

# Python : fonctions

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille  
Chercheur en programmation par contrainte (IA)  
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



- 1 Introduction
- 2 Déclaration et appel
- 3 Fonction récursive
- 4 Fonction à multi-valeurs de retour
- 5 Fonction retournant une fonction
- 6 Typage
  - Typage de paramètre
  - Typage de valeur de retour
  - Vérification de typage
  - Union de type
  - `Optional`

# Plan

- 7 Paramètres
  - Valeur par défaut
  - Paramètres nommés
  - Opérateur `*`
  - Paramètre `**kwargs`
  - Passage par référence
- 8 Fonction de retour (callback)
- 9 Références de fonction
- 10 Fonction génératrice
- 11 Fonction `eval`
- 12 Variables locales et globales
  - `global`
  - `globals`
- 13 LEGB
- 14 Fonction Lambda
- 15 PEP et les fonctions

## Les fonctions en **Python**

- Un bloc d'instructions réalisant une fonctionnalité bien déterminée.
- Pouvant retourner une ou plusieurs valeurs
- Pouvant prendre 0 ou plusieurs paramètres
- Pouvant appeler d'autres fonctions
- Pouvant retourner une autre fonction
- Pouvant avoir comme paramètre le nom d'une autre fonction

# Python

## Déclarer une fonction

```
def nom_function([parameters]):  
    # instructions
```

© Achref EL MOUELHI ©

# Python

## Déclarer une fonction

```
def nom_fonction([parameters]):  
    # instructions
```

## Exemple

```
def somme(a, b):  
    return a + b
```

© Achre  
QUELHI ©

# Python

## Déclarer une fonction

```
def nom_fonction([parameters]):  
    # instructions
```

## Exemple

```
def somme(a, b):  
    return a + b
```

## Appeler une fonction

```
resultat = somme(1, 3)  
  
print(resultat)  
# affiche 4
```

## Exercice 1

Écrire une fonction `maximum_2(a, b)` qui retourne le maximum entre `a` et `b`.

© Achref EL MOUADJIB

# Python

## Exercice 1

Écrire une fonction `maximum_2(a, b)` qui retourne le maximum entre `a` et `b`.

## Exercice 2

Écrire une fonction `maximum_3(a, b, c)` qui retourne le maximum entre `a`, `b` et `c` (vous pouvez utiliser la fonction `maximum_2`).

## Dans un script Python

- Définissez les fonctions avant de les appeler.
- Si une fonction  $A$  appelle une fonction  $B$ , assurez-vous que la fonction  $B$  est définie avant la fonction  $A$  dans le script.

## Exercice 3

Écrire une fonction `est_premier` qui permet de déterminer si un nombre passé en paramètre est premier. La méthode retourne un booléen.

# Python

## Exercice 4

Écrire une fonction **Python** qui retourne la factorielle d'un nombre passé en paramètre.

## Factorielle : rappel

- $0! = 1$
- $1! = 1$
- $n! = n * (n - 1)!$

# Python

## Solution

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

**Une version récursive de `factorielle` (une fonction récursive est une fonction qui s'appelle elle-même)**

```
def factorielle(n):  
    if (n == 0 or n == 1):  
        return 1  
    return n * factorielle(n - 1)
```

# Python

## Exercice

Écrire une fonction récursive

- acceptant comme paramètre un entier positif  $n$
- retournant le  $n$ ième terme de la suite de **Fibonacci**

## Suite de **Fibonacci**

- $U_0 = 0$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

# Python

## Fonction récursive : avantages

- Clarté du code
- Élégance mathématique

## Fonction itérative : avantages

- Plus performant et moins coûteux en mémoire
- Pouvant souvent être optimisée par le compilateur

© Achref EL M

# Python

## Fonction récursive : avantages

- Clarté du code
- Élégance mathématique

## Fonction itérative : avantages

- Plus performant et moins coûteux en mémoire
- Pouvant souvent être optimisée par le compilateur

## Fonction récursive : inconvénients

- Moins performant et plus coûteux en mémoire
- Difficulté de débogage

## Fonction itérative : inconvénients

- Complexité du code
- Moins compréhensible

# Python

Contrairement à la majorité des langages de programmation, une fonction Python peut retourner plusieurs valeurs

```
liste = [2, 3, 8, 5]
def pairs_et_impairs(liste):
    evens = odds = 0
    for elt in liste:
        if (elt % 2 == 0):
            evens += 1
        else:
            odds += 1
    return evens, odds

print(pairs_et_impairs([2, 3, 5, 8, 1]))
# affiche (2, 3)
```

# Python

Les valeurs retournées par une fonction peuvent être récupérées dans des variables de la manière suivante

```
def pairs_et_impairs(liste):  
    evens = odds = 0  
    for elt in liste:  
        if (elt % 2 == 0):  
            evens += 1  
        else:  
            odds += 1  
    return evens, odds
```

```
pairs, impairs = pairs_et_impairs([2, 3, 5, 8, 1])
```

```
print(f'#pairs = {pairs} et #impairs = {impairs}')
```

# affiche #pairs = 2 et #impairs = 3

## Exercice

- Écrire une fonction `compter_voyelles_consonnes(ch)` qui
  - accepte un paramètre `ch` de type chaîne de caractère
  - retourne le nombre de voyelles et le nombre de consonnes dans `ch`
- Le paramètre reçu peut contenir des voyelles, des consonnes ou des espaces.

# Python

## Une fonction peut avoir comme valeur de retour une autre fonction

```
def b():  
    print("b()")  
  
def a():  
    print("a()")  
    return b  
  
returned_function = a()  
  
returned_function()  
# affiche  
# a()  
# b()
```

# Python

## Pour le cas où les fonctions ont des paramètres

```
def afficher_resultat(ch, result):  
    print(f"Le résultat de la {ch} est {result}")  
  
def somme(a, b):  
    return afficher_resultat("somme", a + b)  
  
somme(2, 5)  
# affiche Le résultat de la somme est 7
```

# Python

Depuis La version 3.5, il est devenu possible de typer les paramètres de la fonction

```
def somme(a: int, b: int):  
    return a + b
```

© Achref EL MOUËZ

# Python

Depuis La version 3.5, il est devenu possible de typer les paramètres de la fonction

```
def somme(a: int, b: int):  
    return a + b
```

Pour tester (rien à changer)

```
resultat = somme (1, 3)  
  
print(resultat)  
# affiche 4
```

# Python

Il est également possible de typer la valeur de retour de la fonction

```
def somme(a: int, b: int) -> int:  
    return a + b
```

© Achref EL MOUËZ

# Python

Il est également possible de typer la valeur de retour de la fonction

```
def somme(a: int, b: int) -> int:  
    return a + b
```

Pour tester (rien à changer)

```
resultat = somme (1, 3)
```

```
print(resultat)
```

```
# affiche 4
```

# Python

## Quelques types pour les fonctions (paramètres et valeur de retour, à importer de `typing`)

- Les types connus autorisés pour les variables
- `Iterable`
- `Any`
- `None`
- `Callable`
- `TypeVar`
- `Union`
- ...

# Python

## Le typage n'est pas vérifié par Python

```
print(somme("2", "3"))  
# affiche 23
```

© Achref EL MOULI

# Python

## Le typage n'est pas vérifié par Python

```
print(somme("2", "3"))  
# affiche 23
```

### Question

Le typage, à quoi sert-il alors ?

# Python

## Typage : avantages

- Documentation améliorée
- Auto-complétion dans les éditeurs

# Python

## Vérification de type

- **Python** ne vérifie pas les types.
- Des packages, comme **MyPy**, permettent de valider et vérifier le respect des types utilisés.

# Python

Pour indiquer qu'un paramètre pourrait avoir plusieurs types, on utilise l'union de type [Depuis Python 3.5]

```
from typing import Union

def somme(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    return a + b

print(somme(2, 5))
# affiche 7

print(somme(2.5, 5.5))
# affiche 8.0
```

# Python

Depuis Python 3.10, l'écriture a été simplifiée avec l'opérateur |

```
def somme(a: int | float, b: int | float) -> int | float:  
    return a + b
```

```
print(somme(2, 5))  
# affiche 7
```

```
print(somme(2.5, 5.5))  
# affiche 8.0
```

# Python

## Exercice

Écrire une fonction **Python** nommée `char_in_string(char, string)` qui

- prend deux paramètres `char` (un caractère) et `string` (une chaîne de caractère)
- retourne la position de la première occurrence de `char` dans `string` (bien sûr si `char` est dans `string`), `None` sinon.

© Achref EL M...

# Python

## Exercice

Écrire une fonction **Python** nommée `char_in_string(char, string)` qui

- prend deux paramètres `char` (un caractère) et `string` (une chaîne de caractère)
- retourne la position de la première occurrence de `char` dans `string` (bien sûr si `char` est dans `string`), `None` sinon.

## Exemple d'appel

```
print(char_in_string('o', 'bonjour'))  
# affiche 1  
  
print(char_in_string('a', 'bonjour'))  
# affiche None
```

# Python

## Solution

```
def char_in_string(char, string) -> int | None:
    for i in range(len(string)):
        if string[i] == char:
            return i
    return None
```

```
print(char_in_string('o', 'bonjour'))
# affiche 1
```

```
print(char_in_string('a', 'bonjour'))
# affiche None
```

# Python

Depuis Python 3.5, l'union de `None` et un autre peut être remplacée par `Optional[Type]`

```
from typing import Optional

def char_in_string(char, string) -> Optional[int]:
    for i in range(len(string)):
        if string[i] == char:
            return i
    return None

print(char_in_string('o', 'bonjour'))
# affiche 1

print(char_in_string('a', 'bonjour'))
# affiche None
```

# Python

Il est possible d'attribuer une valeur par défaut aux paramètres d'une fonction

```
def division(x, y=1):  
    return x / y  
  
print(division(10))  
# affiche 10.0  
  
print(division(10, 2))  
# affiche 5.0
```

# Python

## Considérons la fonction suivante

```
def plus_fois(a=0, b=0, c=1):  
    return (a + b) * c
```

© Achref EL MOUËZ

# Python

## Considérons la fonction suivante

```
def plus_fois(a=0, b=0, c=1):  
    return (a + b) * c
```

On peut appeler la fonction en respectant l'ordre des paramètres (paramètres positionnels)

```
print(plus_fois(2, 3, 5))  
# affiche 25
```

# Python

Ou en nommant les paramètres (et sans avoir besoin de respecter l'ordre)

```
print(plus_fois(c=5, b=3))  
# affiche 15  
  
print(plus_fois(2, c=5, b=3))  
# affiche 25  
  
print(plus_fois(c=5, b=3, a=2))  
# affiche 25
```

## Règles à respecter pour les paramètres nommés

- Pas de paramètres dupliqués : chaque paramètre doit être spécifié une seule fois.
- Si vous utilisez à la fois des arguments positionnels (sans nom) et des arguments nommés dans un appel de fonction, les arguments positionnels doivent être placés en premier, suivi des arguments nommés.

L'opérateur \* peut être utilisé pour

- récupérer les paramètres restants (**packing**)
- décomposer une liste en un ensemble de paramètre (**unpacking**)

# Python

Il est possible de définir une fonction prenant un nombre indéfini de paramètres avec l'opérateur \* (le paramètre `autres` est un tuple)

```
def somme(x, *autres):  
    for elt in autres:  
        x += elt  
    return x  
  
print(somme(2))  
# affiche 2  
  
print(somme(10, 2))  
# affiche 12  
  
print(somme(10, 2, 5))  
# affiche 17  
  
print(somme(10, 2, 5, 3))  
# affiche 20
```

Considérons la fonction `produit` suivante

```
def produit(a, b, c):  
    return a * b * c
```

© Achref EL MOUELHI ©

Considérons la fonction `produit` suivante

```
def produit(a, b, c):  
    return a * b * c
```

Pour appeler la fonction `produit`, il faut lui passer trois paramètres `number`

```
print(produit (1, 3, 5))  
# affiche 15
```

© Achref EL MOU...

**Considérons la fonction** `produit` **suyvante**

```
def produit(a, b, c):  
    return a * b * c
```

**Pour appeler la fonction** `produit`, **il faut lui passer trois paramètres** `number`

```
print(produit (1, 3, 5))  
# affiche 15
```

**Et si les valeurs se trouvent dans une liste, on peut faire**

```
liste = [1, 3, 5]  
print(produit(*liste))
```

Considérons la fonction `produit` suivante

```
def produit(a, b, c):  
    return a * b * c
```

Pour appeler la fonction `produit`, il faut lui passer trois paramètres `number`

```
print(produit(1, 3, 5))  
# affiche 15
```

Et si les valeurs se trouvent dans une liste, on peut faire

```
liste = [1, 3, 5]  
print(produit(*liste))
```

On peut utiliser partiellement la décomposition

```
liste = [1, 3]  
print(produit(*liste, 5))
```

# Python

## Exercice

Écrire une fonction **Python** qui reçoit un nombre variable de paramètres et retourne un dictionnaire contenant les types présents dans la liste `liste` ainsi que leur nombre d'occurrence.

## Exemple

```
print(compute_types(2, 'bonjour', 'True', True, 3, '3', 9, 7, False))  
{'int': 4, 'str': 3, 'bool': 2}
```

# Python

## Une solution possible

```
def returns_types(*params):  
    compteur = {}  
    for e in params:  
        if (type(e).__name__ in compteur):  
            compteur[type(e).__name__] += 1  
        else:  
            compteur[type(e).__name__] = 1  
  
    return compteur
```

# Python

L'opérateur \* peut aussi être utilisé pour forcer les appels à nommer les paramètres

```
def somme(*, a, b):  
    return a + b
```

```
print(somme(2, 3))
```

```
# erreur car les paramètres ne sont pas nommés
```

```
print(somme(a=2, b=3))
```

```
# affiche 5
```

# Python

Il est possible d'utiliser le mot-clé `kwargs` (keyword arguments) pour récupérer les paramètres nommés dans un dictionnaire

```
def dire_bonjour(**kwargs):  
    for key, value in kwargs.items():  
        print(key, value, end=" ")  
  
dire_bonjour(nom="wick", prenom="john")  
# affiche nom wick prenom john  
  
dire_bonjour(nom="abruzzo", age=45)  
# affiche nom abruzzo age 45
```

# Python

Vous pouvez également utiliser la syntaxe `**kwargs` lors de l'appel de fonction en construisant un dictionnaire d'arguments et en le passant à la fonction :

```
def dire_bonjour(**kwargs):  
    for key, value in kwargs.items():  
        print(key, value, end=" ")  
  
kwargs = {"nom": "wick", "prenom": "john"}  
  
dire_bonjour(**kwargs)  
# affiche nom wick prenom john
```

# Python

Exercice : sans tester, qu'affiche le programme suivant ?

```
def my_function(*params, **kwargs):  
    print("params: ", params)  
    print("kwargs: ", kwargs)
```

```
my_function('un', 'deux', 'trois', quatre="4", cinq="5", six="6")
```

© Achref EL

# Python

Exercice : sans tester, qu'affiche le programme suivant ?

```
def my_function(*params, **kwargs):  
    print("params: ", params)  
    print("kwargs: ", kwargs)
```

```
my_function('un', 'deux', 'trois', quatre="4", cinq="5", six="6")
```

Le résultat est :

```
params: ('un', 'deux', 'trois')  
kwargs: {'quatre': '4', 'cinq': '5', 'six': '6'}
```

# Python

## Remarques

- Le terme `kwargs` n'est pas spécial par lui-même : c'est la notation `**` qui est importante en **Python**.
- Vous pouvez remplacer `kwargs` par n'importe quel autre nom de variable valide et le comportement restera le même.
- L'utilisation de `kwargs` comme nom de variable est une convention largement adoptée par les développeurs **Python** pour améliorer la lisibilité du code, mais le mécanisme fonctionnerait avec n'importe quel nom de variable.

# Python

## Remarques

- Une fonction définie avec `**kwargs` ne peut pas être appelée avec des paramètres positionnels.
- Une fonction définie avec `*args` ne peut pas être appelée avec des paramètres nommés, car elle s'attend à des paramètres positionnels.
- Une fonction définie sans les paramètres restant (`*args` et `**kwargs`) peut être appelée avec des paramètres nommés et/ou des paramètres positionnels.

# Python

## Passage par référence

- **Python** ne propose pas de passage par référence explicite comme certains autres langages de programmation.
- Mais, la gestion des objets mutables permet d'obtenir un comportement similaire.
- Pour les objets immuables, il est généralement préférable de renvoyer les nouvelles valeurs et de les affecter aux variables concernées.

# Python

## Fonction de retour (callback)

- fonction appelée comme un paramètre d'une deuxième fonction
- très utilisée en **Python** pour simplifier certaines opérations (sur les tableaux par exemple)

© Achref EL MOU

# Python

## Fonction de retour (callback)

- fonction appelée comme un paramètre d'une deuxième fonction
- très utilisée en **Python** pour simplifier certaines opérations (sur les tableaux par exemple)

## Considérons les deux fonctions suivantes

```
def somme(a, b):  
    return a + b
```

```
def produit(a, b):  
    return a * b
```

## Utilisons les fonctions précédentes comme callback d'une fonction

```
operation()
```

```
def operation(a, b, fonction):  
    print(fonction(a, b))
```

```
# appeler la fonction opération  
operation(3, 5, somme)  
# affiche 8
```

```
operation(3, 5, produit)  
# affiche 15
```

## Exercice

- Modifier la fonction opération pour qu'elle puisse accepter deux fonctions callback ensuite retourner le résultat de la composition des deux.

- Par exemple :

`operation (2, 3 , 6, somme, produit)` retourne  $30 = (2 + 3) * 6$

# Python

## Solution

```
def operation(a, b, c, f1, f2):  
    print(f2(f1(a, b), c))  
  
# appeler la fonction opération  
operation (2, 3 , 6, somme, produit)  
# affiche 30  
  
operation (2, 3 , 6, produit, somme)  
# affiche 12
```

## Références de fonction

- une fonctionnalité qui permet d'appeler une fonction en utilisant une variable contenant son nom.
- si une variable contient le nom d'une fonction, vous pouvez l'appeler avec `()`.

© Achref EL MOULI

# Python

## Références de fonction

- une fonctionnalité qui permet d'appeler une fonction en utilisant une variable contenant son nom.
- si une variable contient le nom d'une fonction, vous pouvez l'appeler avec `()`.

### Exemple

```
def somme(a, b):  
    return a + b  
  
operation = somme  
print(operation(10, 5))  
# affiche 15
```

## Remarque

L'utilisation des fonctions variables peut poser des problèmes de sécurité si elle est utilisée avec des entrées (valeurs saisies par l'utilisateur) non contrôlées.

# Python

En Python, il est possible d'utiliser un dictionnaire pour stocker plusieurs fonctions et les appeler dynamiquement

```
def somme(a, b):  
    return a + b  
  
def produit(a, b):  
    return a * b  
  
operations = {  
    "add": somme,  
    "mul": produit  
}  
  
print(operations["add"](3, 4))  
# appelle somme(3, 4) et affiche 7  
  
print(operations["mul"](3, 4))  
# appelle produit(3, 4) et affiche 12
```

## Exercice

- Demander à l'utilisateur d'entrer deux nombres (a et b).
- Demander à l'utilisateur de choisir une opération parmi +, -, \* et /.
- Exécuter l'opération choisie en utilisant un dictionnaire de fonctions.
- Afficher le résultat du calcul sous la forme "Résultat : a [op] b = résultat".

# Python

## Solution (Première partie)

```
def somme(a, b):  
    return a + b
```

```
def produit(a, b):  
    return a * b
```

```
def soustraction(a, b):  
    return a - b
```

```
def division(a, b):  
    return a / b if b != 0 else "Erreur : Division par zéro"
```

# Python

## Solution (Deuxième partie)

```
operations = {
    "+": somme,
    "-": soustraction,
    "*": produit,
    "/": division
}

a = float(input("a = "))
op = input("op (+, -, *, /) = ")
b = float(input("b = "))

if op in operations:
    resultat = operations[op](a, b)
    print(f"Résultat : {a} {op} {b} = {resultat}")
else:
    print("Opération non valide !")
```

## Pourquoi c'est puissant ?

- Évite les longues structures `if...elif...else`
- Facile à étendre (ajouter une nouvelle opération est simple)

## Fonction génératrice

- définie comme toute autre fonction **Python**
- utilisant le mot-clé `yield` pour générer plusieurs valeurs (même une séquence infinie)
- contrairement aux fonctions normales qui ne peuvent pas contenir plusieurs `return` consécutifs (seul le premier sera exécuté), une fonction génératrice peut contenir plusieurs `yield`
- pouvant utiliser le mot-clé `return`
- très efficace en mémoire par rapport aux fonctions normales qui renvoie une séquence de valeurs entièrement chargée en mémoire

# Python

## Exemple

```
def generateur():  
    for i in range(0, 3):  
        yield i
```

© Achref EL MOUL

# Python

## Exemple

```
def generateur():  
    for i in range(0, 3):  
        yield i
```

Pour appeler la fonction et récupérer les valeurs générées

```
for value in generateur():  
    print(value, end=" ")  
  
# affiche 0 1 2
```

# Python

Pour récupérer les valeurs une par une, on utilise la méthode `__next__()` ou la fonction `next()`

```
x = generateur()
print(type(x))
# affiche <class 'generator'>

print(x.__next__())
# affiche 0

print(next(x))
# affiche 1

print(x.__next__())
# affiche 2

print(x.__next__())
# affiche exception
```

# Python

Utiliser un `return` dans une fonction permet de retourner la dernière valeur

```
def generateur():  
    for i in range(0, 5):  
        if (i == 3):  
            return 3  
        yield i  
  
for value in generateur():  
    print(value, end=" ")  
  
# affiche 0 1 2
```

# Python

Il est possible d'utiliser plusieurs fois `yield`

```
def generateur():  
    yield 0  
    for i in range(1, 5):  
        if (i == 3):  
            return 3  
        yield i  
  
for value in generateur():  
    print(value, end=" ")  
  
# affiche 0 1 2
```

Il est possible de créer une fonction génératrice à partir d'une boucle `for`

```
liste = [2, 3, 8, 5]

double = (elt * 2 for elt in liste)

for i in double:
    print(i, end=" ")

# affiche 4 6 16 10
```

## Exercice

En utilisant `yield`, écrire une fonction génératrice **Python**

- acceptant comme paramètre un entier positif  $x$
- retournant à chaque appel un nombre premier inférieur à  $x$

# Python

## Solution

```
def est_premier(n):  
    for i in range(2, n):  
        if (n % i == 0):  
            return False  
    return True  
  
def get_all_premier(n):  
    for i in range(1, n+1):  
        if (est_premier(i)):  
            yield i  
  
for value in get_all_premier(100):  
    print(value, end=" ")
```

# Python

## Exercice

En utilisant `yield`, écrire une fonction génératrice **Python**

- acceptant comme paramètre un entier positif  $n$
- retournant les  $n$  premiers termes de la suite de **Fibonacci**

## Suite de **Fibonacci**

- $U_0 = 0$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

# Python

## Solution

```
def fibonacci(x):  
    n = 0  
    m = 1  
    yield n  
    yield m  
    for i in range(2, x + 1):  
        result = n + m  
        yield result  
        n = m  
        m = result  
  
for value in fibonacci(10):  
    print(value, end=" ")
```

# Python

La fonction `eval` permet d'évaluer une expression sous forme de chaîne de caractères et retourne le résultat.

```
x = 5
print(eval("2+3*4"))
print(eval("x*3"))
print(eval("type(x)"))
```

© Achref EL M...

# Python

La fonction `eval` permet d'évaluer une expression sous forme de chaîne de caractères et retourne le résultat.

```
x = 5
print(eval("2+3*4"))
print(eval("x*3"))
print(eval("type(x)"))
```

Le résultat est

```
14
15
<class 'int'>
```

L'utilisation de `eval` peut être dangereuse si l'entrée est contrôlée par un utilisateur non fiable, car cela peut exécuter du code malveillant.

```
code = "__import__('os').system('rm -rf /')"  
eval(code)  
# Peut avoir des conséquences graves
```

# Python

Une variable déclarée dans une fonction est dite locale et n'est donc pas accessible de l'extérieur

```
def fonction():  
    x = 200
```

```
fonction()
```

```
print(x)  
# affiche
```

```
Traceback (most recent call last):
```

```
  File "c:/Users/User/cours-python/main.py", line 4, in <module>
```

```
    print(x)
```

```
NameError: name 'x' is not defined
```

# Python

Pour donner une portée globale à une variable, on utilise le mot-clé `global`

```
def fonction():  
    global x  
    x = 200  
  
fonction()  
  
print(x)  
# affiche 200
```

# Python

## Une fonction peut redéclarer une variable globale

```
x = 100
```

```
def fonction():  
    x = 200  
    print(x)
```

```
fonction()  
# affiche 200
```

```
print(x)  
# affiche 100
```

# Python

Dans une fonction, on peut utiliser le mot-clé `global` pour indiquer qu'on utilise la valeur définie dans le contexte global

```
x = 100

def fonction():
    global x
    x = 200
    print(x)

fonction()
# affiche 200

print(x)
# affiche 200
```

# Python

## Remarques

- Il est généralement recommandé de limiter l'usage des variables globales quand c'est possible.
- Elle peut rendre le code plus difficile à comprendre et à maintenir.
- Elle peut être remplacée par l'utilisation des paramètres mutables (comme une liste, un dictionnaire...) ou via des valeurs de retour.

# Python

`globals()` est une fonction qui retourne un dictionnaire contenant les variables globales actuelles de l'espace de noms global du programme

```
print(globals())  
# affiche  
{'__name__': '__main__', '__doc__': None, '__package__': None,  
  '__loader__': <_frozen_importlib_external.SourceFileLoader  
  object at 0x00000206CBABBCE0>, '__spec__': None, '  
  __annotations__': {}, '__builtins__': <module  
'builtins' (built-in)>, '__file__': 'c:\\Users\\elmou\\OneDrive  
  \\Bureau\\test-python\\main.py', '__cached__': None, 'x':  
  20, 'fonction': <function fonction at 0x00000206CBAAA2A0>}
```

# Python

## LEGB

Décrit l'ordre de résolution des noms de variables : l'ordre dans lequel l'interpréteur de **Python** recherche les noms utilisés dans le code

- **Local** : Ce sont les noms définis dans la fonction actuelle. Ils sont recherchés en premier. Cela inclut les variables locales et les paramètres de la fonction.
- **Enclosing** (Englobant) : Ce sont les noms définis dans les fonctions englobantes (les fonctions qui entourent la fonction actuelle). Ils sont recherchés après les variables locales.
- **Global** : Ce sont les noms définis au niveau global du module (en dehors de toute fonction). Si la variable n'est pas trouvée dans les deux précédentes portées, elle est recherchée ici.
- **Built-in** (Intégré) : Ce sont les noms des fonctions et objets intégrés fournis par **Python**, comme `print()`, `len()`... Si le nom n'est trouvé dans aucune des portées précédentes, il est recherché ici.

## Fonctions Lambda

- fonction anonyme
- acceptant un nombre indéfini de paramètres
- limitée à une seule instruction
- déclarée avec le mot-clé `lambda`

# Python

## Syntaxe

```
lambda arguments : instruction
```

© Achref EL MOUELHI ©

# Python

## Syntaxe

```
lambda arguments : instruction
```

## Exemple

```
somme = lambda a, b: a + b
```

© Achref EL... QUELHI ©

# Python

## Syntaxe

```
lambda arguments : instruction
```

## Exemple

```
somme = lambda a, b: a + b
```

## Appel d'une fonction lambda

```
resultat = somme (1, 3)
```

```
print(resultat)  
# affiche 4
```

Cas d'utilisation : les fonctions **lambda** sont souvent utilisées pour travailler sur les listes

- filtrer [à voir dans le chapitre suivant]
- transformer [à voir dans le chapitre suivant]
- rechercher [à voir dans le chapitre suivant]
- trier
- ...

# Python

Étant donnée la liste suivante

```
liste = [2, 3, 8, 5]
```

© Achref EL MOUËL

# Python

Étant donnée la liste suivante

```
liste = [2, 3, 8, 5]
```

Pour trier la liste sans la modifier

```
print(sorted(liste))  
# affiche [2, 3, 5, 8]
```

# Python

Étant donnée la liste suivante

```
fruits = ["abricot", "figue", "Clémentine", "cerise"]
```

© Achref EL MOUELHI ©

# Python

## Étant donnée la liste suivante

```
fruits = ["abricot", "figue", "Clémentine", "cerise"]
```

## Pour trier une liste de chaînes de caractères par leur longueur

```
fruits_tries = sorted(fruits, key=len)
print(fruits_tries)
# affiche ['figue', 'cerise', 'abricot', 'Clémentine']
```

# Python

## Étant donnée la liste suivante

```
fruits = ["abricot", "figue", "Clémentine", "cerise"]
```

## Pour trier une liste de chaînes de caractères par leur longueur

```
fruits_tries = sorted(fruits, key=len)
print(fruits_tries)
# affiche ['figue', 'cerise', 'abricot', 'Clémentine']
```

## Pour inverser l'ordre

```
fruits_tries = sorted(fruits, key=len, reverse=True)
print(fruits_tries)
# affiche ['Clémentine', 'abricot', 'cerise', 'figue']
```

# Python

Pour une liste plus complexe on peut spécifier une fonction lambda

```
employees = [  
    {"nom": "Sophie", "departement": "RH", "salaire": 50000},  
    {"nom": "Marie", "departement": "Comptabilité", "salaire": 52000},  
    {"nom": "Cristophe", "departement": "RH", "salaire": 47000}  
]  
  
employees_tries = sorted(employees, key=lambda x: (x['departement'], x['nom']))  
print(employees_tries)  
# affiche  
# {'nom': 'Marie', 'departement': 'Comptabilité', 'salaire': 52000},  
# {'nom': 'Cristophe', 'departement': 'RH', 'salaire': 47000},  
# {'nom': 'Sophie', 'departement': 'RH', 'salaire': 50000}
```

# Python

## Les règles PEP 8 pour les fonctions

- Ne pas entourer = par des espaces pour les valeurs par défaut des paramètres.

```
def fonction(param=10):
```

- Ne pas entourer = par des espaces lors de l'appel d'une fonction avec des paramètres nommés.

```
fonction(param=10)
```

- Entourer = par des espaces lorsqu'on spécifie le type et une valeur par défaut pour les paramètres.

```
def fonction(param: int = 10):
```

- Entourer -> par des espaces pour les annotations de type de retour.

```
def fonction() -> int:
```

- Les noms de fonctions et de variables doivent respecter le **snake case** : utiliser des minuscules avec des underscores.

```
def ma_fonction():
```