 's', 's']
```

`filter` retourne un itérateur, pouvant générer une liste par conversion, retournant les éléments d'un conteneur ou d'une séquence pour lesquelles une condition est vérifiée.

In [ ]:
```
nombres = range(5)
list(filter(lambda x: x % 2 == 0, nombres))
```

Out[ ]:
```
[0, 2, 4]
```

In [ ]:
```
# Liste équivalente par définition en compréhension
[n for n in nombres if n % 2 == 0]
```

Out[ ]:
```
[0, 2, 4]
```

`reduce` du module **functools** permet d'appliquer une fonction à un ensemble en vue d'obtenir un résultat synthétique (unique) à partir de ce dernier. Elle procède par accumulation et prend une fonction à 2 arguments : *valeur*, *accumulateur*. Lors du premier appel (pour le premier élément), *accumulateur* vaudra la valeur d'initialisation passée en 3eme argument de l'appel de *reduce* (**None** si ce dernier n'est pas fourni) et pour les appels successifs *accumulateur* recevra la valeur retournée par la fonction lors du traitement de l'élément précédent. La fonction *reduce* renverra la valeur retournée par la fonction à l'issue de l'appel pour le dernier élément de l'ensemble.

In [ ]:
```
import functools
print(functools.reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])) # Accumulation par addition, sans valeur de départ (qui sera de fait None de valeur numérique nulle)
print(functools.reduce(lambda x, y: x+y, [1, 2, 3, 4, 5],10)) # Accumulation par addition, valeur initiale à 10.
```

```
15
25
```

Exercice N

1. A partir de la liste *sbf_120* ci-dessous, générez la liste des actions non cotées à Paris (c'est à dire ne débutant pas par les lettres FR) par la commande *filter* et une fonction *lambda* adaptée.

In [ ]:
```
sbf_120=['FR0000120404-ACCOR','FR0010340141-ADP','FR0000031122-AIR FRANCE -KLM','FR0000120073-AIR LIQUIDE','NL0000235190-AIRBUS','FR0000060402-ALBIOMA','FR0013258662-ALD','FR001
```

2. A partir de la liste sbf_120, générez par une fonction *map* et une fonction *lambda* adaptée une liste formée des seuls n° ISIN.

In [ ]:
```

```

3. Par application conjointe de *filter*, *map* et *reduce*, affichez le nom d'action le plus court (ayant le moins de caractères une fois l'ISIN retiré) coté uniquement sur la place de Paris

In [ ]:
```

```

9.c - Générateurs

Un *générateur* est une fonction permettant de générer des listes élément par élement. Le mot clé **yield** permet de définir l'élément à retourner lors d'une itération.

Un générateur peut à tout moment être converti en une liste des éléments générés par l'instruction **list(*générateur*)** mais peut être utilisé à la place d'une liste pour contrôler une boucle par exemple.

L'intérêt d'un générateur est sa faible occupation mémoire car chaque élement est généré uniquement lorsqu'il est nécessaire en comparaison d'une liste qui stocke la totalité des éléments.

In [ ]:
```
def zeros(nombre):
    """Générateur produisant successivement autant de zéros qu'indique le
    paramètre nombre"""
    i=0
    while i<nombre:
        yield 0 # yield est suivi par la valeur de l'élément à générer
        i+=1

for _ in zeros(5):
    print(_)
print(list(zeros(10)))
```

```
0
0
0
0
0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Les générateurs complètent les listes définies en compréhension d'un dispositif analogue (que nous appelerons *expression génératrice*) n'ayant pas recours au mot clé `yield`, se distinguant uniquement par l'usage syntaxique de parenthèses plutôt que de crochets pour encadrer la formule de génération.

In [ ]:
```
liste_en_comprension = [x ** 2 for x in range(10) if x % 2 == 0]
print(liste_en_comprension)
expression_generatrice = (x ** 2 for x in range(10) if x % 2 == 0)
print( list(expression_generatrice) )
```

```
[0, 4, 16, 36, 64]
[0, 4, 16, 36, 64]
```

Exercice O

1. Créez une expression génératrice qui produit des taux compris entre 5 et 25%.

In [ ]:

2. Créez un générateur de nombres aléatoires en distribution normale suivant la méthode de Box-Mueller, en écriture polaire présentée par Marsaglia et Bray.

Ce générateur recevra en paramètre l'effectif de valeurs aléatoires à produire.

On tire un couple de variables (X, Y) sachant les variables X et Y tirées au hasard uniformément et indépendamment sur le segment [-1,1] puis on calcule $U = X^2 + Y^2$. Le couple de coordonnées (X, Y) doit être dans le disque unité et on rejette X et Y pour reprendre les tirages tant que $U \geq 1$ ou $U = 0$.

On obtient alors $Z_0 = X \cdot \sqrt{\frac{-2 \ln U}{U}}$ et $Z_1 = Y \cdot \sqrt{\frac{-2 \ln U}{U}}$ qui sont deux variables aléatoires indépendantes suivant une loi normale centrée réduite que l'on peut retourner.

Le package **random** vous met à disposition la fonction *uniform(a,b)* pour obtenir un nombre aléatoire en distribution uniforme dans l'intervalle $[a; b]$. Le package **math** vous met à disposition les fonctions *log()* pour les logarithmes et *sqrt()* pour la racine carrée. Pour votre information indépendante de l'exercice, le package **random** vous propose également une fonction produisant directement des nombres aléatoires suivant une distribution normale. De même, le module **statistics** met à disposition une fonction *NormalDist(moyenne, écart-type).samples(nombre, seed=graine éventuelle pour reproductibilité du tirage)* pour générer un échantillon aléatoire.

In [ ]:

3. A partir d'1 million de tirages du générateur aléatoire, vérifiez à l'aide des fonctions *fmean()* et *stdev()* proposées par la package **statistics** que la moyenne de cet échantillon est proche de 0 et son écart-type proche de 1.

In [ ]:

4. Nous utiliserons le générateur aléatoire précédent pour nourrir les valeurs successives de la variable $\tilde{x}$ correspondant à un tirage aléatoire en distribution normale.
   Soit un sous-jacent de valeur courante S_0=50 et de volatilité annuelle σ=20%, une option de type *call européen* d'échéance T=1 et de prix d'exercice X=50 puis un taux sans risque en composition continue r=10% par an, calculez pour chaque tirage de $\tilde{x}$ une valeur du sous-jacent à l'échéance correspondante selon une évolution log-normale pour en déduire la valeur intrinsèque du call à l'échéance.
   L'évaluation *Monte-Carlo* du call européen n'est autre que l'actualisation au taux sans risque de la moyenne des valeurs intrinsèques obtenues.
   Soit n le nombre de tirages :

$$Estimation\ du\ call = \frac{\mathrm{e}^{-r \times T}}{n} \times \sum_{i=1}^{n} \max\left[0\ ;\ S_0 \times \exp\left(\left(r - \frac{1}{2} \times \sigma^2\right) \times T + \sigma \times \sqrt{T} \times \tilde{x}\right) - X\right]$$

   En faisant progresser n, demandez des évaluations de plus en plus précises du call, sachant qu'une évaluation Black-Scholes retourne 6,634838292 pour les paramètres courants.
   Le package **math** vous met à disposition la fonction *exp(x)* pour le calcul de l'exponentielle.

In [ ]:
```
S0,sigma,T,X,r=50,0.2,1,50,0.1
n=10_000_000
```

In [ ]:

10 - Gestion des erreurs d'exécution en Python

Outil	Description	Exemple

`try` / `except`	Capture une exception pour empêcher l'arrêt du programme.	try:
		  x = 10 / 0
		except ZeroDivisionError:
		  print("Division par zéro non autorisée.")
		# Résultat : Division par zéro non autorisée.

`try` / `except` / `else`	Exécute un bloc `else` si aucune exception n'est levée.	try:
		  x = 10 / 2
		except ZeroDivisionError:
		  print("Division par zéro non autorisée.")
		else:
		  print("Division réussie :", x)
		# Résultat : Division réussie : 5.0

`try` / `except` / `finally`	Le bloc `finally` s'exécute toujours, qu'une exception ait été levée ou non.	try:
		  x = 10 / 0
		except ZeroDivisionError:
		  print("Erreur détectée.")
		finally:
		  print("Bloc finally exécuté.")
		# Résultat :
		# Erreur détectée.
		# Bloc finally exécuté.

`raise`	Lève une exception manuellement.	x = -1 if x < 0: raise ValueError("x ne peut pas être négatif.") # Résultat : ValueError: x ne peut pas être négatif.
`assert`	Vérifie une condition et lève une `AssertionError` si la condition est fausse.	x = 10 assert x > 0, "x doit être positif." # Résultat : Aucun, car la condition est vraie.

Créer une exception personnalisée	Définit une classe pour gérer des erreurs spécifiques.	class CustomError(Exception):
		  pass
		try:
		  raise CustomError("Erreur personnalisée.")
		except CustomError as e:
		  print(e)
		# Résultat : Erreur personnalisée.

Capturer plusieurs exceptions	Utilise un tuple pour gérer plusieurs types d'exceptions.	try:
		  x = int("abc")
		except (ValueError, TypeError):
		  print("Une erreur de conversion a eu lieu.")
		# Résultat : Une erreur de conversion a eu lieu.

Capturer l'exception avec détail	Utilise le mot-clé `as` pour accéder à l'objet exception.	try:
		  x = 10 / 0
		except ZeroDivisionError as e:
		  print("Erreur :", e)
		# Résultat : Erreur : division by zero

`try` imbriqués	Gère des erreurs dans des blocs imbriqués.	try:
		  try:
		  x = 10 / 0
		  except ZeroDivisionError:
		  print("Erreur interne.")
		  raise
		except Exception as e:
		  print("Erreur capturée :", e)
		# Résultat :
		# Erreur interne.
		# Erreur capturée : division by zero

Exercice P

Dans une boucle qui perdure jusqu'à la saisie du mot `fin`, utilisez la fonction **input** pour demander une saisie à l'utilisateur. A chaque itération, dès lors que la saisie peut être convertie en un nombre affichez la valeur correspondante sinon affichez le texte « *Pas de conversion numérique possible* »

In [3]: