Révisions : boucles, tests et fonctions en Python

1 Rappels sur les tests et les boucles

1.1 Notions de base en Python

Rappelons qu'en Python, la commande `x=a` permet d'attribuer la valeur `a` au contenu de la variable `x`. De plus, au sein d'un script, la commande # permet de désactiver la ligne qui suit ce symbole (Python se contentera d'ignorer toute la ligne en question). L'intérêt ici est de permettre de laisser des informations dans le script sans que ces dernières soient effectuées. De plus, la commande `print` permet d'afficher le contenu d'une variable (sous la forme `print(x)`) ou un texte (sous la forme `print("x")`). En outre, la commande `input` permet (lors de l'exécution d'un script) de demander à l'utilisateur de rentrer une valeur ou une expression dont le script aura besoin pour tourner. On l'utilisera sous la forme `x=input()` ou `x=input("entrer la valeur de x")`. Par défaut, Python considèrera que `x` est une chaîne de caractères. On doit donc spécifier le type de la variable `x` si l'on veut effectuer des calculs avec. Si `x` est un entier (resp. un nombre à virgule), on tapera plutôt `x=int(input("entrer un entier"))` (resp. `x=float(input("entrer un réel"))`. Enfin, on rappelle que les opérations élémentaires sont codées en Python sous la forme suivante :

Code Python	Opérations de base
`a+b`	$a+b$
`a-b`	$a-b$
`a*b`	ab
`a/b`	a/b
`a**b`	a^b

On dispose aussi en Python de quelques racourcis. Par exemple, `x+=y` correspond à `x=x+y`. Idem pour les raccourcis `/=`, `*=`, `-=` et `**=`. Enfin, il est possible dans Python de charger des bibliothèques, et ce à l'aide de la commande `import`. Une bibliothèque rassemble tout un ensemble de commandes ou de fonctions qui sont inconnues pour la version de base de Python, mais qui nous serviront constamment par la suite. La commande `import` s'utilise en général sous la forme :

`import` <nom de la bibliothèque> `as` <raccourci d'appel de la bibliothèque>

En particulier, la bibliothèque `numpy` rassemble beaucoup de commandes indispensables en mathématiques (notamment les fonctions de base). Si l'on souhaite en disposer, on pourra commencer par taper `import numpy as np`, puis utiliser les fonctions élémentaires rassemblées dans le tableau suivant (pour les fonctions trigonométriques, les angles doivent être exprimés en radians) :

Code Python	Fonction
`np.sin( )`	sinus
`np.cos( )`	cosinus
`np.abs( )`	valeur absolue
`np.sqrt( )`	racine carrée
`np.exp( )`	exponentielle
`np.log( )`	logarithme népérien
`np.floor( )`	partie entière
`np.pi`	π
`np.e`	e (base du logarithme népérien)

1.2 Rappels sur les tests

En Python (comme dans tout langage de programmation informatique), on utilisera régulièrement des tests et des boucles. Ces tests et ces boucles seront toujours à écrire dans l'éditeur (qui sert à la fois pour les scripts et les fonctions), jamais dans l'invité de commande. En Python, le test `if` se présentera sous la forme suivante :

```
if <condition à vérifier>:
    <liste d'instructions à effectuer si la condition est vérifiée>
else:
    <liste d'instructions à effectuer si la condition n'est pas vérifiée>
```

A noter ici que les instructions doivent être indentées, de façon à ce que Python puisse les effectuer (sinon il ne le fera pas), et que chaque ligne démarrant avec `if` ou `else` devra se terminer avec `:`. A noter aussi que les objets qui apparaissent dans les conditions des tests `if` peuvent être considérés aussi comme des variables informatiques qui peuvent prendre deux valeurs, à savoir : `True` et False. Voici la liste des premiers symboles en Python à utiliser dans les tests `if`, avec leur signification à droite (ici P et Q renvoient à des propositions) :

`a==b`	a est égal à b
`a!=b`	a est différent de b
`a<b`	a est inférieur strictement à b
`a>b`	a est supérieur strictement à b
`a<= b`	a est inférieur ou égal à b
`a>= b`	a est supérieur ou égal à b
`P and Q`	Renvoie `True` si P et Q est vraie, `False` sinon
`P or Q`	Renvoie `True` si P ou Q est vraie, `False` sinon
`not P`	Renvoie `True` si P est fausse, `False` sinon

Exemple 1.1 *La condition "$a < x \leq b$" se traduit en Python par :* `(a<x) and (x<=b)`.

Exemple 1.2 *Etant donné un réel a donné au préalable, le script ci-dessous remplace a par $a+1$ si $a > 1$ et a par $\frac{a}{2}$ sinon, puis affiche le résultat obtenu :*

```
# a : variable réelle
if   a>1:
    a=a+1
else:
    a=a/2
print(a)
```

A noter que, si l'on veut écrire un script qui réalise une alternative plus complexe, du type ci-dessous :

Si <condition1>
    <liste 1 d'instructions>
sinon si <condition2>
    <liste 2 d'instructions>
sinon
    <liste 3 d'instructions>

alors on pourra soit imbriquer plusieurs tests `if` les uns dans les autres, soit utiliser le test `if` avec la commande `elif` de la façon suivante :

```
if <condition1>:
    <liste d'instructions 1>
elif <condition2>:
    <liste d'instructions 2>
else:
    <liste d'instructions 3>
```

Exemple 1.3 *Considérons deux réels a et b donnés au départ. Le programme suivant permet d'en donner le plus grand ou, le cas échéant, d'indiquer qu'ils sont égaux.*

```
# Classement de deux nombres
# Variables: a,b réels
if a>b:
    print("a est le plus grand")
elif a<b:
    print("b est le plus grand")
else:
    print("a vaut b")
```

Ici, on voit qu'on a utilisé la commande `elif`. On aurait tout aussi bien pu insérer deux tests `if` l'un dans l'autre, sous la forme suivante :

```
if a>b:
    print("a est le plus grand")
else :
    if a<b:
        print("b est le plus grand")
    else:
        print("a vaut b")
```

A noter ici le respect de l'indentation pour que les deux tests imbriqués l'un dans l'autre fonctionnent !

1.3 Rappels sur la boucle `for`

En informatique, on est souvent amené à répéter la même suite d'instructions dans un algorithme ou un programme. D'où l'idée d'inventer une seule instruction qui remplacera automatiquement cette répétition en boucle, que l'on appelle une boucle `for`. Attention, dans le cas d'une boucle `for`, il faut contrôler le nombre d'itérations, donc savoir au départ combien de fois on itérera le processus. En Python, une boucle `for` se présente toujours comme suit :

```
for <indice> in <liste>:
    <instruction 1>
    <instruction 2>
          ⋮
    <instruction n>
```

A noter ici que, comme pour le test `if`, les instructions doivent être indentées, de façon à ce que Python puisse les effectuer. A noter aussi que chaque ligne démarrant avec `for` devra se terminer avec un `:`. Le symbole <indice> est l'indice de la boucle qui prend successivement toutes les valeurs de <liste>. Plus précisément, pour chaque valeur de <liste>, la série des instructions ci-dessous est exécutée. La liste des valeurs prises par l'indice est créée en général à l'aide de la commande `range`. Plus précisément, la commande `range(a,b,p)` va nous donner la liste de tous les réels compris entre a et b (**avec b exclu !**), en progression arithmétique de raison (ou de pas) p. Si le pas vaut 1, on ne sera pas obligé de le spécifier, et on pourra écrire `range(a,b)` au lieu de `range(a,b,1)`. Enfin, si la valeur de départ a est égale à 0, on pourra écrire tout simplement `range(b)` au lieu de `range(0,b)` ou `range(0,b,1)`. A titre d'exemple, la commande `range(1,n+1)` renvoie la liste $1, 2, 3, ..., n$, et la commande `range(n+1)` la liste $0, 1, 2, ..., n$.

Exemple 1.4 *Pour calculer et afficher la somme des $n+1$ premiers termes de la suite géométrique $(q^k)_{k\geq 0}$ (et ce sans utiliser la formule du cours de mathématiques), on pourra utiliser le script suivant :*

```
#  Somme des termes d'une suite géométrique
s=1
for k in range(1,n+1):
    s=s+q**k
print(s)
```

Exemple 1.5 *Considérons la suite (u_n) définie par $u_0 = 0$ et par la relation de récurrence :*

$$\forall n \in \mathbb{N}, \quad u_{n+1} = {u_n}^2 - 3.$$

Pour calculer et afficher le n-ième terme de cette suite, on utilisera le script suivant :

```
# Calcul du n-ième terme de la suite
u=0
for k in range(1,n+1):
    u=u**2-3
print(u)
```

1.4 Rappels sur la boucle `while`

En informatique, on a vu que l'on était souvent amené à répéter la même opération plusieurs fois de suite dans un algorithme ou un programme. Le premier exemple d'instruction qui permet d'effectuer des répétitions en boucle est la boucle `for`. Celle-ci a l'avantage de permettre de répéter une même opération un nombre de fois défini au départ. Cependant, il arrive que, dans certains cas, on ne puisse pas spécifier au préalable le nombre de fois qu'une opération donnée doit être effectuée. Dans ce cas, on utilise une autre

boucle, à savoir la boucle `while`. Cette dernière fonctionnera de la façon suivante : le programme réitèrera l'opération demandée tant qu'une certaine condition (spécifiée au départ) sera satisfaite, et s'arrêtera dès que cette condition ne l'est plus. Un programme faisant intervenir la boucle `while` se présente de la façon générale suivante :

```
while <condition à vérifier>
    <instruction 1>
    <instruction 2>
           ⋮
    <instruction n>
```

Le fonctionnement de la boucle `while` est alors le suivant. Si la condition à vérifier est vraie (ou satisfaite), alors la liste d'instructions <instruction 1>,<instruction 2>,...,<instruction n> est effectuée, puis on recommence. Si elle est fausse (ou non satisfaite), alors la boucle s'arrête là. A noter que la condition à vérifier est rédigée à l'aide des mêmes conventions qu'un test `if`. A noter aussi qu'une boucle `for` peut toujours se représenter à l'aide d'une boucle `while`. En effet, supposons qu'au sein d'un programme, on ait les instructions suivantes :

```
for k in range(1,n+1):
    <instruction 1>
    <instruction 2>
           ⋮
    <instruction n>
```

Alors on peut remplacer ce groupe d'instructions de façon équivalente par :

```
k=1
while k <=n:
    <instruction 1>
    <instruction 2>
           ⋮
    <instruction n>
    k=k+1
```

Exemple 1.6 *On cherche à écrire un programme qui demande un entier $n > 0$, puis qui calcule $n!$. On procède alors comme suit :*

```
n=int(input("entrer un entier n strictement positif"))
N=1
k=1
while k<=n:
    N=N*k
    k=k+1
print(N)
```

Mais de façon générale, on utilise une boucle `while` lorsque l'on veut réaliser un certain nombre d'itérations sans savoir au préalable exactement combien il faut en faire. On peut même inclure des tests `if` au sein d'une telle boucle.

Exemple 1.7 *On cherche à écrire un programme qui demande deux nombres réels a et b strictement supérieurs à 1, puis qui calcule et affiche la plus petite puissance entière de a supérieure ou égale à b. On procède alors comme suit :*

```
a=float(input('entrer un reel a'))
b=float(input('entrer un reel b'))
x=a
while x<b:
    x=a*x
print(x)
```

2 Rappels sur les fonctions

En Python, il arrive fréquemment que l'on ait besoin d'introduire de nouvelles fonctions (hormis celles de Python) pour pouvoir faire des calculs ou écrire des programmes. Pour définir une nouvelle fonction, on écrit la définition de la fonction dans un fichier avec une extension *.pi*. La syntaxe d'une fonction sera la suivante :

```
def  <nom de la fonction>(<arguments d'entrée>):
      ⋮
     return <arguments de sortie>
```

A noter que le nom de la fonction peut être arbitraire, et qu'il doit être composé de caractères alphanumériques sans accent ou de _ et commencer par une lettre. A noter aussi que l'on peut n'avoir aucun paramètre d'entrée (dans ce cas, on ne met rien entre parenthèses après le nom de la fonction). Idem pour ce qui est des paramètres de sortie. Quant à la commande `return`, elle permet d'affecter les valeurs des arguments de sortie à des variables (ce que ne permet pas la commande `print` par exemple). Rappelons enfin qu'il ne faut pas oublier de charger une bibliothèque au préalable de la définition de la fonction (par exemple : `numpy`) si jamais on a besoin de commandes mathématiques spécifiques pour écrire les instructions de la dite fonction !

Exemple 2.1 *Supposons que l'on veuille introduire une nouvelle fonction* `truc` *qui calcule pour tout réel* x *le nombre* $truc(x) = x^2 + x$. *Pour ce faire, on écrira le script suivant :*

```
def truc(x):
    y=x**2+x
    return y
```

Exemple 2.2 *Supposons que l'on veuille introduire une nouvelle fonction* `cercle` *qui calcule le périmètre et l'aire d'un cercle dont on donne le rayon* r. *Dans ce cas, on pourra utiliser le script suivant :*

```
import numpy as np

def cercle(r):
#  argument d'entrée: le rayon r
#  arguments de sortie: le périmètre (peri) et l'aire (aire)
    x=2*np.pi*r
    y=np.pi*(r**2)
    return x,y
```

A noter que, si l'on exécute cette fonction, puis que l'on effectue la commande `[a,b]=cercle(r)`, *alors Python va affecter les résultats obtenus à* `a` *et* `b`, *et donc les reconnaîtra comme tels par la suite.*

Enfin, rappelons qu'une fonction est dite *récursive* si elle s'appelle elle-même jusqu'à ce qu'elle ne le fasse plus, c'est-à-dire si le résultat attendu est obtenu en réitérant un certain nombre de fois le même processus. C'est le genre de fonction qui intervient, par exemple, lorsque l'on veut construire le n-ème terme d'une suite définie par une relation de récurrence. Pour pouvoir s'arrêter, une fonction récursive doit avoir une condition d'arrêt, donnée par un test if. En voici des exemples :

Exemple 2.3 *Supposons que l'on veuille introduire une nouvelle fonction* `suite` *qui calcule le* n*-ème terme de la suite* $(u_n)_{n\geq 0}$ *définie par* $u_0 = 1$ *et par la relation de récurrence suivante :* $\forall n \in \mathbb{N}$, $u_{n+1} = \sin(u_n)$. *Dans ce cas, on pourra utiliser le script suivant :*

```
import numpy as np

def suite(n):
    if n==0:
        return 1
    else:
        return np.sin(suite(n-1))
```

Exemple 2.4 *Supposons que l'on veuille introduire une nouvelle fonction* `fibo` *qui calcule le* n*-ème terme de la suite de Fibonacci* $(u_n)_{n\geq 0}$ *définie par* $u_0 = 0$, $u_1 = 1$ *et par la relation de récurrence suivante :* $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$. *Dans ce cas, on pourra utiliser le script suivant :*

```
def fibo(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```