

# T.P. III - Variables & Fonctions

Code Capytale : e602-764562

## I - Ce qu'il faut savoir

- \* Définir, utiliser, modifier, afficher le contenu d'une variable.
- \* Utiliser les opérations usuelles ainsi que quelques fonctions du module `numpy` pour effectuer des calculs simples.
- \* Définir une fonction avec les mots clés `def` / `return`.
- \* Calculer la valeur d'une fonction en un réel.
- \* Tracer le graphe d'une fonction avec les fonctions `plot` / `show` du module `matplotlib.pyplot`.
- \* Utilisation de `range`, `np.arange`, `np.linspace` pour créer la liste des abscisses.
- \* Création de la liste des ordonnées, listes par compréhension.

## II - Modification du contenu des variables

### II.1 - Les variables

L'instruction `x = 2` **stocke** l'entier 2 dans la variable **nommée** `x`. Dans la suite, les calculs seront effectués en remplaçant `x` par la valeur 2. La fonction `print` permet d'afficher le contenu de la variable.

```
x = 2
y = 3 * x + 1
print("x", x)
print("y", y)
```

qui affiche

```
x 2
y 7
```

Remarquons que la quantité à **droite** du signe `=` est **calculée** puis stockée dans la variable qui porte le nom indiqué à **gauche** du signe `=`. Ainsi, la

modification d'une variable **après** un calcul n'affecte **que** la variable qui est modifiée.

```
x = 2
y = 3 * x + 1
x = 4
print("x", x)
print("y", y)
```

qui affiche

```
x 4
y 7
```

**Exercice 1. (Variables)** Pour chacune des questions suivantes, indiquer les variables créées ainsi que leur contenu à l'issue de la suite d'instructions suivantes.

1.

```
a = 12
b = 10
c = a
a = b
b = c
print("a", a)
print("b", b)
print("c", c)
```

2.

```
a = 12
b = 10
a = b
b = a
print("a", a)
print("b", b)
```

**3.**

```

a = 2
b = 4
c = a
b = a
c = b - a
print("a", a)
print("b", b)
print("c", c)

```

**4.**

```

a = 4
b = 2
c = b
b = a
print(..., ...)
print(..., ...)
print(..., ...)

```

**5.**

```

a = 4
b = 2
c = b
b = a
a = c
print(..., ...)
print(..., ...)
print(..., ...)

```

**6.**

```

x = 5
y = 2 * x + 3
x = 100
print(..., ...)
print(..., ...)

```

**7.**

```

a = 3
b = 4
x = a
a = b
b = x
print(..., ...)
print(..., ...)
print(..., ...)

```

Pour échanger le contenu de deux variables, il est conseillé d'utiliser une variable auxiliaire

```

x = 3
y = 2
print("x, y avant :", x, y)
auxiliaire = x
x = y
y = auxiliaire
print("x, y après :", x, y)

```

affiche

```

x, y avant : 3 2
x, y après : 2 3

```

## II.2 - Les opérations

Les opérations usuelles sont les suivantes :

- \* l'addition + et la soustraction -,
- \* la multiplication \* et la division /,
- \* la fonction puissance \*\*.

Les lignes de codes suivantes

```

x = 3
y = 12 * x - 125 + x**2
print("x", x)
print("y", y)
x = 15
print("x", x)
print("y", y)

```

affichent

```
x 3
y -80
x 15
y -80
```

Les fonctions usuelles ne sont pas disponibles directement en Python, elles sont disponibles dans le module `numpy`.

```
import numpy

print(numpy.log(3))
print(numpy.exp(1.14))
```

qui affiche

```
1.0986122886681096
3.1267683651861553
```

Vous remarquerez la syntaxe `numpy.log` qui *dit* à Python qu'il faut aller chercher la fonction `log` du module `numpy`. Cette syntaxe est lourde et il est préférable de donner un *surnom* au module `numpy` de manière à écrire moins de texte. On le surnommera généralement `np` (à noter que la fonction logarithme népérien est appelée `log`). On écrira alors :

```
import numpy as np

print(np.log(3))
print(np.exp(1.14))
```

qui affiche

```
1.0986122886681096
3.1267683651861553
```

## III - Fonctions

### III.1 - Définition

Le mot-clef `def` permet de définir une fonction dont on précise les paramètres entre parenthèses. L'indentation (c'est-à-dire le décalage par rapport au début de la ligne) permet de délimiter le début et la fin de la définition. L'instruction `return` permet de préciser la valeur renvoyée lors de l'appel de la fonction. Par exemple, la fonction suivante correspond à la fonction mathématique  $f : x \mapsto 2x + 3$

```
def f(x):
    return 2 * x + 3
```

On peut ensuite évaluer cette fonction en choisissant différents paramètres :

```
print("f(4)", f(4))

x = 12
print("f(12)", f(x))

y = 20
print("f(20)", f(y))
```

qui affiche

```
f(4) 11
f(12) 27
f(20) 43
```

**Exercice 2. (Fonctions)** Déterminer le contenu de la variable `x` à l'issue de la suite d'instructions suivante.

1.

```
x = 5
def fct1(y) :
    return 3 * y + 25

x = fct1(x)
```

2.

```
x = 5
def fct1(x) :
    return 3 * x + 25

x = fct1(x)
```

## III.2 - Fonctions de plusieurs variables

Il est également possible de définir des fonctions qui prennent plusieurs paramètres en entrée :

```
def fonct(x, y):
    z = x**2 + 3 * np.exp(y)
    return z

print("Evaluation de f en (1,5) :", fonct(1, 5))
```

affiche

```
Evaluation de f en (1,5) : 446.23947730772977
```

## III.3 - Tracé de courbes

Les rendus graphiques en Python sont possibles à l'aide du module `matplotlib.pyplot`, importé ici avec le *surnom* `plt`. La fonction

- \* `plt.figure()` permet de créer un nouveau graphique,
- \* `plt.plot(abscisses, ordonnees)` permet de tracer les points dont la liste des abscisses est `abscisses` et la liste des ordonnées est `ordonnees`,
- \* `plt.show()` permet d'afficher le graphique.

Pour tracer le graphe de la fonction exponentielle sur l'intervalle  $[-5, 5]$ , on utilisera ainsi :

```
import matplotlib.pyplot as plt
X = np.arange(-5, 5.1, 0.1)
Y = [np.exp(x) for x in X]

plt.figure()
plt.plot(X, Y)
plt.show()
```

Pour tracer un graphe, il faut disposer de la liste des abscisses des points à tracer. Il y a trois options :

- \* `range(a, b)` liste les entiers compris entre `a` et `b-1`. Cette fonction est limitée aux entiers.
- \* `np.arange(a, b, pas)` liste les réels `a`, `a+pas`, `a+2*pas`,... et s'arrête juste avant `b`.

- \* `np.linspace(a, b, num)` liste `num` réels répartis uniformément entre `a` et `b`.

Pour tracer un graphe, il faut également la liste des ordonnées, c'est-à-dire de la liste des images des abscisses par une fonction. Pour construire l'image des éléments de la liste `X` par la fonction `f`, il existe plusieurs solutions :

- \* si `f` est une fonction `numpy simple` (définie sans utiliser de conditionnelle), on peut écrire `Y = f(X)`.

```
def f(x):
    return x**2 * np.exp(x)

X = np.arange(1, 3, 0.5)
Y = f(X)
print(Y)
```

qui affiche

```
[ 2.71828183 10.08380041 29.5562244 76.14058725]
```

- \* sinon, on peut utiliser la notion de *liste par compréhension* : `Y = [f(x) for x in X]`. Ceci se lit *Y est la liste des éléments f(x) lorsque x parcourt X*.

```
def f(x):
    if x < 2:
        return x
    else:
        return x + 1

X = np.arange(1, 3, 0.5)
Y = [f(x) for x in X]
print(Y)
```

qui affiche

```
[1.0, 1.5, 3.0, 3.5]
```

**Exercice 3.** Pour tout  $n$  entier naturel, on définit

$$u_n = \frac{4^n}{3n^3 - 2n + e^n}.$$

1. Compléter le code suivant pour que la fonction renvoie la valeur de  $u_n$ .

```
def u(n):  
    return ...
```

2. Compléter le code suivant afin d'afficher les valeurs de  $u_{10}$  puis de  $u_{35}$ .

```
print("u_10", u(...))  
print("u_35", u(...))
```

3. Modifier le code suivant pour qu'il affiche les points de coordonnées  $((n, u_n))_{0 \leq n \leq 25}$ .

```
import matplotlib.pyplot as plt  
  
X = ...  
Y = [u(...) for n ...]  
  
plt.figure()  
plt.plot(..., ..., 'd', color="orange")  
...
```