La Mérci MONTPELLIER

Les fonctions, modules et packages avec Python

Nous allons créer des outils appelés fonctions qui seront réutilisables dans différents programmes.
On peut regrouper des **fonctions** selon leur fonctionnalité dans un fichier :
→on appelle **module** un ensemble de fonctions.
On peut regrouper des modules, on parle alors de **package**.

Voici une **liste de modules de base** qui vous seront utiles tôt ou tard :

random :	fonctions permettant de travailler avec des valeurs aléatoires
math :	toutes les fonctions utiles pour les opérations mathématiques (cosinus, sinus, exp, etc.)
sys	fonctions systèmes
os :	fonctions permettant d'interagir avec le système d'exploitation
time :	fonctions permettant de travailler avec le temps
calendar :	fonctions de calendrier
profile :	fonctions permettant d'analyser l'exécution des fonctions
urllib2 :	fonctions permettant de récupérer des informations sur internet
re :	fonctions permettant de travailler sur des expressions régulières

Les extensions des fichiers python

Il existe plusieurs extensions de fichier qui tournent autour de python:
.py → script modifiable
.pyc → script compilé
.pyw → script exécuté sans lancement de terminal (sous windows)

Les fonctions

Une **fonction** (ou *function*) est une suite d'instructions que l'on peut appeler avec un nom, qui peut réaliser un processus ou un calcul, et qui peut renvoyer ou afficher un résultat.

Exemples de fonctions :

Ex 1 :

```
def ajouter(a):
    print(a+2)

ajouter(1)
```

→3
→ cette fonction ajoute 2 à la valeur saisie en paramètre

```
def ajouter(a):
    return (a +2)

ajouter(1)
```

→rien
→ cette fonction n'affiche aucun résultat

```
def ajouter(a):
    return (a +2)

y=ajouter(1)
print(y)
```

→3

Il est d'ailleurs possible d'utiliser plusieurs paramètres:

Ex 2 :

```
def ajouter(a, b):
    return 30 + a + b

y=ajouter(1, 2)
print(y)
```

→33

On peut récupérer les valeurs renseignées via une liste :

Ex 3 :

```
def ajouter(*param):
    print(param[0] + param[1] + param[2])

ajouter(1, 2, 3)
```

→6

```
ajouter(10, 20, 30)
```

→60

```
def ajouter2(*param):
  a=0
  for i in range(len(param)):
    a+=param[i]
  print(a)

ajouter2(1,10,100,1000,10000)
```

→11 111

Quelques remarques :

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

Ainsi :

$f(2)$ → on obtient : 8

$f(f(2))$ → **erreur type** car il ne peut multiplier un nombre avec une expression « print »

→ en effet, $f(2)$ correspond à un print = nonetype (print n'est pas un nombre)

On utilise beaucoup **return** pour renvoyer des données afin de pouvoir les manipuler dans des calculs.

→ ***ici, on privilégiera donc un return pour renvoyer une valeur :***

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

Ainsi :

print $(f(f(2)))$ → on obtient : 8

print $(f('to'))$ → on obtient : toto

Ex 5 :

```
def fonction( x ) :
    print( x )
    return(2* x )          → après un return, le module s'arrête
    print(3* x )
    return(4* x )
```

fonction(10) → 10

```
y = fonction(10)
print(y)
```
→ 10 , 20

Culture informatique :

Une fonction n'est pas obligée de renvoyer une valeur, on parlera alors dans ce cas plutôt de **procédure**.

Portée des variables (variable globale et variable locale)

Une variable **déclarée à la racine d'un module** est visible dans tout ce module.
On parle alors de **variable globale**.

```
def test():                # possibilité d'utiliser la commande global
    print x

x = "hello"
test()                     → on obtient : hello
```

Et une variable **déclarée dans une fonction** ne sera visible que dans cette fonction.
On parle alors de **variable locale**.

```
def test():
    x = "hello"
    print( x )

x = "False"
test()          # on appelle la fonction
print( x )      # on demande la valeur de x
```
→ hello
  false

x = false car la fonction n'a pas utilisé x en paramètre en entrée,
→ le paramètre à l'intérieur de la fonction aurait pu tout aussi bien s'appeler y.