**Boucles et dictionnaires**

**Boucles For et dictionnaires**

Nous pouvons parcourir à la fois les clés et les valeurs d’un dictionnaire.

my\_books = {

"title": "Harry Potter",

"author": "JK Rowling",

}

for x, y in my\_books.items():

print("the" + x + "is" + y)

>> the title is Harry Potter

the author is JK Rowling

**Opérateur de boucles**

* range(start, stop[, step]) : itérateur dans les boucles.

print(list(range(1, 10, 2)))

>> [1, 3, 5, 7, 9]

* enumerate(iterable) : énumérer chaque élément de l'itérable

for item in enumerate('abcd'):

print(item)

>>

(0, 'a') # Syntax : (index , value)

(1, 'b')

(2, 'c')

(3, 'd')

for (index\_count, letter) in enumerate('abcd'):

print('At index {} the letter is {}'.format(index\_count, letter))

>>

At index 0 the letter is a

At index 1 the letter is b

At index 2 the letter is c

At index 3 the letter is d

* zip(itérable,..) : concat [itérables, …] dans un tuple.

list1 = [1,2,3]

list2 = ['a','b','c']

list3 = [1.1, 2.2, 3.3, 4.4, 5.5]

for item in zip(list1, list2, list3): # only go as far it is possible

print(item)

>>

(1, 'a', 1.1)

(2, 'b', 2.2)

(3, 'c', 3.3)

**Pour le reste**

La partie else est facultative. Lorsqu'elle est incluse, elle est toujours exécutée une fois la boucle for terminée, sauf si une instruction break est rencontrée.

for i in range(1, 3):

print(i)

else:

print('The for loop is over')

>>

1

2

The for loop is over

**Tandis que d'autre part**

while some\_condition:  
# faire  
autre chose:  
# faire une autre chose

x = 0

while x < 2:

print(f'x is {x}')

x += 1

else:

print('x is bigger than 2')

>>

x is 0

x is 1

x is bigger than 2

**Pause, Continuer, Passer**

* break : interrompre la boucle (arrêter l'exécution d'une instruction en boucle)  
  si vous sortez d'une boucle for ou while, tout autre bloc de boucle correspondant n'est pas exécuté.

for letter in 'Leonardo':

if letter == 'a':

break

print(letter, end='') # end='' renders each letter next to the other

>> Leon

while True:

s = input('Enter something : ')

if s == 'quit':

break

print('Length of the string is', len(s))

print('Done')

* continuer : revenir au début de la boucle (continuer à l'itération suivante de la boucle)

for letter in 'Leonardo':

if letter == 'o':

continue

print(letter, end='') # dont execute for 'o' letter

>> Lenard

while True:

s = input('Enter something : ')

if s == 'quit':

break

if len(s) < 3:

print('Too small')

continue

print('Input is of sufficient length')

* passer : ne rien faire

for item in [1,2,3]:

# comment

pass # to avoid the error

print('Finish the script')

>> Finish the script

**🚀 Défi**

Implémentez une boucle demandant à l'utilisateur d'ajouter des entrées au dictionnaire jusqu'à ce que « quit » soit saisi. Affichez le dictionnaire final.

**Conclusion**

Grâce à cette leçon, vous avez appris à combiner efficacement boucles et dictionnaires pour gérer et manipuler des données de manière dynamique. Ces compétences vous aideront à créer des applications Python plus robustes et interactives, capables de gérer efficacement diverses opérations de données.

**Boucles**

**Boucles**

Les boucles sont un concept de programmation fondamental qui vous permet d'itérer sur les éléments d'une liste ou de répéter une action donnée un nombre défini ou même infini de fois.

Utilisées conjointement avec des listes, les boucles permettent d'accéder successivement à chaque élément de la liste. Ce processus est appelé **« itération »** .

En programmation informatique, une *boucle* est une séquence d'instructions qui est répétée continuellement jusqu'à ce qu'une condition spécifique soit atteinte.

Il existe deux principaux types de boucles : forles boucles et whileles boucles.

Voir la documentation officielle [ici](https://docs.python.org/3/tutorial/controlflow.html#for-statements)

**Boucles for**

Les boucles For sont principalement utilisées avec les listes : vous parcourez chaque élément de la liste dans l'ordre ; elle s'arrête automatiquement une fois qu'elle atteint la fin de la liste.

Les boucles For commencent par le mot-clé for. Vient ensuite ce que l'on appelle un « itérateur », une variable dont la valeur change à chaque tour de boucle.

Nous utiliserons cette variable dans la boucle comme valeur actuelle pour une itération donnée.

Tout d’abord, la syntaxe :

for <variable\_name> in <sequence\_name>:

CODE

CODE

Notez l’utilisation des deux points :et l’indentation du code.

**Exemple**

fruits = ['apple', 'banana', 'kiwi', 'pear']

for fruit in fruits:

print(fruit)

>> apple

>> banana

>> kiwi

>> pear

Dans la forboucle ci-dessus, fruit(singulier) est notre itérateur. Ses valeurs changent à chaque tour de boucle. Il est successivement : 'apple', 'banana', 'kiwi'et enfin 'pear'.

**Un autre exemple**

cities = ["London", "San Francisco", "Paris", "Barcelona"]

for city in cities:

print("I once went to", city)

La première ligne de la boucle est :

for city in cities:

Cette ligne indique à Python de récupérer la première valeur de la liste citieset de la stocker dans la variable city.

Vous pouvez choisir le nom que vous souhaitez pour cette variable ; cependant, il est utile de choisir un nom significatif représentant un seul élément de la liste.

La première valeur est « Londres ». Python lit ensuite la ligne suivante :

print("I once went to", city)

Python affiche la valeur de I once went to London. Comme la liste contient plus de valeurs, il revient à la première ligne.

for city in cities:

Python récupère maintenant l'élément suivant de la liste et le stocke dans la variable city(sa valeur est écrasée).

print("I once went to", city)

Maintenant, Python va imprimer I once went to San Francisco.

Python continue ainsi jusqu'à ce qu'il atteigne la fin de la liste.

**Gammes**

Les plages vous permettent de générer une séquence de nombres d'une valeur de départ à une valeur de fin.

Pour créer une plage, utilisez la rangefonction et transmettez-lui une valeur maximale et une valeur minimale. **La dernière valeur est non inclusive,** ce qui signifie que si vous parcourez une plage, elle ne sera pas incluse.

**Syntaxe**

numbers = range(4, 19)

>> range(4, 19)

**Exemple d'itération**

numbers = range(4, 19)

for number in numbers:

print(number)

>> 4

>> 5

>> 6

>> ...

>> 16

>> 17

>> 18

Pour imprimer des nombres de 1 à 5, utilisezrange(1,6)

Bien qu'il puisse être itéré, le **résultat de range n'est pas unlist** .

Pour le stocker dans un list, vous pouvez convertir les résultats de range()directement en un à listl'aide de la list()fonction. Lorsque vous encapsulez list()un appel à la range()fonction, le résultat sera une liste de nombres.

numbers = list(range(1,6))

print(numbers)

**Certaines fonctions intégrées qui utilisent des boucles for**

Des fonctions comme min, maxou sumutilisent des boucles for en coulisses ; pour vous entraîner, vous pouvez essayer de les programmer vous-même.

**Éviter les erreurs d'indentation**

Python utilise l'indentation pour déterminer quand une ligne de code est connectée à la ligne au-dessus.

Dans les exemples précédents, les lignes qui imprimaient des messages sur chaque ville faisaient partie de la forboucle car elles étaient en retrait.

L’utilisation de l’indentation par Python rend le code très facile à lire.

**Exercice**

Accepter un nombre de l'utilisateur et imprimer sa table de multiplication

**Boucles while**

Bien que les boucles diffèrent des forboucles classiques, elles possèdent ce que l'on appelle un cycle d'itération indéfini. Cela signifie que le nombre d'exécutions n'est pas défini par la longueur d'une liste donnée, mais par un point d'arrêt spécifique qui interrompt l'exécution de la boucle lorsqu'il est atteint.

Une whileboucle est définie de manière similaire à une instruction conditionnelle. Elle s'exécute tant que la condition est remplie Trueet s'arrête lorsque cette condition est évaluée à False.

La whileboucle s'exécute tant qu'une certaine condition est vraie.

Il exécutera toutes les instructions de la boucle, puis testera la condition, si la condition est toujours True, il réexécutera la boucle, et encore jusqu'à ce que la condition devienne fausse.

**Exemple**

current\_number = 1

while current\_number <= 5:

print(current\_number)

current\_number += 1

print("Finished")

>> 1

>> 2

>> 3

>> 4

>> 5

Dans la première ligne, nous commençons à compter à partir de 1 en définissant la valeur de current\_numberà 1.

La boucle while est configurée pour continuer à fonctionner tant que la valeur de current\_numberest inférieure ou égale à 5.

Le code à l'intérieur de la boucle imprime la valeur de current\_numberpuis ajoute 1à cette valeur avec current\_number += 1. (L' +=opérateur est l'abréviation de current\_number = current\_number + 1.)

Python répète la boucle tant que la condition current\_number <= 5est True. Parce 1que est inférieur à 5, Python imprime 1puis ajoute 1, ce qui donne current\_number2. Parce que 2est inférieur à 5, Python imprime 2et ajoute 1à nouveau, ce qui donne current\_number3, et ainsi de suite.

Une fois que la valeur de current\_numberest supérieure à 5, la boucle s'arrête et le programme se termine.

**Exemple**

password = ''

while password != 'hello-world-123':

password = input('What is the top secret password? ')

print('You guessed the right password!')

**Exercice**

Imprimez les nombres de 1 à 10 en utilisant la boucle while

**Boucle infinie**

Certaines conditions seront toujours remplies True, par exemple si nous exécutons ce code :

while 1 == 1:

print("Looping...")

Cela ne s'arrêtera jamais car 1 est toujours égal à 1. On peut aussi utiliser while 1:ou while True:.

**Utiliser un drapeau**

Qu'en est-il des programmes plus complexes dans lesquels de nombreux événements différents pourraient entraîner l'arrêt de l'exécution du programme ?

Pour un programme qui ne doit s'exécuter que tant que de nombreuses conditions sont remplies True, vous pouvez définir une variable qui détermine si le programme entier est actif. Cette variable, appelée *indicateur* , agit comme un signal envoyé au programme.

Nous pouvons écrire nos programmes de manière à ce qu'ils s'exécutent lorsque l'indicateur est défini sur , Trueet s'arrêtent lorsqu'un événement parmi plusieurs définit la valeur de l'indicateur sur False. Par conséquent, notre whileinstruction globale ne doit vérifier qu'une seule condition : si l'indicateur flagest actuellement défini sur True.

Ensuite, tous nos autres tests (pour voir si un événement s'est produit qui devrait définir le flagà False) peuvent être organisés proprement dans le reste du programme.

**Exemple**

active = True

while active:

city = input("Please enter the name of a city you have visited (enter 'quit' when you are finished): ")

if city == 'quit':

active = False

elif city == 'leave me alone':

active = False

elif city == 'stop':

active = False

else:

print("I'd love to go to", city)

print("Goodbye !")

**Pause et Continuer**

breaket continuesont deux instructions qui nous aident à gérer le flux de la boucle.

**casser**

breakquittez immédiatement la boucle while sans exécuter le code restant dans la boucle, quels que soient les résultats de toute condition.

Prenons par exemple un programme qui demande à l’utilisateur quels lieux il a visités.

Nous pouvons arrêter la boucle while dans ce programme en appelant break dès que l'utilisateur entre la valeur « quit » :

while True:

city = input("Please enter the name of a city you have visited (enter 'quit' when you are finished): ")

if city == 'quit':

break

else:

print("I'd love to go to", city)

print("Goodbye !")

Une boucle qui commence par while Trues'exécutera indéfiniment à moins qu'elle n'atteigne une instruction break.

La boucle de ce programme demande à l'utilisateur d'entrer les noms des villes qu'il a visitées jusqu'à ce qu'il entre « quit ».

Lorsqu'ils entrent « quit », l'instruction break s'exécute, obligeant Python à quitter la boucle :

Please enter the name of a city you have visited: (Enter 'quit' when you are finished.) New York

I'd love to go to New York!

Please enter the name of a city you have visited: (Enter 'quit' when you are finished.) San Francisco

I'd love to go to San Francisco!

Please enter the name of a city you have visited: (Enter 'quit' when you are finished.) quit

**Exemple**

secret\_number = 4

while True:

user\_number = input('Guess the secret number: ')

if int(user\_number) == secret\_number:

print('Congrats! You win!')

break

else:

print('Wrong guess...')

**continuer:**

Plutôt que de sortir complètement d'une boucle sans exécuter le reste de son code, vous pouvez utiliser l'instruction continue pour revenir au début de la boucle.

Par exemple, considérons une boucle qui compte de 1 à 10 mais imprime uniquement les nombres qui ne sont pas compris entre 3 et 7 :

current\_number = 0

while current\_number <= 10:

current\_number += 1

if 3 < current\_number < 7: # If the number is between 3 and 7

continue # Go back to the beginning of the loop

print(current\_number)

>> 1

>> 2

>> 3

>> 7

>> 8

>> 9

>> 10

Soyez prudent, cela peut finir en boucle infinie.

current = 0

while current <= 10:

print(current)

if 3 < current < 7:

continue

current += 1

// Here the incrementation is after the if statement, therefore as soon as the variable `current` reachers the number 4, the program enters into an infinite loop.

**Conditionnels**

**Conditions**

Les instructions conditionnelles constituent un paradigme de programmation universel permettant d'effectuer des actions spécifiques en comparant des données. C'est notre façon de communiquer avec l'ordinateur : si une condition est vraie, alors nous exécutons cette action, sinon nous exécutons une autre action.

Voir la documentation officielle [ici](https://docs.python.org/3/tutorial/controlflow.html#if-statements)

Les conditions sont des expressions qui comparent deux éléments.

Python prend en charge les conditions logiques suivantes :

* Égal à:a == b
* Pas égal à :a != b
* Moins que:a < b
* Inférieur ou égal à :a <= b
* Supérieur à :a > b
* Supérieur ou égal à :a >= b

Ces expressions renvoient Vrai ou Faux.

>>> 33 > 20

True

>>> 130 > 190

False

**instructions if**

Le mot-clé principal pour créer une condition est ifsuivi de la condition à implémenter et de deux points. Vient ensuite le bloc de code à exécuter si la condition est True.

ifL'instruction teste une condition et exécute un code si la condition est vraie.

**Assurez-vous que vous mettez correctement en retrait** .

Voici la syntaxe :

...

if <condition>:

CODE

CODE

...

Notez que Python s'appuie sur l'indentation (espace au début d'une ligne) pour définir la portée dans le code.

a = 33

b = 200

if a > b:

print("a is greater than b")

print("Finished")

Testez ce code avec d’autres valeurs !

**Exercice**

1. demander à l'utilisateur de saisir son nom

2. utilisez la len()fonction pour vérifier la longueur du nom. s'il est inférieur à 5 lettresprint('You have a short name :)')

**déclaration elif**

Le elifmot-clé est la façon dont Python dit : « si les conditions précédentes n'étaient pas vraies, alors essayez cette condition ».

a = 33

b = 200

if a > b:

print("a is greater than b")

elif a == b:

print("a is equal to b")

print("Finished")

**instruction else**

Le elsemot-clé capture tout ce qui n'est pas capturé par les conditions précédentes.

Si notre condition est False, nous pouvons définir un comportement par défaut en utilisant else. Tout bloc de code suivant notre elseinstruction s'exécutera, à condition que les conditions de l' ifinstruction soient False.

a = 33

b = 200

if a > b:

print("a is greater than b")

elif a == b:

print("a is equal to b")

else:

print("b is greater than a")

print("Finished")

**Exemples**

# Example 1

if 3 > 4:

print('something is fishy...')

elif 3 <= 4:

print('3 is not equal to 4, but it is lower than 4')

else:

print('4 is actually greater than 3')

# Example 2

name = input('Please state your name: ')

if name == 'Frank':

print('You are Frank Sinatra')

elif name == 'Miles':

print('You are Miles Davis')

elif name == 'Tony':

print('You are Tony Benett')

else:

print('I do not know who you are!')

**Instructions AND & OR**

Ils sont tous deux utilisés pour combiner des instructions conditionnelles.

andl'instruction sera activée si les deux conditions sont vraies.

a = 200

b = 33

c = 500

if a > b and c > a:

print("Both conditions are True")

orl'instruction sera activée si une seule des conditions est vraie.

a = 200

b = 33

c = 500

if a > b or a > c:

print("At least one of the conditions is True")

**en mot-clé**

inLe mot-clé peut être utilisé pour vérifier si une valeur est dans une séquence.

L'expression my\_var in my\_listest un booléen ; elle sera remplacée par TrueouFalse

>>> "A" in "ABCD"

True

my\_hobbies = "sport, code, food, icecreams, netflix"

if "code" in my\_hobbies:

print("Hello world")

**pas de mot-clé**

Le notmot clé sera renvoyé Truesi la condition est fausse.

**Exercice**

Demander à l'utilisateur un nombre compris entre 1 et 100

Si le nombre est divisible par trois, imprimezFizz

Si le nombre est divisible par cinq, imprimez Buzz.

Si le nombre est divisible par trois et par cinq, imprimez- FizzBuzzle plutôt.

**Dictionnaires**

**Définition**

Le dictionnaire est un type de données fourni par Python, qui est une autre collection d'objets.  
Un dictionnaire est similaire à une liste à bien des égards :

* Il est mutable (ses éléments peuvent être modifiés)
* C'est dynamique (il peut grandir selon les besoins)

Les dictionnaires sont des collections non ordonnées d'éléments stockant des valeurs selon une paire clé-valeur.  
Avec un dictionnaire, vous créez des données structurées et récupérez des valeurs à l'aide d'une clé.  
On peut les comparer à un tableau de feuille de calcul : ils contiennent des colonnes avec des noms (clés) et des cellules avec des données (valeurs).

**Différence entre les dictionnaires et les listes**

Comment accéder aux éléments :  
Dans les listes, les éléments sont accessibles par leur position dans la liste (via l'indexation)  
Exemple :

my\_list = ['Rick', 'Sanchez']

print("My last name is:", my\_list[1])

Pour accéder au mot « Sanchez », il faut mettre son index entre crochets [].

Dans les dictionnaires, c'est un peu différent : les éléments n'ont pas d'index ; on y accède par des clés. Un dictionnaire est un tableau associatif, constitué d'une collection de paires clé-valeur. Chaque paire clé=valeur associe la clé à sa valeur associée.  
Par exemple, la clé first\_nameest associée à Rick, et la clé last\_nameest associée à Sanchez.

**Définir un dictionnaire**

Alors que les listes sont définies par des crochets [], les dictionnaires sont définis en encadrant une liste de paires clé-valeur entre accolades {}.  
Le signe deux-points : sépare chaque clé de sa valeur associée :

mon\_dict = {  
:,  
:,  
:  
}

rick\_dict = {

'first\_name':'Rick',

'last\_name':'Sanchez'

}

Les clés doivent être déclarées sous forme de chaînes entre guillemets. Les valeurs peuvent être de tout type.

my\_dog = {

'name': 'Rufus',

'age': 4,

'good\_dog': True

}

**Accéder aux données**

Pour accéder aux valeurs d'un dictionnaire, il suffit de référencer la clé de cette valeur entre parenthèses. C'est similaire à l'accès aux valeurs d'une liste, mais au lieu d'utiliser l'index de la valeur, on utilise le nom de sa clé.

my\_dog = {

'name': 'Rufus',

'age': 4

}

print(my\_dog['name'])

>> 'Rufus'

Exemple:

rick\_dict = {

'first\_name':'Rick',

'last\_name':'Sanchez'

}

print("The last name of rick is:", rick\_dict['last\_name'])

Si vous faites référence à une clé qui n’est pas dans le dictionnaire, vous obtiendrez une erreur :

print("The last name of rick is:", rick\_dict['age'])

Traceback (dernier appel le plus récent) :  
Fichier «« , ligne 1, dans  
rick\_dict['age']  
KeyError : 'âge'

Vous pouvez utiliser le déballage de tuples pour parcourir les clés et les valeurs du dictionnaire utilisé.  
Pour ce faire, il suffit de décompresser les éléments de chaque élément dans deux variables distinctes représentant la clé et la valeur :

a\_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}

print(a\_dict.items())

# output :

dict\_items([('color', 'blue'), ('fruit', 'apple'), ('pet', 'dog')])

# The items() method returns a view object that displays

# a list of dictionary's (key, value) tuple pairs.

for key, value in a\_dict.items():

print(key, '->', value)

# output

color -> blue

fruit -> apple

pet -> dog

**Types de données**

Vous pouvez stocker tout type de données dans un dictionnaire. Il peut s'agir de chaînes de caractères et d'entiers, mais aussi d'autres dictionnaires ou listes.

my\_dog = {

'name': 'Rufus',

'age': 4,

'best\_friend': {

'name': 'Felix',

'age': 4.5

},

'favorite\_foods': ['steaks', 'sausages', 'shawarma']

}

**Dictionnaires et listes**

Les dictionnaires peuvent être particulièrement utiles lorsqu'ils sont stockés dans une liste pour représenter une collection de plusieurs éléments représentant la même chose, comme une liste de produits.

shirts = [

{

'name': 'Awesome T-shirt 3000',

'size': 'S',

'price': 20

},

{

'name': 'Awesome T-shirt 3000',

'size': 'M',

'price': 25

},

{

'name': 'Awesome T-shirt 3000',

'size': 'L',

'price': 30

},

]

**🚀 Défi**

Accéder à la valeur de l'historique des clés

sample\_dict = {

"class":{

"student":{

"name":"Mike",

"marks":{

"physics":70,

"history":80

}

}

}

}

**Opérations sur les dictionnaires**

* Modifier une entrée dans un dictionnaire : Modifier un élément dans un dictionnaire fonctionne de la même manière qu'une liste, il faut d'abord le sélectionner (avec sa touche) puis lui attribuer une nouvelle valeur avec le signe égal =.

rick\_dict['last\_name'] = 'SANCHEZ'

* Ajout d'une entrée à un dictionnaire existant : Pour ajouter une entrée à un dictionnaire existant, il suffit d'attribuer une nouvelle clé à sa valeur, comme vous l'auriez modifiée s'il s'agissait d'une clé existante.

rick\_dict['hair\_color'] = 'white'

* Supprimer une entrée dans un dictionnaire : Pour supprimer une entrée, utilisez l'instruction del, en spécifiant la clé à supprimer :

del rick\_dict['hair\_color']

**Restrictions des clés**

Presque tous les types de valeurs peuvent être utilisés comme clé de dictionnaire en Python. Cependant, il existe quelques restrictions.

* Premièrement, les clés sont uniques ; elles ne peuvent apparaître qu'une seule fois dans un dictionnaire ; si vous attribuez une valeur à une clé de dictionnaire existante, la clé n'est pas ajoutée une deuxième fois mais remplace la valeur actuelle.
* Deuxièmement, la clé doit être de type immuable ; elle peut être un entier, un nombre à virgule flottante, une chaîne, un booléen et même un tuple, car tous ces types sont immuables. Une liste ne peut pas être une clé de dictionnaire.  
  En revanche, il n'y a aucune restriction sur les valeurs de dictionnaire.

**Le inmot clé**

Dans une liste, le inmot-clé renvoie Vrai ou Faux selon que l'opérande spécifié apparaît dans la liste.  
Un dictionnaire renvoie Vrai si l'opérande apparaît comme clé dans le dictionnaire.

**Itération**

L'itération sur une liste de dictionnaires est simple ; votre itérateur contient le dictionnaire que vous souhaitez consulter.

for shirt in shirts:

print(shirt['size'])

>> S

>> M

>> L

**Méthodes intégrées**

* keys() : La méthode my\_dict.keys() renvoie un dict\_keys de toutes les clés de my\_dict
* values() : La méthode my\_dict.values() renvoie un dict\_values ​​de toutes les valeurs de my\_dict
* items() : La méthode my\_dict.items() renvoie un dict\_items de tuples contenant les paires clé-valeur dans un dictionnaire.

rick\_dict = {

'first\_name':'Rick',

'last\_name':'Sanchez'

}

print(rick\_dict.items())

* mise à jour() : Siet d sont des dictionnaires, d.update() fusionne les entrées deen d.  
  Pour chaque clé dans:
* Si la clé n'est pas présente dans d, la paire clé-valeur deest ajouté à d.
* Si la clé est déjà présente dans d, la valeur correspondante dans d pour cette clé est mise à jour avec la valeur de.

**🚀 Défi**

Supprimer l'ensemble de clés du dictionnaire Python

sample\_dict = {

"name": "Kelly",

"age":25,

"salary": 8000,

"city": "New york"

}

keys\_to\_remove = ["name", "salary"]

**Conclusion**

Les dictionnaires sont un outil puissant en Python qui offre des capacités efficaces de manipulation des données. Ils sont particulièrement utiles pour l'analyse de données, le développement web et toutes les applications nécessitant une recherche rapide de valeurs de configuration ou de propriétés. La maîtrise des dictionnaires améliore votre capacité à travailler avec de grands ensembles de données et à développer des applications complexes pilotées par les données.

**Fonctions Lambda, Map, Reduce et Filter**

**La fonction map()**

La fonction map() utilise une fonction et une liste (ou tout autre itérable) et exécute la fonction sur chaque objet de la liste. Elle renvoie ensuite la séquence contenant tous les résultats.

Par exemple:

def upper\_string(s):

return s.upper()

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

map\_object = map(upper\_string, fruit)

print(list(map\_object))

>> ['APPLE', 'BANANA', 'PEAR', 'APRICOT', 'ORANGE']

L'objet map\_object renvoyé contiendra tous les fruits en majuscules en raison de chaque fruit passé à la fonction upper\_string.

**La fonction filter()**

Similaire à map(), filter() prend un objet fonction et un itérable et crée une nouvelle liste.

Comme son nom l'indique, filter() crée une nouvelle liste contenant uniquement les éléments satisfaisant une condition particulière. Cette condition correspond à la fonction passée en argument. Si elle renvoie True, l'élément est ajouté à la nouvelle liste ; sinon, il ne l'est pas.  
Par exemple, l'extrait suivant ne renvoie que les chaînes commençant par A.

def starts\_with\_A(s):

return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

filtered\_object = filter(starts\_with\_A, fruit)

print(list(filtered\_object))

>> ['Apple', 'Apricot']

**La fonction functools.reduce()**

reduce() fonctionne différemment de map() et filter(). Elle ne renvoie pas une nouvelle liste basée sur la fonction et l'itérable passés. Elle renvoie une valeur unique.

Par exemple, la fonction somme est une opération réductrice, elle prend une liste comme argument et renvoie un seul nombre.

Une fonction passée à reduce doit recevoir deux arguments qui représenteront le premier et le deuxième élément de l'itérable, puis le deuxième et le troisième, puis le troisième et le quatrième, etc…

Par exemple, un extrait qui recrée la fonction somme :

from functools import reduce

def sum\_numbers(first, second):

return first+second

my\_list = [1, 3, 5, 7]

reduced\_list = reduce(sum\_numbers, my\_list)

print(reduced\_list)

>> 16

**Fonctions lambda**

Les fonctions Lambda sont des « fonctions sur une seule ligne », elles reçoivent uniquement des arguments et renvoient une valeur, et elles n'ont pas besoin d'être stockées dans des variables (mais elles le peuvent !).

Voici la syntaxe d'une fonction lambda :

lambda arg1, arg2 : valeur\_retournée

Par exemple, une fonction qui renvoie une chaîne en majuscules :

lambda s: s.upper()

Vous pouvez également stocker cette fonction dans une variable si vous souhaitez la réutiliser :

my\_function = lambda s: s.upper()

# This is the same as doing:

def my\_function(s):

return s.upper()

Ces fonctions peuvent s'avérer utiles lors de l'utilisation de cartes, de filtres et de réductions, car elles permettent une création rapide. Par exemple, recréons les extraits précédents à l'aide de fonctions lambda.

L'exemple de carte :

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

map\_object = map(lambda s: s.upper(), fruit)

print(list(map\_object))

L'exemple de filtre :

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

filtered\_object = filter(lambda s: s[0] == "A", fruit)

print(list(filtered\_object))

L'exemple de réduction :

from functools import reduce

my\_list = [1, 3, 5, 7]

reduced\_list = reduce(lambda first, second: first+second, my\_list)

**🚀 Défi**

personnes = ["Rick", "Morty", "Beth", "Jerry", "Snowball"]

À l'aide de la carte et du filtre, essayez de dire bonjour à tous ceux dont le nom est inférieur ou égal à 4 lettres

**Conclusion**

Cette leçon vous a présenté les puissants concepts de programmation fonctionnelle en Python, facilitant la création d'un code concis et lisible. En maîtrisant lambda, map, reduce et filter, vous améliorerez votre capacité à écrire du code Python de haute qualité, exploitant efficacement les capacités de Python pour la programmation fonctionnelle.

© 2025 **École supérieure de technologie et de commerce de Paris** . Tous droits réservés.

**Fonctions**

**Fonctions**

Une fonction est un bloc de code réutilisable qui s'exécute lorsque vous l'appelez. Imaginez functionsune variable qui stocke une fonctionnalité sous la forme d'un bloc de code exécutable plutôt que des données functions.

Les fonctions sont des blocs de code nommés qui sont conçus pour effectuer une tâche spécifique.

Lorsque vous souhaitez effectuer une tâche particulière que vous avez définie dans une fonction, vous appelez le nom de la fonction responsable de celle-ci.

Si vous devez effectuer cette tâche plusieurs fois dans votre programme, vous n'avez pas besoin de saisir tout le code de la même tâche encore et encore ; vous appelez la fonction dédiée à la gestion de cette tâche, et l'appel indique à Python d'exécuter le code à l'intérieur de la fonction.

Les fonctions vous permettent de diviser vos programmes en petites parties, chacune effectuant une tâche spécifique.

En utilisant des fonctions, vous serez en mesure d'écrire du code plus efficace, plus facile à dépanner et à entretenir, et qui peut être réutilisé dans de nombreux programmes différents.

**Syntaxe**

**Exemple d'une fonction simple qui dit bonjour**

def say\_hello():

"""A function that says hello"""

print("Hello!")

say\_hello()

Cet exemple montre la structure la plus simple d'une fonction.

**Définition**

La première ligne utilise le mot-clé defpour informer Python que vous définissez une fonction.

Il s'agit de la définition de la fonction, qui indique à Python le *nom* de la fonction.

Les parenthèses contiennent des informations sur les paramètres ; nous le verrons bientôt. Dans ce cas, le nom de la fonction est say\_hello(), et elle n'a besoin d'aucun paramètre ; ses parenthèses sont donc vides (elles sont néanmoins obligatoires).

Enfin, la définition se termine par deux points et tout le code inclus dans la fonction doit être *indenté* .

**Code**

Toutes les lignes indentées qui suivent def say\_hello():constituent le corps de la fonction.

Le texte de la deuxième ligne est un commentaire appelé *docstring* , qui décrit le rôle de la fonction. *Les docstrings* sont entourés de guillemets triples, que Python recherche lors de la génération de la documentation des fonctions de vos programmes.

La ligne print("Hello!")est la seule ligne de code réel dans le corps de cette fonction, elle say\_hello()n'a donc qu'une seule tâche : imprimer « Bonjour !

**1. Appel de fonction**

Lorsque vous souhaitez utiliser cette fonction, vous l' *appelez* .

Un *appel* de fonction indique à Python d'exécuter le code de la fonction. Pour *appeler* une fonction, saisissez son nom, suivi des paramètres nécessaires entre parenthèses, comme indiqué sur la dernière ligne. Comme aucune information n'est requise ici, l'appel de notre fonction est aussi simple que de saisir say\_hello(). Comme prévu, elle affiche Hello!.

**2. Passer des informations à une fonction**

Une fonctionnalité puissante des fonctions est *les arguments* , ou *paramètres* .

Certaines valeurs des fonctions peuvent être modifiées à chaque appel. Par exemple, la say\_hellofonction doit dire bonjour à quelqu'un, comme « Salut Rick ! », mais comme nous voulons pouvoir modifier ce nom (et également afficher bonjour à « Morty »), nous ne le codons pas en dur dans la fonction.

Au lieu de cela, nous déclarons un *paramètre* username et nous lui passons un *argument .*

def say\_hello(username):

print("Hello "+username)

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

* Dans la **définition** de la fonction , nous ajoutons le **paramètre** qui acceptera une valeur.

def say\_hello(username):

print("Hello "+username)

* Dans l' **appel** de fonction , nous passons la valeur (c'est-à-dire un argument).

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

Ici, la fonction exécutera le même code mais avec une valeur différente pour username.

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

**Attention !** Si la fonction attend un argument, vous ne pouvez passer qu'un seul argument ; moins ou plus renverra une erreur.

**Exemple d'une fonction qui accepte plus d'un argument**

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick", "FR")

Cette fonction accepte deux arguments :

* Le premier, username, est utilisé pour stocker la valeur du nom d'utilisateur que nous devons saluer.
* Le deuxième, language, est utilisé pour stocker la langue du message d'accueil.

Lors de *l'appel de* la fonction, nous passons deux arguments dans le bon ordre (d'abord le usernamepuis le language).

Les arguments mis en correspondance de cette manière (par ordre) sont appelés *arguments positionnels.*

**Attention :** vous pouvez obtenir des résultats inattendus si vous mélangez l’ordre des arguments dans un appel de fonction lorsque vous utilisez des arguments positionnels.

Si vous obtenez des résultats étranges, vérifiez que l'ordre des arguments dans votre appel de fonction correspond à l'ordre des paramètres dans la définition de la fonction.

**3. Arguments de mots-clés**

Un *argument mot-clé* est une paire nom-valeur que vous passez à une fonction. Vous associez directement le nom et la valeur dans l'argument, évitant ainsi toute confusion lors du passage de l'argument à la fonction.

Les arguments de mots clés vous évitent de vous soucier de l'ordre correct de vos arguments dans l'appel de fonction et ils clarifient le rôle de chaque valeur dans l'appel de fonction.

Réécrivons l’ say\_helloappel de fonction en utilisant *des arguments de mot-clé* .

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello(username="Rick", language="FR")

**Attention :** lorsque vous utilisez des arguments de mots-clés, assurez-vous d'utiliser les noms exacts des paramètres dans la définition de la fonction

La fonction say\_hellon'a pas changé. Cependant, lorsque nous l'appelons, nous indiquons explicitement à Python le paramètre auquel chaque argument doit correspondre.

Lorsque Python lit l'appel de fonction, il sait stocker l'argument « Rick » dans le paramètre nom d'utilisateur et l'argument « FR » dans la langue.

L'ordre des arguments des mots-clés importe peu, car Python sait où placer chaque valeur. Les deux appels de fonction suivants sont équivalents :

say\_hello(username="Rick", language="FR")

say\_hello(language="FR", username="Rick")

Vous pouvez utiliser à la fois des arguments *de mot-clé* et *de position* dans le même appel de fonction, mais vous devez placer chaque argument *de position* avant ceux *de mot-clé* .

Par exemple, cela fonctionnera :

say\_hello("Rick", language="FR")

Mais cela ne fonctionnera pas :

say\_hello(username="Rick", "FR")

**4. Valeurs par défaut**

Lors de l'écriture d'une fonction, vous pouvez définir une valeur par défaut pour chaque paramètre.

Si un argument est fourni pour un paramètre lors de l'appel de fonction, Python utilise la valeur de l'argument. Dans le cas contraire, il utilise la valeur par défaut du paramètre.

Ainsi, lorsque vous définissez une valeur par défaut pour un paramètre, vous pouvez exclure l'argument correspondant que vous écririez habituellement dans l'appel de fonction.

L'utilisation de valeurs par défaut peut simplifier vos appels de fonctions et clarifier la manière dont vos fonctions sont généralement utilisées.

Par exemple, si vous constatez que la plupart des appels à say\_hello()sont utilisés pour des salutations en anglais, vous pouvez définir la valeur par défaut de languageà « EN ». Désormais, toute personne appelant say\_helloen anglais peut omettre cette information :

def say\_hello(username, language="EN"):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick")

# OR

say\_hello(username="Rick")

Nous avons modifié la définition de say\_hellopour inclure une valeur par défaut, « EN », pour language.

Lorsque la fonction est appelée sans languagespécification, Python sait utiliser la valeur « EN » pour ce paramètre.

Pour utiliser une langue autre que l'anglais, vous pouvez utiliser un appel de fonction comme celui-ci :

say\_hello("Rick", "FR")

# OR

say\_hello(username="Rick", language="FR")

Étant donné qu'un argument explicite pour languageest fourni, Python ignorera la valeur par défaut du paramètre.

Lorsque vous utilisez des valeurs par défaut, tout paramètre doté d'une valeur par défaut doit être listé après tous les paramètres qui n'en ont pas. Cela permet à Python de continuer à interpréter correctement les arguments positionnels.

**5. Éviter les erreurs d'argumentation**

Lorsque vous commencez à utiliser des fonctions, ne soyez pas surpris de rencontrer des erreurs liées à des arguments non appariés. Ces erreurs surviennent lorsque vous fournissez moins ou plus d'arguments que nécessaire à la fonction.

Par exemple, voici ce qui se passe si nous essayons d’appeler say\_hello()sans arguments :

Traceback (most recent call last):

File "app.py", line 6, in <module> say\_hello()

TypeError: say\_hello() missing 2 required positional arguments: 'username' and 'language'

La première ligne du traceback nous indique l'emplacement du problème, nous permettant de regarder en arrière et de voir que quelque chose s'est mal passé dans notre appel de fonction.

Dans le deuxième cas, l'appel de fonction incriminé est écrit pour que nous puissions le voir.

Ensuite, lors du troisième, le traceback nous indique que l'appel manque de deux arguments et indique les noms des arguments manquants.

Python est utile car il lit le code de la fonction pour nous et nous indique les noms des arguments à fournir. C'est un autre avantage de donner à vos variables et fonctions des noms descriptifs. Ainsi, les messages d'erreur de Python seront plus utiles à vous et à toute personne susceptible d'utiliser votre code.

Si vous fournissez trop d'arguments, vous devriez obtenir un traceback similaire qui peut vous aider à faire correspondre correctement votre appel de fonction à la définition de fonction.

**6. Portée locale et mondiale**

**Portée locale**

def number\_by\_three(name, day):

sentence = f'Hello {name}! Today is {day}.'

print(sentence)

print(day)

>> NameError: name 'day' is not defined

**Portée mondiale**

name = 'Avner'

def say\_hi():

print(name)

say\_hi()

>> 'Avner'

**7. Valeurs de retour**

Une fonction n'a pas toujours besoin d'afficher directement sa sortie. Elle peut traiter des données puis renvoyer une valeur ou un ensemble de valeurs.

La valeur renvoyée par la fonction est appelée valeur de retour.

L'instruction de retour prend une valeur à l'intérieur d'une fonction et la renvoie à la ligne qui a appelé la fonction.

Les valeurs de retour vous permettent de déplacer une grande partie du travail de base de votre programme dans des fonctions, ce qui peut simplifier le corps de votre programme.

**Renvoyer une valeur simple**

Regardons une fonction qui prend un prénom et un nom et renvoie un nom complet soigneusement formaté :

def get\_formatted\_name(first\_name, last\_name):

"""Return a full name, neatly formatted."""

full\_name = first\_name + ' ' + last\_name

return full\_name.title()

musician = get\_formatted\_name('jimi', 'hendrix')

print(musician)

Jimi Hendrix

La définition de get\_formatted\_name()prend comme paramètres un prénom et un nom.

La fonction combine ces deux noms, ajoute un espace entre eux et stocke le résultat dans full\_name.

La valeur de full\_nameest convertie en titlecas puis renvoyée à la ligne appelante.

Lorsque vous appelez une fonction qui renvoie une valeur, vous devez fournir une variable dans laquelle la valeur de retour peut être stockée.

Dans ce cas, la valeur renvoyée est stockée dans la variable musician. Le résultat affiche un nom proprement formaté, composé des différentes parties du nom d'une personne :

**Exemple** :

def divide\_by\_three(number):

return number / 3

first\_number = 12

first\_number\_computed = divide\_by\_three(first\_number)

print(first\_number\_computed)

>> 4.0

second\_number = 27

second\_number\_computed = divide\_by\_three(second\_number)

print(second\_number\_computed)

>> 9.0

**Renvoyer plusieurs valeurs avec un tuple**

Nous avons déjà vu qu’un tuple peut être déballé :

my\_tuple = ("jimi", "hendrix")

first\_name, last\_name = my\_tuple

print("First name is: " + first\_name)

print("Last name is: " + last\_name)

First name is jimi

Last name is hendrix

Si c'est le cas, nous pouvons renvoyer un tuple à la fin de la fonction et le décompresser lorsque nous l' *appelons* :

def format\_name(first\_name, last\_name):

return (first\_name.title(), last\_name.title())

first, last = format\_name("RICk", "mORTY")

print(first)

print(last)

Rick

Morty

La fonction reçoit a first\_nameet a last\_name, puis les formate et les renvoie sous forme de tuple.

Lors de l'appel de fonction, le tuple est décompressé en deux variables, firstet last.

**Exercice**

Écrivez une fonction calculation()acceptant deux variables et calculant l'addition et la soustraction. Elle doit également renvoyer l'addition et la soustraction en un seul appel.

For example:

def calculation(a, b):

# Your Code

res = calculation(40, 10)

print(res)

© 2025 **École supérieure de technologie et de commerce de Paris** . Tous

**Liste avancée, dictionnaires et boucles**

**I. Boucles For**

**A. Boucles For et dictionnaires**

Nous pouvons parcourir à la fois les clés et les valeurs d’un dictionnaire.

my\_books = {

"title": "Harry Potter",

"author": "JK Rowling",

}

for x, y in my\_books.items():

print("the" + x + "is" + y)

>> the title is Harry Potter

the author is JK Rowling

**B. Opérateur de boucles**

**range(start, stop[, step]) : itérateur dans les boucles.**

print(list(range(1, 10, 2)))

>> [1, 3, 5, 7, 9]

**enumerate(iterable) : énumérer chaque élément de l' *itérable***

for item in enumerate('abcd'):

print(item)

>>

(0, 'a') # Syntax : (index , value)

(1, 'b')

(2, 'c')

(3, 'd')

for (index\_count, letter) in enumerate('abcd'):

print('At index {} the letter is {}'.format(index\_count, letter))

>>

At index 0 the letter is a

At index 1 the letter is b

At index 2 the letter is c

At index 3 the letter is d

**zip(itérable,..) : concat [itérables, …] dans un tuple.**

list1 = [1,2,3]

list2 = ['a','b','c']

list3 = [1.1, 2.2, 3.3, 4.4, 5.5]

for item in zip(list1, list2, list3): # only go as far it is possible

print(item)

>>

(1, 'a', 1.1)

(2, 'b', 2.2)

(3, 'c', 3.3)

**C. Pour le reste**

Cette elsepartie est facultative. Lorsqu'elle est incluse, elle est toujours exécutée une fois la forboucle terminée, sauf si une breakinstruction est rencontrée.

for i in range(1, 3):

print(i)

else:

print('The for loop is over')

>>

1

2

The for loop is over

**D. Tandis que d'autre part**

while some\_condition:

# do something

else:

# do another thing

x = 0

while x < 2:

print(f'x is {x}')

x += 1

else:

print('x is bigger than 2')

>>

x is 0

x is 1

x is bigger than 2

**E. Pause, Continuer, Passer**

* break: briser la boucle (arrêter l'exécution d'une instruction en boucle)

si vous *sortez* d'une boucle forou , aucun bloc whilede boucle correspondant **n'est** exécuté.else

for letter in 'Leonardo':

if letter == 'a':

break

print(letter, end='') # end='' renders each letter next to the other

>> Leon

while True:

s = input('Enter something : ')

if s == 'quit':

break

print('Length of the string is', len(s))

print('Done')

* continue: revenir au début de la boucle ( *continuer* à l'itération suivante de la boucle)

for letter in 'Leonardo':

if letter == 'o':

continue

print(letter, end='') # dont execute for 'o' letter

>> Lenard

while True:

s = input('Enter something : ')

if s == 'quit':

break

if len(s) < 3:

print('Too small')

continue

print('Input is of sufficient length')

* pass: ne rien faire

for item in [1,2,3]:

# comment

pass # to avoid the error

print('Finish the script')

>> Finish the script

**II. Compréhension de liste : un moyen rapide de créer une liste**

De nombreux objets en Python sont « itérables », ce qui signifie que nous pouvons parcourir chaque élément de l’objet.

Les compréhensions de listes sont un moyen unique de créer rapidement une liste avec Python.

**A. La manière de base d'ajouter un élément à une liste**

my\_number = '1234'

my\_list = []

for num in my\_number:

my\_list.append(num)

print(my\_list)

>> ['1', '2', '3', '4']

**B. La méthode de compréhension de liste**

my\_number = '1234'

my\_list = []

my\_list = [num for num in my\_number]

print(my\_list)

>> ['1', '2', '3', '4']

**Exemples avec la méthode de portée**

my\_list = [x for x in range(0,6)]

print(my\_list)

>> [0, 1, 2, 3, 4, 5]

my\_list = [x\*\*2 for x in range(0,6)] # square

print(my\_list)

>> [0, 1, 4, 9, 16, 25]

my\_list = [x for x in range(0,11) if x%2 == 0] # only even

print(my\_list)

>> [0, 2, 4, 6, 8, 10]

**C. La méthode de base pour ajouter un élément à une liste avec une boucle imbriquée**

my\_list = []

for i in [2, 3, 4]:

for j in [100, 200, 300]:

my\_list.append(i\*j)

print(my\_list)

>> [200, 400, 600, 300, 600, 900, 400, 800, 1200]

**D. La méthode de compréhension de liste**

my\_list = []

my\_list = [(i\*j) for i in [2, 3, 4] for j in [100, 200, 300]]

print(my\_list)

>> [200, 400, 600, 300, 600, 900, 400, 800, 1200]

**E. Compréhension du dictionnaire**

dictionary = {key: value for var in iterable}

family\_age = {'Lea': 12, 'Mark': 25, 'George': 50}

new\_year = 1

new\_family\_age = {name: age+new\_year for (name, age) in family\_age.items()}

print(new\_family\_age)

>> {'Lea': 13, 'Mark': 26, 'George': 51}

**Séquences, Liste, Ensembles, Tuple**

**Séquences**

Une séquence n'est pas un type de données ; c'est une **catégorie de types de données** .

Voici plusieurs types de séquences en Python ; les trois suivants sont les plus importants.

ListsIl s'agit du type de séquence le plus polyvalent. Les éléments d'une liste peuvent être n'importe quel objet, et les listes sont mutables : elles peuvent être modifiées. Les éléments peuvent être réaffectés ou supprimés, et de nouveaux éléments peuvent être insérés.

Par exemple:

my\_name = "Jack"

hello = "Hello World"

my\_age = 27

my\_list = [my\_name, my\_age, hello]

Tuplessont comme des listes, mais elles sont immuables ; elles ne peuvent pas être modifiées.  
*Remarque* : les tuples utilisent des parenthèses.

my\_tuple = (1+3, 2.7, 'Thursday')

StringsIl s'agit d'un type particulier de séquence ne pouvant stocker que des caractères et possédant une notation spécifique. Cependant, toutes les opérations de séquence décrites ci-dessous peuvent également être appliquées aux chaînes de caractères.

my\_string = "I love cheese!"

**Séquences d'indexation**

Les séquences peuvent être indexables, ce qui signifie que chaque élément peut être récupéré avec son index : sequence\_name[index\_num].

Par exemple, en utilisant unlist

my\_name = "Jack"

hello = "Hello World"

my\_age = 27

my\_list = [my\_name, hello, my\_age]

print(my\_list[0]) # The result gives us "Jack"

Vous vous demandez peut-être pourquoi my\_list[0]me donne le premier élément de ma liste ? Notez que les index commencent à [0], qui correspond au premier élément de la liste, et index [1], nous donne le deuxième élément de notre liste, soit la hellovariable de valeur "Hello World".

Il existe d’autres méthodes d’indexation :

* Index négatifs, c'est-à-dire commençant de la fin vers le début ( sequence\_name[-index\_num]) ceci est un exemple utilisant un tuple:

my\_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9)

print(my\_tuple[-2]) # The result gives us the number 8

* Les index de plage renvoient une liste d'un index à l'autre ( sequence\_name[start:end]), voici un exemple utilisant un list:

my\_list = [10, 20, 30, 40, 50, 60, 70, 80, 90]

print(my\_list[0:4]) # The result gives us [10, 20, 30, 40]

Lors de l'utilisation de plages, l'index de fin **n'est pas inclus**

**cordes**

En fait, strc'est une séquence ; c'est une suite de lettres.  
Vous pouvez indexer les lettres d'une chaîne grâce à la technique d'indexation.

>>> my\_name = "Rick"

>>> print(my\_name[0])

"R"

>>> print(my\_name[2])

"c"

>>> print(my\_name[-1])

"k"

>>> print(my\_name[1:3])

"ic"

**Listes**

Les listes permettent de stocker une collection d'éléments. Ces éléments peuvent être de tout type : entiers, chaînes, booléens ou même d'autres listes.  
Consultez la documentation officielle [**ici.**](https://docs.python.org/3/tutorial/datastructures.html#more-on-lists)

Une liste est une collection ordonnée et modifiable. Elle autorise les doublons.

Pour créer une liste, utilisez des crochets []et insérez des éléments séparés par une virgule ,. Les éléments peuvent être de tous types de données.

Par exemple:

>>> my\_hobbies = ["Eat", "Sleep", "Code"]

Pour accéder aux éléments, reportez-vous au numéro d'index.

>>> my\_hobbies[0]

"Eat"

>>> my\_hobbies[2]

"Code"

>>> my\_hobbies[-2]

"Sleep"

* **Modifier un élément**

La liste est modifiable ; son contenu peut donc être mis à jour. Pour modifier un élément d'une liste, reportez-vous à son numéro d'index et attribuez-lui une nouvelle valeur.

**Attention :** les index dans la liste commencent à 0 et non à 1.

>>> print(my\_hobbies)

["Eat", "Sleep", "Code"]

>>> my\_hobbies[1] = "Meditate"

>>> print(my\_hobbies)

["Eat", "Meditate", "Code"]

Attention ! Si vous essayez de faire référence à un index qui n'existe pas, votre programme plantera.IndexError

* **Ajouter un élément**

L'ajout d'un élément à une liste s'appelle l'ajout.

Pour ajouter un élément à une liste, utilisez la append()méthode.

>>> print(my\_hobbies)

["Eat", "Meditate", "Code"]

>>> my\_hobbies.append("Baseball")

>>> print(my\_hobbies)

["Eat", "Meditate", "Code", "Baseball"]

* **Supprimer un élément**

Il existe plusieurs méthodes pour supprimer des éléments d’une liste.

* Pour supprimer un élément spécifié, utilisez la removeméthode ; cette méthode supprime la première occurrence de l'élément.

>>> print(my\_hobbies)

["Eat", "Meditate", "Code"]

>>> my\_hobbies.remove("Meditate")

>>> print(my\_hobbies)

["Eat", "Code"]

* Pour supprimer un élément par son index, utilisez la popméthode ; cette méthode supprime l'élément à l'index donné et le renvoie.

>>> print(my\_hobbies)

["Eat", "Code"]

>>> my\_hobbies.pop(0)

>>> print(my\_hobbies)

["Code"]

* **Ajout de deux listes**

L'ajout de deux listes les concatènera.

>>> my\_hobbies = ["Food", "Code"]

>>> additional\_hobbies = ["Sport", "More code"]

>>> my\_hobbies + additional\_hobbies

["Food", "Code", "Sport", "More code"]

**Fonctions de liste**

La liste contient plusieurs fonctions intégrées que vous pouvez utiliser.

* **Longueur**  
  L'utilisation de la len()fonction permet de récupérer le nombre d'éléments contenus dans une liste donnée.

fruits = ["apple","pear", "banana", "melon"]

len(fruits)

>> 4

* **Somme**  
  Si vous avez une liste de nombres, vous pouvez obtenir à nouveau la somme de tous les nombres via une fonction.

numbers = [3, 12, 1, -4]

sum(numbers)

>> 12

* **Tri**  
  Il existe différentes manières de trier les éléments d'une liste, sorted()vous permet de le faire directement.

Si vous avez des numéros, une liste sera triée du plus petit au plus grand.

numbers = [3, 5, 1, 2]

sorted(numbers)

>> [1, 2, 3, 5]

Si vous avez des chaînes, elles seront triées par ordre alphabétique.

letters = ['d', 'a', 'g', 'b']

sorted(letters)

>> ['a', 'b', 'd', 'g']

Cela ne fonctionnera pas si vous avez des valeurs mixtes.

mixed = [3, 'd', 1, 'a']

sorted(mixed)

>> error!

**Plus de méthodes de liste**

food = ['spam', 'eggs', 'ham']

food.append('sushi')

print(food)

>> ['spam', 'eggs', 'ham', 'sushi']

food.insert(0, 'beans')

print(food)

>> ['beans', 'spam', 'eggs', 'ham', 'sushi']

food.extend(['bread', 'water'])

print(food)

>> ['beans', 'spam', 'eggs', 'ham', 'sushi', 'bread', 'water']

**En savoir plus sur les listes**[**ici**](https://www.w3schools.com/python/python_lists.asp)

**Exercice**

Étant donné cette liste :

liste1 = [5, 10, 15, 20, 25, 50, 20]

Recherchez la valeur 20 dans la liste et, si elle est présente, remplacez-la par 200. Mettez à jour uniquement la première occurrence d'une valeur

**Astuce : Regardez la méthode d'index**

Expected output:

list1 = [5, 10, 15, 200, 25, 50, 20]

**Tuples**

Les tuples sont des listes immuables, ce qui signifie que les éléments **ne peuvent pas être modifiés** .

Pour créer un tuple, utilisez des parenthèses :

>>> my\_tuple = (5,6,7)

L'avantage des tuples est qu'ils peuvent être décompressés, ce qui signifie que chaque valeur ira à une variable :

>>> a, b, c = my\_tuple

>>> print(a)

5

>>> print(b)

6

>>> print(c)

7

**Exercice**

Décompressez le tuple suivant en 4 variables

a\_tuple = (10, 20, 30, 40)

Expected output:

a\_tuple = (10, 20, 30, 40)

# Your code

print(a) # should print 10

print(b) # should print 20

print(c) # should print 30

print(d) # should print 40

**Ensembles**

**Important!!**

Listset tuplessont des types de données Python standard qui stockent des valeurs dans un fichier sequence.

**Les ensembles et les dictionnaires n'ont pas de notion d'indice d'élément** et ne peuvent donc pas être considérés comme des suites. En mathématiques, de manière informelle, une suite est une liste ordonnée d'objets (ou d'événements). Comme un ensemble, elle contient des membres (également appelés éléments ou termes).

SetsLes ensembles sont un autre type de données Python standard qui stocke également des valeurs. La principale différence réside dans le fait que, contrairement aux listes ou aux tuples :

* ne peut pas avoir plusieurs occurrences du même élément
* et stocker des valeurs non ordonnées

Nous avons l'habitude Setsde créer une liste sans doublons.

Pour créer un ensemble, utilisez la set()fonction

Pour ajouter un élément à un ensemble, utilisezset.add(element)

>>> my\_set = set()

>>> my\_set.add("Rick")

>>> my\_set.add("Morty")

>>> my\_set.add("Rick")

>>> print(my\_set)

{"Rick", "Morty"}

>>> this\_set = {"banana", "apple", "cherry", "apple"}

>>> print(this\_set)

{'banana', 'cherry', 'apple'}

# the set named this\_set is unordered

**Variables**

**Pourquoi devrions-nous utiliser des variables ?**

Les variables sont là pour rendre votre code dynamique ; lors de la programmation, vous souhaitez exécuter le même code mais avec des valeurs différentes ; c'est pourquoi les variables ont été créées.

**Syntaxe**

Pour définir une variable, nous écrivons son nom puis utilisons le signe égal pour lui attribuer la valeur que nous souhaitons.

hello = "Hello World."

Les variables doivent être écrites sous la forme d'un seul mot, mais vous pouvez utiliser des traits de soulignement au lieu d'espaces.

my\_age = 23

Les noms de variables doivent toujours être explicites.

**1. Déclarer une variable**

Déclarer une variable est vraiment simple : variable\_name = "variable value", par exemple :

my\_name = "Rick"

my\_age = 43

**2. Noms des variables**

Les noms de variables doivent respecter certaines conventions, décrites dans la documentation [PEP 8.](https://www.python.org/dev/peps/pep-0008/#function-and-variable-names) Cette documentation définit les normes que votre code doit respecter.  
Les noms de variables doivent :

* Être composé uniquement de lettres minuscules ou de chiffres, avec des mots séparés par des traits de soulignement
* Ne jamais commencer par un chiffre
* Soyez facile à lire et à comprendre
* Ne pas être l'un des [mots-clés Python](https://www.programiz.com/python-programming/keywords-identifier)

**3. Utilisation des variables**

Pour utiliser une variable, tapez son nom et Python le remplacera par sa valeur.

my\_age = "43"

my\_name = "Rick"

print("My name is " + my\_name + ", and I am " + my\_age + " years old.")

"My name is Rick, and I am 43 years old."

**Important!!**

Vous vous demandez peut-être pourquoi « » my\_age = "43"est entouré de guillemets.  
Il est important de se rappeler que les valeurs numériques non entourées de guillemets sont considérées comme des entiers.

my\_age = 43

my\_age1 = "43"

These two are not the same!

type(my\_age) # The 43 value makes this an integer

type(my\_age1) # The " " turns the number 43 into a string

Comme nous l'avons vu auparavant dans la leçon précédente, les entiers ne peuvent pas être concaténés ou formatés en phrases avec des chaînes, par exemple, si je devais essayer d'imprimer ceci :

my\_age = 43

print("My name is Rick, and I am " + my\_age + " years old.")

Une erreur apparaîtra

TypeError: can only concatenate str (not "int") to str

Cette erreur de type explique que les entiers et les chaînes ne peuvent pas être concaténés. Si cela se produit, pensez donc à placer vos nombres entre guillemets.  
Il existe cependant une solution pour contourner cette erreur grâce au formatage de chaîne ; vous trouverez des exemples ci-dessous.

**4. Afficher la valeur d'une variable**

Pour afficher la valeur d'une variable, vous pouvez utiliser la printfonction. Par exemple :

>>> my\_hair\_color = "black"

>>> print(my\_hair\_color)

"black"

Vous pouvez également passer plusieurs arguments à print, en les séparant par une virgule ,. Python affichera tous les éléments séparés par un espace. Par exemple :

my\_hair\_color = "black"

print("My hair is", my\_hair\_color)

"My hair is black"

**5. Incrémenter une variable**

Incrémenter signifie « Ajouter un à », pour incrémenter une variable, remplacez sa valeur par « elle-même plus un » :

my\_number = 5

# Increment it

my\_number = my\_number + 1

# Now my number is 6

On peut aussi écrire +=; cet opérateur a la même signification que +.

my\_number = 5

my\_number += 5

# += means "= my\_number +"

#In this example the value of "my\_number" will be 10

Cette syntaxe peut être utilisée avec **tous les opérateurs** .

**6. Formatage des chaînes**

Lorsque vous travaillez avec des chaînes de caractères contenant plusieurs variables, leur formatage peut s'avérer complexe. Python propose des solutions pour simplifier cette tâche, et à partir de Python 3.6, **les chaînes f** (littérales de chaîne formatées) sont devenues la méthode privilégiée pour le formatage des chaînes en raison de leur simplicité et de leur rapidité. Cependant, il est important de comprendre l'ancienne str.format()méthode, car vous pourriez la rencontrer dans du code obsolète.

**f-strings (préféré dans Python 3.6+)**

**Les chaînes de caractères (f-strings)** permettent d'insérer des expressions dans des chaînes littérales, à l'aide d'accolades {}. Voici leur fonctionnement :

1. Commencez la chaîne avec un favant les guillemets d'ouverture.
2. Placez toutes les variables ou expressions à l'intérieur des accolades {}.

Exemple:

name = "Frank"

age = 65

print(f"Hello, {name}. You are {age}.")

**Sortir:** 'Hello, Frank. You are 65.'

Les chaînes de caractères f sont évaluées à l'exécution, ce qui signifie que vous pouvez y intégrer n'importe quelle expression Python valide, et pas seulement des variables. Par exemple :

print(f"The result of 2 \* 35 is {2 \* 35}")

**Sortir:** 'The result of 2 \* 35 is 70'

C'est pourquoi les chaînes de caractères f sont plus rapides et plus lisibles que les anciennes méthodes de formatage. C'est l'approche la plus couramment utilisée dans le code Python moderne.

**La str.format()Méthode (Héritage)**

Avant les chaînes f, cette str.format()méthode était largement utilisée. Bien qu'elle soit moins courante dans Python 3, vous pourriez la rencontrer dans des bases de code plus anciennes ; il est donc important de la comprendre.

Avec str.format(), les champs de remplacement sont indiqués par des accolades {}. Voici un exemple :

name = "Frank"

age = 65

print("Hello, {}. You are {}.".format(name, age))

**Sortir:** 'Hello, Frank. You are 65.'

Dans cette méthode, le nombre d'accolades {}doit correspondre au nombre d'arguments passés à format(). Vous pouvez également réorganiser ou réutiliser les arguments en utilisant leur index entre les accolades :

print("Hello, {1}. You are {0}.".format(age, name))

**Sortir:** 'Hello, Frank. You are 65.'

**Exemple de comparaison :**

Voici un exemple de formatage d'une date utilisant à la fois des chaînes f etstr.format() :

* **En utilisantstr.format()** :

birthday\_day = 8

birthday\_month = 2

birthday\_year = 1979

print("I was born the {}/{}/{}".format(birthday\_day, birthday\_month, birthday\_year))

**Sortir:** "I was born the 8/2/1979"

* **Utilisation des f-strings** :

birthday\_day = 8

birthday\_month = 2

birthday\_year = 1979

print(f"I was born the {birthday\_day}/{birthday\_month}/{birthday\_year}")

**Sortir:** "I was born the 8/2/1979"

**Remarques importantes :**

* **Les chaînes f** sont évaluées au moment de l'exécution, elles peuvent donc inclure n'importe quelle expression Python à l'intérieur des accolades {}, ce qui les rend très flexibles.
* **str.format()** est toujours valide, mais est désormais considéré comme obsolète par rapport aux chaînes f. Il est utile de le savoir si vous devez maintenir ou comprendre du code existant.

**Exercice 1**

Vous avez une amie nommée Alice et vous souhaitez lui envoyer un message avec les détails suivants :

Nom : Alice

Âge : 30 ans

Ville : New York

Tâches:

Utilisez des chaînes f pour imprimer un message indiquant :

« Bonjour Alice ! Tu as 30 ans et tu vis à New York. »

Utiliser str.format()pour imprimer le même message.

**Exercice 2**

Demandez à l'utilisateur son âge à l'aide de la input()fonction et enregistrez-le dans une variable d'âge.

Convertissez l'âge saisi en un entier et calculez le nombre d'années jusqu'à ce qu'ils atteignent 100 ans.

Afficher un message : « Vous aurez 100 ans dans X années », où X est le nombre d'années calculé.

**7. Prendre les données de l'utilisateur**

La fonction de saisie permet de demander à l'utilisateur de saisir une information. Elle renvoie toujours une stringréponse.

input('Please state your name: ')

**Stockage des entrées utilisateur**

name = input('Please state your name: ')

**Opérations numériques Exemple**

Nous pouvons utiliser des variables pour effectuer toutes sortes d'opérations, y compris des opérations mathématiques. Si vous les utilisez avec une input()fonction, soyez prudent : elles input()renvoient toujours une chaîne de caractères ; vous devez donc convertir cette chaîne en entier si vous souhaitez effectuer une opération mathématique avec elle.

number = input('Multiply me by three: ')

number \* 3

>> '444' # incorrect

int(number) \* 3

>> 12 # correct

**Exercice**

Analysez le code ci-dessous et anticipez le résultat. Vérifiez les résultats dans votre shell Python.

age = input("How old are you? ")

print(f"You are {age} years old")

**\*args et \*\*kwargs**

**\*args et \*\*kwargs**

**1. Quel est le but de l’astérisque ?**

Python a une syntaxe spéciale, \*(opérateur de décompression) et \*\*(opérateur de décompression), qui vous permet de passer un **nombre variable d'arguments** à une fonction.

Par convention, on les écrit \*argset \*\*kwargs, mais seuls les astérisques sont indispensables ; on pourrait également noter \*varset \*\*varspour obtenir le même résultat.

**2. \*args**

\*argsest approprié lorsque nous ne savons pas à l'avance combien d'arguments nous devons passer dans une fonction.

def check\_arguments(\*args):

print(f"These are the arguments {args}")

check\_arguments(1, 2, 'hey')

>> These are the arguments (1, 2, 'hey') // You get a tuple

\*args: est un tuple d'arguments.

def check\_tuple(a,b):

# Returns the sum of 'a' and 'b'

return sum((a,b))

print(check\_tuple(10,30))

>> 40

# Multiple \*args

def check\_multiple\_arguments(\*args):

return sum(args)

print(check\_multiple\_arguments(10,20,100,30))

>> 160

**3. \*\*kwargs**

\*\*kwargsest utile lorsque nous ne savons pas à l'avance combien d'arguments avec mots-clés nous devons passer dans une fonction.

\*\*kwargs: est un dictionnaire d'arguments (mots-clés).

def check\_keywordedarguments(\*\*kwargs):

print(kwargs)

check\_keywordedarguments(name="Sarah", age=24)

>> {'name': 'Sarah', 'age': 24} // You get a dictionary

def check\_keywordedarguments(\*\*kwargs):

for key, value in kwargs.items():

print(key,":",value)

check\_keywordedarguments(name="Sarah", age=24)

>>

name : Sarah

age : 24

def check\_keywordedarguments(\*\*kwargs):

return kwargs

print(check\_keywordedarguments(fruit='apple', ordered= 2))

>> {'fruit': 'apple', ordered: 2}

**4. \*args et \*\*kwargs ensemble**

Souvent \*args, et \*\*kwargssont utilisés ensemble dans une fonction où nous avons au moins un argument obligatoire.

Dans ces cas, l'ordre est important. \*argset \*\*kwargssont placés après tous les arguments requis.

def check\_arguments\_keywordedarguments (required\_arg, \*args, \*\*kwargs):

print(required\_arg)

if args:

print(args)

if kwargs:

print(kwargs)

check\_arguments\_keywordedarguments("required argument")

check\_arguments\_keywordedarguments("required argument", 1, 2, 'hey')

check\_arguments\_keywordedarguments("required argument", 1, 2, 'hey', name="Sarah", age=24)

**Il faut préserver l'ordre !**

def check\_arguments\_keywordedarguments(\*args,\*\*kwargs):

print('\*args', args)

print('\*\*kwargs', kwargs)

check\_arguments\_keywordedarguments(10,20,30,name='John',surname='Doe')

>>

\*args (10, 20, 30)

\*\*kwargs {'name': 'John', 'surname': 'Doe'}

check\_arguments\_keywordedarguments(10,20,30,name='John',surname='Doe', 2)

>> SyntaxError: positional argument follows keyword argument

Lorsque nous déclarons un paramètre étoilé tel que \*param, tous les arguments positionnels à partir de ce point jusqu'à la fin sont collectés sous forme d'un tuple appelé param.

Lorsque nous déclarons un paramètre à double étoile tel que \*\*param, tous les arguments de mots-clés à partir de ce point sont collectés dans un dictionnaire appelé param.

def check(a, \*numbers, \*\*person):

print('Greetings : ', a)

#iterate through all the items in tuple

for num in numbers:

print('num - ', num)

#iterate through all the items in dictionary

for key, value in person.items():

print(key + ': ' + value)

check("hello", 1,2,3,name="John",surname="Doe")

>> Greetings : hello

num - 1

num - 2

num - 3

name: John

surname: Doe

**5. Passer \*args et \*\*kwargs comme arguments**

def check(a, b, c):

print(a, b, c)

a = [1,2,3]

check(\*a)

>> 1 2 3

a = {'a':'Sarah', 'b': 24}

check(\*\*a)

>> TypeError: check() missing 1 required positional argument: 'c'

a = {'a':'Sarah', 'b':24, 'c': 180}

check(\*\*a)

>> Sarah 24 180

**Introduction à la POO**

**Programmation orientée objet**

**Abstrait**

La programmation orientée objet, ou **POO** en abrégé, signifie structurer des programmes de manière à ce que les propriétés et les comportements soient regroupés dans des objets individuels.

Par exemple, un objet pourrait représenter une personne avec un nom, une propriété, un âge, une adresse, etc., et des comportements comme marcher, parler, respirer et courir. Ou un e-mail avec des propriétés comme la liste des destinataires, l'objet, le corps, etc., et des comportements comme l'ajout de pièces jointes et l'envoi.

En termes simples, la programmation orientée objet est une approche permettant de modéliser des choses concrètes du monde réel comme les voitures et les relations entre des choses comme les entreprises et les employés, les étudiants et les enseignants, etc. La POO modélise des entités du monde réel sous forme d'objets logiciels, auxquels sont associées certaines données et qui peuvent exécuter certaines fonctions.

**I. Cours**

Concentrons-nous d’abord sur les données : chaque chose ou objet est une instance d’une classe.

Les structures de données primitives disponibles en Python, comme les nombres, les chaînes et les listes, sont conçues pour représenter des choses simples comme le prix d'un objet, le nom d'un poème et vos couleurs préférées. Mais que se passerait-il si vous souhaitiez exprimer quelque chose de beaucoup plus complexe ?

Par exemple, imaginons que vous souhaitiez suivre le nombre d'animaux différents. Si vous utilisiez une liste, le premier élément pourrait être le nom de l'animal, tandis que le second représenterait son âge. Comment savoir quel élément correspond à quoi ? Et si vous aviez 100 animaux différents ? Êtes-vous certain que chaque animal possède un nom, un âge, etc. ? Et si vous souhaitiez ajouter d'autres propriétés à ces animaux ? Cela manque d'organisation, et c'est précisément la raison d'être des classes.

Les classes permettent de créer de nouvelles structures de données définies par l'utilisateur contenant des informations arbitraires sur un élément. Dans le cas d'un animal, nous pourrions créer une Animalclasse pour suivre ses propriétés, telles Animalque son nom et son âge.

Il est important de noter qu'une **classe fournit une structure** : c'est un plan directeur pour définir quelque chose, mais elle n'offre aucun contenu concret. La Animalclasse peut préciser que le nom et l'âge sont nécessaires pour identifier un animal, mais elle n'indiquera pas le nom ni l'âge d'un animal spécifique. Il peut être judicieux de considérer une classe comme une idée de définition.

**II. Objets (instances)**

Alors que la classe est le plan directeur, **une instance est une copie de la classe** avec des valeurs réelles, littéralement un objet appartenant à une classe spécifique. Ce n'est plus une idée ; c'est un animal réel, comme un chien nommé Roger, âgé de huit ans.

Autrement dit, un cours est comparable à un formulaire ou à un questionnaire. Il définit les informations nécessaires. Une fois le formulaire rempli, votre copie spécifique constitue une instance du cours ; elle contient les informations de base qui vous concernent.

Vous pouvez remplir plusieurs copies pour créer de nombreuses instances différentes , mais sans le formulaire comme guide, vous seriez perdu et ne sauriez quelles informations sont nécessaires. Ainsi, avant de créer des instances individuelles d'un objet, vous devez d'abord spécifier ce qui est nécessaire en définissant une classe.

**III. Définition d'une classe**

Définir une classe est simple en Python :

class Dog():

pass

* Vous commencez par le classmot clé pour indiquer que vous créez une classe,
* puis ajoutez le nom de la classe (en utilisant la notation *CamelCase* , en commençant par une majuscule.)

Nous avons également utilisé le mot-clé Python passici. Il est souvent utilisé comme espace réservé pour le code. Il nous permet d'exécuter ce code sans générer d'erreur.

**IV. Création d'un objet (instance)**

Comme nous l’avons dit, la classe n’est qu’un plan.

Pour créer un objet ou une instance, vous souhaitez *appeler* la classe de cette classe.

class Dog():

pass

shelter\_dog = Dog()

Ici, nous avons créé shelter\_dogun **objet de la classe** Dog .

**V. Attributs**

Même dans la vie réelle, tous les objets ont des attributs, et par exemple, les chiens ont **une taille** , **une couleur** et **une race** , nous pouvons implémenter cela dans nos classes, de sorte que tous les objets de la même classe ont les mêmes attributs (mais pas les mêmes valeurs pour ces attributs).

Les attributs sont comme des variables ; ils peuvent être de n'importe quel type de données, la seule différence est qu'ils appartiennent à un objet.

Pour cibler un attribut, il faut faire **référence à l'objet** ( shelter\_dog), suivi d'un point .et du nom de l'attribut. Par exemple

shelter\_dog.color

Ceci fera référence à l' colorattribut du chien. Bien sûr, lorsque vous ciblez l'attribut d'un objet, vous devez cibler un attribut existant, sinon vous obtiendrez une erreur.

Pour définir un nouvel attribut (ou modifier un existant), ciblez-le et attribuez-lui sa nouvelle valeur avec le signe égal =.

class Dog():

pass

shelter\_dog = Dog()

shelter\_dog.color = "Brown"

**VI. La \_\_init\_\_méthode**

Lorsqu'un objet est créé, **Python exécute automatiquement** la \_\_init\_\_()méthode (il faut l'appeler ainsi) de la classe.

Cette méthode doit avoir au moins un argument self(elle n'a pas besoin d'être appelée self,mais c'est une convention Python).

* **selffait référence à l'objet lui-même.**

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self):

print("A new dog has been initialized !")

shelter\_dog = Dog()

Bien que cette fonction reçoive un argument ( self), nous n'avons pas besoin de le passer, **il sera passé automatiquement par Python comme premier argument.**

Vous pouvez ajouter des arguments à \_\_init\_\_. Ces arguments seront transmis lors de la création de l'objet ( shelter\_dog = Dog()).

Par exemple :

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

shelter\_dog = Dog(name\_of\_the\_dog="Rex")

# or

shelter\_dog = Dog("Rex")

Ici, nous ne passons qu'un seul argument ( "Rex"), mais en réalité, deux valeurs sont passées, la valeur name\_of\_the\_doget la shelter\_dogvaleur elle-même, sous la forme self. Voici ce qu'exécute Python :

shelter\_dog = Dog(shelter\_dog, "Rex")

Cet argument est passé *implicitement* .

La plupart du temps, vous souhaitez initialiser les attributs d'un objet lors de sa création (comme un chien nouveau-né aurait une couleur, une race et une taille initiales...).

Pour ce faire, vous pouvez passer des arguments à la \_\_init\_\_()fonction, puis initialiser les attributs de la fonction.

Pour attribuer une valeur à un attribut, sélectionnez-le avec object.attributeet attribuez-lui le =signe. Dans \_\_init\_\_une fonction, l'objet est appeléself ; pour définir un attribut, il faut donc self.attribute = value.

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

shelter\_dog = Dog('Rex')

other\_shelter\_dog = Dog("Jimmy")

Ici, j'ai créé deux Dogobjets différents avec deux autres noms.

Autre exemple

Créez une classe nommée Person, utilisez la \_\_init\_\_()fonction pour attribuer des valeurs pour le nom et l'âge :

class Person():

def \_\_init\_\_(self, name, age):

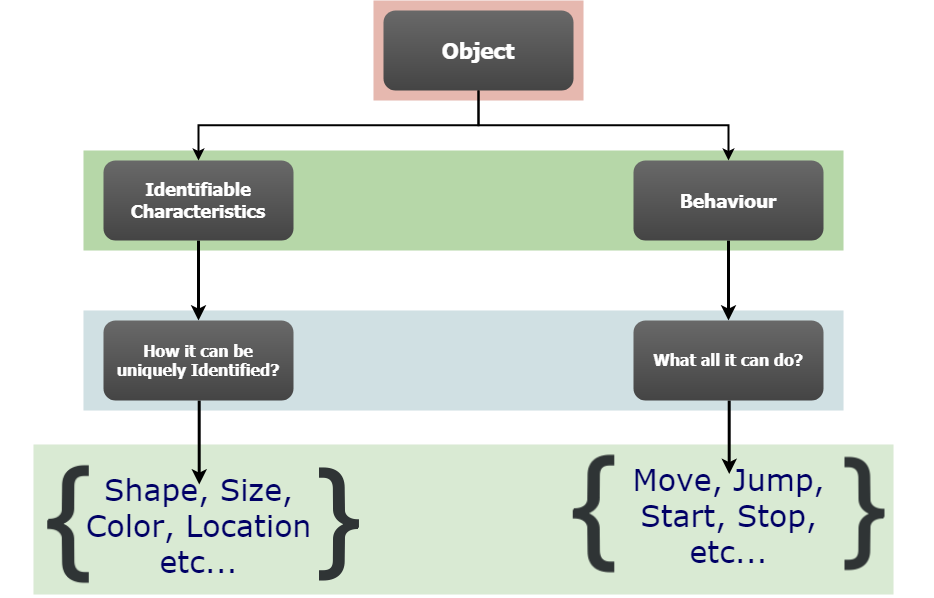
self.name = name

self.age = age

first\_person = Person("John", 36)

print(first\_person.name)

print(first\_person.age)



**Exercice**

* Analysez le code ci-dessous. Quel sera le résultat ?
* Expliquez le but de la \_\_init\_\_()méthode

class Point():

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

## create an instance of the class

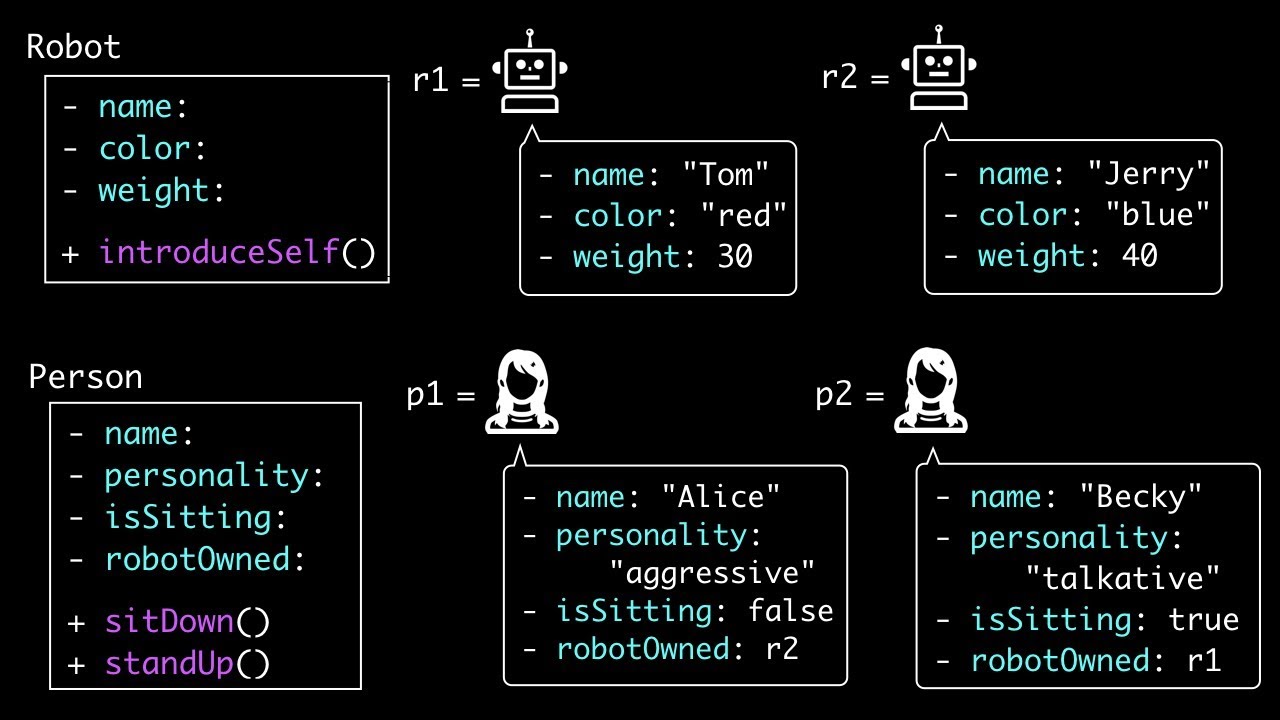
p = Point(3,4)

## access the attributes

print("p.x is:", p.x)

print("p.y is:", p.y)

**Méthodes d'instance**



**Méthodes d'instance**

Les méthodes d'instance sont définies dans une classe et servent à décrire une fonction appartenant à cette classe . Par exemple, dans la réalité, la fonction « aboyer » appartient à la classe « Chien ».

Les méthodes d'instance peuvent être utilisées pour effectuer des opérations avec les attributs, obtenir le contenu d'une instance et bien d'autres choses.

Pour définir une méthode, utilisez le defmot-clé dans la classe, comme nous l'avons fait avec la \_\_init\_\_méthode. Toutes les méthodes d'instance doivent recevoir selfun premier argument ; cela nous permet de manipuler l'objet à l'intérieur de la méthode.

Définissons la méthode de l'écorce.

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

def bark(self):

print(f"{self.name} barks ! WAF")

Pour appeler une méthode d'instance, saisissez le nom de l'instance (objet) suivi d'un point .et du nom de la fonction. Créons un Dogobjet et appelons la barkfonction.

shelter\_dog = Dog("Rex")

shelter\_dog.bark()

La première ligne affichera :

>> A new dog has been initialized

>> His name is Rex

Et le deuxième :

>> Rex barks ! WAF

Après la méthode ' self ', les méthodes peuvent également recevoir des arguments, qui seront passés implicitement.

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

def bark(self):

print(f"{self.name} barks ! WAF")

def walk(self, number\_of\_meters):

print(f"{self.name} walked {number\_of\_meters} meters")

shelter\_dog = Dog("Rex")

shelter\_dog.walk(10)

Vous pouvez également utiliser des méthodes d'instance pour modifier les attributs d'un objet.  
Voici une fonction permettant de modifier le nom du chien :

class Dog():

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

def bark(self):

print(f"{self.name} barks ! WAF")

def walk(self, number\_of\_meters):

print(f"{self.name} walked {number\_of\_meters} meters")

def rename(self, new\_name):

self.name = new\_name

shelter\_dog = Dog("Rex")

shelter\_dog.rename("Paul")

N'oubliez pas que le selfmot-clé fait référence à l'objet lui-même.

**Exercice**

* Analysez le code ci-dessous. Quel sera le résultat ?
* Expliquer le but des méthodes
* Créer une méthode qui modifie le nom de la personne

class Person():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def show\_details(self):

print("Hello my name is " + self.name)

first\_person = Person("John", 36)

first\_person.show\_details()

**Exercice**

Analysez le code ci-dessous. Quel sera le résultat ?

class Computer():

def description(self, name):

"""

This is a totally useless function

"""

print("I am a computer, my name is", name)

#Analyse the line below

print(self)

mac\_computer = Computer()

mac\_computer.brand = "Apple"

print(mac\_computer.brand)

dell\_computer = Computer()

Computer.description(dell\_computer, "Mark")

# IS THE SAME AS:

dell\_computer.description("Mark")

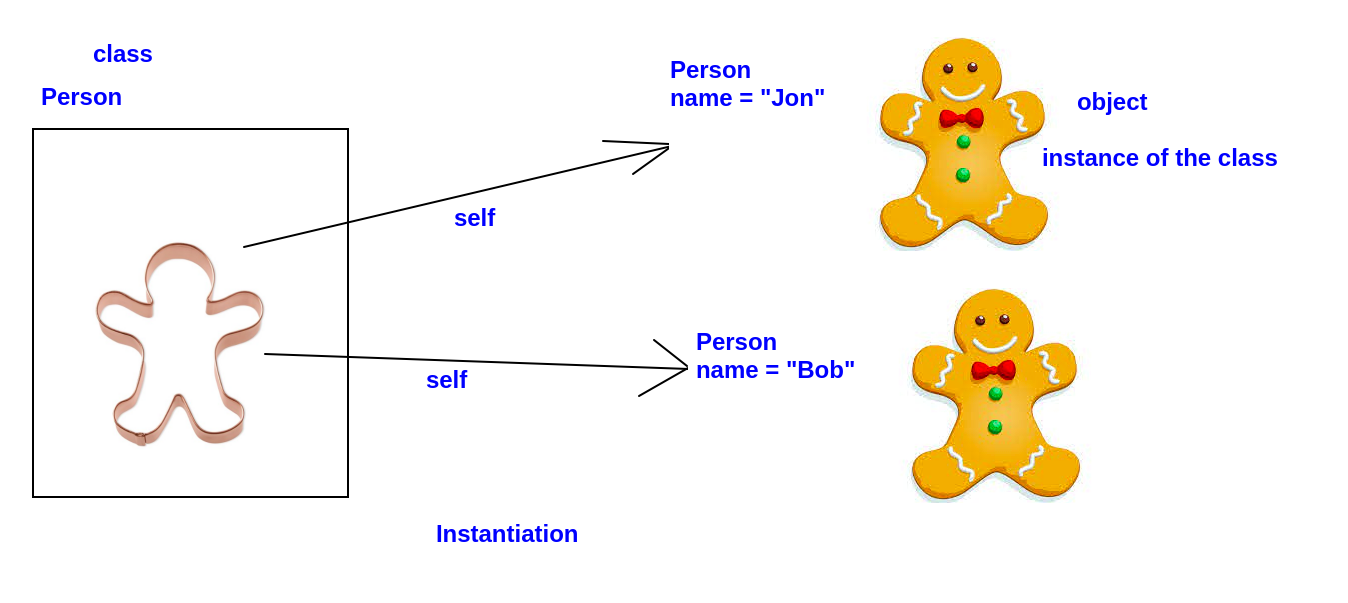
**Exemple de code de classes et d'objets**

**Table des matières**

* + [Classes et objets](https://pstb.developers.institute/courses/collection/117/course/614/section/1675/chapter/971#Classes_and_Objects)

**Dernière mise à jour : 27 janvier 2025**

**Classes et objets**



class BankAccount:

def \_\_init\_\_(self, account\_number, balance=0):

self.account\_number = account\_number

self.balance = balance

self.transactions = []

def view\_balance(self):

self.transactions.append("View Balance")

print(f"Balance for account {self.account\_number}: {self.balance}")

def deposit(self, amount):

if amount <= 0:

print("Invalid amount")

elif amount < 100:

print("Minimum deposit is 100")

else:

self.balance += amount

self.transactions.append(f"Deposit: {amount}")

print("Deposit Succcessful")

def withdraw(self, amount):

if amount > self.balance:

print("Insufficient Funds")

else:

self.balance -= amount

self.transactions.append(f"Withdraw: {amount}")

print("Withdraw Approved")

return amount

def view\_transactions(self):

print("Transactions:")

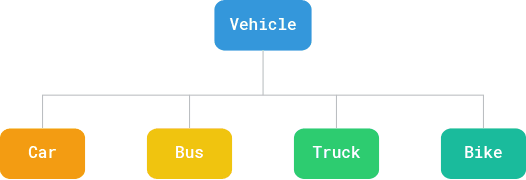
print("-------------")

for transaction in self.transactions:

print(transaction)

**Cours sur l'héritage OOP**

**I. Héritage**



L'héritage est le processus par lequel une classe hérite des attributs et des méthodes d'une autre. Les classes nouvellement créées sont appelées classes enfants, et les classes dont elles dérivent sont appelées classes parents.

Par exemple, dans la réalité, Chien est une classe enfant d'Animal ; chaque Chien est un Animal, mais chaque Animal n'est pas un Chien. Les enfants héritent de tous les attributs et comportements du parent, mais peuvent également spécifier un comportement différent.

Pour faire hériter une classe d'une autre, ajoutez-la simplement entre parenthèses lors de la définition de la classe :

class Animal():

def \_\_init\_\_(self, name):

self.name = name

class Dog(Animal):

def bark(self):

print(f"{self.name} barked, WAF !")

*L'héritage* se produit lorsqu'une classe utilise du code construit dans une autre classe. Les classes appelées *classes enfants* ou *sous-classes* héritent des méthodes et des variables des *classes parentes* ou *classes de base* .

class Animal():

def \_\_init\_\_(self, type, number\_legs, sound):

self.type = type

self.number\_legs = number\_legs

self.sound = sound

def make\_sound(self):

print(f"I am an animal, and I love saying {self.sound}")

class Dog(Animal):

pass

rex= Dog("dog", 4, "wouaf")

print('This animal is a:', rex.type)

# >> This animal is a dog

print('This dog has', rex.number\_legs , ' legs')

# >> This dog has 4 legs

print('This dog makes the sound ', rex.sound)

# >> This dog makes the sound wouaf

rex.make\_sound()

# >> I am an animal, and I love saying wouaf

Nous avons créé un objet Dog *rex* qui utilise la méthode et les attributs de la classe Animal même si nous n'avons pas défini ces méthodes et attributs dans la classe enfant Dog.

class Animal():

def \_\_init\_\_(self, type, number\_legs, sound):

self.type = type

self.number\_legs = number\_legs

self.sound = sound

def make\_sound(self):

print(f"I am an animal, and I love saying {self.sound}")

class Dog(Animal):

def fetch\_ball(self):

print("I am a dog, and I love fetching balls")

rex = Dog('dog', 4, "Wouaf")

print('This animal is a:', rex.type)

# >> This animal is a dog

rex.fetch\_ball()

# >> I am a dog, and I love fetching balls

roger = Animal('Roger', 4, "Grr")

roger.fetch\_ball()

# >> AttributeError: 'Animal' object has no attribute 'fetch\_ball'

C'est parce que la méthode fetch\_ball()appartient uniquement à la *classe enfant* Dog et non à la *classe parent* Animal .

**Exercice**

Analysez le code ci-dessous avant de l'exécuter. Quels seront les résultats ?

class Circle:

color = "red"

class NewCircle(Circle):

color = "blue"

nc = NewCircle

print(nc.color)

# >> What will be the output ?

class Circle:

def \_\_init\_\_(self, diameter):

self.diameter = diameter

def grow(self, factor=2):

"""grows the circle's diameter by factor"""

self.diameter = self.diameter \* factor

class NewCircle(Circle):

def grow(self, factor=2):

"""grows the area by factor..."""

self.diameter = (self.diameter \* factor \* 2)

nc = NewCircle(1)

print(nc.diameter)

nc.grow()

print(nc.diameter)

# >> What will be the output

**II. Remplacement des méthodes parentes**

Lorsque vous créez la même méthode dans la classe enfant, vous surchargez la méthode de la classe parent.  
Il est important de noter que les classes enfants surchargent ou étendent les fonctionnalités des classes parentes. La classe enfant aura toutes les fonctions de la classe parente, plus les siennes.

class Animal():

def \_\_init\_\_(self, type, number\_legs, sound):

self.type = type

self.number\_legs = number\_legs

self.sound = sound

def make\_sound(self):

print(f"I am an animal, and I love saying {self.sound}")

class Dog(Animal):

def fetch\_ball(self):

print("I am a dog, and I love fetching balls")

def make\_sound(self):

print("I am an DOGGG !!! WOUAFFF!!")

rex = Dog('dog', 4, "Wouaf")

rex.make\_sound()

# >> I am an DOGGG !!! WOUAFFF!!

**III. La fonction super()**

Avec cette super()fonction, vous pouvez accéder aux méthodes héritées qui ont été écrasées dans un objet de classe.

class Animal():

def \_\_init\_\_(self, type, number\_legs, sound):

self.type = type

self.number\_legs = number\_legs

self.sound = sound

class Dog(Animal):

def \_\_init\_\_(self, type, number\_legs, sound, fetch\_ball):

super().\_\_init\_\_(type, number\_legs, sound)

# Or : Animal.\_\_init\_\_(self,type, number\_legs, sound)

self.fetch\_ball = fetch\_ball

rex = Dog('dog', 4, "wouaf", True)

print('This animal is a:', rex.type)

# >> This animal is a dog

print('This dog has', rex.number\_legs , ' legs')

# >> This dog has 4 legs

print('This dog makes the sound ', rex.sound)

# >> This dog makes the sound wouaf

print('Does this dog fetchs balls ? ', rex.fetch\_ball)

# >> Does this dog fetchs balls ? True

Nous utilisons super()la fonction avant \_\_init\_\_()la méthode. Nous souhaitons extraire le contenu de \_\_init\_\_()la méthode de la classe parent vers la classe enfant.

Vous pouvez également spécifier à partir de quelle classe vous souhaitez utiliser la super()fonction.

class MyClass(object):

def func(self):

print("I'm being called from the Parent class")

class ChildClass(MyClass):

def func(self):

print("I'm actually being called from the Child class")

print("But...")

# Calling the `func()` method from the Parent class.

super(ChildClass, self).func()

my\_instance\_2 = ChildClass()

my\_instance\_2.func()

**Exercice**

Quelle est la différence entre ces 2 morceaux de code ?

class A(B):

def \_\_init\_\_(self, \*args, \*\*kwargs)

B.\_\_init\_\_(self, \*args, \*\*kwargs)

pass

class A(B):

def \_\_init\_\_(self, \*args, \*\*kwargs)

super().\_\_init\_\_(\*args, \*\*kwargs)

pass

**Exercice sur l'héritage et la composition : classe Door**

Essayez de recréer la classe expliquée ci-dessous :

Nous avons une classe appelée Doorqui a un attribut is\_openeddéclaré lorsqu'une instance est initiée.

Nous pouvons créer une classe appelée BlockedDoorqui hérite de la classe de base Door.

Nous remplaçons simplement les fonctions de la classe parente open\_door()et close\_door()par notre propre BlockedDoorversion de ces fonctions, ce qui génère simplement une **erreur** indiquant qu'une porte bloquée ne peut pas être ouverte ou fermée.

**Encapsulation, polymorphisme**

**I. Encapsulation**

Nous pouvons restreindre l'accès aux méthodes et aux variables. Cela empêche toute modification directe des données, ce qu'on appelle **l'encapsulation**\_ . En Python, les attributs privés sont indiqués par un trait de soulignement (single ou double) comme préfixe \_\_.

Définissons une classe d’objets Computeret essayons d’accéder à ses variables et méthodes à la fois privées et globales.

class Computer():

def \_\_init\_\_(self):

self.name = "Apple Computer" # public

self.\_\_max\_price = 900 # private

def sell(self): # public method

print(f"Selling Price: {self.\_\_max\_price}")

def \_\_sell(self): # private method

print('This is private method')

def set\_max\_price(self, price):

self.\_\_max\_price = price

c = Computer()

**Remarque** : une fois que nous essayons d’accéder aux attributs et aux méthodes de notre classe, nous devrions recevoir une restriction s’ils sont privés.

c.sell()

# >> Selling Price: 900

Explication de la sortie Selling Price: 900:  
qui \_\_max\_priceest un attribut privé, est affiché en utilisant la sell()méthode publique.

c.\_\_sell()

# >> AttributeError: 'Computer' object has no attribute '\_\_sell'

Explication de la sortieAttributeError: 'Computer' object has no attribute '\_\_sell' :  
L'interpréteur ne peut pas exécuter la \_\_sell()fonction à cause du \_\_trait de soulignement, ce qui signifie qu'elle est privée et que l'utilisateur n'y accède pas.

# change the price

c.\_\_max\_price = 1000

c.sell()

# >> The private attribute \_\_max\_price cannot be changed

# >> Selling Price: 900

# using setter function

c.set\_max\_price(1000)

c.sell()

# >> Selling Price: 1000

**II. Polymorphisme**

En programmation, le polymorphisme désigne des classes différentes ou apparentées qui utilisent les mêmes noms pour leurs fonctions.

Le polymorphisme permet d'utiliser une interface standard pour plusieurs formulaires ou types de données.

Définissons deux classes, Parrotet Penguin, toutes deux avec des fonctions appelées flyet swim, puis nous appellerons ces fonctions avec la flying\_test(bird)fonction .

class Parrot():

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin():

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

# >> Parrot can fly

flying\_test(peggy)

# >> Penguin can't fly

**Décorateurs en Python**

**Décorateurs**

Les décorateurs sont des fonctions qui « décorent » ou améliorent d'autres fonctions. Pour comprendre ce que cela signifie, voyons d'abord comment passer des fonctions à d'autres fonctions. N'oubliez pas qu'en Python, tout est un objet, et les objets sont de première classe !

def shout():

return "WHOA!"

def whisper():

return "Shhhh"

def perform\_action(func):

print("something is happening")

return func()

perform\_action(shout)

# something is happening

# 'WHOA!'

perform\_action(whisper)

# something is happening

# 'Shhhh'

Nous pouvons écrire le comportement d'un décorateur comme ceci :

def new\_decorator(func):

def wrap\_func():

print("code before func!")

func()

print("code after func!")

return wrap\_func

def decorate\_me():

print("decorate me!")

decorate\_me = new\_decorator(decorate\_me)

decorate\_me() # What do you think this will print?

Utilisons maintenant la syntaxe du décorateur pour faire la même chose ! Lorsqu'on utilise un décorateur, la fonction utilisée pour décorer est préfixée par un @symbole. La fonction à décorer est alors définie ci-dessous. Regardez :

def new\_decorator(func):

def wrap\_func():

print("code before func!")

func()

print("code after func!")

return wrap\_func

@new\_decorator

def decorate\_me():

print("decorate me!")

Notez comment le code à l’intérieur de la new\_decoratorfonction décore ou améliore le code à l’intérieur de la decorate\_mefonction.

Reprenons le premier exemple mais refactorisons-le pour utiliser la syntaxe du décorateur.

def perform\_action(func):

def wrap\_func():

print("something is happening")

return func()

return wrap\_func

@perform\_action

def whisper():

return "Shhhh"

@perform\_action

def shout():

return "WHOA!"

whisper()

# something is happening

# 'Shhhh'

shout()

# something is happening

# 'WHOA!'

Ce code fonctionnera très bien, mais si nous examinons l’ attribut \_\_name\_\_or \_\_doc\_\_de notre fonction, il ne sera pas correct !

shout.\_\_name\_\_ # 'wrap\_func' - oops!

Nous pouvons corriger cela manuellement ou utiliser le décorateur wraps du module functools.

from functools import wraps

def perform\_action(func):

@wraps(func)

def wrap\_func():

print("something is happening")

return func()

return wrap\_func

@perform\_action

def whisper():

return "Shhhh"

@perform\_action

def shout():

return "WHOA!"

shout.\_\_name\_\_ # 'shout' - much better

**Qu'est-ce qu'un décorateur ?**

Les décorateurs sont des wrappers pour les fonctions, ils nous permettent de modifier le comportement d'une fonction sans la modifier de manière permanente.

*Syntaxe*

* Le décorateur prend une **fonction comme argument** ,
* et définissez ensuite une fonction wrapper, qui contient les arguments de la fonction.
* Ensuite, il renvoie l'adresse du wrapper.

# The decorator

def is\_called(func\_as\_arg):

# The wrapper function

def is\_returned():

print("Hello")

func\_as\_arg()

# returns the wrapper address

return is\_returned

def func\_as\_arg():

print("Bye")

new = is\_called(func\_as\_arg)

new()

# >> Output : Hello Bye

**Décoré pour ajouter un message :**

Travaillons avec cette fonction

def print\_helloworld():

print("Hello world from the function")

**Exemple sans décorateur**

# Example without decorator

def my\_decorator(called\_function):

def new\_function():

print("Hello world from the decorator")

called\_function()

return new\_function

def print\_helloworld():

print("Hello world from the function")

# What happens in the line below ?

print(my\_decorator(print\_helloworld))

# >> <function my\_decorator.<locals>.new\_function at 0x7f5e4e58d790>

# The output is the function returned by the function my\_decorator()

# So, we have to call this function to print the two sentences

print(my\_decorator(print\_helloworld)())

# >> Hello world from the decorator

# >> Hello world from the function

**Exemple avec décorateur**

# Example with the decorator

def my\_decorator(called\_function):

def new\_function():

print("Hello world from the decorator")

called\_function()

return new\_function

@my\_decorator

def print\_helloworld():

print("Hello world from the function")

print\_helloworld()

# >> Hello world from the decorator

# >> Hello world from the function

# The same as writing

print\_helloworld = my\_decorator(print\_helloworld)

print\_helloworld()

**Exercice**

Analysez ce code avant de l'exécuter. Commentez chaque chiffre pour comprendre son fonctionnement. Quel sera le résultat ?

def my\_decorator(my\_function): # <-- (4)

def inner\_decorator(): # <-- (5)

print("This happened before!") # <-- (6)

my\_function() # <-- (7)

print("This happens after ") # <-- (10)

print("This happened at the end!") # <-- (11)

return inner\_decorator

# return None

@my\_decorator # <-- (3)

def my\_decorated(): # <-- (2) <-- (8)

print("This happened!") # <-- (9)

my\_decorated() # <-- (1)

**Décorer une fonction avec des arguments**

def cap\_decorator(func):

def wrapper(name):

name = name.capitalize()

func(name)

return wrapper

@cap\_decorator

def print\_my\_name(name):

print("Hello world from",name)

@cap\_decorator

def say\_hello\_to\_me(name):

print("Hello to",name)

print\_my\_name("eyal")

say\_hello\_to\_me("eyal")

>> Hello world from Eyal

>> Hello to Eyal

**Décorer une fonction avec un nombre inconnu d'arguments**

def cap\_decorator(func):

def wrapper(\*args, \*\*kwargs):

args = [arg.capitalize() for arg in args]

func(\*args, \*\*kwargs)

return wrapper

@cap\_decorator

def describe\_me(first\_name, last\_name, favourite\_activity):

print("I am {} {} and I love {}".format(first\_name, last\_name, favourite\_activity))

@cap\_decorator

def describe\_my\_family(father\_name, mother\_name, brother\_name, sister\_name):

print("The name of my father is", father\_name)

print("The name of my mother is", mother\_name)

print("The name of my brother is", brother\_name)

print("The name of my sister is", sister\_name)

describe\_me("john", "ricotta", "coding")

describe\_my\_family("John","Valentina","mario","luigi")

>> I am John Ricotta and I love Coding

>> The name of my father is John

>> The name of my mother is Valentina

>> The name of my brother is Mario

>> The name of my sister is Luigi

**Exercice**

Analysez ce code avant de l'exécuter. Quel sera le résultat ?

import datetime

def my\_decorator(inner):

def inner\_decorator(num\_copy):

print(datetime.datetime.utcnow())

inner(int(num\_copy) + 1)

print(datetime.datetime.utcnow())

return inner\_decorator

@my\_decorator

def decorated(number):

print("This happened : " + str(number))

decorated(5)

**Ajouter des arguments au décorateur**

def decorator\_creator(city):

def decorator(func):

def new\_function(\*args, \*\*kwargs):

print("Hello world from", city)

func(\*args, \*\*kwargs)

return new\_function

return decorator

@decorator\_creator("Tel Aviv")

def my\_function():

print("Hello from my function")

my\_function()

# >> Hello world from Tel Aviv

# >> Hello from my function

**Enchaînement de décorateurs**

Ordre d'exécution des décorateurs : les décorateurs sont exécutés dans le même ordre qu'ils apparaissent dans le fichier

def first\_decorator(f):

def wrapper():

print("Hello from decorator 1")

f()

return wrapper

def second\_decorator(f):

def wrapper():

print("Hello from decorator 2")

f()

return wrapper

@first\_decorator

@second\_decorator

def simple\_function():

print("Hello world!")

simple\_function()

# >> Hello from decorator 1

# >> Hello from decorator 2

# >> Hello world!

**@méthode de classe, @méthode statique, @property**

**Décorateurs de classe**

Les décorateurs @classmethod, @staticmethodet @propertysont utilisés sur les fonctions définies au sein des classes. Ce sont des décorateurs intégrés.

**@staticmethod**

Une *méthode statique* est une méthode qui ne reçoit pas de self.

class MyClass:

@staticmethod

def add(a, b):

return a + b

print(MyClass.add(3, 6))

# >> 9

* Le code appartient à une classe mais n'utilise pas l'objet lui-même.
* Cela facilite la lisibilité du code : lorsque nous utilisons @staticmethod, nous savons que la méthode ne dépend pas de l'état de l'objet lui-même.
* Il lie une méthode à la définition de classe.

**Exemple:**

class Man(Person):

sex = "Male"

@staticmethod

def get\_nicknames():

return ["Bro", "Dude", "Buddy"]

**@méthode de classe**

Les méthodes de classe sont des méthodes qui ne sont pas liées à un objet, mais à une classe. Leur premier argument est la classe elle-même (rappelons que les classes sont aussi des objets).

class MyClass:

\_\_counter = 0

@classmethod

def add(cls,a):

return cls.\_\_counter + a

my\_class\_add = MyClass.add(3)

print(my\_class\_add)

# >> 3

new\_class = MyClass()

new\_class.\_\_counter = 1

print(new\_class.add(3))

# >> 3

# The output is still three because the method add refers to the class definition, not the counter of the new\_class instance

**@propriété**

**1. Getter**

Méthodes utilisées dans la programmation orientée objet (OOPS) qui permettent d'accéder aux attributs privés d'une **classe** .

Sans@property

class MyClass:

def \_\_init\_\_(self, first\_name, last\_name):

self.first\_name = first\_name

self.last\_name = last\_name

def email(self):

return f"{self.first\_name}.{self.last\_name}@gmail.com"

newClass = MyClass("John", "Doe")

print(newClass.email())

# >> John.Doe@gmail.com

Avec@property

class MyClass:

def \_\_init\_\_(self, first\_name, last\_name):

self.\_\_first\_name = first\_name

self.\_\_last\_name = last\_name

@property

def email(self):

return f"{self.\_\_first\_name}.{self.\_\_last\_name}@gmail.com"

newClass = MyClass("John", "Doe")

print(newClass.email())

# >> TypeError: 'str' object is not callable

print(newClass.email)

# >> John.Doe@gmail.com

**2. Setter**

Méthodes utilisées dans la fonctionnalité de programmation orientée objet qui permettent de définir la valeur des attributs privés dans une **classe** .

class MyClass:

def \_\_init\_\_(self, first\_name, last\_name):

self.\_\_first\_name = first\_name

self.\_\_last\_name = last\_name

@property

def email(self):

return f"{self.\_\_first\_name}.{self.\_\_last\_name}@gmail.com"

@email.setter

def email(self, name):

self.\_\_first\_name = name

newClass = MyClass("John", "Doe")

newClass.email = "Sarah"

print(newClass.email)

# >> Sarah.Doe@gmail.com

**Exemple**

class Person:

used\_names = set()

def \_\_init\_\_(self, name, age):

if name in self.used\_names:

name = input("That name is taken. Enter another name: ")

self.name = name

self.age = age

self.used\_names.add(name)

@classmethod

def fromYear(cls, name, birth\_year):

THIS\_YEAR = 2020

return cls(name, THIS\_YEAR - birth\_year)

@property

def professional\_name(self):

return "Mr " + self.name

**Exercice**

Analysez le code ci-dessous. Quel sera le résultat ?

class MyClass(object):

count = 0

def \_\_init\_\_(self, val):

self.val = val

MyClass.count += 1

def set\_val(self, newval):

self.val = newval

def get\_val(self):

return self.val

@classmethod

def get\_count(cls):

return cls.count

object\_1 = MyClass(10)

print("\nValue of object : %s" % object\_1.get\_val())

print(MyClass.get\_count())

object\_2 = MyClass(20)

print("\nValue of object : %s" % object\_2.get\_val())

print(MyClass.get\_count())

**Exercice**

Analysez le code ci-dessous. Quel sera le résultat ?

class MyClass(object):

count = 0

def \_\_init\_\_(self, val):

self.val = self.filterint(val)

MyClass.count += 1

@staticmethod

def filterint(value):

if not isinstance(value, int):

print("Entered value is not an INT, value set to 0")

return 0

else:

return value

a = MyClass(5)

b = MyClass(10)

c = MyClass(15)

print(a.val)

print(b.val)

print(c.val)

print(a.filterint(100))

**Écrire nos propres modules**

**Modules**

L'un des avantages des fonctions est la façon dont elles séparent les blocs de code de votre programme principal. En utilisant des noms descriptifs pour vos fonctions, votre programme principal sera beaucoup plus facile à suivre.

**Qu'est-ce qu'un module ?**

Vous pouvez aller plus loin en stockant vos fonctions dans un fichier séparé appelé **module** , puis **en important** ce module dans votre programme principal.

Une importinstruction indique à Python de **rendre le code d’un module disponible** dans le fichier programme en cours d’exécution.

Stocker vos fonctions dans un fichier séparé vous permet de les réutiliser dans de nombreux programmes. En réservant vos fonctions dans des fichiers individuels, vous pouvez les partager avec d'autres programmeurs sans avoir à partager l'intégralité de votre programme.

Savoir importer des fonctions vous permet également d'utiliser des bibliothèques de fonctions que d'autres programmeurs ont écrites.

Il existe plusieurs façons d'importer un module ; nous les décrirons dans ce chapitre. Mais commençons par **créer notre propre module** !

**Créer un module**

Un module est simplement un fichier Python (un fichier avec l' extension *.py* ).

Créons un fichier appelé pizza.pycontenant la fonction make\_pizza().

def make\_pizza(size, \*toppings):

"""

Summarize the pizza we are about to make.

"""

print(f"\n Making a {size}-inch pizza with the following toppings:")

for topping in toppings:

print("- " + topping)

Nous allons créer un autre fichier appelé making\_pizzas.py(dans le même répertoire que pizza.py).  
Ce fichier **importera la fonction** que nous venons de créer et lui effectuera deux appels.

**Importer un module entier**

Pour importer un module entier :

* utiliser la importdéclaration,
* suivi du nom du module (sans l'extension).

Par exemple, pour importer le pizza.pyfichier :

import pizza

Lorsque Python lit ce fichier, import pizzaindiquez à Python d'ouvrir le fichier **pizza.py** et de copier toutes les fonctions de celui-ci dans ce programme.

Vous ne voyez pas le code copié entre les fichiers car Python copie le code en arrière-plan pendant l'exécution du programme.

Tout ce que vous devez savoir est que toute fonction définie dans **pizza.py** sera désormais **disponible** dans **making\_pizzas.py** .

Pour appeler une fonction à partir d'un module importé, entrez le nom du module que vous avez importé, pizza, suivi du nom de la fonction, make\_pizza(), séparé par un point.

import pizza

pizza.make\_pizza(16, 'pepperoni')

pizza.make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

**Importation de fonctions spécifiques**

Vous pouvez également importer une fonction spécifique depuis un module. Voici la syntaxe générale de cette approche :

from module\_name import function\_name

Vous pouvez importer autant de fonctions que vous le souhaitez à partir d'un module en séparant le nom de chaque fonction par une virgule :

from module\_name import function\_0, function\_1, function\_2

L' making\_pizzas.pyexemple ressemblerait à ceci si nous voulons importer uniquement la fonction que nous allons utiliser :

from pizza import make\_pizza

make\_pizza(16, 'pepperoni')

make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

**Utilisation d'un alias**

Vous pouvez attribuer un surnom à chaque module/fonction importé. Ce surnom, un alias court et unique (un nom alternatif similaire à celui de la fonction), est défini comme « mot-clé ».

from pizza import make\_pizza as mp

La syntaxe générale d'une fonction est

from module\_name import function\_name as alias\_name

Et pour un module :

import module\_name as alias\_name

**Exercice**

Créez un fichier appeléoperators.py

Créez 2 fonctions : addOperator(x,y) qui renvoie l'addition de 2 nombres, et divideOperator(x,y)qui renvoie la division de 2 nombres

Créez un autre fichier nommé calculator.pyet importez le module opérateurs. Appelez les deux fonctions et affichez les résultats.

Effectuez ce processus 3 fois :

* une fois en important le module entier
* la deuxième fois en important des fonctions spécifiques
* la troisième fois en utilisant un alias

**Installation d'un module externe**

Python dispose d'une énorme communauté *open source* , ce qui signifie que de nombreux développeurs mettent leur code source en ligne.

Vous pouvez télécharger leurs modules pour pouvoir utiliser leur fonction.

Python dispose d'un installateur de modules intégré, appelé *pip* (parfois appelé *pip3* ).

Pour vérifier votre installation pip, entrez la commande suivante dans un terminal :

* Pour Linux et Mac OS :pip3 --version
* Pour les fenêtres :python -m pip --version

Pip dispose d'une liste de modules qu'il peut télécharger, disponibles sur [pypi](https://pypi.org/" \t "_blank) .

Pour installer l'un des modules, vérifiez son nom pypiet utilisez :

$ pip install module\_name

**Travailler avec un fichier d'exigences**

Un fichier d'exigences est utilisé pour permettre aux autres développeurs d'installer plus facilement les versions correctes du package Python requis pour exécuter le code Python que nous avons écrit.

La plupart des programmes Python sont fournis avec un fichier texte, requirements.txtcontenant généralement la liste de tous les modules requis utilisés dans votre code Python, ainsi que leur version.  
Il est recommandé de générer et de partager un requirements.txtfichier lors du téléchargement de votre code . Github  
Un requirements.txtfichier ressemble à ceci :

appnope==0.1.0

backcall==0.1.0

beautifulsoup4==4.6.3

bleach==2.1.4

certifi==2018.8.24

chardet==3.0.4

Click==7.0

cycler==0.10.0

decorator==4.3.0

defusedxml==0.5.0

entrypoints==0.2.3

Flask==1.0.2

Vous pouvez installer chaque module requis à partir d'un requirements.txtfichier avec l' pip install -r (filename)argument dans pip.

pip install -r requirements.txt

Vous pouvez générer un fichier d'exigences pour votre projet Python avec *pipreqs* , installez-le avec :

$ pip install pipreqs

Pour générer un fichier d’exigences, exécutez simplement ceci dans votre dossier de projet :

$ pipreqs .

Il existe également une autre façon de procéder en utilisant pip freezel'intérieur de votre dossier de projet :

$ pip freeze > requirements.txt

Cette commande crée le fichier requirements.txtde la même manière pipreqsqu'elle écrirait le fichier.

**Modules**

**Modules**

Les modules sont des fichiers Python uniques qui peuvent être importés et utilisés dans un autre fichier.

**1. importer**

Nous utilisons importl'instruction pour importer un fichier Python dans un autre fichier Python ; elle s'exécutera lors de l'importation du module.  
Dans l'autre fichier Python, toutes les fonctions et classes des modules sont accessibles via module\_name.function\_name.

Par exemple, importez un fichier appelé my\_program.py, écrivez import my\_program-y votre code.

# math module

import math

print(math.e) # Euler's number

# >> 2.718281828459045

La convention dit que vous devez écrire toutes vos déclarations d'importation en haut de la page.

**2. de**

Nous pouvons importer des fonctions spécifiques d'un module avec l' frominstruction.

Il n'est pas nécessaire d'appeler module\_name.function\_name lors de l'utilisation de l'importation from; il vous suffit d'appeler la fonction régulièrement puisqu'elle a été directement importée dans votre application.

Exemple:

from mymodule import myfunction

from math import floor

print(floor(9.8923))

# >> 9

# Instead of

import math

print(math.floor(9.8923))

# >> 9

**3. comme**

Nous pouvons utiliser des alias pour changer le nom du module à l'intérieur du programme.

Exemple:  
import mymodule as mod

# Calculus of the square root of a number.

# alias called 'm'

import math as m

print(m.sqrt(121))

print(m.sqrt(729))

# >> 11

# >> 27

print(m)

# >> <module 'math' from '/usr/local/lib/python3.8/lib-dynload/math.cpython-38-x86\_64-linux-gnu.so'>

print(dir(m))

# >> ['\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

**4. pépin**

Pip est un gestionnaire de paquets Python. Il peut télécharger et installer des paquets de manière autonome ; il les récupère pypi.orget les installe automatiquement à partir d'un paquet téléchargé. Vous pouvez l'utiliser depuis le terminal.

* Emballer

Un package est composé de plusieurs fichiers Python et peut même inclure des bibliothèques en C ou C++ ; c'est un dossier entier.

* \_\_init\_\_.py

En observant la structure des paquets, vous verrez un fichier nommé \_\_init\_\_.py. Il s'agit de l'ancienne syntaxe de création de paquets Python. Ce n'est plus nécessaire aujourd'hui, mais la plupart des paquets sont construits ainsi. Ce fichier est exécuté au chargement du paquet.  
Chaque module du paquet peut être chargé individuellement en utilisantfrom my\_package import my\_module

**5. Quelques exemples de modules**

* temps
* aléatoire
* demandes
* os

**6. Exemple utilisant les modules Fakerettabulate**

*Fonction principale*

if \_\_name\_\_ == "\_\_main\_\_":

*Créer de fausses données à l'aide de faker*

Depuis le terminal, installez le package Faker avec la commande :pip install Faker

Dans votre programme Python, importez Faker et demandez-lui de créer un fichier de fausses dates

from faker import Faker

fake = Faker()

print(fake.date())

Écrivez-les dans deux fichiers différents.

**Créer une fausse date**

from faker import Faker

fake = Faker()

def create\_file(num):

f = open("datefile" + num + ".txt", "w")

for \_ in range(100):

f.write(f"{fake.date()}\n")

f.close()

print(fake.date())

create\_file('1')

create\_file('2')

**Trouve la médiane entre deux dates.**

import sys

import typing

from tabulate import tabulate

def main(file1, file2):

file1\_contents = extract\_file\_contents(file1)

file2\_contents = extract\_file\_contents(file2)

display\_files(file1\_contents, file2\_contents)

dates = sorted(set(file1\_contents).union(file2\_contents))

print(dates[len(dates) // 2])

def display\_files(file1\_contents, file2\_contents):

table = {

'file1': file1\_contents,

'file2': file2\_contents

}

print(tabulate(table))

def extract\_file\_contents(file\_path: str) -> typing.List[str]:

"""

Extract file contents and strip whitespaces from each row.

: param file\_path: The path to the file to extract

: return: A list of date data rows

"""

with open(file\_path, 'r') as f:

file\_contents = f.readlines()

new\_file\_contents = []

for date\_data\_record in file\_contents:

date\_data\_record = date\_data\_record.strip()

if date\_data\_record:

new\_file\_contents.append(date\_data\_record)

return new\_file\_contents

if \_\_name\_\_ == '\_\_main\_\_':

args = sys.argv[1:]

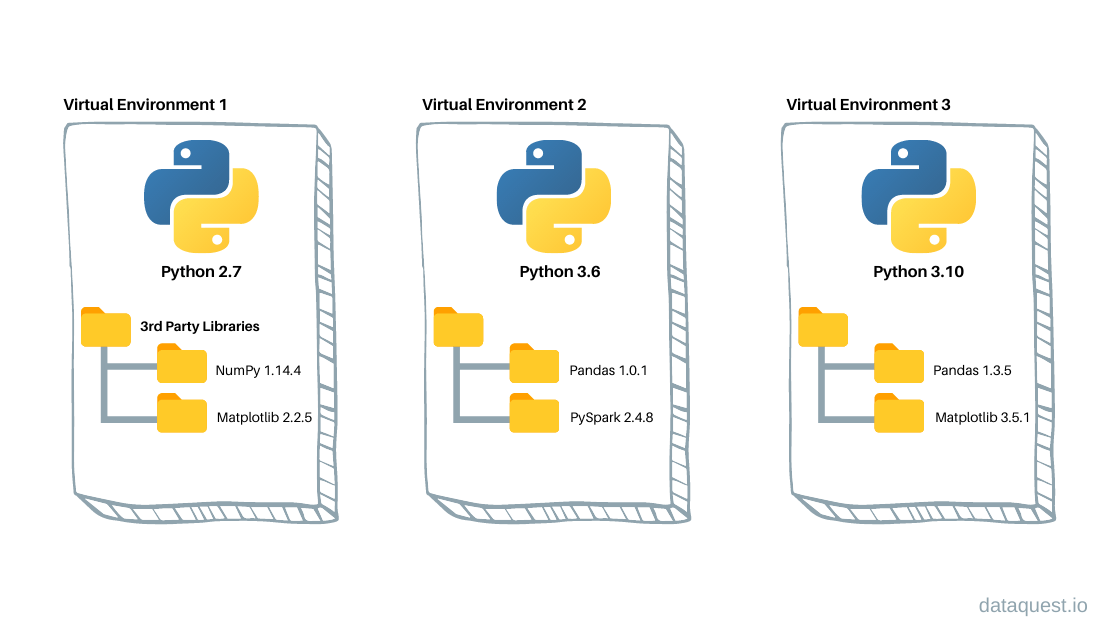
if len(args) != 2:

print(f"Expected two arguments as files. Got {len(args)} instead.")

sys.exit(1)

main(\*args)

**Environnements virtuels en Python**



Pourquoi les environnements virtuels sont utiles et importants :

* **Gestion des dépendances** : différents projets peuvent nécessiter différentes versions du même package, et les environnements virtuels aident à éviter les conflits entre ces dépendances.
* **Reproductibilité** : L'utilisation d'environnements virtuels garantit qu'un projet peut être configuré avec exactement les mêmes dépendances sur n'importe quel système, ce qui facilite le partage et la collaboration avec d'autres.
* **Stabilité du système** : en gardant les dépendances spécifiques au projet isolées, les environnements virtuels aident à empêcher les modifications apportées à votre installation Python à l'échelle du système, ce qui pourrait autrement entraîner des problèmes avec d'autres projets ou applications.

**Étapes pour utiliser un environnement virtuel en Python**

**Comment installer un environnement virtuel en Python**

Pour créer un environnement virtuel, Python fournit un module intégré appelé venv. Voici comment l'installer :

* **Étape 1** : Assurez-vous que Python est installé sur votre système. Vous pouvez le vérifier en exécutant :

python --version

* **Étape 2** : Si Python est installé, vous pouvez installer le venvmodule à l’aide de la commande suivante :

python -m pip install --upgrade pip

**Comment créer un environnement virtuel**

* **Étape 1** : Accédez au dossier de votre projet dans le terminal :

cd /path/to/your/project

* **Étape 2** : Créez un environnement virtuel avec venv. Remplacez env\_namepar le nom souhaité pour votre environnement virtuel :

python -m venv env\_name

Cela créera un dossier nommé env\_namecontenant les fichiers nécessaires à l'environnement virtuel.

**Comment activer l'environnement virtuel**

* **Étape 1** : Pour activer l’environnement virtuel, utilisez la commande suivante en fonction de votre système d’exploitation :
* Sous **Windows** :

env\_name\Scripts\activate

* Sur **MacOS/Linux** :

env\_name/bin/activate

* **Étape 2** : Après l'activation, vous devriez voir le nom de votre environnement virtuel dans l'invite du terminal, comme ceci :

(env\_name) $

Cela indique que l'environnement virtuel est actif et que tous les packages que vous installez seront installés dans cet environnement.

**Comment installer des packages dans l'environnement virtuel**

* **Étape 1** : L'environnement virtuel étant actif, vous pouvez installer les paquets à l'aide de pip. Par exemple, pour installer le requestspaquet :

bash pip install requests

* **Étape 2** : Pour vérifier que le package a été installé, exécutez :

bash pip show requests

**Gestion des erreurs**

Si vous obtenez l’erreur suivante lors de l’exécution des commandes ci-dessus :

pip : The term 'pip' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the

path is correct and try again.

Exécutez cette commande au lieu de simplement « pip » :

py -m pip install requests

**Comment créer un requirements.txtfichier**

Ce requirements.txtfichier permet de lister toutes les dépendances de votre projet. Vous pouvez le créer en suivant ces étapes :

* **Étape 1** : Avec votre environnement virtuel actif, exécutez :

pip freeze > requirements.txt

* **Étape 2** : Cela créera un requirements.txtfichier dans le répertoire de votre projet contenant tous les packages installés et leurs versions.

**Comment désactiver l'environnement virtuel**

* **Étape 1** : Pour désactiver l’environnement virtuel, exécutez simplement :

deactivate

* **Étape 2** : une fois désactivé, l'invite de votre terminal reviendra à son état normal et vous ne travaillerez plus dans l'environnement virtuel.

**Comment supprimer l'environnement virtuel**

* **Étape 1** : Pour supprimer l’environnement virtuel, supprimez simplement le répertoire où se trouve l’environnement virtuel :

rm -rf env\_name

* **Étape 2** : Cela supprimera le env\_namedossier et tout son contenu, y compris les packages installés.

**Résumé**

En utilisant des environnements virtuels, vous pouvez créer des environnements isolés pour chacun de vos projets Python. Cela garantit l'absence de conflits de dépendances entre les différents projets. Grâce aux étapes ci-dessus, vous pouvez créer, activer, installer des packages et gérer efficacement votre environnement virtuel.

**Attribut de classe**

Vous pouvez également définir un attribut propre à la classe et non à chaque instance ; on l'appelle un attribut de classe. Tout élément assigné dans la portée de la classe est un attribut de classe ; chaque instance de la classe partage le même attribut. Par exemple, définissons le nom du roi des chiens comme attribut de classe :

class Dog():

dogs\_king = "Bernie IV"

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

def bark(self):

print(f"{self.name} barks ! WAF")

def walk(self, number\_of\_meters):

print(f"{self.name} walked {number\_of\_meters} meters")

def rename(self, new\_name):

self.name = new\_name

my\_dog = Dog("Rex")

my\_dog.rename("Paul")

La dogs\_kingvariable est maintenant définie comme « Bernie IV », nous n’avons pas besoin d’avoir un Dogobjet pour appeler cette variable ; utilisez :

print("The king of the dogs is:", Dog.dogs\_king)

Par exemple, nous pouvons enregistrer le nombre de chiens jamais créés dans une variable de classe et l'incrémenter à chaque fois qu'un chien est créé (dans la \_\_init\_\_fonction).

class Dog():

number\_of\_dogs = 0

dogs\_king = "Bernie IV"

# Initializer / Instance Attributes

def \_\_init\_\_(self, name\_of\_the\_dog):

print("A new dog has been initialized !")

print("His name is", name\_of\_the\_dog)

self.name = name\_of\_the\_dog

# Increment the number of dogs

Dog.number\_of\_dogs += 1

def bark(self):

print(f"{self.name} barks ! WAF")

def walk(self, number\_of\_meters):

print(f"{self.name} walked {number\_of\_meters} meters")

def rename(self, new\_name):

self.name = new\_name

my\_dog = Dog("Rex")

my\_dog2 = Dog("Bernie V")

print(f"Curently, there are {Dog.number\_of\_dogs} dogs")

**Exercice**

Analysez le code ci-dessous. Quels seront les résultats ?

Expliquer le but des méthodes

class Circle:

color = "red"

def \_\_init\_\_(self, diameter):

self.diameter = diameter

def grow(self, factor=2):

self.diameter = self.diameter \* factor

def get\_color(self):

return Circle.color

circle1 = Circle(2)

print(circle1.color)

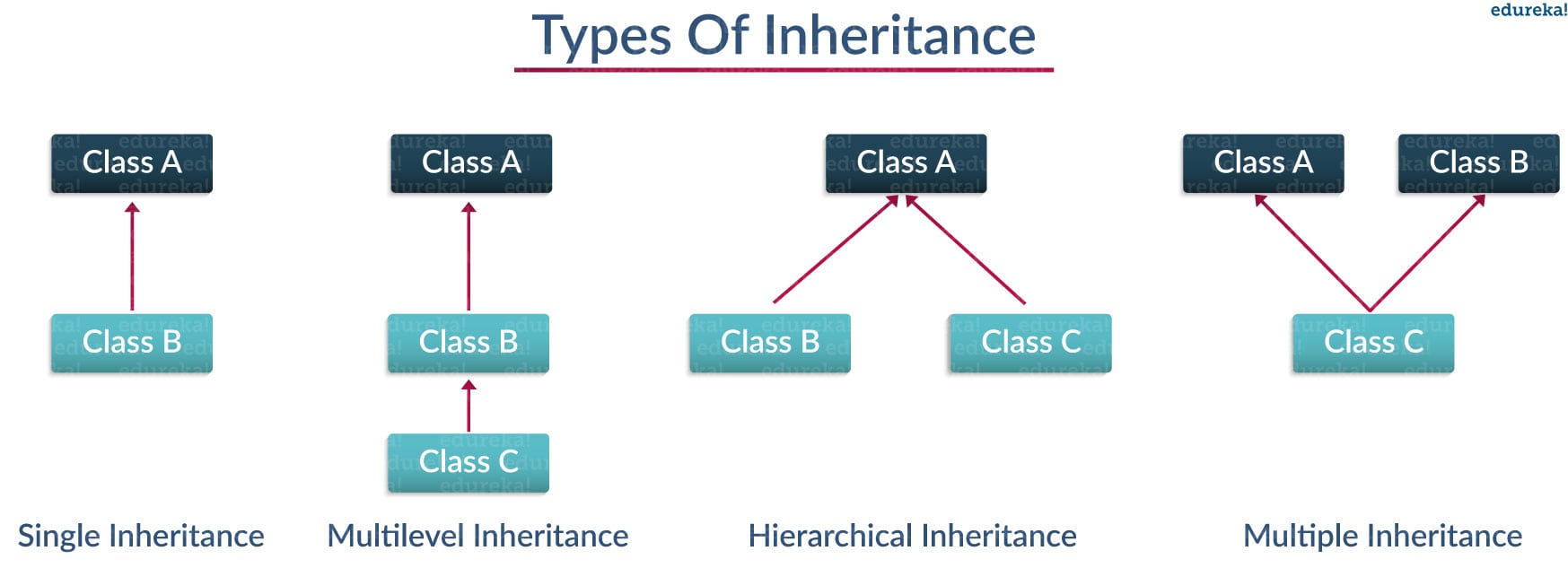
print(Circle.color)

print(circle1.get\_color())

circle1.grow(3)

print(circle1.diameter)

**héritage multiple**



Une classe peut hériter de deux classes différentes ; dans ce cas, l'ordre de la classe parent dans la définition de classe détermine ce qui sera hérité. La première classe parent priorise les classes suivantes (les dernières fonctions héritées peuvent donc remplacer les fonctions de la classe parent).

class Alien():

def \_\_init\_\_(self, name, planet):

self.name = name

self.planet = planet

def fly(self):

print(self.name, 'is flying!')

def sleep(self):

print("Aliens don't sleep")

class Animal():

def \_\_init\_\_(self, name):

self.name = name

def sleep(self):

print("zzzZZZZZ")

class Dog(Animal):

def bark(self):

print(f"{self.name} barked, WAF !")

class AlienDog(Alien, Dog):

def bark(self):

print(f"{self.name} barked, 0ul10ul0u (that's how aliens dogs bark..) !")

my\_normal\_dog = Dog("Roger")

my\_normal\_dog.sleep()

# >> zzzZZZZZ

my\_normal\_dog.bark()

# >> Roger barked, WAF !

my\_alien\_dog = AlienDog("Rex", "Neptune")

print(my\_alien\_dog.planet)

# >> Neptune

my\_alien\_dog.fly()

# >> Rex is flying!

my\_alien\_dog.sleep()

# >> Aliens don't sleep

my\_alien\_dog.bark()

# >> Rex barked, 0ul10ul0u (that's how aliens dogs bark..) !

J'ai créé ici deux nouvelles classes, Alienet AlienDog, AlienDogqui héritent de Alienet Dog, mais toutes deux possèdent une méthode \_\_init\_\_et une sleepméthode. Cependant, les fonctions de Alienseront transférées à AlienDogcar Alienelles sont situées avant Dogdans la définition de classe ( class AlienDog(Alien, Dog)).

**Exercice**

Analysez le code ci-dessous avant de l'exécuter. Quel sera le résultat ? Pourquoi ?

class A():

def dothis(self):

print("doing this in A")

class B(A):

pass

class C():

def dothis(self):

print("doing this in C")

class D(B, C):

pass

d\_instance = D()

d\_instance.dothis()

**Exercice**

Analysez le code ci-dessous avant de l'exécuter. Quel sera le résultat ? Pourquoi ?

class Book():

def \_\_init\_\_(self, title, author, publication\_date, price):

self.title = title

self.author = author

self.publication = publication\_date

self.price = price

def present(self):

print(f'Title: {self.title}')

class Fiction(Book):

def \_\_init\_\_(self, title, author, publication\_date, price, is\_awesome):

super().\_\_init\_\_(title, author, publication\_date, price)

self.genre = 'Fiction'

self.is\_awesome = is\_awesome

if self.is\_awesome:

self.bored = False

print('Woow this is an awesome book')

else:

self.bored = True

print('I am very bored')

if \_\_name\_\_ == '\_\_main\_\_':

foundation = Fiction('Asimov', 'Foundation', '1966', '5£', True)

foundation.present()

print(foundation.price)

print(foundation.bored)

boring\_book = Fiction('boring\_guy', 'boring\_title', 'boring\_date', '9000£', False)

print(boring\_book.bored)

**Méthodes Dunder**

**Méthodes Dunder**

**1. Qu'est-ce qu'une méthode Dunder ?**

Anecdote : en raison de la convention de nommage utilisée pour ces méthodes, on les appelle aussi méthodes dunder, abréviation de « méthodes de **double sous** -score ». On les appelle aussi parfois méthodes spéciales ou méthodes magiques. Nous préférons toutefois les méthodes dunder !

\_\_init\_\_, \_\_str\_\_sont des méthodes stupides.

La magie des méthodes Dunder est que l’invocation est réalisée en coulisses.

*Par exemple :*  
lorsque vous créez une instance x de la classe A avec l'instruction

x = A()

Python effectuera les appels nécessaires à \_\_new\_\_et \_\_init\_\_.

**2. Quelques méthodes Dunder que vous connaissez déjà**

**1. La \_\_init\_\_méthode**

Vous avez déjà remarqué la syntaxe étrange de la \_\_init\_\_méthode ; ces quatre traits de soulignement entourant le nom de la méthode sont là volontairement ; en fait, ils représentent des méthodes uniques que Python appelle automatiquement lorsqu'il en a besoin.

Par exemple, la \_\_init\_\_fonction est la fonction que Python appelle automatiquement lorsqu'il crée un objet.

**2. La \_\_str\_\_méthode**

La \_\_str\_\_fonction est la fonction que Python appelle lorsqu'il doit convertir un objet en chaîne, par exemple lors de son impression.

Par exemple:

mylist = [1, 3, 5]

print(str(mylist))

C'est en fait :

mylist = [1, 3, 5]

print(mylist.\_\_str\_\_())

**3. La \_\_len\_\_méthode**

Cette \_\_len\_\_méthode permet de renvoyer la longueur de l'objet. C'est la méthode appelée lorsque vous tentez d'atteindre len(my\_object).

**4. La \_\_call\_\_méthode**

L'une des méthodes uniques essentielles sera utilisée lorsque vous tenterez d'appeler l'objet (c'est-à-dire en ajoutant ()à la fin du nom : my\_object()).

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_call\_\_(self):

print (f"Person: {self.name}, Age: {self.age}")

person1 = Person("Sarah", 25)

person1()

# Person: Sarah, Age: 25

**5. \_\_repr\_\_et\_\_str\_\_**

La plupart des classes devraient au moins avoir ces méthodes uniques :

* object.\_\_str\_\_:

Appelé par la str()fonction intégrée et par la fonction print pour calculer la représentation *informelle* sous forme de chaîne d'un objet.  
\_\_str\_\_sera toujours une représentation sous forme de chaîne,

* object.\_\_repr\_\_:

Appelé par la repr()fonction intégrée pour calculer la représentation *officielle* de la chaîne d'un objet.  
\_\_repr\_\_peut être un aperçu plus « en coulisses » de l'objet.

**4. Comment utiliser les méthodes Dunder ?**

**Exemple avec\_\_repr\_\_()**

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_repr\_\_(self):

return f"{self.\_\_class\_\_.\_\_name\_\_} : {self.name} {self.age}"

newPerson = Person('Sarah', 24)

print(newPerson)

# >> Person : Sarah 24

# \_\_repr\_\_ is the representation of an object

**Exemple avec\_\_add\_\_()**

class Person:

def \_\_init\_\_(self, name, lastName):

self.name = name

self.lastName = lastName

def \_\_repr\_\_(self):

return f"{self.\_\_class\_\_.\_\_name\_\_} : {self.name} {self.lastName}"

def \_\_add\_\_(self, other):

return Person(self.name, other.lastName)

father = Person('John', 'Snow')

mother = Person('Kaleesi', 'MotherOfDragons')

# using the \_\_add\_\_() method

dragonChild = father + mother

print(dragonChild)

# >> Person : John MotherOfDragons // \_\_add\_\_ is called to add two objects

print(type(dragonChild))

# >> <class '\_\_main\_\_.Person'>

print(dir(dragonChild))

# >> ['\_\_add\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'lastName', 'name']

**5. Comprendre ce qui se passe derrière le rideau**

Quand vous écrivez :

dir(list)

>> ['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_','\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Vous obtenez toutes les méthodes Dunder que vous pouvez utiliser avec le typelist

Donc quand vous écrivez :

first\_list = ["a", "b", "c"]

second\_list = ["d", "e", "f"]

print("\nCalling the `+` builtin with both lists")

print(first\_list + second\_list)

# >> Calling the `+` builtin with both lists

# ['a', 'b', 'c', 'd', 'e', 'f']

print("\nCalling `\_\_add\_\_()` with both lists")

print(first\_list.\_\_add\_\_(second\_list))

# >> Calling `\_\_add\_\_()` with both lists

# ['a', 'b', 'c', 'd', 'e', 'f']

Utiliser l'opérateur + pour ajouter deux listes revient à utiliser la \_\_add\_\_()méthode Dunder avec deux objets (c'est-à-dire les listes)

**6. Comment surcharger une méthode dunder ?**

Surcharger une méthode dunder signifie lui donner une signification étendue au-delà de sa signification native. La surcharge d'une méthode permet de modifier le comportement de fonctions telles que len, abs, hash, divmod, etc. Pour surcharger une méthode, il suffit de définir la méthode correspondante dans votre classe. Prenons quelques exemples :

#To overload a dunder method, we need to implement it in a class.

class Person:

def \_\_init\_\_(self, name, lastName):

self.name = name

self.lastName = lastName

#Here we overloaded the method by redefining '\_\_repr\_\_ 'using 'def' and passed the argument '(self)'

def \_\_repr\_\_(self):

# We can write whatever we want inside this method, but we have to return an object.

return f"{self.\_\_class\_\_.\_\_name\_\_} : {self.name} {self.lastName}"

def \_\_add\_\_(self,other):

name = self.name[0] + other.name[1:]

lastname = other.lastName

return Person(name,lastname)

father = Person('John', 'Snow')

mother = Person('Kaleesi', 'MotherOfDragons')

# using the \_\_add\_\_() method

dragonChild = father + mother

print(dragonChild)

# >>Person : Jaleesi MotherOfDragons

**Exercice**

Analysez le code ci-dessous avant de l'exécuter. Quel sera le résultat ?

class PrintList(object):

def \_\_init\_\_(self, my\_list):

self.mylist = my\_list

def \_\_repr\_\_(self):

return str(self.mylist)

printlist = PrintList(["a", "b", "c"])

print(printlist.\_\_repr\_\_())

**Aide-mémoire sur les méthodes Dunder**

**Méthodes des opérateurs**

Lorsque vous utilisez des opérateurs comme +ou -, il n'y a pas de magie : Python exécute uniquement les méthodes de l'objet à additionner/soustraire. En réalité, Python utilise les méthodes \_\_add\_\_et \_\_sub\_\_.

**opérateurs binaires**

| **Opérateur** | **Méthode** |
| --- | --- |
| + | object.\_\_add\_\_(self, other) |
| - | object.\_\_sub\_\_(self, other) |
| \* | object.\_\_mul\_\_(self, other) |
| // | object.\_\_floordiv\_\_(self, other) |
| / | object.\_\_div\_\_(self, other) |
| % | object.\_\_mod\_\_(self, other) |
| \*\* | object.\_\_pow\_\_(self, other[, modulo]) |
| << | object.\_\_lshift\_\_(self, other) |
| >> | object.\_\_rshift\_\_(self, other) |
| & | object.\_\_and\_\_(self, other) |
| ^ | object.\_\_xor\_\_(self, other) |

**Opérateurs d'affectation**

| **Opérateur** | **Méthode** |
| --- | --- |
| += | object.\_\_iadd\_\_(self, other) |
| -= | object.\_\_isub\_\_(self, other) |
| \*= | object.\_\_imul\_\_(self, other) |
| /= | object.\_\_idiv\_\_(self, other) |
| //= | object.\_\_ifloordiv\_\_(self, other) |
| %= | object.\_\_imod\_\_(self, other) |
| \*\*= | object.\_\_ipow\_\_(self, other[, modulo]) |
| <<= | object.\_\_ilshift\_\_(self, other) |
| >>= | object.\_\_irshift\_\_(self, other) |
| &= | object.\_\_iand\_\_(self, other) |
| ^= | object.\_\_ixor\_\_(self, other) |

**Opérateurs unaires**

| **Opérateur** | **Méthode** |
| --- | --- |
| - | object.\_\_neg\_\_(self) |
| + | object.\_\_pos\_\_(self) |
| abdos() | object.\_\_abs\_\_(self) |
| ~ | object.\_\_invert\_\_(self) |
| complexe() | object.\_\_complex\_\_(self) |
| int() | object.\_\_int\_\_(self) |
| long() | object.\_\_long\_\_(self) |
| flotter() | object.\_\_float\_\_(self) |
| octobre() | object.\_\_oct\_\_(self) |
| hexadécimal() | object.\_\_hex\_\_(self) |

**Opérateurs de comparaison**

| **Opérateur** | **Méthode** |
| --- | --- |
| < | object.\_\_lt\_\_(self, other) |
| <= | object.\_\_le\_\_(self, other) |
| == | object.\_\_eq\_\_(self, other) |
| != | object.\_\_ne\_\_(self, other) |
| >= | object.\_\_ge\_\_(self, other) |
| > | object.\_\_gt\_\_(self, other) |

**Exemple n°1**

class Racecar():

def \_\_init\_\_(self, model, reg\_no, top\_speed=0, nitros=False):

self.model = model

self.reg\_no = reg\_no

self.top\_speed = top\_speed

self.nitros = nitros

if self.nitros:

self.top\_speed += 50

def \_\_str\_\_(self):

return self.model.capitalize()

def \_\_repr\_\_(self):

return f"This is a Racecar with registration: {self.reg\_no}"

def \_\_call\_\_(self):

print(f"Vroom Vroom. The {self.model} Engines Started")

def \_\_gt\_\_(self, other):

if self.top\_speed > other.top\_speed:

return True

else:

return False

def drive(self, km):

print(f"You drove the {self.model} {km} km in {km / self.top\_speed} hours.")

def race(self, other\_car):

if self > other\_car:

print(f"I am the winner")

else:

print(f"The {other\_car.model} is the winner")

class PrintList():

def \_\_init\_\_(self, my\_list):

self.mylist = my\_list

def \_\_repr\_\_(self):

return str(self.mylist)

# printlist = PrintList(["a", "b", "c"])

# print(printlist.\_\_repr\_\_())

# printlist

**Exemple n°2**

from random import choice, randint

class Dog:

def \_\_init\_\_(self, name, age, weight, breed, gender):

self.name = name

self.age = age

self.weight = weight

self.breed = breed

self.gender = gender

def \_\_str\_\_(self):

return "I'm a DOG called " + self.name

def \_\_repr\_\_(self):

return f"Dog: name = {self.name}"

def \_\_len\_\_(self):

if breed == "dachshund":

return self.weight \* 5

else:

return self.weight \* 3

def \_\_gt\_\_(self, other):

return "All dogs are created equal"

def \_\_ge\_\_(self, other):

return self.age >= other.age

def \_\_add\_\_(self, other):

if self.gender == other.gender:

return None

if self.breed != other.breed:

breed = self.breed + "-" + other.breed

else:

breed = self.breed

age = 0

weight = (self.weight + other.weight)/20

return [Dog(f"Puppy {i+1}", age, weight, breed, choice(["M", "F"])) for i in range(randint(1, 6))]

def \_\_mul\_\_(self, other):

return self.\_\_add\_\_(other)