Jusqu'à présent, la plupart des programmes que nous avons réalisés n'utilisaient qu'une suite d'instructions sans architecture particulière. Mais alors que vos programmes deviennent progressivement plus complexes, vous pouvez simplifier votre tâche et améliorer la lisibilité en les subdivisant en sous-ensembles dénommés **« fonctions »**. Cette façon de développer s'appelle réaliser une programmation structurée.

Une fonction désigne une entité de données et d'instructions qui fournie une solution à une (petite) partie bien définie d'un problème plus complexe. Elle peut faire appel à d'autres fonctions, leur transmettre des données ou bien en recevoir en retour. L'ensemble des fonctions ainsi reliées doit alors être capable de résoudre le problème global.

Il arrive que nos programmes utilisent très fréquemment un même groupe d'instructions. Il devient très avantageux alors d'en faire une ou plusieurs fonctions particulières, que nous pouvons même regrouper dans une bibliothèque (en dehors du fichier du programme principal).

Ainsi, avec cette démarche, nous réalisons une étude une fois pour toute, et ensuite chaque programme profite de cet effort avec une utilisation beaucoup plus simplifiée. Cette démarche est fondamentale. Le programme principal devient alors une boîte à outils, chaque outil étant représenté par une fonction.

Un programme est très souvent développé en équipe. L'identification des fonctions du programme permet de répartir le travail au sein de cette équipe. Chaque programmeur aura ensuite la charge du développement d'une ou de plusieurs fonctions. La tâche du programmeur sera alors de les coder et de les tester, afin de garantir leur fonctionnement.

La programmation structurée permet de réduire la complexité, chacun réalisant un travail plus modeste et plus facile à implémenter. Il sera également plus facile de faire des mises à jour et de faire de la maintenance à posteriori.

Définition d'une fonction personnalisée

Une fonction correspond à un ensemble d'instructions spécifiques, mais s'utilise comme s'il n'y en avait qu'une seule. L'utilisateur y fait appel autant de fois qu'il en a besoin. Cela évite d'avoir des duplications de code à plusieurs endroits du script. Elle est donc définie une fois pour toute et peut s'utiliser autant de fois que nécessaire.

*Une fonction est représentée par un nom spécifique et unique dans le cas de Python. Dans ce cadre là, il n'est pas possible d'avoir plusieurs fonctions qui font des traitements totalement différents et qui portent le même nom. La **« surcharge »**, contrairement à beaucoup d'autres langages, n'est donc pas tolérée dans le langage Python.*

*Les opérandes d'une fonction, appelée **« paramètres »**, sont spécifiés dans une liste entourée de parenthèses **« () »**, les paramètres étant séparés par des virgules **« , »**. Il est possible de ne pas avoir de paramètre, les parenthèses restent tout de même obligatoires. C'est d'ailleurs ce qui permet de reconnaître une fonction.*

*Une fonction par essence, à l'image des fonctions mathématiques, renvoie généralement une valeur de retour (ou plusieurs grâce aux tuples dans Python), mais ce n'est pas obligatoire. Dans ce cas particulier, cette fonction s'appelle une procédure. Les actions qu'exécutent une fonction sont spécifiées dans le corps de la fonction. Toutefois, Python, comme d'autres langages, ne fait pas la différence entre procédure ou fonction classique. Pour lui, il s'agit d'une fonction. Nous avons d'ailleurs déjà utilisé ces deux cas de figure avec la fonction **« input() »** et la procédure **« print() »**.*

factorielles.py

```
# Fonction factorielle
def factorielle(x) :
    i, calcul = 2, 1
    while i <= x : calcul, i = calcul*i, i+1
    return calcul

# Fonction d'affichage
def afficher(motif, valeur, resultat) : print(motif.format(valeur, resultat))

# Fonction de saisie
def saisie():
    print("Calcul de la factorielle d'un nombre : n!")
    return int(input("Introduisez la valeur de n : "))

# Utilisation des fonctions
n = saisie()
r = factorielle(n)
afficher('{}! = {}', n, r)
```

Shell Python : [évaluer factorielles.py]

```
Calcul de la factorielle d'un nombre : n!
Introduisez la valeur de n : 5
5! = 120
```

*Pour définir une nouvelle fonction personnalisée, vous devez commencer par le mot réservé **« def »** (define, en anglais), qui constitue le prélude à toute construction de fonction. Généralement vous définissez vos fonctions en début de script pour qu'elle soit déjà connue avant de l'utiliser.*

À la suite de ce mot réservé, vous spécifiez le nom de la fonction à votre convenance, mais en respectant les mêmes critères que pour la déclaration des variables. Ce nom, je le rappelle doit être unique.

*Vous précisez ensuite la liste des paramètres, s'il y en a, séparés par des virgules **« , »**, et la liste, même vide, doit être encadrée par des parenthèses **« () »**. Les paramètres peuvent être de n'importe quel type. Il n'y a aucune restriction.*

*Vous spécifiez ensuite votre bloc d'instructions, et comme tout bloc d'instruction, **« while »**, **« if »**, etc., vous devez d'abord le ponctuer par le double point **« : ».** Ainsi, à la suite de cette signature de fonction, vous précisez votre ensemble d'instructions en utilisant l'indentation, comme nous avons l'habitude de le faire lorsque nous définissons un bloc. Éventuellement, pour faciliter la concision, si la fonction ne possède qu'une seule instruction, vous pouvez la mettre sur la même ligne que la signature.*

*Si la fonction réalise un traitement dont le résultat doit être transmis à l'extérieur, vous devez alors utiliser le mot réservé **« return »** suivi de la valeur à retourner, sous formes de constante littérale, de variable, d'expression complexe, etc. Lorsque le mot **« return »** est atteint, la fonction s'arrête automatiquement, même si d'autres instructions suivent dans la fonction.*

Interprétation du script précédent

Pour bien comprendre le fonctionnement du script précédent, nous nous intéressons plus particulièrement à la fonction **« factorielle() »**. Elle prend un paramètre **« x »** dont le type attendu est un entier. À l'intérieur de cette fonction, deux variables supplémentaires **« i »** et **« calcul »** (**« x »** étant la première) sont déclarées pour permettre de réaliser l'enchaînement des calculs dans l'itérative.

*Ces variables sont **« locales »** à cette fonction et ne sont donc pas accessibles de l'extérieur. Du coup, plusieurs fonctions peuvent utiliser ces mêmes noms de variables sans qu'il y ait de conflit particulier. Lorsque la fonction se termine, ces variables **« locales »** n'existent plus, elles sont automatiquement détruites.*

*À la fin du script, dans le programme principal, lorsque nous appelons cette fonction **« factorielle() »**, nous devons bien évidemment lui indiquer quel est le nombre correspondant au calcul à réaliser. Cette information que nous transmettons à la fonction au moment même où nous l'appelons s'appelle un **« argument »**. Ici, **« l'argument »** est la variable **« n »**. Juste à ce moment là, le **« paramètre x »** copie la valeur de **« l'argument n »**. La fonction effectue alors ce qu'elle doit faire sans plus se préoccuper de **« l'argument »**. Du coup, si vous modifiez le **« paramètre »**, cela n'a aucune conséquence directe sur **« l'argument »**.*

Lorsque le « paramètre » est un objet, le comportement est différent, le « paramètre » fait directement référence à « l'argument » et reste donc en relation pendant toute la durée de la fonction. Du coup, une modification éventuelle du « paramètre » modifie également « l'argument » puisqu'il s'agit en réalité de la même entité. Nous aborderons cette situation lorsque nous proposerons des listes ou des dictionnaires à une fonction.

*Toujours à l'intérieur de cette fonction **« factorielle() »**, lorsque l'itérative est terminée, le résultat est alors prêt à être diffusé, nous le transférons donc à l'extérieur pour qu'il puisse être exploité par le programme principal, ceci au moyen de l'opérateur **« return »**.*

*Dans ce programme principal, c'est la variable **« r »** qui récupère la valeur renvoyée par la fonction et qui correspond bien au résultat prévu par le calcul de cette factorielle. Une fois que la variable **« r »** a reçue sa valeur, la fonction **« factorielle() »** s'arrête automatiquement, et le programme principal peut suivre son déroulement normal en se servant de cette variable **« r »** pour d'autres traitements.*

Utilité de la programmation structurée

L'intérêt ici de ce type de programmation, je le rappelle, est toujours de réduire la complexité du code, d'une part pendant la conception afin d'éviter de réfléchir à plusieurs problèmes en même temps, et d'autre part après coup pour obtenir une meilleure lisibilité afin d'avoir une maintenance plus facile à mettre en œuvre.

Dans ce programme d'exemple, nous avons dans un premier temps la définition de trois fonctions personnalisées, à la suite de quoi, nous avons notre programme principal qui se réduit simplement à trois lignes de codes et qui utilise ces fonctions qui vient d'être définies (les fonctions doivent d'abord être définies avant de pouvoir être utilisées).

Cela correspond bien à la démarche générale du programme savoir, faire une saisie, réaliser le calcul de la factorielle et enfin afficher le résultat obtenu. Tout est parfaitement clair et facile à comprendre.

Le script que nous venons de réaliser est pour l'instant assez rudimentaire et fonctionne très bien si nous sommes dans le cadre de l'ensemble de définition prévu pour l'utilisation d'une factorielle, savoir calculer avec des entiers. Mais que se passe-t-il si l'utilisateur propose une valeur réelle ou si il rentre une valeur négative ? Faisons l'expérience !

Shell Python : [évaluer factorielles.py]

```
Calcul de la factorielle d'un nombre : n!
Introduisez la valeur de n : 3.5
Retraçage (dernier appel le plus récent) :
  Fichier "/home/manu/CouldStation/Projets python/factorielles.py", ligne 16, dans <module>
    n = saisie()
  Fichier "/home/manu/CouldStation/Projets python/factorielles.py", ligne 13, dans <module>
    return int(input("Introduisez la valeur de n : "))
builtins.ValueError: invalid literal for int() with base 10: '3.5'
```

Une erreur se produit bien puisque nous n'avons pas pris en compte cette situation, et un message assez iconoclaste apparaît alors. Ici, le seul problème concerne la saisie, et pas une autre fonction, comme celle par exemple concernant le calcul de la factorielle puisque nous avons vu qu'elle réalise son traitement de façon correcte.

L'intérêt de la programmation structurée est qu'il est facile de cibler un problème et de réagir uniquement sur la partie qui pose problème sans se préoccuper des autres fonctions.

*Ici donc, nous devons retravailler sur la fonction **« saisie() »** uniquement et résoudre ces contraintes supplémentaires. Bien que cela ne soit pas nécessaire, je changerais également la fonction **« factorielle() »** afin d'avoir une écriture plus concise en utilisant le principe de récursivité.*

factorielles.py

```
# Fonction factorielle
def factorielle(x) :
    if x==0: return 1
    else: return x*factorielle(x-1)

# Fonction d'affichage
def afficher(motif, valeur, résultat) : print(motif.format(valeur, résultat))

# Fonction de saisie
def saisie():
    print("Calcul de la factorielle d'un nombre : n!")
    clavier = 'valeur non entière'
    while not clavier.isdecimal():
        clavier = input("Introduisez la valeur de n : ")
        if clavier.isdecimal() : return int(clavier)
        else : print("Ce n'est pas une valeur entière positive, recommencez !")

# Utilisation des fonctions
n = saisie()
afficher('{}! = {}', n, factorielle(n))
```

Shell Python : [évaluer factorielles.py]

```
Calcul de la factorielle d'un nombre : n!
Introduisez la valeur de n : bonjour
Ce n'est pas une valeur entière positive, recommencez !
Introduisez la valeur de n : 5.6
Ce n'est pas une valeur entière positive, recommencez !
Introduisez la valeur de n : -7
Ce n'est pas une valeur entière positive, recommencez !
Introduisez la valeur de n : 7
7! = 5040
```

*Le programme est maintenant beaucoup plus fiable. Quelque soit les erreurs de saisie, un message d'avertissement est engendré. Heureusement, la classe **« str »** possède une méthode **« isdecimal() »** qui vérifie si le texte saisie correspond bien à une valeur entière naturelle.*

*Pour changer la définition de la fonction **« factorielle() »**, je me suis tenu au critère suivant **« n ! = n (n-1) ! »** avec **« 0!=1 »**. Cette écriture est bien récursive, c'est-à-dire que la fonction s'appelle elle-même. Python le permet. Nous avons du coup une écriture beaucoup plus concise. Toutefois, dans les écritures récursives, il faut qu'il y ait systématiquement une condition de sortie **« 0!=1 »** sinon nous avons une infinité d'appels.*

Des objets en paramètres de fonction

Comme évoqué plus haut, il est possible d'avoir n'importe quel type de paramètre et même des objets. N'oubliez pas que dans ce cas là, le paramètre fait directement référence à l'argument. Nous allons illustrer nos propos au travers d'un ensemble de fonctions qui permettent de gérer une liste de notes.

notes.py

```
# Définition des fonctions
def moyenne(notes):
    if len(notes) == 0 : return 0.0
    somme = 0.0
    for note in notes : somme += note
    return somme / len(notes)

def plusHaute(notes):
    max = 0.0
    for note in notes:
        if note>max : max = note
    return max

def plusBasse(notes):
    if len(notes)==0: return 0.0
    min = 20.0
    for note in notes:
        if note<min : min = note
    return min

def ajout(notes, note): notes.append(note)
```

```
# Programme principal
élève = [8.5, 12, 13.5, 18]
ajout(élève, 15.0)
ajout(élève, 5.0)
affichage='Notes : {}, moyenne = {}, plus haute = {}, plus basse = {}'
print(affichage.format(élève, moyenne(élève), plusHaute(élève), plusBasse(élève)))
```

Shell Python : [évaluer notes.py]

```
Notes : [8.5, 12, 13.5, 18, 15.0, 5.0], moyenne = 12.0, plus haute = 18, plus basse = 5.0
```

*Ce script dispose d'un certain nombre de fonctions qui prennent toutes le paramètre **« notes »**. Ce paramètre est systématiquement en relation avec l'argument **« élève »** qui est une liste de valeurs numériques correspondant aux notes de l'élève en question.*

*La fonction **« ajout() »** modifie le paramètre **« notes »**, et du coup, instantanément, l'argument **« élève »** s'en trouve automatiquement mis à jour. Les autres fonctions travaillent alors avec cette nouvelle liste de notes.*

Variables locales – variables globales

Les variables globales sont des variables qui sont définies en dehors de toute fonction. Leurs portées et leurs durées de vie correspondent à la durée de vie du programme. Toutes les fonctions peuvent accéder directement à ces variables sans contraintes particulières. Les variables locales (les paramètres en font partie) sont des variables qui sont définies à l'intérieur d'une fonction et à ce titre leurs durées de vie correspond à la durée de vie de l'utilisation de la fonction. Lorsque cette fonction est terminée, toutes les variables locales (déclarées à l'intérieur) n'existent plus.

Il faut limiter au maximum l'utilisation des variables globales. C'est une source de conflit éventuel. Il est toujours préférable d'utiliser les paramètres quand cela est possible plutôt qu'une fonction utilise directement ces variables globales. Toutefois, si vous manipulez des types simples, comme les valeurs numériques, et que vous devez modifier leurs valeurs à l'aide d'une fonction spécifique, il peut être intéressant que cette fonction agisse directement sur ces variables globales.

Je rappelle que pour une donnée numérique, une copie est réalisée entre l'argument et le paramètre, de telle sorte que lorsque nous manipulons le paramètre, cela n'a aucune conséquence sur l'argument. L'argument et le paramètre sont bien des variables séparées.

*Par contre, dans votre code, à l'intérieur de cette fonction, précisez alors que vous êtes conscient de manipuler une variable globale en le précisant à l'aide du mot réservé **« global »**.*

notes.py

```
# Définition des fonctions
def moyenne():
    global notes
    if len(notes) == 0 : return 0.0
    somme = 0.0
    for note in notes : somme += note
    return somme / len(notes)

def plusHaute():
    global notes
    max = 0.0
    for note in notes:
        if note>max : max = note
    return max

def plusBasse():
    global notes
    if len(notes)==0: return 0.0
    min = 20.0
    for note in notes:
        if note<min : min = note
    return min

def ajout(note):
    global notes
    notes.append(note)

# Programme principal
notes = [8.5, 12, 13.5, 18]
ajout(15.0)
ajout(5.0)
affichage='Notes : {}, moyenne = {}, plus haute = {}, plus basse = {}'
print(affichage.format(notes, moyenne(), plusHaute(), plusBasse()))
```

Shell Python : [évaluer notes.py]

```
Notes : [8.5, 12, 13.5, 18, 15.0, 5.0], moyenne = 12.0, plus haute = 18, plus basse = 5.0
```

Autant vous dire tout de suite que je préfère largement la première version. Avec ce style d'écriture, c'est très difficile de s'y retrouver facilement. Vous êtes constamment en train de regarder le code de votre fonction, mais aussi où se trouve cette variable globale. Même si Python le permet, évitez autant que possible ce style d'écriture.

Par ailleurs, remarquez bien que ce script s'est largement alourdi par rapport au script précédent. Nous avons plus de lignes de code dues à chaque déclaration de variable globale dans chaque fonction.

Valeurs par défaut des paramètres

Dans la définition d'une fonction, il peut être judicieux de définir une valeur par défaut pour chacun des paramètres. Nous obtenons ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus. Si vous ne précisez pas spécifiquement un argument pour un paramètre, c'est la valeur par défaut qui est alors pris en compte pour réaliser le traitement prévu pour ce paramètre dans le corps de la fonction.

fonctions.py

```
# Un paramètre par défaut pour la fonction afficher()
def afficher(nom, prénom, séparateur=' '):
    print('-------------------------------------------------')
    if séparateur==' ': print(prénom.capitalize(), nom.upper())
    elif séparateur=='\n':
        print('Nom :', nom.upper())
        print('Prénom :', prénom.capitalize())

# Programme principal
nom = input('Votre nom : ')
prénom = input('Votre prénom : ')
afficher(nom, prénom)               # utilisation de la valeur par défaut
afficher(nom, prénom, '\n')         # prise en compte du troisième argument
```

Shell Python : [évaluer fonctions.py]

```
Votre nom : rémy
Votre prénom : emmanuel
-------------------------------------------------
Emmanuel RÉMY
-------------------------------------------------
Nom : RÉMY
Prénom : Emmanuel
```

*La fonction **« affiche() »** possède trois paramètres dont le dernier possède une valeur par défaut. Cette fonction peut donc être utilisée de deux façons différentes. Soit avec nous l'utilisons avec trois arguments, tous les paramètres copient alors ce qui est proposé par chacun des arguments. Soit nous utilisons cette fonction avec uniquement deux arguments, le dernier paramètre prend alors la valeur par défaut qui est proposée dans la définition de la fonction.*

Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que nous fournissons lors de l'appel d'une fonction doivent être transmis exactement dans le même ordre que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction possèdent chacun une valeur par défaut, sous la forme déjà décrite dans le chapitre précédent, nous pouvons faire appel à la fonction en fournissant les arguments correspondants, dans n'importe quel ordre, à la condition de nommer les paramètres correspondants.

fonctions.py

```
# Définition de la fonction afficher()
def afficher(nom, prénom='', civilité='Monsieur', séparateur=' '):
    print('-------------------------------------------------')
    if séparateur==' ': print(civilité, prénom.capitalize(), nom.upper())
    elif séparateur=='\n':
        print('Nom :', nom.upper())
        if prénom!='': print('Prénom :', prénom.capitalize())
        print('Civilité : ', civilité)

# Programme principal
afficher('rémy', séparateur='\n')
afficher('durand', civilité='Mademoiselle', prénom='martine')
afficher(civilité='Madame', nom='dupond')
```

Shell Python : [évaluer fonctions.py]

```
-------------------------------------------------
Nom : RÉMY
Civilité :  Monsieur
-------------------------------------------------
```

```
Mademoiselle Martine DURAND
-----------------------------------------------
Madame  DUPOND
```

Le premier paramètre ne possède pas de valeur par défaut, vous êtes donc obligé de spécifier impérativement le premier argument. Toutefois, si vous précisez l'étiquette, vous pouvez le placer dans un ordre différent. En réalité, la précision des étiquettes est optionnelle.

Fonction renvoyant plusieurs valeurs

Grâce au mécanisme des **« tuples »**, vu dans l'étude précédente, le langage Python permet une utilisation assez exceptionnelle, qui n'existe pas dans les autres langages de programmation, de pouvoir renvoyer simplement, au moyen de l'opérateur **« return »,** plusieurs valeurs, chacune séparée par une virgule (fonctionnement inhérent aux tuples).

fonctions.py

```
# Définition des fonctions
def division(dividende, diviseur):
    return dividende//diviseur, dividende%diviseur

def réel(nombre):
    if type(nombre) is float: nombre = str(nombre)
    entière, décimale = nombre.split('.')
    return int(entière), float('0.'+décimale)


# Programme principal
quotient, reste = division(8, 3)
print('Quotient = {}, Reste = {}'.format(quotient, reste))

flottant = 3.56
entière, décimale = réel(flottant)
print('Nombre = {}, partie entière = {}, partie décimale {}'.format(flottant, entière, décimale))

chaîne = '3.56'
entière, décimale = réel(flottant)
print('Nombre = {}, partie entière = {}, partie décimale {}'.format(chaîne, entière, décimale))
```

Shell Python : [évaluer fonctions.py]

```
Quotient = 2, Reste = 2
Nombre = 3.56, partie entière = 3, partie décimale 0.56
Nombre = 3.56, partie entière = 3, partie décimale 0.56
```

Nombre de paramètres variable dans une fonction

Comme beaucoup d'autres langages, Python permet de construire (de définir) des fonctions dont le nombre de paramètres est inconnu, ou si vous voulez le nombre d'arguments proposé durant l'appel de la fonction est variable. Intrinsèquement, cela consiste à utiliser une liste de paramètres.

*La mise en œuvre est très simple. Nous déclarons un seul paramètre qui prend en compte cette liste d'arguments. Toutefois, afin de bien comprendre qu'il s'agit d'une liste variable, vous devez préfixer votre paramètre à l'aide de l'opérateur **« * »**.*

*Le fait de préciser l'opérateur **« * »** devant le nom du paramètre fait que Python va automatiquement placer tous les paramètres de la fonction dans un **« tuple »** (arguments séparés par des virgules **« , »**), que nous pouvons ensuite traiter comme nous le désirons.*

Il est bien entendu possible de mixer vos paramètres de telle sorte que la fonction attende plusieurs paramètres qui doivent être fournis quoi qu'il arrive, suivis d'une liste de paramètres variables.

*A titre d'exemple, la fonction par excellence qui utilise ce principe est la fonction **« print() »** que nous avons largement utilisée. Voici d'ailleurs sa signature :*

print(*valeur, sep=' ', end='\n', file=sys.stdout) :

notes.py

```
# Définition de la fonction moyenne()
def moyenne(*notes, qui='Anonyme'):
    somme = 0.0
    for note in notes : somme += note
    trame = '{} : Notes = {}, moyenne = {}'
    print(trame.format(qui, notes, somme / len(notes)))

# Programme principal
martin = [8.5, 12, 13.5, 18, 4.5]
moyenne(12.5, 19, 5, 8.5, 15.5)     # plusieurs arguments dont le nombre est variable
```

```
moyenne(12, 10, 8, qui='Michel')    # plusieurs arguments dont le nombre est variable
moyenne(*martin, qui='Martin')      # un seul argument qui est une liste transformée en tuple
```

Shell Python : [évaluer notes.py]

```
Anonyme : Notes = (12.5, 19, 5, 8.5, 15.5), moyenne = 12.1
Michel : Notes = (12, 10, 8), moyenne = 10.0
Martin : Notes = (8.5, 12, 13.5, 18, 4.5), moyenne = 11.3
```

L'exemple ci-dessus est assez édifiant. Dans la première partie du programme principal, nous proposons le nombre (variable) de valeurs requises pour le calcul de la moyenne. Si vous possédez déjà une liste de valeurs et si vous désirez proposer cette liste à la fonction, il faut arriver à décomposer cette liste en une suite de valeurs individuelles. Cela se fait grâce, de nouveau, à l'opérateur ***« * »****.*

Nous utilisons donc l'opérateur « * » dans deux cas spécifiques. Si nous le proposons dans la définition de la fonction, cela signifie que les arguments fournis lors de l'appel seront capturés dans le paramètre, sous la forme d'un « tuple ». Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs arguments envoyés à la fonction.

Alléger le nombre de paramètres dans une fonction en utilisant les dictionnaires

Dans python, nous venons de le découvrir dans cette étude, il est possible, durant l'appel de fonction, de nommer les paramètres que nous utilisons et dans l'ordre que nous voulons, en spécifiant la valeur de chacun des arguments requis. Si nous regardons ce principe de plus prêt, cela ressemble beaucoup à l'utilisation des **« dictionnaires ».**

Python a justement prévu d'élargir le principe des ***« dictionnaires »*** *dans la définition de la fonction plutôt que d'avoir une grande liste de paramètres qualifiés (par défaut), surtout lorsque le nombre de paramètres est relativement conséquent. Pour cela, nous utilisons de nouveau l'opérateur* ***« * »*** *qui sera cette fois-ci dédoublé :* ***« ** »****.*

fonctions.py

```
from math import sqrt

# Définition des fonctions
def affichage(**identité):
    print('Identité :', identité['civilité'], identité['prénom'].capitalize(), identité['nom'].upper())

def module(réel=0, imaginaire=0):
    calcul = sqrt(réel**2 + imaginaire**2)
    print('réel={}, imaginaire={}, module={:0.2f}'.format(réel, imaginaire, calcul))

# Programme principal
personne = {'nom':'durand', 'prénom':'michèle', 'civilité': 'Mme'}
complexe = {}
complexe['réel'] = 5.6
complexe['imaginaire'] = -7.8;
nombre = 5

affichage(civilité='Mr', nom='rémy', prénom='emmanuel')
affichage(**personne)

module(1, 1)
module(réel=2, imaginaire=1)
module(**complexe)
module(réel=nombre)
module(imaginaire=1)
```

Shell Python : [évaluer fonctions.py]

```
Identité : Mr Emmanuel RÉMY
Identité : Mme Michèle DURAND
réel=1, imaginaire=1, module=1.41
réel=2, imaginaire=1, module=2.24
réel=5.6, imaginaire=-7.8, module=9.60
réel=5, imaginaire=0, module=5.00
réel=0, imaginaire=1, module=1.00
```

La première fonction utilise cette technique. Elle possède un seul paramètre ***« identité »*** *qui représente une liste de paramètres sous forme de* ***« dictionnaire »****. À l'intérieur de la fonction, vous devez récupérer chaque élément utile pour le traitement de la fonction en utilisant la syntaxe classique prévu pour les* ***« dictionnaires »****. Je ne vous cache pas que je ne suis pas favorable à ce type de pratique puisque cela finalement alourdi le code plutôt que de l'alléger.*

Lorsque vous utilisez une telle fonction, vous pouvez l'appeler de deux façons différentes, soit en proposant votre liste d'arguments en passant par les étiquettes, soit en ayant un seul argument de type ***« dictionnaire »*** *qui peut être pré-rempli en une seule fois ou par étape, auquel cas vous devez utiliser la syntaxe du double opérateur* ***« ** »*** *.*

Par contre, vous pouvez définir une fonction de façon classique, en listant nommément tous les paramètres requis avec éventuellement des valeurs par défaut, et utiliser ensuite, lors de l'appel de cette fonction, un dictionnaire qui comportera exactement les mêmes éléments que les paramètres requis. Cette technique est vraiment intéressante dans le cas où vous devez récupérer des valeurs au fur et à mesure, avant de les proposer définitivement à votre fonction.

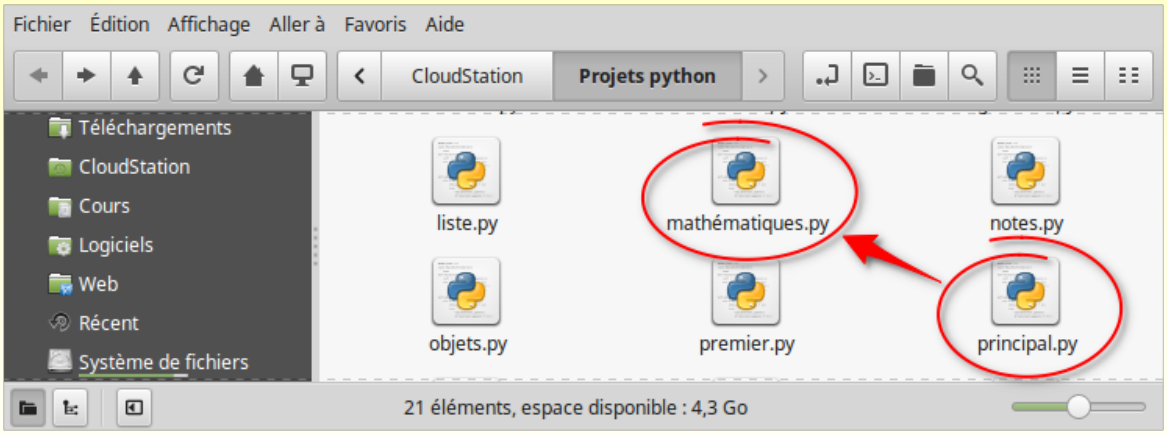
Vous avez un bel exemple, dans le code ci-dessus, qui utilise toutes les différentes techniques possibles sur la fonction ***« module() ».***

Les modules – scripts multi-fichiers

Lorsque votre script commence à s'étoffer, il peut être judicieux de le découper en plusieurs parties, en regroupant les fonctions suivant des thèmes particuliers afin de favoriser la réutilisation et le travail en équipe.

Dans ce cadre là, nous travaillons plutôt sous forme de modules séparés, des fichiers distincts, qu'il sera possible par la suite de réutiliser pour d'autres programmes. Effectivement, une fois que nous avons construit quelques fonctions génériques, elles peuvent constituer une base d'outils indispensables par la suite, cela permet d'avoir un rendement optimal. Ce n'est pas la peine de réinventer la roue à chaque fois. Autant utiliser les compétences déjà acquises.

C'est d'ailleurs ce que nous faisons lorsque nous faisons appel à la fonction racine carré « sqrt() » du module « math », déjà créé par les développeur de Python, que nous intégrons facilement dans nos différents scripts.



Un module est grossièrement une portion de code que nous enfermons dans un fichier spécifique. Nous emprisonnons ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si nous désirons travailler avec les fonctionnalités prévues par le module (celles qui sont intégrées dans le module), il suffit ***« d'importer »*** *le module et utiliser ensuite toutes les fonctions qui vous semblent nécessaires avec éventuellement les variables déclarées à l'intérieur.*

À titre d'exemple, je vous propose de construire deux modules, le premier ***« mathématiques.py »*** *qui intègre trois fonctions à connotation mathématiques, et un deuxième module* ***« principal.py »*** *qui correspond, comme son nom l'indique, au programme principal et qui utilise les compétences du premier module. Voici le code correspondant :*

mathématiques.py

```
# Fonction factorielle
def factorielle(n):
    if n==0: return 1
    else: return n*factorielle(n-1)

# Fonction fibonacci
def fibonacci(n):
    if n<=1: return n
    else: return fibonacci(n-1)+fibonacci(n-2)

# Fonction recherche nombre premier
from math import sqrt
def ispremier(n):
    for diviseur in range (2, int(sqrt(n)+1)):
        if n%diviseur == 0: return False
    return True
```

principal.py

```
import mathématiques # importation du module "mathématiques"

# utilisation des fonctions intégrées dans le module "mathématiques"
print('5! =', mathématiques.factorielle(5))
print('fibonnaci(7) =', mathématiques.fibonacci(7))
print('13 premier ?', mathématiques.ispremier(13))
print('49 premier ?', mathématiques.ispremier(49))
```

Shell Python : [évaluer principal.py]

```
5! = 120
fibonnaci(7) = 13
13 premier ? True
49 premier ? False
```

*Le résultat ci-dessus correspond au lancement du script **« principal.py »** ou à l'exécution de la commande qui suit lorsque nous sommes en mode terminal du système d'exploitation (dans le répertoire où se situent ces modules) :*

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
manu@manu-M70Vn ~/CloudStation/Projets python $ python3 principal.py
5! = 120
fibonnaci(7) = 13
13 premier ? True
49 premier ? False
manu@manu-M70Vn ~/CloudStation/Projets python $
```

*Lorsqu'un module a besoin des compétences d'un autre module, il suffit d'utiliser la commande **« import »** suivi du nom du module souhaité. À partir de là, vous pouvez utiliser toutes les fonctions intégrées, par contre vous devez systématiquement préfixer votre fonction du module concerné en utilisant l'opérateur de séparation **« . »**.*

Lorsque vous réalisez une importation, c'est comme si nous avions un seul script, le code du module importé est alors intégré au code principal, pour ne former qu'une seule et même entité.

*Un module correspond donc à un fichier, mais c'est plus que cela, il s'agit aussi d'un **« espace de nom »**. En effet, si plusieurs modules possèdent des noms de fonction identiques, c'est le nom du module qui permet de savoir exactement de quelle fonction il s'agit, et d'éviter ainsi des conflits éventuels.*

Changer d'espace de nom dans le module principal

Dans certains cas, vous pouvez également changer de nom pour **« l'espace de nom »** dans lequel sera stocké le module importé pour avoir par exemple un raccourci d'écriture. Il suffit alors d'utiliser le mot réservé **« as »** comme cela vous est présenté ci-dessous :

principal.py

```
import mathématiques as m # importation du module "mathématiques"

# utilisation des fonctions intégrées dans le module "mathématiques"
print('5! =', m.factorielle(5))
print('fibonnaci(7) =', m.fibonacci(7))
print('13 premier ?', m.ispremier(13))
print('49 premier ?', m.ispremier(49))
```

Shell Python : [évaluer principal.py]

```
5! = 120
fibonnaci(7) = 13
13 premier ? True
49 premier ? False
```

Importer juste ce qui est nécessaire sans problème de conflit d'espace de nom

Si vous êtes sûr de ne pas avoir de conflit avec les fonctions que vous avez besoin, il peut être embêtant de préfixer systématiquement chacune des fonctions que vous avez besoin avec le nom du module correspondant. Il existe une deuxième syntaxe, plus utilisée, qui permet juste de choisir la ou les fonctions nécessaires pour votre programme principal.

*Il n'est pas toujours indispensable de tout intégrer, notamment si vous avez besoin que d'une seule fonction. La syntaxe est alors la suivante : **« from module import fonction »**.*

principal.py

```
# importation des fonctions du module "mathématiques"
from mathématiques import factorielle, fibonacci, ispremier

# utilisation des fonctions intégrées dans le module "mathématiques"
print('5! =', factorielle(5))
print('fibonnaci(7) =', fibonacci(7))
print('13 premier ?', ispremier(13))
print('49 premier ?', ispremier(49))
```

*Vous remarquez que maintenant, nous pouvons utiliser les fonctions directement grâce à cette syntaxe. Par ailleurs, si vous avez besoin de toutes les fonctions du module, il existe un raccourci d'écriture grâce à l'emploi du joker **« * »**.*

principal.py

```
# importation des fonctions du module "mathématiques"
from mathématiques import *

# utilisation des fonctions intégrées dans le module "mathématiques"
print('5! =', factorielle(5))
print('fibonnaci(7) =', fibonacci(7))
print('13 premier ?', ispremier(13))
print('49 premier ?', ispremier(49))
```