Chapitre 2 : Instructions itératives

Table des matières

1 Instructions élémentaires

1.1 Instructions conditionnelles

Définition: *(if)*
Les instructions conditionnelles se définissent à l'aide de l'instruction `if` et prennent la forme suivante :

```
if expression booleenne:
    bloc..............
    instructions 1..
else:
    bloc..............
    instructions 2..
```

Le fonctionnement de cette instruction est le suivant : si l'expression booléenne de la première ligne s'évalue en `True`, le premier bloc d'instructions est exécuté, si elle s'évalue en `False` c'est le second bloc qui est exécuté.

Exemple :

```
if 2 <= 3 :
    print("Le resultat est vrai")
else :
    print("Le resultat est faux")
```

ou encore :

```
x = 2
if x == 3 :
    print("Le resultat est vrai")
else :
    print("Le resultat est faux")
```

Il est par ailleurs possible de multiplier les tests grâce à l'instruction `elif`, le fonctionnement est le suivant :

```
if expression booleenne 1:
    bloc..............
    instructions 1..
elif expression boolenne 2:
    bloc..............
    instructions 2..
else :
    bloc..............
    instructions 3..
```

- Si l'expression booléenne 1 s'évalue en `True`, le bloc d'instructions 1 est réalisé.
- Si l'expression booléenne 1 s'évalue en `False` et l'expression booléenne 2 s'évalue en `True`, le bloc d'instructions 2 est réalisé.
- Sinon, le bloc d'instruction 3 est réalisé.

1.2 Instructions itératives

1.2.1 Boucles énumérées

Réaliser une itération, ou encore une boucle, c'est répéter un certain nombre de fois des instructions semblables.
Dans la plupart des langages de programmation, il existe deux instructions pour réaliser une boucle, suivant qu'on peut calculer à l'avance le nombre d'itérations à réaliser (on parle alors de boucles énumérées) ou que le nombre d'itérations dépend de la réalisation ou non d'une certaine condition (on parle dans ce cas de boucles conditionnelles) ; le langage Python ne fait pas exception à la règle.

La commande `range()` peut prendre entre 1 et 3 arguments entiers :

Vocabulaire:

- `range(b)` : énumère les entiers $0,1,2,..,b-1$.
- `range(a,b)` : énumère les entiers $a,a+1,a+2,..,b-1$.
- `range(a,b,c)` : énumère les entiers $a,a+c,a+2c,..,a+nc$ où n est le plus grand entier tel que $a+nc<b$.

Exemple :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(5, 15))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>> list(range(1, 20, 3))
[1, 4, 7, 10, 13, 16, 19]
```

Définition: *(for)*
Les instructions itératives énumérées (boucles bornées) se définissent à l'aide de l'instruction `for` et prennent la forme suivante :

```
for ... in range(...):
    bloc ..........................
    ..............................
    instructions .................
```

Immédiatement après le mot-clé `for` doit figurer le nom d'une variable, qui va prendre les différentes valeurs de l'énumération produite par l'instruction `range`. Pour chacune de ces valeurs, le bloc d'instructions qui suit sera exécuté.

Exercice :
Que va afficher la commande suivante ?

```
for x in range(2, 10):
     print(x, x**2)
```

De même que pour les instructions conditionnelles, il est bien entendu possible d'imbriquer des boucles à l'intérieur d'autres boucles; attention seulement à respecter les règles d'indentation pour délimiter chacun des blocs d'instructions.

Exercice :
Que va afficher la commande suivante ?

```
for x in range(1, 6):
    for y in range(1, 6):
        print(x * y, end='␣')
    print('/')
```

1.2.2 Boucles conditionnelles

Définition: (*while*)
Les instructions itératives conditionnelles (boucles non bornées) se définissent à l'aide de l'instruction `while` et prennent la forme suivante :

```
while condition:
    bloc ..........................
    ..............................
    instructions ................
```

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée; elle peut donc tout aussi bien ne jamais réaliser cette suite d'instructions (lorsque la condition n'est pas réalisée au départ) que de les réaliser un nombre infini de fois (lorsque la condition reste éternellement vérifiée).

Exemple :

```
while 1+1==3:
    print('abc')
print('def')

'def'
```

Exercice :
Que va afficher la commande suivante ?

```
x=10
while x>0:
    print(x)
    x=x-1
```

Exercice :
Ecrire une instruction pour la demande suivante: Afficher la liste des $\frac{1}{x}$ pour x entre 1 et 5.

2 Notion de fonction

2.1 Définition d'une fonction en Python

Lorsqu'on utilise le langage Python de base, seul un nombre très limité de fonctions est disponible; heureusement, il existe une multitude de modules additionnels que l'utilisateur peut ajouter en fonction de ses besoinset qui lui fournissent des fonctions supplémentaires. Par exemple, les modules `math` ou `numpy` enrichissent le corpus utilisable des fonctions mathématiques suivantes :

- les fonctions trigonométriques sin, cos, tan ainsi que leurs fonctions réciproques arcsin, arccos, arctan (les angles s'expriment par défaut en radians).
- la fonction racine carrée sqrt.

Afin de pouvoir utiliser les fonctions d'un module, il faut l'insérer dans le script avec la commande ci-dessous (insérer tous les éléments du module avec *) :

```
1 from numpy import *
2 from numpy import sin,cos,tan
```

Dans ce cas, toutes les fonctions de ce module devront être préfixées du nom du module pour être utilisées : pour calculer un sinus il faudra utiliser la fonction `numpy.sin`.
Comme les noms des modules peuvent être longs on peut leur donner un nom d'usage plus court en écrivant par exemple :

```
1 import numpy as np
2
3 >>>np.sqrt(2)
4 1.4142135623730951
5
6 >>>np.sin(np.pi)
7 0
```

Les deux modules primordiaux que nous allons utiliser majoritairement sont `numpy` (importé en `np`) qui contient de nombreuses fonctions mathématiques et `matplotlib.pyplot` (importé en `plt`) qui sert à représenter graphiquement des fonctions (c.f. Chapitre 4).

Définition: (*def*)
On définit une fonction en python à l'aide du mot clé `def`. Il faut lui attribuer un nom, préciser la liste de ses paramètres, lui décrire les différentes instructions à réaliser et enfin le résultat qu'elle doit retourner. La syntaxe générale est la suivante :

```
1 def nomdelafcn(liste de parametres):
2     bloc ..........................
3     ..............................
4     instructions ................
5     return resultats
```

On peut donc définir des fonctions classiques comportants un seul ou plusieurs arguments.

Exemple :

- Pour la fonction $f : x \mapsto 2x + 3$:

```
1 def f(x):
2     return 2x+3
3
4 >>>f(4)
5 11
```

- Pour la fonction qui à un vecteur $\overrightarrow{u} = \begin{pmatrix} x \\ y \end{pmatrix}$ associe sa norme $||\overrightarrow{u}|| = \sqrt{x^2 + y^2}$:

```
1 import numpy as np
2 def norme(x,y):
3     return np.sqrt(x**2 + y**2)
4
5 >>>norme(3,4)
6 5
```

Exercice :
Ecrire en python les fonctions suivantes :

- $f : x \mapsto 5x^2 - 6$
- $g : (x, y, z) \mapsto x + 2y + 3z$

2.2 Applications aux Probabilités/Statistiques

2.2.1 Statistiques

Rappel: (*Moyenne et écart-type*)
Soit $(x_1, x_2, ..., x_n)$ une suite de n valeurs.

- Moyenne : $\overline{x} = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n} x_k = \dfrac{x_1 + x_2 + ... + x_n}{n}$
- Variance : $v = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n} (x_k - \overline{x})^2$
- Ecart-type : $\sigma = \sqrt{v}$

On définit donc la fonction `moyenne` suivante qui prend en argument une liste de données et qui renvoie la moyenne.

```
1 def moyenne(L):
2     somme = 0
3     for x in L:
4         somme = somme + x
5     return somme/ len(l)
```

Exercice :
Rédiger une fonction en python qui prend en argument une liste et qui renvoie l'écart type associé.

2.2.2 Probabilités

En anglais `random` signifie aléatoire. C'est ainsi le nom du module de fonctions Python à charger en premier dans un programme Python: tout programme Python dans lequel on veut utiliser des calculs "au hasard" doit commencer par la ligne:

```
1 from random import *
```

Vocabulaire:

- `random()` : donne un nombre au hasard, dans l'intervalle $[0; 1[$.
- `randint(a,b)` : donne un nombre entier au hasard entre `a` et `b`.

Exemple :

```
from random import *

>>> random()
0.34970520325568466

>>> random()
0.7179949466450943

>>> random()
0.6152473549959072

>>> randint(3,12)
3

>>> randint(3,12)
8

>>> randint(3,12)
7
```

Pour simuler un jet de dé, nous allons utiliser la fonction `randint(1, 6)` qui retourne un entier pris au hasard entre 1 et 6. Le script qui consiste à simuler un jet de deux dés jusqu'à obtenir un double-six est :

```
s=0
while randint(1,6)!=6 and randint(1,6)!=6:
    s+=1
print(s)
```

Cette boucle a pour effet d'itérer la variable `s` jusqu'à la réalisation de l'événement attendu : un double-six.
Pour obtenir le nombre moyen de jets nécessaire à la réalisation de cet événement, il suffit de réaliser cette expérience un nombre suffisant de fois puis de calculer la moyenne de la valeur prise par `s` au sortir de cette boucle.

```
def Moyenne_jets(n):
    e=0
    for k in range(n):
        s=0
        while randint(1,6)!=6 and randint(1,6)!=6:
            s+=1
        e+=s
    return e/n
```

Exercice :

1. La fonction `randint` tire un nombre entier aléatoirement. Que fait le programme suivant ? Qu'affiche-t-il ?

```
from random import *

d=randint(1,6)
print(d)
if d==6:
    print("Gagne")
else:
    print("Dommage")
```

2. Modifier le programme précédent avec une boucle en un programme qui lance 10 fois un dé et qui compte et affiche le nombre de 6 obtenus. Calculer et afficher alors, le pourcentage de 6 obtenus. Que devient ce pourcentage lorsqu'on augmente le nombre de lancers (10 lancers, puis 100, 1000, . . .)