Chapitre (S1) 1 Fondamentaux

1 Fonction .....
2 Boucle for .....
3 Tests conditionnels .....
4 Boucle while .....

Objectifs

- Savoir déclarer une fonction et exécuter celle-ci.
- Savoir réaliser une boucle à l'aide de l'instruction `for`.

1. FONCTION

1.1. Déclaration et exécution

Une fonction a pour objectif de produire un **résultat** à partir des valeurs d'un ou plusieurs **paramètres**.

On distingue :

- la phase de **déclaration** de la fonction, décrivant de manière générale comment se calcule le résultat à partir des paramètres, ces derniers n'ayant pas à cette étape de valeurs déterminées ;
- la phase d'**exécution** de la fonction, qui produit le résultat correspondant à des valeurs spécifiées des paramètres.

La syntaxe générale de la déclaration d'une fonction est la suivante :

```
def nom_de_la_fonction(paramètre_1,...,paramètre_n):
    instructions
    return(valeur_à_renvoyer)
```

La première ligne commence toujours par le mot-clé `def` (qui permet à l'ordinateur de savoir que la partie de code qui va suivre forme la déclaration d'une fonction), suivi du nom que portera cette fonction. Ce nom, comme tous les noms de variables, peut être choisi assez librement en utilisant des lettres, des chiffres et le caractère « _ » (quelques contraintes : les autres caractères ne sont pas autorisés ; le nom ne peut pas commencer par un chiffre ; il n'est pas possible d'utiliser comme nom un mot déjà utilisé en tant qu'instruction du langage, comme `def` par exemple). Suit entre parenthèses, séparés par des virgules, une liste de noms de variables qui seront les paramètres de la fonction.

Enfin, la première ligne se termine impérativement par le caractère « : » (à ne pas oublier, sans quoi une erreur serait provoquée à l'exécution du code).

Les lignes suivantes forment une suite d'instructions permettant de déterminer, à partir des paramètres, la valeur qui sera renvoyée par la fonction grâce à l'instruction `return`. Notez que, si une fonction peut avoir plusieurs paramètres, en revanche elle ne peut renvoyer qu'**une seule** valeur. D'ailleurs, s'il reste des instructions dans le corps de la fonction **après** une instruction `return`, celles-ci seront ignorées par l'ordinateur : la fonction prend fin lorsque l'instruction `return` est rencontrée. Enfin, à part la première ligne, les instructions suivantes sont décalées vers la droite, on dit qu'elles sont **indentées**. Outre que cette présentation facilite la lecture, c'est en fait un **impératif** du langage python, et ne pas respecter cette indentation provoque une erreur à l'exécution. L'espacement choisi est généralement de quatre caractères d'espace (il s'agit là d'une convention non obligatoire, il est possible d'utiliser un espacement différent, mais le même doit être impérativement figurer pour chacune des lignes).

Une fois correctement déclarée, la fonction peut être exécutée autant de fois qu'on le souhaite, en indiquant après son nom une liste de valeurs entre parenthèses, chacune de ces valeurs étant associée à un paramètre dans l'ordre spécifié lors de la déclaration de la fonction (bien sûr, il faut que le nombre de valeurs corresponde à celui des paramètres, sans quoi une erreur sera provoquée à l'exécution).

Donnons un premier exemple, très simple, de fonction. Celle-ci aura pour nom f, deux paramètres x et y, et renverra la valeur $(x+y)^2$. Sa déclaration est la suivante :

```
def f(x,y):
    return (x+y)**2
```

Notez que l'opérateur de puissance s'écrit ** en langage python.

Cette fonction peut être exécutée, par exemple par l'appel `f(1,2)` qui renverra bien sûr l'entier 9 (la valeur renvoyée `(x+y)**2` étant ici évaluée pour x contenant 1 et y contenant 2).

Sur cet exemple, le corps de la fonction est réduit à une seule instruction. Bien sûr, le traitement est souvent plus complexe et nécessite plusieurs lignes. En anticipant sur le rappel qui sera fait au chapitre suivant à propos de l'instruction de test **`if`**, voici une autre fonction renvoyant le plus grand de deux nombres :

```
def maximum(x,y):
    if x <= y :
        return y
    else :
        return x
```

Notez en particulier comment les instructions **`return`** se retrouvent deux fois indentées, une pour la fonction et l'autre pour le test.

L'appel `maximum(2,5)` renverra la valeur 5 puisque le test `x<=y`, évalué avec x contenant 2 et y contenant 5, est vrai donc c'est la valeur de y qui est renvoyée.

L'appel `maximum(7,3)` renverra la valeur 7 puisque le test `x<=y`, évalué avec x contenant 7 et y contenant 3, est faux donc c'est la valeur de x qui est renvoyée.

1.2. Signature, documentation & commentaires

Chaque paramètre d'une fonction est généralement d'un certain `type` fixé (nombre entier, nombre décimal, booléen, …), et les valeurs correspondantes lors de l'appel de la fonction devront être de ce type. De même, la valeur de retour de la fonction est généralement toujours d'un même type. Les types usuels que vous rencontrerez au fur et à mesure de l'avancement du cours sont les suivants :

- `int` : nombre entier ;
- `float` : nombre flottant (correspond à un nombre décimal en première approximation, mais nous verrons ultérieurement que c'est en fait un peu plus compliqué) ;
- `bool` : booléen (résultat d'un test : **`True`** ou **`False`** ; attention à ne pas oublier la majuscule !) ;
- `str` : chaîne de caractères (autrement dit un texte) ;
- `list` : liste (un chapitre ultérieur y sera consacré) ;
- `tuple` : n-uplet (idem) ;
- `dict` : dictionnaire (idem) ;
- **`None`** : type particulier correspondant à une absence de valeur (nous verrons dans quels cas ce type est utilisé).

Il est possible d'indiquer ces informations de type, appelées **signature** de la fonction, en complétant la première ligne de sa déclaration de la manière suivante :

```
def nom_de_la_fonction(paramètre_1 : type_1,...,paramètre_n : \
↪ type_n) ->  : type :
```

Par exemple, la fonction `maximum`, définie dans la partie précédente, peut être spécifiée en choisissant que ses deux paramètres x et y doivent être des nombres entiers, et que la valeur de retour est aussi un nombre entier (on pourrait tout aussi bien choisir que tous ces nombres sont décimaux) :

```
def maximum(x : int, y : int) -> int :
    if x <= y :
        return y
    else :
        return x
```

À noter que cette signature sert principalement de **documentation** pour le programmeur, et que l'ordinateur ne produit pas d'erreur à l'appel de la fonction si les valeurs fournies pour les paramètres ne sont pas des bons types. Par exemple, l'appel de `maximum(2.3,4.5)` fonctionnera en renvoyant le nombre décimal 4.5 même si 2.3 et 4.5 ne sont pas des nombres entiers comme la signature le réclame. Il est donc uniquement de la responsabilité du programmeur de respecter la signature d'une fonction, ce que nous nous imposerons toujours en pratique. Une exception toutefois : lorsque la signature indique qu'un paramètre doit prendre des valeurs décimales, il nous arrivera régulièrement de l'appeler pourtant avec une valeur entière, ce qui peut paraître cohérent puisque mathématiquement un entier est en particulier un nombre décimal, mais ce n'est pas le cas informatiquement et nous verrons dans un chapitre ultérieur que les valeurs entières ne sont pas stockées de la même manière que les valeurs flottantes en mémoire de l'ordinateur (par exemple, la valeur entière 2 n'est pas identique à la valeur flottante 2.0).

Dans le même ordre d'idées, il est d'usage de **commenter** une fonction en ajoutant à la suite de la première ligne un texte précisant son effet, encadré par trois caractères « " » ouvrants et fermants. Par exemple :

```
def maximum(x : int, y : int) -> int :
    """
    Renvoie la plus grande valeur entre celle de x et y.
    """
    if x <= y :
```

```
        return y
    else :
        return x
```

Ces lignes de commentaire, à destination du programmeur, ne forment bien sûr pas un code exécutable et sont tout simplement ignorées par l'ordinateur. À noter qu'elles doivent être **indentées** au même titre que les suivantes. D'autre part, la signature et ce texte de commentaire sont affichés lors de l'appel de l'instruction `help(nom_de_la_fonction)`.

Par exemple l'appel `help(maximum)` affiche à l'écran les informations suivantes :

```
>>> help(maximum)
Help on function maximum:

maximum(x: int, y: int) -> int
    Renvoie la plus grande valeur entre celle de x et y.
```

En plus de ce commentaire général sur l'effet de la fonction, il est possible d'ajouter ponctuellement des explications sur l'effet d'une ligne particulière. Cela est possible en utilisant à la fin de la ligne correspondante le caractère « # » : toute la suite de cette ligne sera alors considérée comme du commentaire (notez qu'il n'est pas nécessaire de terminer la ligne par le caractère « # »).

Par exemple, même si ce n'est guère utile ici, ajoutons quelques commentaires à la déclaration de la fonction `maximum` :

```
def maximum(x : int, y : int) -> int :
    """
    Renvoie la plus grande valeur entre celle de x et y.
    """
    if x <= y : # si y est le plus grand
        return y
    else :      # si x est le plus grand
        return x
```

1.3. Utilisation de variables locales dans une fonction

Pour calculer la valeur renvoyée par une fonction, celle-ci peut utiliser des variables qui ne seront créées par l'ordinateur que le temps de l'exécution de la fonction : on parle de **variables locales** à la fonction.

Écrivons par exemple une fonction g, d'un seul paramètre décimal x, et renvoyant la valeur $\sqrt{x} \times \cos(\sqrt{x})$.

Tout d'abord, il y a nécessité d'**importer** les fonctions racine carrée et cosinus (qui ne sont pas définies par défaut en langage Python) à partir d'un **module** appelé `math`, ce qui peut se faire ainsi (bien sûr `sqrt` pour square root désignant la racine carrée) :

```
from math import sqrt, cos
```

La déclaration de la fonction g peut alors se faire de cette manière :

```
def g(x : float) -> float :
    """
    Renvoie la valeur sqrt(x) * cos(sqrt(x))
    """
    return sqrt(x)*cos(sqrt(x))
```

Ce code est bien sûr correct, mais on constate que l'évaluation de la racine carrée du nombre x est faite deux fois, ce qui peut être jugé maladroit. La version suivante, utilisant la variable locale `rac`, sera préférable :

```
def g(x : float) -> float :
    """
    Renvoie la valeur sqrt(x) * cos(sqrt(x))
    """
    rac = sqrt(x)
    return rac*cos(rac)
```

Profitons de cette partie pour faire un bref rappel sur le fonctionnement des variables en informatique. Pour simplifier, une variable peut se concevoir comme une « boite » , désignée par un nom (obéissant aux contraintes déjà précisées pour les noms de fonctions), et dans laquelle on peut stocker une valeur (correspondant aux différents types que nous avons décrits). Les deux opérations que l'on peut effectuer sur une variable sont :

- **[stocker]** une valeur dans la variable, se fait par l'instruction `nom_de_la_variable = valeur` (si la variable contenait déjà une valeur, celle-ci est effacée et remplacée par la nouvelle) ;
- **[lire]** la valeur stockée dans la variable, se fait en écrivant tout simplement le nom de la variable, qui est remplacé par sa valeur dans l'évaluation d'une expression (il faut qu'une valeur ait déjà été stockée dans la variable sinon une erreur est produite à l'exécution).

Voici par exemple une succession d'instructions manipulant des variables :

```
x = 1
x = 5
y = 2*x
x = x+1
```

L'effet successif de ces lignes peut se simuler en notant dans un tableau le contenu des variables ligne après ligne (les deux premières lignes ne sont pas remplies pour y car cette variable n'existe pas encore à ces étapes) :

ligne	x	y
#1	1	
#2	5	
#3	5	10
#4	6	10

Notons que lors de l'exécution d'une instruction de la forme `nom_de_variable=expression`, l'ordinateur commence par évaluer l'expression, puis stocke la valeur résultat dans la variable. C'est ainsi que l'instruction `x = x+1` en ligne #4, provoque d'abord l'évaluation de $x+1$, avec x contenant la valeur 5 à l'issue de l'étape #3, et le résultat 6 est utilisé pour modifier la valeur de x. Ainsi, l'instruction `x=x+1` consiste à augmenter de 1 la valeur stockée dans x avant cette étape. Cette instruction, très classique, possède une syntaxe simplifiée consistant à écrire `x += 1`. Notons d'autre part qu'à l'étape #3 la valeur de y est initialisée comme étant le double de celle de x, mais que cette relation n'est pas maintenue en ligne #4 quand la valeur de x est modifiée. Enfin, insistons sur la différence entre l'usage du symbole « = » en informatique, correspondant à l'**affectation** (le nom de la variable est toujours à gauche et celui de la valeur est à droite, écrire `1=x` provoquerait une erreur), et son usage habituel qui correspond plutôt au **test d'égalité** qui évalue si deux valeurs sont égales, et que l'on retrouve en Python sous la syntaxe « == » celle-ci étant évaluée en les booléens `True` ou `False`. Par exemple les instructions :

```
x = 1
y = x == 0
```

ont pour effet :

ligne	x	y
#1	1	
#2	1	`False`

Pour finir, il est possible d'affecter simultanément plusieurs variables en les séparant par des virgules, ce qui a de nombreux avantages pratiques. Là encore, l'**ensemble** des valeurs à droite du signe « = » est évalué **avant** de procéder à l'affectation. Par exemple :

```
x,y = 1,2
x,y = 2*x, x+y
x,y = y,x
```

ont pour effet :

ligne	x	y
#1	1	2
#2	2	3
#3	3	2

Notons en particulier l'intérêt de la dernière commande, utile pour échanger les contenus de deux variables.

1.4. Fonctions imbriquées

Il est bien sûr possible d'appeler une fonction à l'intérieur de la déclaration d'une autre fonction.

Considérons l'exemple très simple suivant, dans lequel nous déclarons successivement deux fonctions f et g

```
def f(x : float)-> float :
    """
    Renvoie x**2.
    """
    return x**2

def g(x : float)-> float :
    """
    Renvoie (x+1)**2 * x.
    """
    return f(x+1)*x
```

Le résultat de l'appel `g(2)` par exemple renverra $(2+1)^2 \times 2 = 18$.

Ceci fonctionne car les variables correspondant au paramètre x dans les deux fonctions, pour lesquelles nous avons choisi le **même nom**, sont en fait considérées par l'ordinateur comme des variables **différentes**. Si cela **n'avait pas** été le cas, le programme aurait dû produire le résultat 27 au lieu de 18. En effet :

- Appel de `g(2)` : la variable x prend la valeur 2 et l'ordinateur évalue l'expression `f(x+1)*x`, et commence donc par évaluer `f(x+1)`, ie. `f(3)` ;
- Appel de `f(3)` : la variable x prend maintenant la valeur 3 et l'ordinateur évalue l'expression `x**2`, qui vaut 9 ;
- Cette valeur permet de terminer l'appel de `g(2)` en évaluant l'expression `9*x` qui vaut 27 puisque la variable x a changé de valeur à l'étape précédente.

Pour que le bon résultat soit produit, l'ordinateur a fait en sorte que les deux variables x de chacune des fonctions soient en fait distinctes en les **renommant** : le programme réellement exécuté par l'ordinateur est (pour simplifier, en pratique le choix des nouveaux noms est plus complexe) :

```
def f(x1 : float)-> float :
    """
    Renvoie x1**2.
    """
    return x1**2

def g(x2 : float)-> float :
    """
    Renvoie (x2+1)**2 * x2.
    """
    return f(x2+1)*x2
```

Dans ce cas, on vérifie aisément que l'appel `g(2)` fonctionne correctement, puisque la valeur de $x2$ initialisée à 2 n'est pas modifiée par l'appel de `f(3)`, qui lui travaille à l'aide de la variable $x1$.

Ce mécanisme de renommage, entièrement automatique et dont le programmeur n'a pas à se soucier, et très important en pratique puisqu'il permet de réutiliser les mêmes noms de variables dans différentes fonctions d'un même programme, et s'applique bien sûr aux **variables locales** de la même manières qu'aux paramètres de la fonction.

2. BOUCLE FOR

La commande `for` permet de répéter plusieurs fois un ensemble d'instructions, en faisant prendre à une variable une valeur différente à chaque répétition. Ces valeurs décrivent souvent une liste de nombres entiers (mais nous verrons dans un chapitre ultérieur que l'on peut décrire des listes de valeurs beaucoup plus générales). La syntaxe d'une boucle `for` est la suivante :

```
for nom_de_variable in liste_à_décrire :
    instructions
```

Notez la présence de caractère « : » à la fin de la première ligne, ainsi que l'indentation des lignes suivantes.

La liste des valeurs à décrire peut être construite à l'aide de la commande `range`, dont la syntaxe est (les valeurs de a, b, c doivent être des entiers) :

- `range(a,b)` permet de décrire les entiers de a **inclus** à b **exclu**, donc a, $a+1$, …, $b-1$.
  Par exemple `range(3,10)` décrit $3, 4, 5, 6, 7, 8, 9$.
- `range(b)` a le même effet que `range(0,b)`, et décrit donc 0, 1, …, $b-1$ (à noter qu'il y a donc b itérations).
  Par exemple `range(10)` décrit $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.
- `range(a,b,c)` ne conserve dans la liste `range(a,b)` que les termes successifs espacés de `c` à partir de `a` (le dernier terme du résultat n'est donc pas nécessairement `b-1`).
  Par exemple `range(1,10,3)` décrit $1, 4, 7$.
  Notons que l'entier `c` peut être négatif, ce qui permet de décrire une liste dans l'ordre **décroissant**.
  Par exemple `range(10,0,-1)` décrit $10, 9, 8, 7, 6, 5, 4, 3, 2, 1$.

Notons que dans certains cas la liste produite peut être **vide**, par exemple pour `range(0)`. Alors, la boucle `for` n'exécutera bien sûr **aucune** fois les instructions indentées correspondantes.

Voici par exemple, un programme affichant les puissances de deux de 2^1 à 2^9 (l'instruction `print` permet d'afficher une valeur à l'écran de l'ordinateur :

```
>>> for k in range(1,10):
...     print(2**k)
...
2
4
```

```
8
16
32
64
128
256
512
```

Il arrive que la variable sur laquelle itère la boucle for n'intervienne pas dans les instructions indentées : celles-ci sont alors répétées **à l'identique** le nombre de fois correspondant. C'est le cas dans l'exemple :

```
>>> for k in range(5):
...     print("Hello world")
...
Hello world
Hello world
Hello world
Hello world
Hello world
```

Dans ce cas particulier, on peut (mais ce n'est pas obligatoire) remplacer le nom de la variable par le caractère « _ » et écrire :

```
for _ in range(5):
    print("Hello world")
```

Terminons en illustrant l'utilisation que nous ferons de boucles **for** pour écrire des fonctions. Voici par exemple une fonction renvoyant la quantité $n \times x$, où n est un entier naturel et x un flottant, en utilisant uniquement des additions grâce à la propriété :

$$n \times x = \underbrace{x + x + \cdots + x}_{n \text{ fois}}.$$

```
def produit(n : int, x : float) -> float :
    """
    Renvoie n*x pour n entier naturel.
    """
    R = 0
    for k in range(n):
        R = R+x
    return R
```

Notons que la variable k n'intervenant pas dans l'instruction itérée, on aurait pu la remplacer par « _ » . D'autre part, l'instruction `R=R+x` aurait pu s'écrire aussi `R += x` :

```
def produit(n : int, x : float) -> float :
    """
    Renvoie n*x pour n entier naturel.
    """
    R = 0
    for _ in range(n):
        R += x
    return R
```

On peut détailler le calcul lié à un appel de cette fonction, par exemple `produit(4,1.2)`, en utilisant un tableau dans lequel la ligne i précise le contenu des variables après le i^e passage dans la boucle **for**. Notez bien que i ne correspond pas à un numéro de ligne du texte de la déclaration de la fonction, mais à un nombre de passages dans la boucle. La première ligne pour laquelle $i = 0$ précise par convention le contenu des variables **juste avant** de passer dans la boucle **for** pour la première fois (la variable de boucle k n'existe alors pas encore) :

i	k_i	R_i
0		0
1	0	1.2
2	1	2.4
3	2	3.6
4	3	4.8

La valeur contenue dans la variable R au sortir de la boucle **for** est donc 4.8, et c'est la valeur renvoyée par la fonction.

Notons que la fonction produit le bon résultat aussi pour $n = 0$, puisque dans ce cas `range(0)` est vide, et que la boucle **for** n'exécutant aucune instruction, la valeur renvoyée est celle placée initialement dans R, à savoir 0.

Exercice 1 [Sol 1] Écrire une fonction `puissance` calculant sur le même modèle x^n, et détailler dans un tableau l'appel `puissance(4, 2)`.

Exercice 2 [Sol 2] Écrire une fonction

```
somme(n : int, p : float) -> float
```

calculant la somme $\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p$. Dérouler dans un tableau l'appel `somme(4,2)`.

Exercice 3 [Sol 3] Que retourne la fonction suivante?

```
def mystere(n : int, x : float) -> float :
    """
    Mystère
    """
    S = 0
    R = 1
    for k in range(n):
        S += R
        R = R*x
    return S
```

On commencera par dérouler dans un tableau l'appel `mystere(4, 2)`.

3. TESTS CONDITIONNELS

3.1. Instruction `if-else`

L'instruction **`if`**, éventuellement accompagnée de l'instruction **`else`**, a pour syntaxe générale :

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

- `condition` est un booléen (ayant pour valeur **`True`** ou **`False`**), souvent issu d'un test,
- `bloc de commandes 1` est un ensemble de lignes de code qui sont exécutées si et seulement si `condition` a pour valeur **`True`**,
- `bloc de commandes 2` est un ensemble de lignes de code qui sont exécutées si et seulement si `condition` a pour valeur **`False`**

Exemple 1 (Fonction `positif`)

```
def positif(a:float)->bool:
    if a >= 0:
        return True
    else:
        return False
```

Cette fonction renvoie **`True`** si et seulement si la variable `a` passée en entrée est positive ou nulle, et **`False`** sinon.

Exemple 2 (Fonction `nb_div`)

```
def nb_div(n:int)->int:
    N = 0
    for k in range(2,n):
        if n%k == 0: # si k divise n
            N = N+1
    return N
```

Cette fonction renvoie le nombre de diviseurs de l'entier `n` passé en entrée (autres que 1 et `n`).

Exercice 4 [Sol 4] On donne ci-dessous le code de la fonction `mult_3(n)`, où `n` est supposé être un entier naturel.

```
def mult_3(n:int)->tuple:
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

1) Que renvoie `mult_3(4)`, `mult_3(10)`?

2) Décrire à l'aide d'une phrase ce que renvoie `mult_3(n)`.

agraphPrécisions syntaxiques

- La syntaxe du langage impose un symbole « : » après la condition et un autre après le mot **`else`**.
- Les lignes formant les deux blocs de commandes doivent être indentées (décalées vers la droite) par rapport à celles commençant par **`if`** et **`else`**. Dans les éditeurs de texte, cette indentation s'effectue automatiquement par une retour à la ligne après le symbole « : » .

- L'instruction **else** est optionnelle. En l'absence de **else**, si la condition du **if** est fausse, les instructions du `bloc de commandes 1` sont simplement ignorées et le programme poursuit son exécution.

3.2. Instruction if-elif-...-else

Considérons le problème suivant : pour un entier $0 \leq n < 10000$, on souhaite écrire une fonction `nb_chiffres(n)` qui renvoie le nombre de chiffres nécessaires pour écrire n (il faut 1 chiffre si $0 \leq n < 10$, deux chiffres si $10 \leq n < 100$ etc.). On peut écrire cette fonction à l'aide d'instructions **if** et **else** successives imbriquées :

```
def nb_chiffres(n:int)->int:
    if n < 10:
        Nb = 1
    else:
        if n < 100:
            Nb = 2
        else :
            if n < 1000:
                Nb = 3
            else:
                Nb = 4
    return Nb
```

On voit ici que l'emploi des tests imbriqués rend peu lisible le code. Une alternative est d'utiliser les instructions **if**, **elif** et **else**, selon la syntaxe générale :

```
if condition 1 :
    bloc de commandes 1
elif condition 2 :
    bloc de commandes 2
elif condition 3 :
    bloc de commandes 3
...
elif condition n :
    bloc de commandes n
else :
    bloc de commandes n+1
```

Lors de l'exécution, les conditions sont examinées successivement (dans l'ordre d 'écriture), et on exécute *uniquement* le bloc de commande relatif à la première condition rencontrée ayant pour valeur **True** (si certaines conditions placées ultérieurement ont pour valeur **True**, les blocs correspondants sont donc ignorés).

Le problème précédent peut donc se traiter également à l'aide de la fonction `nb_chiffres_bis(n)` suivante, plus lisible :

```
def nb_chiffres_bis(n:int)->int:
    if n < 10:
        Nb = 1
    elif n < 100:
        Nb = 2
    elif n < 1000:
        Nb = 3
    else:
        Nb = 4
    return Nb
```

Exercice 5 [Sol 5] On donne les deux fonctions suivantes, dont le but est de préciser si l'entier a passé en argument est divisible par 2 et/ou par 3.

```
def div(a:int)->tuple:
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

```
def div_bis(a:int)->tuple:
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

Préciser, en justifiant, ce que renvoie chacun des appels `div(6)` et `div_bis(6)`.

3.3.1. Tests élémentaires

La condition qui suit les instructions **if** ou **elif** est très souvent exprimée sous forme de test. Il peut s'agir d'un test élémentaire, dont on peut rappeler les différentes syntaxes :

math	python
$<$	<
$\leq$	<=
$>$	>
$\geq$	>=
$=$	==
$\neq$	!=

3.3.2. Tests multiples

Il est également possible d'utiliser des opérateurs logiques suivants pour écrire des tests plus complexes. La syntaxe de ces opérateurs est la suivante :

math	python
et	**and**
ou	**or**
non	**not**
ou exclusif	^

et suivent les règles de logique et de priorité usuelles (vues en cours de mathématiques).

Ainsi pour savoir si un entier n est positif et divisible par 3, on écrira `n >= 0 and n%3 == 0`.

Remarque 1 On peut signaler une particularité de l'évaluation d'un test du type `a and b`, où `a` et `b` sont des tests élémentaires : si `a` a pour valeur **False**, alors le résultat du test `a and b` est **False**, quelque soit le résultat de `b`. Il n'est donc pas nécessaire d'évaluer le test `b`, et c'est ce que fait Python. Cette propriété, parfois nommée évaluation paresseuse, peut être utile dans certains cas pour éviter des erreurs, comme nous le verrons un plus tard.

Remarque 2 Dans l'écriture des scripts Python, il est (fortement) recommandé de laisser un espace avant et après chaque opération de test (= =, ! =, etc.), et ce pour plus de lisibilité (il en est de même pour l'instruction d'affectation =). Ainsi on préfèrera écrire `unecertainevariable = = uneautrevariable` (et pour l'affectation `unecertainevariable = unecertainevaleur`) plutôt que `unecertainevariable= =uneautrevariable` (et `unecertainevariable=unecertainevaleur`).

3.3.3. Généralisation

Dans la présentation précédente

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

on a vu que `condition` est un booléen (ayant pour valeur **True** ou **False**). Il n'est pas obligatoire d'écrire `condition` sous forme de test. En réalisé, toute variable booléenne peut convenir, ainsi que toute fonction renvoyant un booléen.

Ainsi, supposons que l'on dispose d'une fonction `premier(n)` qui renvoie **True** si `n` est un nombre entier premier, et **False** sinon. Alors, la fonction suivante :

```
def somme_prem(n:int)->int:
    S = 0
    for k in range(n):
        if premier(k):
            S = S+k
    return S
```

renvoie la somme des nombres premiers strictement inférieurs à `n`.

4. BOUCLE WHILE

4.1. Présentation

L'instruction `while` permet la répétition d'un bloc de commandes *tant qu*'une certaine condition est vérifiée.

La syntaxe de l'instruction `while` est :

```
while condition :
    bloc de commandes
```

où

- `condition` est un booléen (ayant pour valeur `True` ou `False`), souvent issu d'un test,
- `bloc de commandes` est un ensemble de lignes de code.

Lors de l'exécution, `condition` est évalué, puis :

- si `condition` a pour valeur `False`, les instructions de `bloc de commandes` ne sont pas exécutées, et on passe à la suite du programme (on dit qu'on sort de la boucle `while`),
- si `condition` a pour valeur `True`, les instructions de `bloc de commandes` sont exécutées, puis **on retourne évaluer le booléen `condition`**, et ainsi de suite jusqu'à ce que la condition devienne fausse pour la première fois, auquel cas l'ordinateur sort immédiatement de l'instruction `while`.

On peut préciser quelques points importants :

- les instructions de `bloc de commandes` seront donc généralement exécutées plusieurs fois : c'est la différence fondamentale avec une commande `if` sans partie `else`,
- pour que le programme ne boucle pas indéfiniment, il faut que le bloc de commandes influe sur l'expression formant la condition pour finir par la rendre fausse (sinon on a une boucle infinie, qui est une erreur fréquemment rencontrée),
- la répétition d'instructions permet de réaliser une boucle, qui comme nous le verrons possède d'autres possibilités que celles d'une boucle `for`.

4.2. Premier exemple : calcul de somme

Considérons la fonction `somme(n)` qui permet de calculer la somme des entiers strictement inférieurs à l'entier `n` :

```
def somme(n:int)->int:
    S = 0
    for k in range(n):
        S = S+k
    return S
```

Il est possible de programmer cette fonction à l'aide d'une boucle `while` :

```
def somme(n:int)->int:
    S = 0
    k = 0
    while k < n:
        S = S+k
        k = k+1
    return S
```

Il y a deux modifications relatives à la variable `k` qui parcours les entiers qui apparaissent dans cette deuxième fonction :

- il est nécessaire d'initialiser la variable `k` (instruction `k = 0`),
- il est nécessaire d'incrémenter la variable `k` (instruction `k = k+1`). En l'absence de cette dernière instruction, on aurait une boucle infinie.

> **Remarque 3** Il est toujours possible (mais pas toujours souhaitable) de réécrire une boucle `for` à l'aide d'une boucle `while`. Dans l'exemple de la fonction `somme` précédente, il n'y a aucun intérêt à utiliser une boucle `while` plutôt qu'une boucle `for`, et l'emploi de la boucle `for` est même à privilégier, car son écriture est plus simple.

4.3. Deuxième exemple : suite de SYRACUSE

Certaines répétitions ne peuvent s'exprimer qu'avec une boucle while, et non par une boucle for. C'est dans ces cas que l'instruction `while` prend tout son intérêt. Considérons par exemple la suite de SYRACUSE définie par :

$$u_0 = p \ (p \in \mathbb{N}) \quad \text{et :} \quad \forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Une conjecture célèbre et non démontrée à ce jour est qu'une telle suite finit toujours par faire apparaître le nombre 1 (puis se poursuivre par une répétition de la séquence $4, 2, 1$) quelque soit l'entier naturel non nul p choisi pour initialiser la suite. La fonction `indice_syracuse` ci-dessous renvoie, pour p donné, le premier indice k tel que $u_k = 1$.

```python
def indice_syracuse(p:int)->int:
    u = p
    k = 0
    while u != 1:
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

On peut noter que :

- Cette fonction ne peut pas être programmée à l'aide d'une boucle `for`,
- la boucle termine pour tout entier p uniquement si la conjecture de SYRACUSE est vérifiée,
- l'emploi de la division entière `//` a pour unique rôle de maintenir le type `int` de la variable `u`.

4.4. for ou while ?

Une question que l'on doit forcément se poser lorsqu'on est amené à créer une boucle est la suivante : *doit-on utiliser une boucle* `for` *ou bien une boucle* `while` ?

Pour répondre à cette question, il faut se demander si, pour une entrée connue, mais quelconque, on peut connaitre avant d'exécuter la boucle le nombre d'itérations qu'elle comportera. Si oui, alors, il vaut mieux utiliser une boucle `for` (on peut utiliser une boucle `while`, mais c'est souvent un peu plus difficile à programmer - et à lire), sinon, on doit utiliser une boucle `while`. Par exemple, pour la suite de SYRACUSE précédente :

- Si l'on veut calculer u_{20} à partir d'un entier $u_0 = p$ quelconque, alors on sait qu'on devra faire 20 itérations (une première pour calculer u_1, une deuxième pour u_2 et ainsi de suite jusqu'à u_{20}). Il vaut mieux dans ce cas là utiliser une boucle `for`.
- Si l'on souhaite comme dans l'exemple précédent déterminer le plus petit indice k tel que u_k soit égal à 1, à partir d'un entier $u_0 = p$ quelconque, alors on ne sait pas combien d'itérations seront nécessaires.

Exercice 6 [Sol 6] On désire écrire une fonction `nb_div2(n)` qui renvoie le plus grand entier k tel que 2^k divise n. Par exemple, pour $n = 24$, la fonction devra renvoyer 3 car 24 est divisible par $2^3 = 8$, mais pas par $2^4 = 16$.

1) Faut-il utiliser une boucle `for` ou une boucle `while` pour écrire cette fonction ?

2) Écrire la fonction demandée.

SOLUTIONS DES EXERCICES

Solution 1

```
def puissance(n : int, x : float) -> float :
    """
    Renvoie x**n pour n entier naturel.
    """
    R = 1
    for k in range(n):
        R = R*x
    return R
```

i	k_i	R_i
0		1
1	0	2
2	1	4
3	2	8
4	3	16

Solution 2

```
def somme(n : int, p : float) -> float :
    """
    Renvoie 1**p + 2**p + ... + n**p pour n>0
    et 0 sinon.
    """
    S = 0
    for k in range(1,n+1):
        S += k**p
    return S
```

i	k_i	S_i
0		0
1	1	1
2	2	5
3	3	14
4	4	30

On peut aussi éviter l'utilisation du symbole ** si on veut.

```
def somme(n : int, p : float) -> float :
    """
    Renvoie 1**p + 2**p + ... + n**p pour n>0
    et 0 sinon.
    """
    S = 0
    for k in range(1,n+1):
        S += puissance(p, k)
    return S
```

On peut contrôler ensuite dans la console.

```
>>> somme(4, 2)
30
```

Solution 3

i	k_i	S_i	R_i
0		0	1
1	1	1	2
2	2	3	4
3	3	7	8
4	4	15	16

On peut contrôler ensuite dans la console.

```
>>> mystere(4, 2)
15
```

La fonction retourne la somme géométrique $\sum_{k=0}^{n-1} x^k$.

Solution 4

1) `mult_3(4)` renvoie `(1,3)` et `mult_3(10)` renvoie `(3,18)`.

2) Plus généralement, `mult_3(n)` renvoie le tuple `(N,S)` où `N` est le nombre d'entiers k multiples de 3 vérifiant $1 \leq k < n$ et `S` est la somme de ces mêmes entiers.

Solution 5 Pour la fonction `div`, la condition relative au premier **`if`** est vrai, donc on affecte 2 à `d1`. On examine ensuite le deuxième **`if`**, dont la condition est aussi vraie, et on affecte 3 à `d2`. On a donc pour retour `(2,3)`. Pour la fonction `div_bis`, la condition relative au premier **`if`** est vrai, donc on affecte 2 à `d1`, mais on n'exécute pas le bloc relatif au **`elif`**. la variable `d2` reste inchangée et la fonction renvoie `(2,None)`.

Solution 6

1) Pour réaliser la fonction, il faut effectuer des divisions par 2 successives. Pour n quelconque, on ne connait pas le nombre de divisions nécessaires avant d'avoir exécuté l'algorithme. Il faut donc utiliser une boucle **`while`**.

2) Fonction `nb_div2(n)`

```
def nb_div2(n:int)->int:
    k = 0
    while n%2 == 0:
        n = n//2
        k = k+1
    return k
```