Informatique et Calcul Scientifique

Cours 3 : Fonctions et modularité

12.03.2025

La fois passée, on a vu..

Trois instructions de flux de contrôle qui permettent de contrôler le déroulement d'un programme.

- L'instruction `if` qui crée une structure conditionnelle :

```
if condition_booleenne :
    # bloc d'instructions
elif autre_condition:
    # autre bloc d'instructions
else :
    # autre bloc d'instructions
# fin de la structure de controle
```

- L'instruction `while` qui permet de répéter un bloc d'instructions en fonction de la validité d'une condition :

```
while condition_booleenne :
    # bloc d'instructions
    # maj de la variable d'iteration
# fin de la boucle while
```

- L'instruction `for` , pour parcourir une structure itérable :

```
for i in variable_iterable:
    # bloc d'instructions utilisant (ou non)
    # la variable i
# fin de la boucle for
```

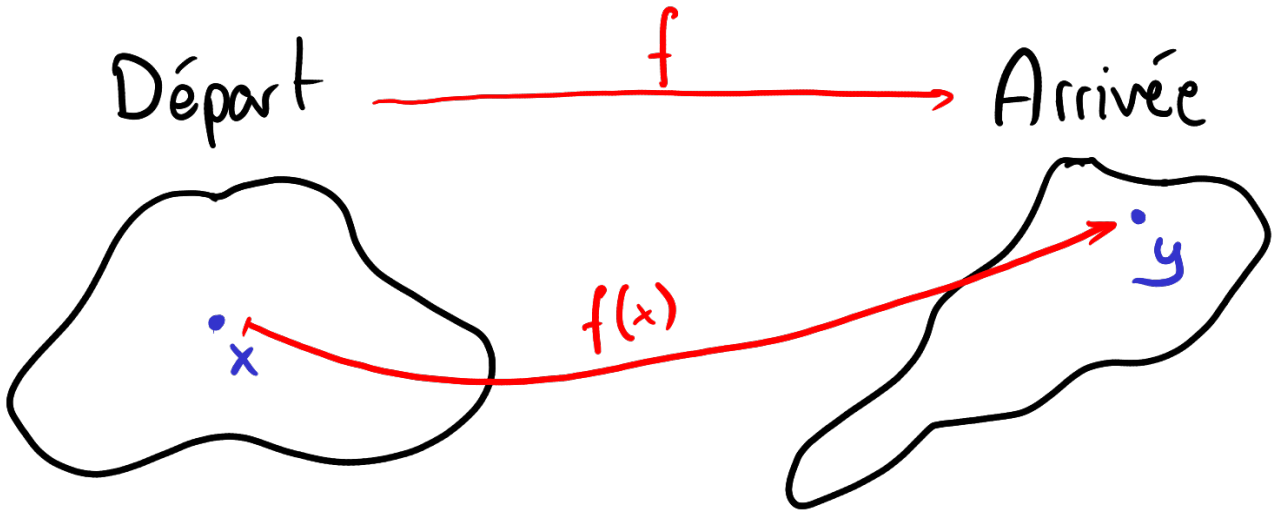
But de la leçon

Aujourd'hui on...

- présentera la notion de fonction informatique, et son utilisation pour simplifier notre code,
- étudiera les différentes sortes de paramètres que peut prendre une fonction,
- comprendra la différence entre une variable globale et locale,
- se familiarisera avec les différents modules en Python.

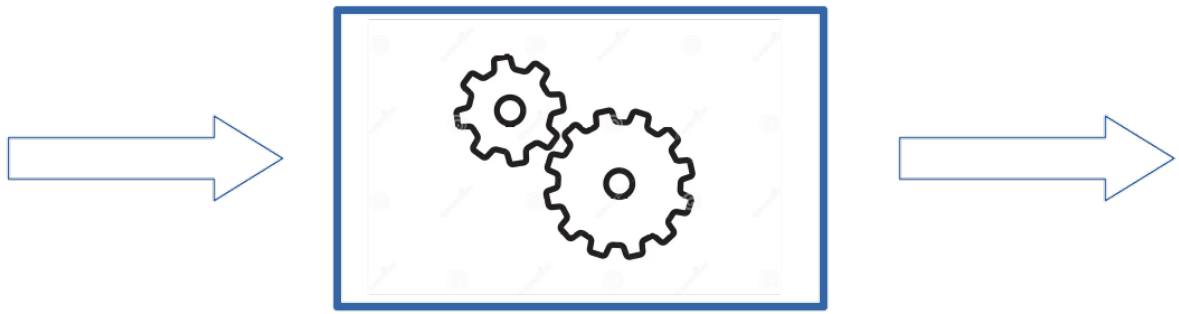
Fonction mathématique :

Une fonction f est une transformation qui associe à chaque élément x de l'ensemble de départ un élément y de l'ensemble d'arrivée tel que $y = f(x)$



Les fonctions en informatique

En informatique, une **fonction** est un ensemble d'instructions qui accomplit une tâche spécifique.



Elle prend en entrée des données (ou non) et retourne (ou non) le résultat du traitement de ces données.

Les fonctions en informatique

On utilise une fonction pour :

- Effectuer plusieurs fois la même tâche au sein d'un programme mais avec des valeurs différentes, et simplifier sa mise à jour.
- Rendre le code plus lisible et cacher certaines instructions superflues à l'utilisateur et d'autres programmeurs.
- Partager du code entre plusieurs développeurs.

Dans ce cours, nous avons déjà vu quelques fonctions internes à Python `print()`, `input()`, `len()` et `range()`, mais il en existe bien d'autres.

De plus, il est possible de définir ses propres fonctions !

Les fonctions en informatique

Voici un exemple de fonction personnalisée. On peut distinguer la *définition* de la fonction de son *appel*.

```
# ================================= Debut de la definition de la fonction
def Fahrenheit_to_Celsius(T):
    """ Cette fonction convertit une temperature de degre Fahrenheit
    a degre Celsius et affiche un message si celle-ci est superieure
    a 40C ou inferieure a -20C"""
    T_C = (T-32)*5/9
    if T_C < -20:
        print(f"{T_C} C: Attention trop froid!")
        return T_C, 1
    elif T_C > 40:
        print(f"{T_C} C: Attention surchauffe!")
        return T_C, 2
    else:
        return T_C, 0
# ================================= Fin de la definition et code principal
liste_T1 = range(0,150,1)
for i in liste_T1:
    T_C, flag = Fahrenheit_to_Celsius(i)      # <=== Appel #1 de la fonction
    if flag != 0:
        break

liste_T2 = range(0,-100,-1)
for j in liste_T2:
    T_C, flag = Fahrenheit_to_Celsius(j)      # <=== Appel #2 de la fonction
    if flag != 0:
        break
```

Définition d'une fonction

Avant de pouvoir *appeler*, ou utiliser, une fonction, il faut d'abord la *définir*. La **définition** d'une fonction se fait toujours avant son appel, dans les premières lignes du script.

- On **déclare** une fonction par le mot clé `def` suivi par le nom de la fonction puis des paramètres entre parenthèses si nécessaire, et finalement de deux points.
- Le **corps de la fonction** est ensuite défini après les deux points. Comme pour les structures de contrôle (`if` , `while`), ceux-ci indiquent que la suite est un bloc d'instruction. Attention à l'indentation !
- Les **valeurs renvoyées** par une fonction sont déclarées dans le corps de la fonction par le mot clé `return` .

Ajoutons qu'une fonction n'admet pas forcément de valeurs d'entrée ni de sortie !

Définition d'une fonction

Ces différents éléments sont illustrés sur le schéma ci-dessous :

```
def nom_de_la_fonction(param1, param2, ...):
    """Docstring : petit texte permettant
    de documenter la fonction, insere entre
    une triple paire de guillemets"""

    bloc d instructions pouvant faire
    intervenir des 'if', 'for', 'while'
    ou meme d autres fonctions

    return out1, out2, ...

fin de la fonction et retour au code
principal
```

/!\ *Indentation !!* /!\

Appel d'une fonction

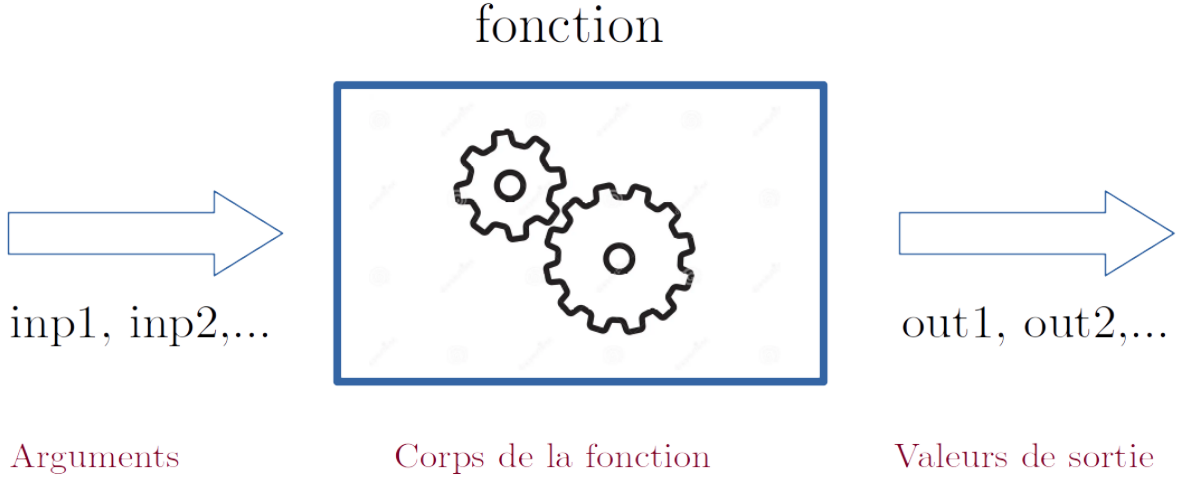
Pour exécuter le corps de la fonction, il suffit de l'*appeler* :

```
out1, out2, ... = nom_de_la_fonction(inp1, inp2, ...)
```

On peut décomposer l'appel d'une fonction en trois parties.

- Les arguments (*input*)
- Le nom de la fonction
- Les valeurs de sortie (*output*).

Les arguments sont alors passés dans le corps de la fonction qui est exécuté, puis retourne des valeurs de sortie.



Appel d'une fonction

Voici plusieurs exemples d'appels de fonctions :

- `x = input()`
- `a, p = compute_area_triangle(base, hauteur)`
- `print()`
- `print("Ceci est une fonction")`
- `print("Ceci", "est, "une", "fonction")`
- `print("Ceci", "est, "une", "fonction", sep = "-")`
- `print("Ceci", "est", "une", "fonction", sep = "-", end = ".")`

Remarquons déjà qu'une même fonction peut prendre 0, 1, 2 ou même plus d'arguments.

Paramètres

Les variables définies dans l'en-tête d'une fonction sont appelées **paramètres**.

```
def calcul(x,y):
    return 100*x + 10*y
```

Chaque paramètre défini implique la création d'une variable :

- portant le même nom,
- existant dans le corps de la fonction uniquement,
- à laquelle on peut affecter un objet de différentes manières.

Arguments

Les valeurs transmises à la fonction lors de son appel sont appelées **arguments**.

```
a,b = 5,4
s1 = calcul(a,b)
print(s1)
s2 = calcul(b,a)
print(s2)
```

```
Output :


540
450
```

La valeur de chaque paramètre est **liée** à l'argument correspondant en fonction de l'ordre dans lequel ceux-ci sont transmis à la fonction lors de son appel.

- Il faut donc *généralement* fournir autant d'arguments que de paramètres.

Paramètres obligatoires et optionnels

Il est possible d'assigner une valeur **par défaut** à un (ou plusieurs) paramètre dans la définition de la fonction. On parle alors de paramètres **optionnels**.
Ceux-ci doivent toujours être déclarés **après** les paramètres obligatoires (sans valeur par défaut).

```
def calcul(x,y,z=3):
    return 100*x + 10*y + z
```

Lors de l'appel, la valeur du paramètre n'a pas besoin d'être précisée par un argument. Si elle l'est, la valeur de l'argument fait foi, sinon il prend sa valeur par défaut.

```
s1 = calcul(1,2)
print(s1)
s2 = calcul(1,2,3)
print(s2)
s3 = calcul(1,2,5)
print(s3)
```

```
Output :
123
123
125
```

Arguments positionnels et par mots-clés

Il faut toujours fournir un argument pour chaque paramètre ne possédant pas de valeur par défaut.

De plus, on peut lier un argument à un paramètre de manière plus explicite en utilisant la syntaxe `nom_param = valeur_arg` dans son appel.

- On parle alors d'argument **par mot-clé**, ou argument nommé.

```
def calcul(x,y,z=3):
    return 100*x + 10*y + z

s1 = calcul(1,2,3)
s2 = calcul(1, y=2, z=3)
s3 = calcul(1, z=3, y=2)
s4 = calcul(z=3, x=1, y=2)
print(s1, s2, s3, s4, sep='\n')
```

```
Output :
123
123
123
123
```

Lors de l'appel de la fonction, l'argument par mot-clé est associé au paramètre correspondant à son nom.

Exemples

Dans la fonction `print()`, les paramètres `sep` et `end` prennent les valeurs par défaut `sep = ' '` et `end = '\n'` et n'ont ainsi pas besoin d'être appelées explicitement.

```
print("Param", "par", "defaut")
print("Param", "par", "defaut", sep=' ')
```

Output :

```
Param par defaut
Param par defaut
```

Afin de modifier leur valeur, il faut les nommer explicitement lors de l'appel de la fonction :

```
print("Param", "par", "defaut", sep='_')
print("Param", "par", "defaut", sep='_',
                      end='!!')
```

Output :

```
Param_par_defaut
Param_par_defaut!!
```

Remarques sur l'ordre des arguments

Si on utilise plusieurs arguments positionnels et plusieurs arguments par mot-clés, ces derniers doivent obligatoirement être appelés **après** les arguments positionnels.

```
out = calcul(2, y = 3)
print(f"Resultat : {out}")

out = calcul(x = 2, 3)
print(f"Resultat : {out}")
```

```
Output :
Resultat : 233

SyntaxError:
positional
argument follows
keyword argument
```

Si **tous** les arguments sont définis par mot-clés, on peut les passer dans n'importe quel ordre

```
out = calcul(x = 2, y = 3, z = 4)
print(f"Resultat : {out}")

out = calcul(z = 4, x = 2, y = 3)
print(f"Resultat : {out}")
```

```
Output :
Resultat : 234
Resultat : 234
```

Remarque sur le nombre d'arguments

La fonction `print()` fonctionne avec un nombre variable d'arguments :

```
print("Un")
print("Un", "exemple")
print("Un", "joli", "exemple")
print("Un", "exemple", sep='.')
print("Un", "exemple", sep='.', end='!')
```

Output :

```
Un
Un exemple
Un joli exemple
Un.exemple
Un.exemple!
```

Nous verrons au cours 5 qu'il est possible de définir une fonction prenant un nombre **indéfini** d'arguments positionnels ou nommés en les groupant dans un *tuple* ou dans un *dictionnaire*.

Valeur de sortie

En Python, une fonction peut avoir zéro, une ou plusieurs valeurs de sortie définies après le mot-clé `return`. Il est donc pratique de combiner l'appel d'une fonction à une affectation multiple :

```python
def aire_perim_rectangle(x,y):
    area = x*y
    perimeter = 2*x + 2*y
    return area, perimeter

a,p = aire_perim_rectangle(2,3)
print(f"aire : {a}, perim : {p}")
```

```
Output :

aire : 6, perim : 10
```

La valeur de sortie est alors de type `tuple`, comme nous allons voir dans deux semaines.

Valeur de sortie

Après l'exécution de la ligne commençant par `return`, le corps de la fonction est immédiatement quitté et le programme retourne à la ligne suivant l'appel de la fonction.

- L'appel de la fonction est alors remplacé par sa valeur de sortie

```
def addition(a, b):
    print("On me voit")
    return a + 2*b
    print("On me voit plus")

somme = addition(1,2)
print(somme)
```

Output :

```
On me voit
5
```

Valeur de sortie

En particulier :

- Si le mot-clé `return` n'est suivi d'aucune expression, la fonction retourne la valeur `None`.
- Le corps d'une fonction peut ne pas contenir de mot-clé `return`. Dans ce cas, la fonction est exécutée dans son intégralité et retourne la valeur `None`.
- Le corps d'une fonction peut contenir plusieurs mots-clé `return` en fonction. C'est notamment le cas si on utilise des structures de contrôle.
  Dans ce cas, le corps de la fonction est quitté lorsque le programme atteint le premier `return`, et retourne la ou les valeurs correspondantes.

Exemple

```python
def operations(x,method):
    if method == 1:
        for i in range(x):
            print(i+1)
    elif method == 2:
        s = 0
        for i in range(x+1):
            s += i
        return s
    elif method == 3:
        p = 1
        i = 1
        while i < x+1:
            p *= i
            i += 1
        return p, not p%27
    else:
        print("Mauvais choix de methode")

print(operations(5,1), end='\n ***\n')
print(operations(5,2), end='\n ***\n')
print(operations(5,3), end='\n ***\n')
```

```
Output :

1
2
3
4
5
None

***

15

***

(120, False)
***
```

Documentation

Au début du corps de la fonction, il est courant d'introduire un commentaire spécial, appelé `Docstring`.

- Celui-ci est délimité par des triples quotes : `"""doc"""`

```
def docstring_example():
    """Ceci est un exemple de Docstring
    decrivant la fonction docstring_example"""
    print("Lire la documentation")

docstring_example()
print(docstring_example.__doc__)
```

- Il sert de documentation à la fonction et doit donc être rendu aussi clair et descriptif que possible
- Généralement, on y décrit le rôle de la fonction ainsi que les différentes variables y apparaissant.
- Il est possible de consulter la Docstring en tapant `nom_de_la_fonction.__doc__`

Variable locale et globale

En programmation, il faut distinguer la notion de variable **globale** et **locale**.

- Une variable globale est définie dans le module principal et sera visible **partout** dans le programme
- Une variable locale est créée dans le corps d'une fonction. Elle n'existe **que** dans celui-ci.

Dans l'exemple suivant [1], les variables `area` et `perimeter` sont locales, alors que les variables `a` et `p` sont globales.

```
def aire_perim_rectangle(x,y):
    area = x*y
    perimeter = 2*x + 2*y
    return area, perimeter

a,p = aire_perim_rectangle(2,3)
print(f"aire : {a}, perim : {p}")
print(area)
```

Output :

```
aire : 6, perim : 10
NameError: name 'area'
is not defined
```

1. Pour une version interactive, voir PythonTutor

Variable locale et globale

Si une variable est définie **localement** dans une fonction, Python ne va jamais chercher de version globale.

- Une variable locale remplace une variable globale du même nom dans le corps d'une fonction.
- Si on crée une variable (globale) avant l'appel d'une fonction, alors celle-ci sera accessible dans la fonction en lecture seule.

```
x = 3
def moyenne(a,b,c):
    return (a+b+c+x)/4
def moyenne2(a,b,c):
    x = 5
    return (a+b+c+x)/4

z = moyenne(4,4,5)
print(f"Moyenne de {z}")

z = moyenne2(4,4,5)
print(f"Moyenne de {z}")
```

```
Output :

Moyenne de 4.0

Moyenne de 4.5
```

Modularité

Toute fonction doit être définie à un endroit de l'ordinateur accessible par Python. Elles peuvent être :

- définies par l'utilisateur dans le fichier `.py` exécuté
- *built-in*, c'est-à-dire définies dans la distribution Python **et** accessibles en tout temps (`print()` , `len()` , `input()`)
- définies par l'utilisateur dans un autre fichier accessible (par exemple `all_functions.py`). Pour les utiliser, il doit tout d'abord les importer dans le programme
- définies dans la bibliothèque standard Python, mais à importer (`random.randint()` , `statistics.mean()` , `math.sqrt()`)
- créées par d'autres utilisateurs et définies dans des modules accessibles en ligne, et à installer avant de pouvoir les importer

Import de modules

Il existe plus de 200 modules dans la librairie standard de Python, et près de 200'000 au total ! Il est donc probable que ce que vous cherchiez à faire existe déjà.

Il existe différentes manières d'importer une fonction d'un module :

Méthode d'import	**Appel**
`from module_name import fonction`	`fonction()`
`from module_name import *`	`fonction()`
`import module_name`	`module_name.fonction()`
`import module_name as alias`	`alias.fonction()`

La deuxième méthode est la moins couramment utilisée car elle risque d'effacer (ou d'être effacée par) une fonction locale possédant le même nom.

On efface un module de la mémoire en tapant `del module_name`.

Modules courants

Voici quelques modules que vous serez susceptibles d'utiliser dans la suite de vos études[2].

`math`	Les principales fonctions mathématiques et nombres remarquables tels que `sin`, `sqrt`, `pi`
`sys`	Pour interagir avec l'interpréteur Python
`os`	Pour exploiter les fonctionnalités dépendantes du système d'exploitation
`time`	Pour accéder aux fonction liées au temps ainsi qu'à l'heure de l'ordinateur
`random`	Pour générer des nombres aléatoires
`Tkinter`	Pour créer un interface graphique

2. La liste complète des modules standards de Python se trouve ici : https://docs.python.org/3/py-modindex.html

Exemples

Un module que vous utiliserez souvent est le module `math`. La plupart des fonctions et opérations mathématiques y sont définies.

```python
from math import sqrt
x = 9
print(f"La racine carree de {x} vaut {sqrt(x)}")

del randint
from random import *
print(f"Voici un nombre aleatoire : {randint(0,10)}")

del cos, pi, sin
import math
print(f"cos(pi) = {math.cos(math.pi)}")

del math
import math as m
print(f"cos(pi) = {m.cos(m.pi)}")

import user_def as ud
ud.fin_du_cours()
```

Output :

```
La racine carree
de 9 vaut 3.0


Voici un nombre
aleatoire : 7


cos(pi) = -1.0


cos(pi) = -1.0


Merci d'avoir
suivi ce cours !!
```

Take Home Message

Pour écrire des programmes plus intéressants, on utilise des structures de contrôle :

- L'instruction `if: ... else:` permet d'exécuter une portion de code différente en fonction du résultat d'un test
- L'instruction `while` permet de répéter une instruction un nombre indéfini de fois
  $\Longrightarrow$ Attention aux boucles infinies !
- L'instruction `for` permet de parcourir chaque élément d'un objet itérable