

Fonctions

Polytech Marseille

Séverine Dubuisson, Simon Vilmin

severine.dubuisson@univ-amu.fr,
simon.vilmin@univ-amu.fr

2024 - 2025



Au programme

Définir et utiliser une fonction

- Principe et syntaxe

- Docstrings et spécifications

- Résumé

Aspects avancés

- Arguments optionnels

- Retour sur la portée des variables

- Méthodes et fonctions

- Python et `return`

- Résumé

Fonctions récursives

- Principe et syntaxe

- Pourquoi faire ?

- Résumé

Définir et utiliser une fonction

Définir et utiliser une fonction

- Principe et syntaxe

- Docstrings et spécifications

- Résumé

Aspects avancés

Fonctions récursives

Exercice

⚙️ **Exercice** : écrivez un programme qui demande à l'utilisateur·ice de saisir trois entiers `annee`, `mois`, `jour` qui représentent sa date de naissance et qui l'affiche au format `jj/mm/aaaa`. Les contraintes sont :

- `1890 <= annee <= 2024`
- `1 <= mois <= 12`
- `1 <= jour <= maxjour` où `maxjour` est le nombre de jours de mois

```
# ==== Exemple
```

```
Entre une annee entre 1890 et 2024 : 1920
```

```
Entre un mois entre 1 et 12 : 3
```

```
Entre un jour entre 1 et 31 : 10
```

```
T'es ne.e le 10/03/1920
```

⚙️ **Exercice** : modifiez le programme précédent pour qu'il affiche "`mauvaise saisie !`" à chaque mauvaise saisie de l'utilisateur·ice.

Une solution

```
annee, mois, jour = 0, 0, 0

while not(1890 <= annee <= 2024):
    annee = int(input("Entre une annee entre 1890 et 2024 : "))
    if not(1890 <= annee <= 2024):
        print("mauvaise saisie !")
while not(1 <= mois <= 12):
    mois = int(input("Entre un mois entre 1 et 12 : "))
    if not(1 <= mois <= 12):
        print("mauvaise saisie !")

maxjour = 31
if mois == 2:
    maxjour = 28
elif mois == 4 or mois == 6 or mois == 9 or mois == 11:
    maxjour = 30

while not(1 <= jour <= maxjour):
    jour = int(input(f"Entre un jour entre 1 et {maxjour} : "))
    if not(1 <= jour <= maxjour):
        print("mauvaise saisie !")
print(f"T'es ne.e le {jour:0>2d}/{mois:0>2d}/{annee}")
```

Fonctions

❌ **Problème** : c'est l'enfer de répéter le code comme ça ...

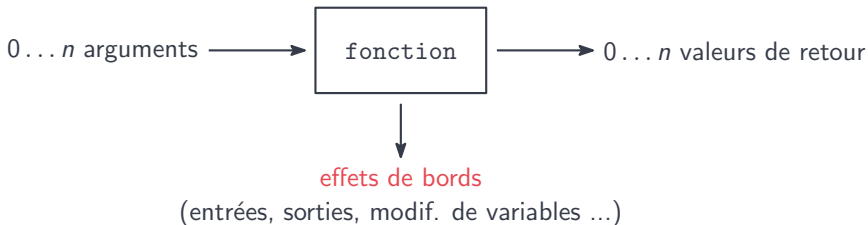
💡 **Idée** : on va utiliser des *fonctions*

📖 **Définition** : une *fonction* est un ensemble d'instructions vu comme une unité et réalisant une tâche particulière. Cette unité peut être *appelée* à tout moment d'un programme pour effectuer la tâche en question.

Les fonctions permettent de :

- *maximiser* la *réutilisation de code* et *minimiser* la *redondance*
- *structurer* et *décomposer* un programme

Principe



Une fonction en Python est une « *boîte* » *indépendante* qui :

- a de 0 à n *arguments*, ou *paramètres*
- *retourne* (ou *renvoie*) de 0 à n *valeurs de sortie*
- peut avoir des *effets de bord* qui modifient l'environnement :
 - interactions entrées/sorties
 - modifications de variables en dehors de la fonction
 - ...


Syntaxe

 **Syntaxe :** `def` pour définir une fonction, `return` pour renvoyer des valeurs :

```
def fonction(arg1, arg2, ...):  
    instructions  
    return valeurs  
  
# ==== Prog principal  
res = fonction(var1, var2, ...)  
# pas oblige d'affecter le resultat a une variable !
```

Remarque :

- on peut mettre *aucun* ou *plusieurs* `return`
- une fonction peut renvoyer *différents types* de valeurs

 **Attention :** attention à l'*indentation* et aux « : » !

Exemple

```
def somme(n):  
    res = 0  
    for i in range(1, n + 1, 1):  
        res += i  
    return res  
  
# ==== Prog principal  
n = 4  
print(f"la somme des {n} premiers entiers est {somme(n)}")  
  
# ==== Resultat  
la somme des 4 premiers entiers est 10
```

Exemple

```
def suppr_ind(phrase, ind):
    # slicing : on renvoie phrase moins la lettre a l'indice ind
    return phrase[:ind] + phrase[ind + 1:]

def suppr_lettre(phrase, lettre):
    res = phrase
    # find donne l'indice de la 1ere apparition de lettre dans res, et -1 sinon
    ind = res.find(lettre)
    while ind >= 0:
        res = suppr_ind(res, ind)
        ind = res.find(lettre)

    return res

# ==== Prog principal
phrase = "salut a toi jeune entrepreneur"
print(suppr_lettre(phrase, "e"))

# == Resultat
salut a toi jun ntrprnur
```

Exemple

```
def f1(x):  
    return 3*x - 5  
  
def f2(x):  
    return 5*x**2 - 7*x + 12  
  
def derivee(x, h, f):  
    return (f(x + h) - f(x)) / h  
  
# ==== Prog principal  
print(f"{derivee(3, 0.001, f1):.2f}")  
print(f"{derivee(3, 0.001, f2):.2f}")  
  
# ==== Resultat  
3.000  
23.005
```

Exercice

⚙️ **Exercice** : écrivez une fonction `reste` qui prend en paramètres deux entiers `n` et `m`, et qui renvoie le reste de la division euclidienne de `n` par `m`.

⚙️ **Exercice** : écrivez une fonction `echange` qui prend en paramètres 3 chaînes de caractères `phrase`, `carac1` et `carac2` et qui renvoie la chaîne de caractères obtenue en remplaçant dans `phrase` toutes les occurrences de `carac1` par `carac2` et vice-versa. On suppose que `carac1` et `carac2` sont juste des lettres. Par exemple,

```
| echange("l'elephant promene un ecureuil", "e", "u")
```


doit renvoyer

```
| "l'uluphant promunu en ucerueil"
```

Notion de portée des variables

 **Définition** : la *portée*, ou le *scope*, d'une variable est la zone du code où elle est accessible.

- chaque fonction a son propre « espace » de variables,
- une variable *initialisée* ou *réaffectée* dans une fonction est *locale* à celle-ci, la fonction est leur *scope* :
 - *accessible* dans (et seulement dans) la *fonction*
 - *pas accessible* depuis le *programme principal*
 - *pas accessible* depuis les *autres fonctions* définies ailleurs dans le code
- une variable définie dans le programme principal est *globale*

 **Remarque** : une variable globale *peut avoir le même nom* qu'une variable locale, mais le *contexte local prévaudra*.

Exemple

```
def somme(n): # n de la fonction somme
    s = 0
    for i in range(1, n + 1):
        s += i
    return s

def message(n, s): # n de la fonction message
    return f"la somme des {n} premiers entiers est : {s}"

# ==== Prog Principal
n = 12 # n du programme principal
print(message(n, somme(n)))

# ==== Resultat
la somme des 12 premiers entiers est : 78
```

Exercice

⚙️ **Exercice** : écrivez un programme qui demande à l'utilisateur·ice de saisir trois entiers `annee`, `mois`, `jour` qui représentent sa date de naissance et qui l'affiche au format `jj/mm/aaaa`. Les contraintes sont :

- `1890 <= annee <= 2024`
- `1 <= mois <= 12`
- `1 <= jour <= maxjour` où `maxjour` est le nombre de jours de mois

```
# ==== Exemple
```

```
Entre une annee entre 1890 et 2024 : 1920
```

```
Entre un mois entre 1 et 12 : 3
```

```
Entre un jour entre 1 et 31 : 10
```

```
T'es ne.e le 10/03/1920
```

⚙️ **Exercice** : modifiez le programme précédent pour qu'il affiche "**mauvaise saisie !**" à chaque mauvaise saisie de l'utilisateur·ice.



Astuce : cette fois-ci, on peut utiliser des fonctions !

Ne rien renvoyer

 **Syntaxe** : au moins *deux manières* de ne rien renvoyer dans une fonction


```
def fonction(arg1, arg2, ...):  
    instructions  
    # pas de return du tout
```

```
def fonction(arg1, arg2, ...):  
    instructions  
    return # return mais sans rien avec
```


Multi-affectation, multiples retours

 **Syntaxe** : Python permet la *multi-affectation* de variables :

```
var1, var2, var3 = val1, val2, val3
```

 **Syntaxe** : c'est ce que l'on utilise quand renvoie plusieurs valeurs dans une fonction :

```
def fonction():  
    instructions  
    return val1, val2  
  
# ==== Prog principal  
x1, x2 = fonction() # x1 = val1, x2 = val2
```

Exercice : variation de AMU

⚙️ **Exercice** : écrivez un programme qui gère les notes des étudiant·e·s d'AMU.
Le programme :

1. demande le nom et le prénom d'un·e étudiant·e
2. demande un nombre de notes, puis demande la saisie de ces notes et en calcule la moyenne. Si une note est ≤ 6 , la moyenne est 0
3. affiche le résultat sous la forme « prénom, nom : moyenne / 20 »

À la suite de ça, le programme demande à l'utilisateur·ice s'il y a d'autres étudiant·e·s à saisir et boucle si tel est le cas.



Astuce : écrivez une fonction par grande « étape »

Docstring



Rappel : commentaires longs avec des triple-guillemets `"""`.




Syntaxe : les `"""` servent aussi à *documenter* des fonctions via les *docstring*

```
def fonction(n, m):  
    """renvoie la somme de n et m"""  
    return n + m  
  
# la fonction help() pour avoir de l'aide...  
help(fonction)  
  
# ==== Resultat  
fonction(n, m)  
    renvoie la somme de n et m
```




Attention : attention à l'*indentation* des `"""`

Spécifications

 **Syntaxe** : la *spécification* d'une fonction est la description de son *comportement*, de ses *arguments*, de ses *valeurs de retour*, ses *erreurs*, etc

```
def fonction(arg1, arg2, ...):  
    """ fonction qui fait ...  
  
    Arguments :  
    arg1 -- ...  
    arg2 -- ...  
  
    Sortie :  
    sor1 -- ...  
    """  
  
    instructions
```

 **Remarque** : pensez à *documenter* les fonctions, c'est-à-dire, écrire leurs *spécifications*. <https://peps.python.org/pep-0257/>

Exemple

```
def affine(a, b, y):  
    """ Resout l'equation  $ax + b = y$ .  
  
    Arguments:  
    a -- coefficient directeur  
    b -- ordonnee a l'origine  
    y -- ordonnee du point a identifier  
  
    Sortie:  
    -- resultat de l'equation, None si pas de solution,  
    0.0 si infinite de solution  
    """  
  
    if a == 0.0 and b != y:  
        print("Pas de solution, on retourne None")  
        return None  
    elif a == 0.0:  
        print("Infinite de solutions, on retourne 0.0")  
        return 0.0  
    else:  
        print("Unique solution")  
        return (y - b) / a
```

Exercice

⚙️ **Exercice :** (*générateur de latin nul*) écrivez une fonction `latin` qui prend en paramètre une phrase en français et qui ajoute aux mots l'une des terminaisons latines suivantes : `ibus`, `um`, `ae`, `us`.

```
# ==== Exemple
phrase = "bien le bonjour frere Simon"
print(latin(phrase))

# ==== Resultat
bien le bonjourus frereum Simonibus
```

Remarque :

- les mots sont séparés par des espaces
- libre à vous de choisir comment les terminaisons doivent être ajoutées (ça peut dépendre de la taille, de la fin, ...) !
- utiliser des fonctions pour segmenter votre programme
- on peut utiliser la fonction `randint` de `random`

Exercice

⚙️ **Exercice** : écrivez un programme qui étant donné un entier n affiche :

- tous les nombres premiers compris entre 0 et n , ET
- tous les nombres parfaits compris entre 0 et n

i Remarque :

- un nombre est *premier* s'il est différent de 1 et qu'il n'est divisible que par 1 et par lui-même
- un nombre est *parfait* s'il est la somme de ses diviseurs propres

```
# ==== Exemple
```

```
pour n = 50
```

```
nombre premiers : 2 3 5 7 11 13 17 19 23 27 29 31 37 41 43 47
```

```
nombre parfaits : 6 28
```

Quelques bonnes pratiques

💡 **Idée :** dans le but de simplifier et rendre lisible du code :

- une fonction fait *une chose* et *c'est tout*
- *moins* d'arguments = *moins de tests* et *plus de clarté*
- les fonctions les plus *courtes* sont les meilleures
- *documenter* ses fonctions
- *éviter* les *variables globales*

Résumé



Rappel :

- une fonction permet de *structurer* le code et éviter la *duplication*
- « *boîte* » *indépendante* du programme principal avec ses propres (noms de) variables
- définie avec `def`, `return` pour renvoyer de 0 à n des valeurs



Attention : penser à la *portée des variables* et aux *effets de bord* !

```
def frequence(a, b):  
    res = 0  
    for i in range(a, b, 1):  
        if premier(i):  
            print(i, end=" ")  
            res += 1  
    return res / (b - a)
```

Aspects avancés

Définir et utiliser une fonction

Aspects avancés

- Arguments optionnels

- Retour sur la portée des variables

- Méthodes et fonctions

- Python et `return`


- Résumé

Fonctions récursives

Arguments optionnels

 **Syntaxe** : on peut donner une *valeur par défaut* aux arguments, ce qui les rend *optionnels* :

```
def fonction(arg1, arg2, ..., arg_opt=defaut):  
    instructions  
  
# on ne precise pas arg_opt, il vaudra default  
fonction(val1, val2, ...)  
  
# on reprecise le nom de l'argument optionnel, il vaudra valeur  
fonction(val1, val2, ..., arg_opt=valeur)
```

 **Remarque** : les arguments optionnels peuvent être utiles quand le comportement d'une fonction ne change que pour quelques valeurs

Exemple

```
def decalage(phrase, shift=1, minus=""):
    resultat = ""
    alphabet = "abcdefghijklmnopqrstuvwxyz"

    for c in phrase:
        # str.find(c) renvoie l'indice de la première apparition de c
        # dans str, et -1 si c n'est pas dans str
        ind = alphabet.find(c)

        if ind >= 0 and c not in minus:
            resultat += minus[(ind + shift) % len(minus)]

    return resultat

# ==== Prog principal
print(decalage("bonjour"))
print(decalage("bonjour", shift=2, minus="o"))

# ==== Resultat
cpokpvs
doplowl
```

Exemple

```
def affine(a, b, y=0.0): # par défaut, y=0.0
    if a == 0.0 and b != y:
        print("Pas de solution, on retourne None")
        return None
    elif a == 0.0:
        print("Infinite de solutions, on retourne 0.0")
        return 0.0
    else:
        print("Unique solution")
        return (y - b) / a

# ==== Prog principal
print(affine(1., 2.))
print(affine(2., 3.5, y=7.))

# ==== Resultat
Unique solution
-2.0
Unique solution
1.75
```

Portée des variables

Rappel :

- chaque fonction définit son propre espace de variables
- le *scope* d'une variable est défini par l'endroit où elle est *(ré)affectée*
- le programme principal a lui aussi un espace « général »
- une variable définie dans une fonction est *locale*, elle est *globale* si elle est définie au niveau du programme principal

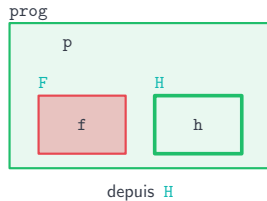
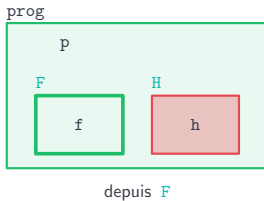
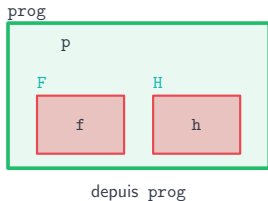
Astuce :

- les espaces sont *hiérarchisés*
- on a accès aux variables des *espaces* « *du dessus* »
- l'espace où l'on est *prioritaire*

Exemple

```
def F():  
    f = "f"  
  
def H():  
    h = "h"  
  
# ==== Prog principal  
p = "p"
```

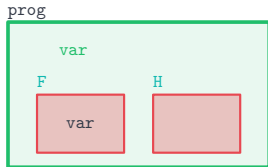
accessibilité	p	f	h
depuis prog	✓	✗	✗
depuis F	✓	✓	✗
depuis H	✓	✗	✓



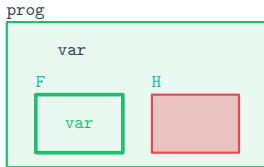
Exemple

```
def F():  
    var = "f"  
  
def H():  
    print(var)  
  
# === Programme principal  
var = "p"
```

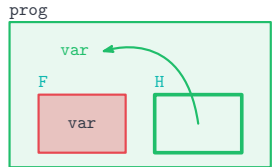
valeur	var
dans prog	"p"
dans f	"f"
dans h	"p"



dans prog




dans F



dans H

Éviter les variables globales


 **Attention** : on essaye *d'éviter* au maximum les *variables globales* !

```
def fonction():  
    res = x + 5 # pas top !  
    return "AH" * res
```

```
# ==== Prog principal  
x = 7  
print(fonction())
```

```
def fonction(x):  
    res = x + 5 # mieux !  
    return "AH" * res
```

```
# ==== Prog principal  
x = 7  
print(fonction(x))
```

 **Astuce** : si on a besoin d'une variable du programme principal dans une fonction, c'est probablement qu'il fallait en faire un *paramètre* !

Notion de méthode et parenthèse objet

i Remarque :

- en Python, on manipule des *objets* : des entités qui ont des *attributs* et des *fonctions propres*, dites *méthodes*
- *classe* : modèle sur lequel on définit des objets
- un objet est une *instance* d'une classe

Méthodes vs. fonctions :


- une *méthode est une fonction*, une *fonction n'est pas forcément une méthode*
- une fonction qui n'est pas une méthode est *indépendante* de tout objet
- une méthode *dépend* de l'objet ou de la classe auxquels elle est rattachée


```
mot = "pouet"           # mot est une instance de la classe str
print(mot)              # fonction print, independante de mot
mot.replace("p", "ch")  # methode replace, dependante de mot
```

Appels de méthode

 **Syntaxe** : pour appeler une méthode, on utilise le point « . » :

```
# on appelle methode sur objet avec args  
objet.methode(args)
```

 **Remarque** : il existe des méthodes spéciales qui peuvent être appelées comme des fonctions, sans préciser l'objet avant.

 **Astuce** : pour savoir quelles méthodes existent pour un type donné, `help(type)`

```
>>> phi = (1 + 5**0.5)/2 # quel est ce nombre ?  
>>> phi.is_integer()      # methode is_integer de float sur phi  
False
```

<https://www.youtube.com/watch?v=ku6bCLx9roY>

Fonctionnement de `return` : exemple

```
def f_une():  
    return 4  
  
def f_rien():  
    pass # mot-cle qui dit "on passe a la suite sans rien faire"  
  
def f_mult():  
    return 1, 2, 3  
  
x = f_une()  
y = f_rien()  
z = f_mult()  
print(x, y, z)  
print(type(x), type(y), type(z))  
  
# ==== Resultat  
4 None (1, 2, 3)  
<class 'int'> <class 'NoneType'> <class 'tuple'>
```

Fonctionnement de `return`


# de valeurs de sorties dans la fonction	valeur renvoyée en Python (implicitement)
aucune	<code>None</code>
une seule	la valeur
plusieurs	les valeurs groupées dans un <i>tuple</i> (une <i>collection</i> de valeurs)

i Remarque : Python ne renvoie implicitement qu'*une seule valeur*, mais permet à plus haut niveau le renvoi *de 0 à n valeurs*

Résumé

Rappel :

- on peut avoir des *arguments optionnels* en fixant leur *valeur par défaut*
- les *méthodes* sont des fonctions dépendantes de leurs objets
- une fonction a son propre *espace de variables*

 **Attention :** la *portée des variables*, éviter les *variables globales*

```
def somme_parlante(a, b, verbose=True):  
    if verbose:  
        print(f"Je suis une fonction bavarde  
              qui fait la somme de {a} et {b}")  
        print(f"(ca fait {a + b})")  
  
    return a + b
```

Exercice

⚙️ **Exercice** : écrire une fonction qui prend en entrée deux chaînes de caractères `ch1`, `ch2` et qui vérifie que `ch1` et `ch2` font la même taille.

❗ **Attention** : quelques contraintes

- pas de `for`, pas de `while`, pas de `len`
- pas d'opérations *arithmétiques* type `+`, `-`, ...
- les opérations booléennes sont autorisées

🐾 **Astuce** : `chaine[1:]` renvoie `chaine` moins son premier caractère

Fonctions récursives

Définir et utiliser une fonction

Aspects avancés


Fonctions récursives

- Principe et syntaxe

- Pourquoi faire ?

- Résumé

 **Remarque :** partie *algorithmique* plus que Python à proprement parler.

 **Définition :** une fonction qui s'appelle elle-même est *récursive*.

Une fonction récursive contient :

- des *cas terminaux* où la récursion s'arrête
- des *appels récursifs* qui réappellent la fonction

 **Remarque :**

- une fonction pas récursive est *itérative* : c'est ce qu'on a fait jusqu'ici
- des fonctions qui s'appellent de manière cyclique sont aussi *récurives*

Exemple

```
def somme(n):  
    if n <= 1:  
        return 1 # cas terminal  
    else:  
        return n + somme(n - 1) # recursion  
  
def produit(n, m):  
    if m <= 1:  
        return n # cas terminal  
    else:  
        return n + produit(n, m - 1) # recursion  
  
# ==== Prog principal  
print(somme(10), produit(4, 3))  
  
# ==== Resultat  
101, 12
```

Exercice

⚙️ **Exercice** : écrire deux fonctions qui calculent le n -ème terme de la suite de Fibonacci (l'une récursive, l'autre itérative), sachant que $u_1 = u_2 = 1$ et $u_n = u_{n-1} + u_{n-2}$

⚙️ **Exercice** : écrire une fonction qui prend en entrée une suite de lettres distinctes et qui énumère tous les sous-ensembles de lettres possibles.

```
# ==== Exemple
```

```
rentrez une sequence : abcde
```

```
  e d de c ce cd cde b be bd bde bc bce bcd bcde a ae ad ade ac  
ace acd acde ab abe abd abde abc abce abcd abcde
```

⚙️ **Exercice** : (*fonction d'Ackermann*). Proposer un programme pour coder la fonction d'Ackermann $A(m, n)$, $n, m \in \mathbb{N}$, définie par

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{sinon} \end{cases}$$

Pourquoi, pourquoi pas

Pourquoi utiliser/connaître le récursif :

- parfois plus *élégant et concis* que de l'itératif
- parfois pas de solution itératives (sans *pile*)
- *très utilisé* en informatique, notamment théorique :
 - parcours d'arbres, de graphes, d'ordres, ...
 - algorithmes d'énumérations,
 - vérifications syntaxiques d'expressions, logique, ...

Mais *pourquoi ne pas l'utiliser* automatiquement :

- algorithmiquement *pas toujours le plus efficace* ...
- le Python est pensé en itératif, le récursif est souvent *plus lent*
- un algo récursif peut être « *aplati* » *en itératif* avec des *piles* (permet à la machine de simuler le récursif)

Résumé



Rappel :

- fonction *récursive* : fonction qui s'*appelle elle-même*
- fonction *pas récursive* : fonction *itérative*
- *très utile* en informatique théorique
- mais *pas très performant* en Python (ni toujours algorithmiquement)



Attention : éviter les récursions *infinies* !

```
def fact(n):  
    if n <= 1:  
        return 1 # cas terminal  
    else:  
        return n * fact(n - 1) # recursion
```