LSIN200A - Cours 4

Coline Gianfrotta

coline.gianfrotta@ens.uvsq.fr

26 février 2026

Les fonctions

- Il est très fréquent d'utiliser la même fonctionnalité plusieurs fois
- La librairie standard de Python ou des librairies externes vont vous fournir des fonctions déjà codées.

Exemples

```
import math

x = 2
math.sqrt(x)
math.cos(x)
print(x)

chaine = "hello"
x = len(chaine)
```

Les fonctions

En mathématiques, une fonction s'écrit $f : x \to f(x)$
Il y a trois parties dans la fonction:

- f son nom
- x son argument
- $f(x)$ son image (valeur/résultat)

Quelques exemples que vous connaissez bien:

$$f(x) = ax + b, \quad f(x) = x^2,$$
$$f(x, y) = x + y, \quad f(x) = 2^x,$$
$$f(x) = |x|$$

Définir une fonction en Python

Syntaxe

```
def nom_fonction(arg1, arg2, ..., argn)
    instruction1
    instruction2
    ...
    instructionn
```

- Une fonction est introduite grâce au mot-clé `def`
- On indique ensuite son nom et on donne sa liste d'arguments entre parenthèses
- Le corps de la fonction est un *bloc d'instructions*, donc indenté, qui vient après sa déclaration.

Définir une fonction en Python

Un premier exemple

```
def multiplie(x):
    """ Cette fonction multiplie son argument par 2
    et l'affiche """
    print(x*2)

multiplie(7)
multiplie("aaa")
```

- Le bloc de la fonction peut être précédé d'un commentaire, appelé **docstring**, qui sert à générer la documentation automatique de help(). Un docstring est entouré de trois guillemets : `""" docstring """`.

Ecrire une fonction

- Les fonctions retournent une **valeur**, donnée après le mot clé `return`
- Même les fonctions qui n'ont pas d'instruction `return`, ou qui terminent en finissant d'exécuter le bloc de code sans atteindre un `return`, retournent la valeur `None`.

Syntaxe

```
def nom_fonction(arg1, arg2, ..., argn)
    instruction1
    instruction2
    ...
    return objet
```

Avec return

```
def multiplie(x):
    return x*2

print(type(multiplie(2)))
```

Sans return

```
def multiplie(x):
    print(x*2)

print(type(multiplie(2)))
```

Signature d'une fonction

- En programmation, la *signature d'une fonction* est donnée par son nom, les paramètres (arguments), leur type et le type renvoyé.
- En Python, pas besoin de donner le type renvoyé lors de la déclaration, ni le type des paramètres.

Que se passe-t-il lors de l'appel de la fonction ?

```
def somme(x,y):
    return x+y

print(somme("hello", 1))
```

Sortir de la fonction

- Attention, une fonction termine dès qu'elle atteint le mot-clé `return`
- C'est utile pour gérer le flot d'exécution d'une fonction

```
def un():
    return 1
    print("je ne serai jamais atteint")
un()

def cherche(x,liste):
    for elem in liste:
        1 / elem #erreur quand elem vaut 0
        if(x == elem):
            return 1
    return 0

cherche(10, [1,10,0])
#la valeur 0 pas atteinte car on trouve 10 avant
```

Type de return

- En Python, on peut utiliser des types de retour complexes (contrairement à C par exemple).
- On peut par exemple renvoyer des tuples de valeurs, ce qui est souvent pratique.

Exemple

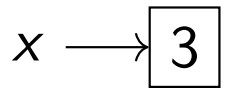
```
def ordonne(a, b):
    if a < b:
        return a, b
    else:
        return b, a

print(ordonne(10, 5))
```

Types mutables et immutables

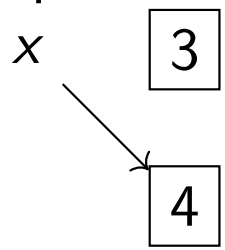
En Python, on distingue deux types d'objets:

- les types dits **immutables**, c'est-à-dire qu'on ne peut pas changer
  - entiers, flottants
  - tuples
  - chaînes de caractère

```
x = 3
x += 1
```



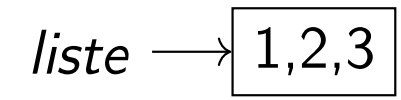
Après incrémentation:



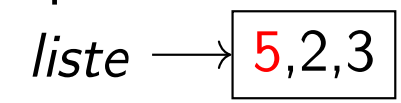
Types mutables et immutables

- les types dits **mutables**, c'est-à-dire dont on peut modifier le contenu
  - listes
  - ensembles (sets)
  - dictionnaires

```
liste = [1,2,3]
liste[0] = 5
```



Après modification:



Questions

Que se passe-t-il à l'exécution ?

```
mon_tuple = (1,2,3)
mon_tuple[0] = 5
```

```
chaine = "hello world"
chaine[0] = 'H'
```

Les variables d'une fonction

- Les variables définies dans une fonction sont appelées variables locales
- La portée d'une variable locale est limitée à la fonction où elle a été définie, on ne peut y faire référence en dehors.

Exemple

```
def incremente(x):
    x += 1
    z = 2

incremente(2)
print(x)
print(z)
```

Lors de l'exécution, l'affichage produit une erreur (NameError, variable non définie).

Evaluation de la fonction

- Les arguments sont des variables locales à la fonction
- Un argument est passé comme **copie** de la référence sur l'objet donné en argument

Exemple

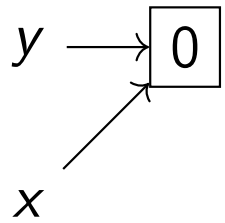
```
def incremente(x):
    x += 1

y = 0
incremente(y)
print(y)
```

Evaluation de la fonction

- Les arguments sont des variables locales à la fonction
- Un argument est passé comme **copie** de la référence sur l'objet donné en argument

Exemple

```
def incremente(x):
    x += 1

y = 0
incremente(y)
print(y)
```



⟵ à cette étape du programme

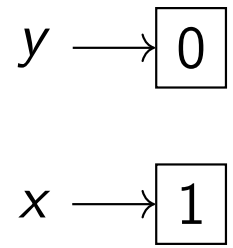
Evaluation de la fonction

- Les arguments sont des variables locales à la fonction
- Un argument est passé comme **copie** de la référence sur l'objet donné en argument

Exemple

```
def incremente(x):
    x += 1

y = 0
incremente(y)
print(y)
```

⟵ à cette étape du programme

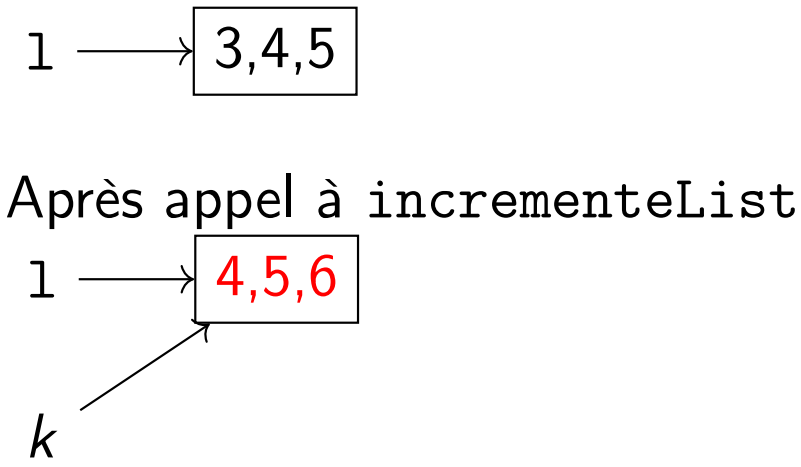


Evaluation de la fonction

- Pour les variables de type **immutables**, une modification du contenu de la variable passée en paramètre n'est donc pas conservée en dehors de la fonction
- En revanche, pour les types **mutables** comme les listes, on peut modifier le contenu de la variable depuis une fonction et conserver cette modification en dehors de la fonction

Exemple

```
def incrementeListe(k):
    for i in range(len(k)):
        k[i] += 1

l = [3,4,5]
incrementeListe(l)
print(l)
```

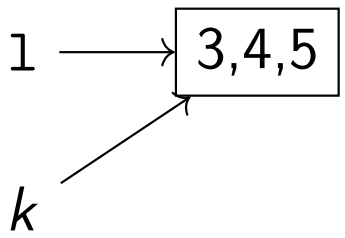


Evaluation de la fonction

- Mais attention, on ne peut pas modifier la variable de type liste elle-même !

Exemple

```
def supprimeListe(k):
    k = []

l = [3,4,5]
supprimeListe(l)
print(l)
```



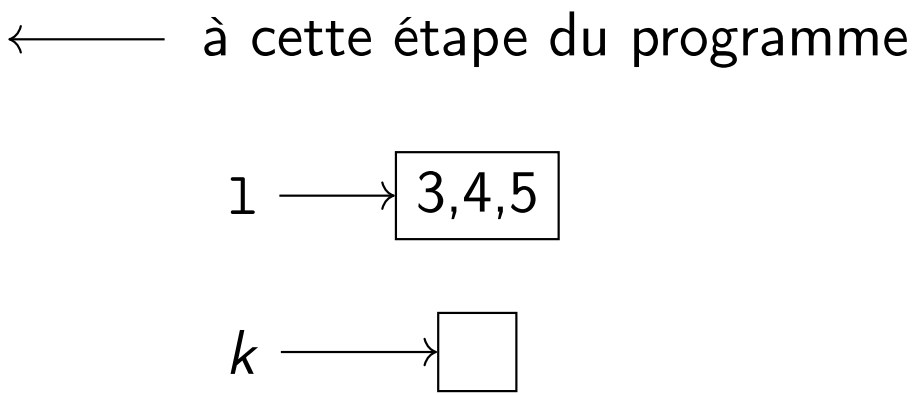
⟵ à cette étape du programme

Evaluation de la fonction

- Mais attention, on ne peut pas modifier la variable de type liste elle-même !

Exemple

```
def supprimeListe(k):
    k = []

l = [3,4,5]
supprimeListe(l)
print(l)
```



Utilisation de variables globales

- Les variables définies hors des fonctions sont **globales** par opposition aux variables **locales** des fonctions.
- On peut les utiliser dans toutes les fonctions
- Cependant, attention, on ne peut pas changer la valeur d'une variable globale à l'intérieur d'une fonction.

```
x = 1 # x est une variable globale
print(x)

def affiche(): #utilisation d'une variable globale
    print(x)
x += 3
affiche()

def ajoute(): #tentative de modification d'une variable globale
    x += 1
ajoute()
print(x)
```

Utilisation de variables globales

- Pour modifier une variable globale, il faut spécifier dans la fonction que la variable est globale par le mot-clé `global`.
- Il est cependant déconseillé de faire usage de ce mot-clé et des variables globales en général

```
x = 1
def ajoute(): # modif. d'une var. globale avec
              # le mot-clé "global"
    global x
    x += 1

ajoute()
print(x)
```

Utilisation de variables globales

Qu'affiche-t-on à l'exécution ?

```
x = 3

def ajoute(): # redéfinir une variable globale
    x = 5
    x += 1

ajoute()
print(x)
```

Ici, une version locale de la variable x est définie dans la fonction.

Représentation des arguments dans un appel de fonction

- On peut donner des valeurs par défaut aux arguments d'une fonction de la manière suivante:
  `def ma_fonction(pays, age = 1, nom = "toto")`
- Les variables ayant une valeur par défaut peuvent être omises.

```
def ma_fonction(pays, age = 1, nom = "toto"):
    print(nom, " a", age, "ans et vit en ", pays)

ma_fonction("France")
ma_fonction("Allemagne", 18, "kurt")
ma_fonction("Italie", 77)
```

Ordre des arguments dans un appel de fonction

- En règle générale, on respecte l'ordre des arguments donné dans la signature
- Mais, on peut également donner les arguments dans le désordre en spécifiant leur nom.

```
def ma_fonction(pays, age = 1, nom = "toto"):
    print(nom, " a", age, "ans et vit en ", pays)

ma_fonction(nom = "Kader", age = 18, pays = "Algérie")
ma_fonction("France", nom = "Sylvie")
```

Appel de fonction depuis une autre fonction

Une fonction peut appeler une autre fonction.

```
def doubler(x):
    return 2*x

def sommer(x,y):
    return x+doubler(y)

print(sommer(2,3))
```

Appel de fonction depuis une autre fonction

Qu'affiche-t-on à l'exécution ?

```
def doubler(x):
    return 2*x

def doubler_liste(l):
    for i in range(len(l)):
        doubler(l[i])
    return l

l = [2,3,4]
print(doubler_liste(l))
```

Si besoin, indiquer comment modifier ce code pour afficher les valeurs de la liste doublées.

Expressions lambda

- Pour définir une fonction courte, il existe une syntaxe alternative utilisant l'opérateur **lambda**.
- La définition doit tenir sur une ligne. On ne peut pas utiliser d'instructions de contrôle (condition if, boucle, etc...)

```
g = lambda x: x*2
print(g(2))
```

Expressions lambda

- La fonction définie par un lambda est anonyme, c'est à dire qu'elle n'a pas de nom.
- Cela peut être utile quand la fonction ne sert qu'une fois

```
# on applique la fonction au paramètre 3
print((lambda x: x*2)(3))

# on applique la fonction à chaque élément de range(10)
list(map(lambda x: x*2, range(10)))
```