Constructions élémentaires

Cours

En Python, l'**indentation**, ou décalage vers la droite du début de ligne, délimite les séquences d'instructions et facilite la lisibilité en permettant d'identifier des **blocs**. La ligne précédant une indentation se termine toujours par le signe deux-points.

PEP 8

Préférer les espaces aux tabulations.

L'indentation est normalement réalisée par quatre caractères « espace ». Python accèpte aussi une tabulation, voire un seul ou un autre nombre d'espaces, mais dans tous les cas il faut être consistant à travers tout un programme.

Instructions conditionnelles

Cours

Une instruction conditionnelle exécute, ou pas, une séquence d'instructions suivant la valeur d'une condition (une expression booléenne qui prend la valeur `True` ou `False`).

```
if condition:
    instructions
```

Par exemple, ce programme détermine le stade de la vie d'une personne selon son age.

PEP 8

Pas d'espace avant le deux-points (`:`).

```
1  age = int(input("Quel age avez-vous ?"))
2  if age >= 18:
3      stade = "adulte"
4      print(f"Vous êtes un {stade}")
5      print(f"Vous avez {age} ans")
```

⚠️ Ne pas oublier les deux-points « `:` » après la condition et l'indentation sur les lignes suivantes.

L'indentation détermine la séquence à exécuter (ou pas), dans ce cas les instructions qui suivent aux lignes 3, 4 et 5. Quand la condition (l'expression booléenne) `age >= 18` n'est pas vérifiée, aucune des trois instructions n'est

exécutée.

Il est possible de ne pas indenter l'instruction en ligne 5 `print(f"Vous avez {age} ans")`, dans ce cas elle ne ferait plus partie de l'instruction conditionnelle et serait alors exécutée dans tous les cas.

```
1  age = int(input("Quel age avez-vous ?"))
2  if age >= 18:
3      stade = "adulte"
4      print(f"Vous êtes {stade}")
5  print(f"Vous avez {age} ans")
```

Par contre, si l'instruction en ligne 4 `print(f"Vous êtes {stade}")` n'était pas indentée, la variable `stade` ne serait pas définie quand la condition n'est pas vérifiée et dans ce cas il y aura un message erreur à la ligne 4.

La structure `if-else` permet de gérer le cas où la condition est fausse :

Cours

```
if condition:
    instructions
else :
    instructions_sinon
```

> L'instruction `else` n'a pas de condition, elle est toujours suivie des deux-points « `:` » .

```
1  age = int(input("Quel age avez-vous ?"))
2  if age >= 18:
3      stade = "adulte"
4  else:
5      stade = "enfant"
6  print(f"Vous êtes {stade}")
7  print(f"Vous avez {age} ans")
```

Dans ce cas, la variable `stade` est toujours définie, et l'instruction en ligne 6 `print(f"Vous êtes {stade}")` peut ne pas être indentée.

La structure `if-elif-else` permet de remplacer des instructions conditionnelles imbriquées pour gérer plusieurs cas distincts :

Cours

```
if condition_1:
    instructions_si_1
elif condition_2:
    instructions_si_2
elif condition_3:
    instructions_si_3
…
else :
    instructions_sinon
```

Ces deux programmes font exactement la même chose, mais le second est plsu lisible :

Instructions conditionnelles imbriquées

```
1  if age >= 18:
2      stade = "adulte"    # au dessus de 18
3  else:
4      if age >= 12:
5          stade = "ado"    # entre 12 et 18
6      else:
7          if age >= 2:
8              stade = "enfant"    # entre 2 et 12
9          else :
10             stade = "bébé"    # moins de 2
11
12 print(f"Vous êtes {stade}, vous avez {age} ans")
```

Chaque fois qu'une condition `if` n'est pas vérifiée, le programme exécute toute la séquence `else` correspondante. Il "descend" ainsi de suite dans les conditions imbriquées jusqu'à ce qu'une condition soit vérifiée, et à ce stade il sort de tous les blocs conditionnels et reprend à l'instruction qui suit tous les blocs conditionnels imbriqués, ici à ligne 12 `print(f"Vous êtes {stade}, vous avez {age} ans")`.

Par exemple, si on affecte la valeur `10` à la variable `age`, la première condition en ligne 1 `if age >= 18:` est fausse, le programme exécute donc toute la partie indentée après le premier `else:` correspondant en ligne 3. Il passe à la seconde condition en ligne 4 `if age >= 12:` qui est encore fausse, il "passe" donc à la séquence indentée après le `else:` correspondant en ligne 6. La troisième condition en ligne 7 `if age >= 2:` est cette fois vraie, il exécute la ligne 8 `stade = "enfant"` et sort de toutes les instructions conditionnelles pour reprendre à la ligne 12 et afficher le message `Vous êtes enfant, vous avez 10 ans`.

Noter qu'il faut bien prendre soin à l'indentation et que ce programme n'est pas très lisible !

Instructions conditionnelles en utilisant la structure `if-elif-else`

```
1  if age >= 18:
2      stade = "adulte"    # au dessus de 18
3  elif age >= 12:
4      stade = "ado"    # entre 12 et 18
5  elif age >= 2:
6      stade = "enfant"    # entre 2 et 12
7  else :
8      stade = "bébé"    # moins de 2
9
10 print(f"Vous êtes {stade}, vous avez {age} ans")
```

Dès que la première conditions `if` ou une condition `elif` est vérifiée, le programme ne teste plus les conditions `elif` suivantes ni le `else` final, mais reprend directement à l'instruction qui suit le bloc conditionnel, ici `print(f"Vous êtes {stade}, vous avez {age} ans")`.

Par exemple, si on affecte la valeur `15` à la variable `age`, la première condition en ligne 1 `if age >= 18:` est fausse, le programme passe donc à la conditions suivante `elif age >= 12:…`, celle-ci est vérifiée, il exécute donc la ligne 4 `stade = "ado"` et n'a pas besoin de vérifier la condition suivante en ligne 5 `elif age >=2 :…` ni d'exécuter le `else:…`. Dès que la condition `elif age >= 12:…` a été vérifiée, il sort de toute l'instruction conditionnelle pour reprendre à la ligne 10 et afficher le message `Vous êtes ado, vous avez 15 ans`.

Rappel

Éviter les conditions d'égalité avec les nombres de type float. Par exemple :

```
if 2.3 - 0.3  == 2:
    print('Python sait bien calculer avec les float')
else:
    print('Python ne sait pas bien calculer avec les float')
```

PEP 8

Éviter de comparer des variables booléennes à `True` ou `False` avec `==` ou avec `is`.

On écrit :

```
cond = True
if cond:
    print("la condition est vraie"
```

mais pas `if cond == True:` ni `if cond is True:`

Exercice corrigé

Écrire un programme qui demande une année et affiche si elle est bissextile ou pas :

1. en utilisant des conditions imbriquées.
2. en utilisant une structure `if-elif-else`.

« Depuis l'ajustement du calendrier grégorien, l'année sera bissextile (elle aura 366 jours) seulement si elle respecte l'un des deux critères suivants :

1. C1 : l'année est divisible par 4 sans être divisible par 100 (cas des années qui ne sont pas des multiples de 100) ;
2. C2 : l'année est divisible par 400 (cas des années multiples de 100).

Si une année ne respecte ni le critère C1 ni le critère C2, l'année n'est pas bissextile ».

Source: https://fr.wikipedia.org/wiki/Année_bissextile.

Réponse 1.en utilisant des conditions imbriquées

Réponse 2.en utilisant une structure `if-elif-else`

Note

il est bien sûr aussi possible d'utiliser l'expression trouvée dans l'exercice corrigé précédent :

```
annee = int(input('annee: ') )
if (annee % 4 == 0 and not annee % 100 == 0) or annee % 400 == 0:
    print(annee, 'est bissextile')
else:
    print(annee, "n'est pas bissextile")
```

mais ce n'est pas très lisible.

Boucles non bornées

Cours

Une boucle non bornée (ou boucle conditionnelle) permet de répéter une instruction ou une séquence d'instructions, tant qu'une condition (une expression booléenne) est vraie.

```
while condition:
    instructions
```

La structure est similaire à l'instruction conditionnelle. La condition qui suit le mot `while` est une expression de type booléen qui prend la valeur `True` ou `False`. Le bloc d'instructions qui suit est exécuté tant que la condition est vraie. Ce bloc d'instruction doit impérativement modifier la valeur de la condition afin qu'elle finisse par ne plus être vérifiée, sinon la boucle est sans fin et le programme ne se terminera jamais !

Exemple :

Rappel

`i += 2` est une abréviation de `i = i + 2`

```
>>> i = 0
>>> while i <= 10:
...     print(i)
...     i += 2
...
0
2
4
6
8
10
```

Il faut toujours impérativement **vérifier que la condition ne sera plus vérifiée** après un nombre fini de passage, sinon le programme ne s'arrête jamais, le **programme boucle** ou diverge. Ici, c'est bien le cas grâce à l'instruction `i += 2`, `i` finira bien par être plus grand que 10.

Exemple de programme qui boucle (erreur d'indentation dans l'instruction `i += 2`):

```
i = 0
while i <= 10:
    print(i)
i += 2
```

⚠️ Comme pour les instructions conditionnelles, il faut faire particulièrement attention aux boucles avec les nombres de type `float`. La boucle suivante qui semble écrite correctement ne finira jamais :

```
i = 1
while i != 2:
    i += 0.1
    print(i)
```

Testons la même boucle écrite correctement :

```
i = 1
while i < 2:
    i += 0.1
    print(i)
```

Elle affiche dans la console :

```
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

`i` ne prend donc jamais la valeur `2`.

Boucles bornées

Cours

Une boucle bornée (ou boucle non conditionnelle) permet de répéter `n` fois, `n` étant un nombre entier connu, une instruction ou une séquence d'instructions.

```
for i in range(n):
    instructions
```

`i` est appelé l'**indice de boucle ou compteur de boucle**, il prend les `n` valeurs entières comprises **entre 0 et n - 1**.

⚠️ **`i` ne prend pas la valeur n**.

Il est aussi possible d'utiliser :

- `range(d, f)` qui énumère les `f-d` nombres entiers compris entre `d` et `f-1`.[1]
- `range(d, f, p)` qui énumère les nombres entiers compris entre `d` et `f-1` avec un pas de `p` : `d`, `d+p`, `d+2p`, etc. [2]

La boucle bornée ci-dessus est très similaire à la boucle non bornée suivante :

```
i = 0
while i < n :
    instructions
    i += 1
```

⚠️mais attention, comparons ces deux programmes :

Programme 1

```
for i in range(5):
    print(i, end=" ")
```

Programme 2

```
i = 0
while i < 5:
    print(i, end=" "))
    i = i + 1
```

Les deux programmes semblent afficher la même chose, tous les chiffres de 0 à 4 : `0 1 2 3 4`

Pourtant ils sont différents. Ajoutons une instruction `print(i)` à la fin les deux programmes.

Programme 1

```
for i in range(5):
    print(i, end=" ")
print(i)
```

affiche : `0 1 2 3 4 4`

La valeur finale de `i` est `4`

Programme 2

```
i = 0
while i < 5:
    print(i, end=" ")
    i = i + 1
print(i)
```

affiche : `0 1 2 3 4 5` La valeur finale de `i` est `5` afin que la condition ne soit plus valide.

Dans le programme 1, la valeur finale de `i` est `4`, alors que dans le programme 2 c'est `5` afin que la condition ne soit plus valide.

Autre différence, quand on modifie l'indice de boucle dans une boucle `for`, il reprend la valeur suivante à la prochaine répétition.

Programme 1

```
for i in range(5):
    print(i, end=" ")
    i = i + 2
    print(i, end=" ")
```

affiche : `0 2 1 3 2 4 3 5 4 6`

La valeur de `i` est modifiée à l'intérieur de la boucle mais reprend la valeur suivante à la prochaine répétition.

Programme 2

```
i = 0
while i < 5:
    print(i, end=" ")
    i = i + 2
    print(i, end=" ")
```

affiche : `0 2 2 4 4 6`

La valeur de `i` est modifiée à l'intérieur de la boucle.

A chaque passage dans une boucle `for`, l'indice de boucle repart de sa valeur au dernier passage dans la boucle, même si cette valeur a changé dans la boucle. En pratique, il n'est pas recommandé de changer sa valeur dans la boucle.

La boucle `for` possède d'autres possibilités très utiles, par exemple elle permet d'énumérer chaque caractère d'une chaine de caractères. Le programme ci-dessous affiche chaque lettre d'une variable message l'une après l'autre.

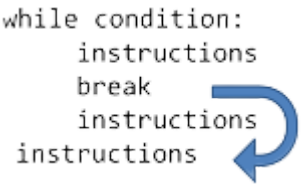
```
message = 'Hello world'
for c in message:
    print(c)
```

Instructions break et continue

Il existe deux instructions qui permettent de modifier l'exécution des boucles `while` et `for`, il s'agit de `break` et de `continue`.

Cours

L'instruction `break` permet de sortir de la boucle courante et de passer à l'instruction suivante.

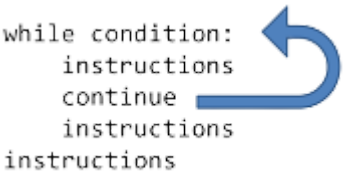


Par exemple, voici un programme qui redemande un mot de passe jusqu'à obtenir le bon :

```
while True:
    mdp = input('mot de passe')
    if mdp == '123456':
        break
print('trouvé')
```

Cours

L'instruction `continue` permet de sauter les instructions qui restent jusqu'à la fin de la boucle et de reprendre à la prochaine itération de boucle.



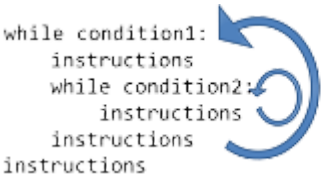
Imaginons par exemple un programme qui affiche la valeur de $1/(n-7)$ pour tous les entiers n compris entre 1 et 10. Il est évident que quand n prend la valeur 7 il y aura une erreur. Grâce à l'instruction `continue`, il est possible de traiter cette valeur à part puis de continuer la boucle.

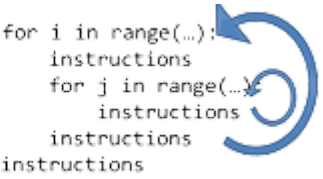
```
for n in range(10):
    if n == 7:
        continue
    print(1 / (7 - n))
```

Boucles imbriquées

Cours

Il est possible d'imbriquer des boucles. A chaque passage dans la première boucle (la boucle externe), la seconde boucle (la boucle interne) est effectuée entièrement.





⚠️ Attention à l'indentation pour déterminer à quelle boucle appartiennent les instructions.

Par exemple le programme suivant affiche toutes les heures, minutes et seconde de la journée (de 0h0min0s à 23h59min59s) :

```
for heure in range(24):
    for minute in range(60):
        for seconde in range(60):
            print(f"{heure}h{minute}min{seconde}s")
```

Exercice corrigé

Exercice corrigé : Écrire un programme en Python qui affiche tous les nombres premiers inférieurs à 100.

Rappel : un nombre est premier s'il n'a que deux diviseurs, 1 et lui-même.

Réponse

---

1. L'avantage d'exclure la borne supérieure apparait clairement dans l'instruction `range(d, f)` qui comprend `f-d` nombres compris entre `d` (inclus) et `f` (exclus), comme l'a expliqué Edsger W. Dijkstra dans une note de 1982 ↩
2. L'instruction `range()` fonctionne sur le modèle `range([début,] fin [, pas])`. Les arguments entre crochets sont optionnels. ↩