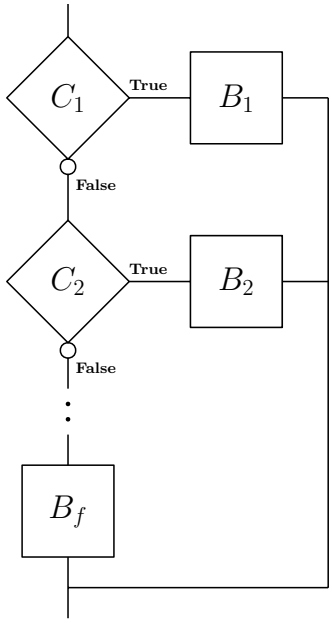
Tests et boucles

I Les tests

I.1 Structure général des tests

La structure générale d'un test en Python est donnée par :

```
if Condition1:
    instruction Bloc1
    instruction Bloc1
    ...
elif Condition2:
    instruction Bloc2
    instruction Bloc2
    ...
elif ...
    ....
else:
   instruction Bloc final
   instruction Bloc final
   ....
```



Seul le premier mot-clé `if` est obligatoire, les `elif` et le `else` sont optionnels. Noter que `else` n'est pas suivi d'une condition. Les conditions sont des variables de type booléen ou des expressions qui sont évaluées sous forme de booléen.

Un seul des blocs d'instructions peut être exécuté : le premier possible. Ceci a lieu y compris si l'état courant rend plusieurs des conditions valides.

I.2 Les différents type de tests

De manière pratique, on rencontre plusieurs cas :

Le **test simple** (avec `if` seul) permet d'effectuer une action lorsqu'une condition et vérifiée et rien sinon. C'est typiquement la situation du comptage.

```
def nbValeur(L, a):
    """ Compte le nombre d'occurrences de la valeur a dans L """

    compteur = 0
    for x in L:
        if x == a:
            compteur += 1
    return compteur
```

1

Le **test avec alternative** (avec `if` et `else`) permet d'effectuer une action lorsqu'une condition et vérifiée et une autre action dans le cas contraire.

La suite de Syracuse qui est définie par la transformation qui associe à un entier n l'entier $n/2$ si n est pair et $3n+1$ sinon fournit un bon exemple..

```
if n%2 == 0:
    p = n//2
else:
    p = 3*n + 1
```
2

Les **tests avec plusieurs alternatives** (avec `if`, `elif` et `else`) permettent d'effectuer plus d'un test.

Imaginons que l'on veuille écrire une fonction `estBissextile(n)` qui renvoie `True` si `n` est le numéro d'une année bissextile et `False` sinon. On rappelle que les années bissextiles sont celles qui sont multiples de 4 sauf si elles sont multiples de 100 mais pas de 400.

```
def estBissextile(n):

    if n % 4 != 0:
        return False
    elif n % 100 != 0:
        return True
    elif n % 400 != 0:
        return False
    else:
        return True
```
3

Exercice 1

Écrire la même fonction en utilisant uniquement les opérateurs logiques : `and`, `or` et `not()` (sans branchements conditionnels).

Les **tests avec de nombreuses alternatives** s'écrivent généralement avec des boucles et non pas avec des quantités de `elif`. Imaginons une fonction qui renvoie un tarif postal en fonction du poids d'une lettre :

```
def prixMasse(x):
    """ Renvoie le tarif pour un courier de x gr (x <= 2000) """

    tarif = [(20, 1), (50, 2), (100, 3), (250, 5), (500, 8),
    (1000, 14), (2000, 18)]

    for (m, p) in tarif:
        if x <= m:
            return p
```
4

Entraînement 1

Écrire une fonction qui renvoie le maximum de trois nombres (sans la fonction `max`).

Exercice 2

On exécute chacun des deux codes dans un état où les variables `a, b` et `c` ont respectivement pour valeur 5, 6 et 1. Quel état en résulte ?

```
1 a = a + c
2 if a == b:
3     b = c
4 else:
5     c = b
6 a = b
```

5

```
1 a = a + c
2 if a == b:
3     b = c
4 else:
5     c = b
6     a = b
```

6

I.3 Comparaisons et opérateurs booléens

Les conditions d'un test sont toujours évaluées sous la forme d'un booléen (on rappelle que le type booléen ne possède que deux valeurs : `True` et `False`). On rappelle les opérateurs de comparaison :

Opérateur	test effectué
==	égalité
!=	différent
<=	inférieur ou égal
>=	supérieur ou égal
<	inférieur strict
>	supérieur strict

Les **opérateurs booléens** permettent de combiner les conditions, il y en a trois, du plus prioritaire au moins prioritaire : `not`, `and` et `or`. Les opérations effectuées s'appellent respectivement *négation*, *conjonction* et *disjonction*.

négation

C	not C
True	False
False	True

conjonction

C1	C2	C1 and C2
True	True	True
True	False	False
False	True	False
False	False	False

disjonction

C1	C2	C1 or C2
True	True	True
True	False	True
False	True	True
False	False	False

Exercice 3

Les opérateurs `and` et `or` sont paresseux en Python. C'est à dire que dans `C1 and C2` et `C1 or C2` après l'évaluation de la première condition la deuxième n'est pas évaluée si cela ne sert à rien pour déterminer le résultat.

Pour chacun des deux opérateurs, indiquez précisément dans quelle situation la condition C2 n'est pas évaluée.

II Boucles inconditionnelles

Quand on veut répéter un certain nombre de fois une action et que le nombre de répétitions est connu à l'avance on utilise une boucle inconditionnelle.

```
>>> for k in range(4):
        print(k)

0
1
2
3
```
7

Ici la fonction range renvoie un *itérateur* qui produit consécutivement les valeurs entières de 0 à 3. La boucle **for** ne fait que parcourir les valeurs de cet itérateur.

Plus généralement, la boucle **for** permet de parcourir tout objet *itérable*. Les chaines de caractères, les listes et les tuples sont itérables.

```
>>> for c in 'Python':
        print(c)
P
y
t
h
o
n
```
8

```
>>> for x in [2, 0, 1, 6]:
        print(x**2)


4
0
1
36
```
9

```
>>> semaine = ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi',
'samedi', 'dimanche')
>>> for jour in semaine:
        print(jour, '- ', end = '')


lundi - mardi - mercredi - jeudi - vendredi - samedi - dimanche -
```
10

La syntaxe générale d'une boucle inconditionnelle **for** est :

```
for element in iterable:
    instruction     #
    instruction     # bloc d'instructions
    instruction     #
```
11

Lors de l'exécution de cette instruction **for**, le bloc d'instructions est exécuté pour les différentes valeurs que l'objet `iterable` fournit à la variable `element`.

La fonction **range** possède trois arguments dont deux sont optionnels :

— **range**(n) retourne un itérateur parcourant les entiers consécutifs entre 0 et n exclu.

— **range**(m, n) retourne un itérateur parcourant les entiers consécutifs entre m compris et n exclu.

— **range**(m, n, s) retourne un itérateur parcourant les entiers consécutifs entre m compris et n exclu avec un pas de s.

La fonction **list**() transforme l'itérateur en liste [1].

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(4, 12))
[4, 5, 6, 7, 8, 9, 10, 11]
>>> list(range(4, 16, 3))
[4, 7, 10, 13]
>>> list(range(5, -1, -1))
[5, 4, 3, 2, 1, 0]
```

12

On peut imbriquer deux boucles, par exemple pour obtenir les tables de multiplications :

```
>>> for i in range(1, 11):
        for j in range(1, 11):
            print(i*j, end=' ')
        print()


1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
.
.
10 20 30 40 50 60 70 80 90 100
```

13

Entraînement 2

En 2020, le mois de juin commence un lundi. À l'aide de deux boucles imbriquées, faire afficher les jours des quatre premières semaines du mois sous la forme :

```
lundi 1 juin
mardi 2 juin
.
.
```

Exercice 4 Écrire une fonction qui calcule la somme $\sum_{i=1}^{n}\sum_{j=1}^{n} ij$.

Entraînement 3

Écrire une fonction qui calcule la somme $\sum_{i=0}^{n}\left(\prod_{j=1}^{p}(i+j)\right)$.

1. La fonction **tuple**() le transformerait en tuple.

L'objet itérable produit ses valeurs les unes après les autres, modifier la valeur de la variable qui contient l'élément courant n'y change rien.

```
>>> for k in range(3):
        print(k)
        k += 2
        print(k)
0
2
1
3
2
4
```

14

Il est par contre possible de sortir d'une boucle **for** avant la fin de la boucle à l'aide de l'instruction **break**. Pour calculer la somme des termes d'une liste `L` tant qu'ils sont positifs on écrira par exemple :

```
S = 0
for x in L:
    if x < 0:
        break
    S += x
```

15

L'utilisation de **break** est acceptée tant qu'elle simplifie la lecture du code. Sachant qu'elle peut souvent être avantageusement remplacée par une boucle conditionnelle, on ne devrait pas la voir souvent...

Exercice 5

Écrire une fonction `sommePositifs(L)` qui réalise cette opération mais sans **break**.

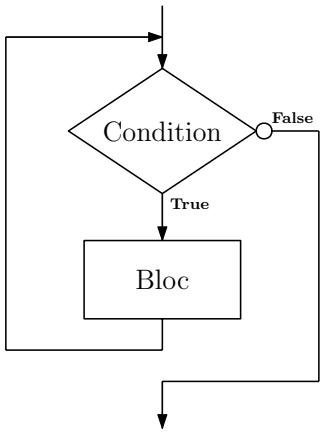
Bilan

III Boucle conditionnelle

Dans une boucle **for**, le nombre d'itérations est connu à l'avance. Lorsque l'on ne sait pas à l'avance combien de fois la boucle devra être exécutée, on utilise une boucle **while**.

La syntaxe est la suivante :

```
while condition:
    instruction du bloc
    instruction du bloc
    instruction du bloc
```



La condition est une expression qui doit pouvoir être évaluée sous forme de booléen. Tant que sa valeur est True, le bloc d'instructions est exécuté.

Attention, lors de l'utilisation de **while**, il faudra bien prendre gare à ce que la condition finisse par prendre la valeur False sans quoi la boucle ne se terminera pas !!

Remarque : Pour faire une opération jusqu'à obtenir une certaine condition il suffit d'ajouter l'opérateur logique **not**().

Un exemple typique est celui de la division euclidienne par soustractions successives. Ici `n` est un entier positif et $\texttt{d} > 0$.

```
def division(n, d):

    q = 0
    while n >= d:
        n -= d
        q += 1
    return (q, n)
```

16

Entraînement 4

Écrire une fonction `valuation2(n)` qui calcule la valuation 2-adique d'un entier n.

On rappelle que la valuation 2-adique d'un entier n est l'exposant de 2 dans la décomposition en facteur premier de n.

Exercice 6

Montrer qu'une boucle **for** k **in range**(a, b, p) peut être obtenue à l'aide d'un **while**.

Exercice 7

Écrire la fonction sommePositifs(L) vu précédemment à l'aide d'une boucle **while**.

Exercice 8

Écrire une fonction `plusPetitDiviseur(n)` qui renvoie le plus petit diviseur premier de n (ici $n \geqslant 2$).

Entraînement 5 

La conjecture de Syracuse énonce que, pour toute suite (u_n) définie par son premier élément $u_0 \in \mathbb{N}^*$ et la relation de récurrence

$$u_{n+1} = \begin{cases} u_n/2 & \text{si} \quad u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon,} \end{cases}$$

il existe un indice n tel que $u_n = 1$.

Écrire une fonction `tempsVol(u)` qui prend en paramètre un entier positif u qui est le premier terme de la suite et renvoie le premier indice n tel que $u_n = 1$.