BOUCLE FOR

1 BOUCLE FOR ET COMMANDE RANGE

A Un premier exemple

La boucle **for** est comme une boucle `while` mais dans laquelle l'incrément de boucle est intégré : on n'a pas à s'en occuper. Python se charge lui-même d'initialiser et d'incrémenter le variant de boucle. En particulier, on n'aura pas de boucles infinies avec **for**.

Si on veut écrire les nombres de 1 à 5, on peut programmer la boucle **while** :

```
i=1
while i <= 5:
  print(i)
  i += 1
print('fini')
```

Python 1: Syntaxe avec une boucle while

Pour le traduire avec une boucle **for** :

```
for i in range(1,6):
    print(i)
print('fini')
```

Python 2: Syntaxe avec une boucle for

range(n,p) est l'écriture en Python de l'intervalle $[\![n,p[\![= [\![n,p-1]\!]$.
$\longrightarrow$ la première borne n est comprise (incluse), mais pas la deuxième (exclue).

Ainsi on peut traduire en français la première ligne :

`for i in range(1,6)` **pour** i variant dans l'intervalle $[\![1,6[\![$.

Une telle écriture sous-entend que i commence par la valeur 1 et termine à $6-1=5$. À chaque boucle, la valeur de i est incrémentée de 1.

B Compléments sur la commande range

Commencer à 0 :

Lorsque l'on commence à compter à partir de 0, ce n'est pas la peine de le préciser et **range** n'a alors qu'un seul argument : la deuxième borne (qui est exclue)

```
#compte de 0 à 10
for i in range(11):      # idem que : for i in range(0,11):
    print(i)
print('fini')
```

Python 3: range à un seul argment

Remarque : on comprend pourquoi la deuxième borne est exclue. Ainsi, dans `range(11)`, le compteur va de 0 à 10, la boucle est parcourue 11 fois.
Il arrive que la valeur exacte du compteur n'ai pas d'importance sur le résultat, mais que seul compte le nombre de fois que la boucle est effectuée. Dans ce cas, il suffit d'écrire `range(n)` pour que la boucle soit effectuée n fois.

Exemple

Programmer une suite de commande qui écrit 1000 fois à l'écran : "J'aime Python".

Solution :

```
for i in range(1000):
    print("J'aime Python")
```

Exemple (Corriger l'erreur)

Pour calculer $n!$ on propose le programme suivant :

```
# trouver l'erreur :
def factoriel(n):
    fact = 1
    for i in range(n+1):
        fact *= i
    return fact
```

Python 4: Calcul de la factorielle n, énoncé faux

Quelle est l'erreur ? Corriger.
Solution :
i varie entre 0 et n et à la première boucle, on multiplie `fact` par 0. `fact` vaudra alors 0, et sa valeur ne changera plus jusqu'à la fin de la boucle (0 multpliplié par n'importe quel nombre donne toujours 0).
Il faut donc commencer par `i=1`, voire même `i=2` pour ne pas perdre de temps avec une boucle inutile.

```
# version corrigée
def factoriel(n):
    fact = 1
    for i in range(2,n+1):
        fact *= i
  return fact
```

Python 5: Calcul de la factorielle n, corrigé

Avancer par sauts :

Il est possible de compter avec un pas de k, en spécifiant la valeur de k en troisième argument.

```
# compte de 0 à 10 de 2 en 2
for i in range(0,11,2):
    print(i)
print('fini')
```

Python 6: range avec un pas fixe

Résumé sur la commande range :

Syntaxe	Signification
`range(`n`)`	entiers de 0 à $n-1$
`range(`p`,`n`)`	entiers de p à $n-1$ (1)
`range(`p`,`n`,`k`)`	entiers de p à $n-1$, avec un pas de k (2)

Notes :

- (1) Si $p \geq n$, alors `range` est vide (mais ne renvoie pas d'erreurs).
  p et n peuvent être négatifs.
- (2) k peut être négatif, pour un parcours décroissant.

Remarques pour les pros :
La variable d'itération est une variable globale : lorsque l'on sort de la boucle, elle garde la dernière valeur prise.

```
for i in range(11):
    pass                    #instruction vide : ne fait rien
print('Hors de la boucle, i vaut ',i)
```

Python 7: Variable d'itération hors de la boucle

Nota : le bloc d'instruction doit contenir au moins une instruction. Ici, comme on ne veut rien faire, on lui donne l'instruction vide (ou instruction nulle) avec la commande Python `pass` (ne rien faire).

C Quel type de boucle choisir ?

Vaut-il mieux programmer avec une boucle **for** ou une boucle **while** ? Ça dépend.

De manière générale, si on sait programmer avec une boucle **for**, alors il est facile d'écrire avec une boucle **while**, le contraire est faux.

La boucle **while** est plus souple et plus générale, mais aussi plus lourde à programmer.

On cherchera toujours à privilégier la boucle **for**.

Plus simplement :

- J'utilise une boucle **for** lorsque je sais à l'avance combien de fois devra tourner la boucle.
  (en général cela correspond au nombre de calculs je dois réaliser pour arriver au résultat.)
- Dans les autres cas, j'utilise une boucle **while**.

boucle while	**BOUCLE FOR**
Avantages : Très souple et très général → on peut avoir une condition d'arrêt très compliquée qui dépend du calcul réalisé dans la boucle, → on n'est pas obligé de savoir à l'avance combien de fois sera réalisée la boucle.	**Avantages : simple et robuste** → pas de risque de boucle infinie, → programmation souvent plus succincte et *plus élégante*, → davantage dans *l'esprit Python*.
Inconvénients → gérer le compteur à la main, → risque de boucles infinies, → souvent un peu *lourd* à programmer.	**Inconvénients** → moins général.
⚠ il faut **initialiser** et **incrémenter** le compteur	**for** le fait tout seul.

Exemple

- Pour calculer le $100^{\text{ème}}$ terme d'une suite définie par récurrence, j'utilise une boucle **for**.
  En effet, je sais à l'avance que je devrai faire tourner ma boucle exactement 100 fois.
- Pour calculer le premier terme d'une suite qui est plus grand que 100, j'utilise une boucle **while**.
  En effet, je ne sais pas à l'avance à quel terme cela correspond : u_5, u_{100}, u_{845} ? et je ne sais donc pas à l'avance combien de calculs je dois réaliser.

Avertissement : la suite de ce chapitre fait intervenir les listes et les chaînes de caractères.
Si vous n'avez pas encore lu les sections correspondantes dans le chapitre sur les types de données... alors c'est le moment.

2 OBJETS ITÉRABLES

range est ce que l'on appelle un objet *itérable*. Cela veut dire qu'on peut le parcourir et c'est justement ce que fait la commande **for** : elle parcours l'intervalle défini pas la commande **range**.

Ceci peut être généraliser à tout autre objet que l'on peut parcourir de la même façon : on parle d'objet *itérable*.

La commande for parcourt un objet itérable.

Il existe d'autres objets itérables que l'on peut parcourir avec une boucle **for**. En particulier les listes et les chaînes de caractères.

```
def parcoursObjet(objet):
    for i in objet:
        print(i)

# utilisation
texte = 'abcdef'
parcoursObjet(texte)  # écrit les lettres les unes à la suite des autres

liste = [1,2,3,4]
parcoursObjet(liste)  # écrit les éléments de la liste les uns à la suite des autres
```

Python 8: Parcours d'objets iterables

Ici, le compteur i prend directement les valeurs successives de l'objet (soit les lettres, soit les éléments de la liste).

On peut aussi faire les choses *à la main* avec la commande **range**, mais c'est beaucoup moins élégant et à éviter si on peut faire autrement.

```
# Parcours avec range
def parcoursObjet(objet):
    lg = len(objet)
    for i in range(lg):
        print(objet[i])
```

Python 9: Parcours d'objets iterables, alternative avec range

Parcourir l'objet ou ses indices ?

On cherchera toujours à privilégier la boucle **for** sur l'objet.

itération sur les indices	ITÉRATION SUR L'OBJET
Avantages : Très général → Contient davantage d'informations : → on n'est pas obligé de savoir à l'avance combien de fois sera réalisée la boucle.	**Avantages : simple et robuste** → pas de risque d'erreur sur les indices, → plus simple à programmer, → plus succinct et plus élégant, → davantage dans *l'esprit Python*.
Inconvénients → risque de se tromper sur le `range`, oublier des indices ou sortir de la liste. → syntaxe plus lourde.	**Inconvénients** → on ne sait pas où on est dans la liste : l'indice n'est pas accessible.
⚠ si on utilise **range**, on risque toujours de se tromper dans les indices (d'aller trop loin, pas assez...).	

3 LISTES EN COMPRÉHENSION

Pour construire des listes, on est souvent amené à utiliser des boucles, ce qui amène à des syntaxes un peu lourdes. La compréhension de liste est une manière très compacte de construire de telles listes. Elle consiste à écrire la boucle dans la liste elle-même [1].

[1] C'est une notation : in fine, la boucle est exécutée et la liste ne contient pas cette boucle à proprement parler.

Exemple (Liste de carrés)

On cherche à faire la liste des carrés de x pour x variant de 1 à 10.

```
# méthode basique
carres = []
for x in range(1,11):
    carres.append(x**2)
print(carres)

#avec compréhension de liste
[x**2 for x in range(1,11)]
```

Python 10: Listes en compréhenion

L'écriture en compréhension correspond exactement à ce que l'on dit en français " les carrés de x pour x variant de 1 à 10".

Exercice

À partir d'une liste de nombres `liste`, construire la liste des mêmes éléments incrémentés de 1.

Remarques pour les pros :
Lorsque la variable d'incrémentation n'est pas utilisée pour construire le résultat, on peut la remplacer par un caractère souligné "_".
Par exemple `[1 for _ in range(10)]`.

4 COMPLÉMENT : RUPTURES DE SÉQUENCES ET DE BOUCLES

L'existence de ces possibilités est intéressante à connaître, mais à utiliser avec la plus grande réserve. Leur utilisation traduit souvent une programmation maladroite.

Ce passage peut être sauté sans préjudice majeur.

A Commande break - else

La commande **break** permet de sortir d'une boucle avant la fin.
La commande **else** au niveau d'une boucle **for** permet de donner un code à exécuter après la boucle, sauf si on est sorti par un **break**.

Voici un exemple tiré du tutoriel Python sur le site officiel :

```
for n in range(2, 10):
  for x in range(2, n):
    if n % x == 0:
      print(n, 'equals', x, '*', n/x)
      break
  else:
    # loop fell through without finding a factor
    print(n, 'is a prime number')
```

Python 11: Commandes break-else - exemple tutorial Python

Cette boucle recherche les nombres premiers compris entre 2 et 9.
Pour chaque nombre n compris entre 2 et 9, elle teste tous les diviseurs possibles $x \in [\![2, n-1]\!]$.
Si x divise n alors le programme donne le produit en question et sort de la boucle **for** du x.
Sinon, à la fin de la boucle, Python effectue le **else** (qui est au niveau du **for** et pas du **if**).

Résumé : le **else** est exécuté en fin de boucle **for**, sauf dans le cas où une commande **break** fait sortir *artificiellement* de la boucle.

Remarque : dans le cas d'une boucle à l'intérieur d'une fonction, la commande **return** permet également d'arrêter le parcours de la boucle pour renvoyer directement la valeur. Contrairement à **break**, cette méthode *spécifique à Python* permet parfois des formulations élégantes. Il nous arrivera de la proposer.

B Commande continue

La commande **continue** permet d'ignorer la suite des instructions dans la boucle et de passer directement à l'incrément suivant.
Voici à nouveau un exemple issu du tutoriel.

```
for num in range(2, 10):
  if num % 2 == 0:
    print("Found an even number", num)
    continue
  print("Found a number", num)
```

Python 12: Commande continue - exemple issu du tutoriel Python