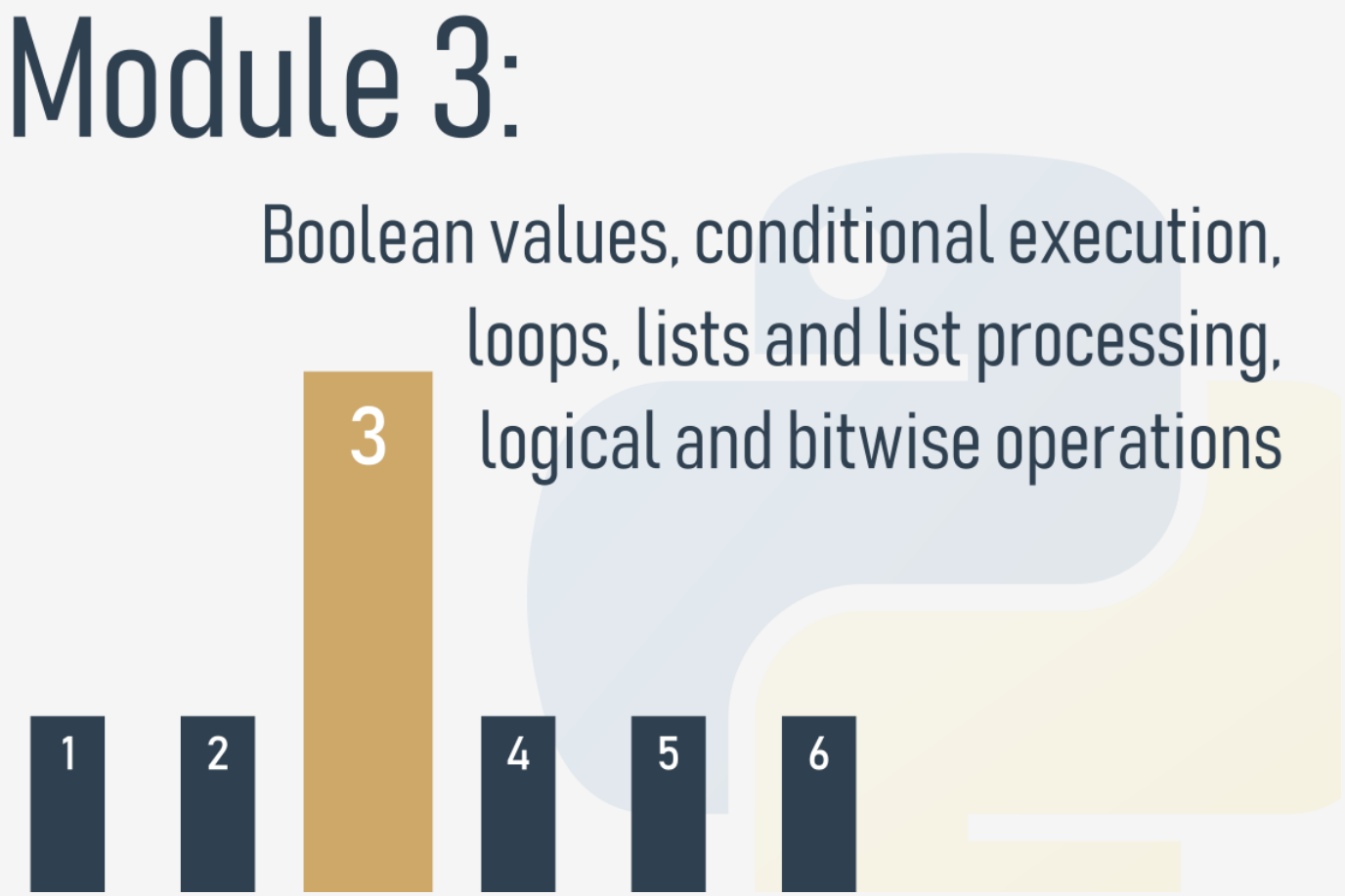
Éléments essentiels de la programmation en Python: Module 3

Dans ce module, vous découvrirez:

- Valeurs booléennes;
- instructions if-elif-else;
- les boucles while et for;
- contrôle de flux;
- opérations logiques et au niveau du bit;
- listes et tableaux.



Questions et réponses

Un programmeur écrit un programme et **le programme pose des questions**.

Un ordinateur exécute le programme et **fournit les réponses**. Le programme doit pouvoir **réagir en fonction des réponses reçues**.

Heureusement, les ordinateurs ne connaissent que deux types de réponses :

- Oui c'est vrai ;

- Non, c'est faux.

Vous n'obtiendrez jamais de réponse comme *Laissez-moi penser ...* , *je ne sais pas* , ou *probablement oui, mais je ne sais pas avec certitude* .

Pour poser des questions, Python utilise un ensemble d'opérateurs très spéciaux . Passons en revue les uns après les autres, illustrant leurs effets sur quelques exemples simples.

Comparaison: opérateur d'égalité

Question: **deux valeurs sont-elles égales** ?

Pour poser cette question, vous utilisez l' `==`opérateur (égal égal).

N'oubliez pas cette distinction importante:

- `=`est un **opérateur d'affectation** , par exemple, `a = b`assigne `a`avec la valeur de `b`;
- `==`la question *est ces valeurs sont-elles égales?* ; `a == b` **compare** `a` et `b`.

Il s'agit d'un **opérateur binaire avec liaison à gauche** . Il a besoin de deux arguments et **vérifie s'ils sont égaux** .

Des exercices

Maintenant posons quelques questions. Essayez de deviner les réponses.

Question # 1 : Quel est le résultat de la comparaison suivante?

`2 == 2` Vérifier

Question # 2 : Quel est le résultat de la comparaison suivante?

`2 == 2.` Vérifier

Question # 3 : Quel est le résultat de la comparaison suivante?

`1 == 2` Vérifier

Égalité: l' opérateur *égal à* (==)

L' opérateur `==` (égal à) compare les valeurs de deux opérandes. S'ils sont égaux, le résultat de la comparaison est `True`. S'ils ne sont pas égaux, le résultat de la comparaison est `False`.

Regardez la comparaison d'égalité ci-dessous - quel est le résultat de cette opération?

`var == 0`

Notez que nous ne pouvons pas trouver la réponse si nous ne savons pas quelle valeur est actuellement stockée dans la variable `var`.

Si la variable a été modifiée plusieurs fois pendant l'exécution de votre programme, ou si sa valeur initiale est entrée à partir de la console, la réponse à cette question ne peut être donnée que par Python et uniquement lors de l'exécution.

Imaginez maintenant un programmeur qui souffre d'insomnie et doit compter séparément les moutons noirs et blancs tant qu'il y a exactement deux fois plus de moutons noirs que de moutons blancs.

La question sera la suivante:

```
black_sheep == 2 * white_sheep
```

En raison de la faible priorité de l' opérateur `==`, la question doit être traitée comme équivalente à celle-ci:

```
black_sheep == (2 * white_sheep)
```

Alors, pratiquons `==`maintenant votre compréhension de l'opérateur - pouvez-vous deviner la sortie du code ci-dessous ?

```
var = 0 # assigning 0 to var
print(var == 0)
```

```
var = 1 # assigning 1 to var
print(var == 0)
```

Exécutez le code et vérifiez si vous aviez raison.

Inégalité: l' opérateur *non égal à* (! =)

L'opérateur `!=` (différent de) compare également les valeurs de deux opérandes. Voici la différence : s'ils sont égaux, le résultat de la comparaison est `False`. S'ils ne sont pas égaux, le résultat de la comparaison est `True`.

Jetez maintenant un œil à la comparaison des inégalités ci-dessous - pouvez-vous deviner le résultat de cette opération?

```
var = 0 # assigning 0 to var
print(var != 0)
```

```
var = 1 # assigning 1 to var
print(var != 0)
```

Exécutez le code et vérifiez si vous aviez raison.

Opérateurs de comparaison : supérieur à

Vous pouvez également poser une question de comparaison à l'aide de l'opérateur `>` (supérieur à).

Si vous voulez savoir s'il y a plus de moutons noirs que de blancs, vous pouvez l'écrire comme suit :

```
black_sheep > white_sheep # greater than
```

`True`le confirme ; `False`le nie.

Opérateurs de comparaison : supérieur ou égal à

L'opérateur *supérieur à* a une autre variante spéciale **non stricte**, mais il est indiqué différemment de la notation arithmétique classique: `>=`(supérieur ou égal à).

Il y a deux signes ultérieurs, pas un.

Ces deux opérateurs (stricts et non stricts), ainsi que les deux autres décrits dans la section suivante, sont des **opérateurs binaires avec liaison à gauche**, et leur **priorité est supérieure à celle indiquée par** `==` **et** `!=` .

Si nous voulons savoir si nous devons ou non porter un chapeau chaud, nous posons la question suivante :

```
centigrade_outside ≥ 0.0 # greater than or equal to
```

Opérateurs de comparaison: inférieur ou égal à

Comme vous l'avez probablement déjà deviné, les opérateurs utilisés dans ce cas sont: l' opérateur `<` (inférieur à) et son frère non strict: `<=`(inférieur ou égal à).

Regardez cet exemple simple:

```
current_velocity_mph < 85 # less than
```

```
current_velocity_mph ≤ 85 # less than or equal to
```

Nous allons vérifier s'il y a un risque d'être condamné à une amende par la police routière (la première question est stricte, la seconde ne l'est pas).

Utiliser les réponses

Que pouvez-vous faire avec la réponse (c.-à-d. Le résultat d'une opération de comparaison) que vous obtenez de l'ordinateur?

Il y a au moins deux possibilités: premièrement, vous pouvez le mémoriser (le **stocker dans une variable**) et l'utiliser plus tard. Comment tu fais ça? Eh bien, vous utiliseriez une variable arbitraire comme celle-ci:

```
answer = number_of_lions >= number_of_lionesses
```

Le contenu de la variable vous indiquera la réponse à la question posée.

La deuxième possibilité est plus pratique et beaucoup plus courante : vous pouvez utiliser la réponse que vous obtenez pour **prendre une décision concernant l'avenir du programme** .

Vous avez besoin d'une instruction spéciale à cet effet, et nous en discuterons très bientôt.

Nous devons maintenant mettre à jour notre **table de priorités** et y mettre tous les nouveaux opérateurs. Il se présente maintenant comme suit:

Priorité	Opérateur	
1	`+`, `-`	unaire
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binaire
5	`<`, `<=`, `>`, `>=`	
6	`==`, `!=`	

LABORATOIRE

Temps estimé

5 minutes

Niveau de difficulté

Très facile

Objectifs

- se familiariser avec la `input()` fonction;
- se familiariser avec les opérateurs de comparaison en Python.

Scénario

À l'aide de l'un des opérateurs de comparaison en Python, écrivez un programme simple à deux lignes qui prend le paramètre `n` en entrée, qui est un entier, et affiche `False` si `n` est inférieur à `100`, et `True` si `n` est supérieur ou égal à `100`.

Ne créez pas de blocs `if` (nous allons en parler très bientôt). Testez votre code à l'aide des données que nous vous avons fournies.

Données de test

Exemple d'entrée: `55`

Production attendue: `False`

Exemple d'entrée: `99`

Production attendue: `False`

Exemple d'entrée: `100`

Production attendue: `True`

Exemple d'entrée: `101`

Production attendue: `True`

Exemple d'entrée: `-5`

Production attendue: `False`

Exemple d'entrée: `+123`

Production attendue: `True`

Conditions et exécution conditionnelle

Vous savez déjà comment poser des questions sur Python, mais vous ne savez toujours pas comment faire un usage raisonnable des réponses. Vous devez avoir un mécanisme qui vous permettra de faire quelque chose **si une condition est remplie et de ne pas le faire si ce n'est pas le cas** .

C'est comme dans la vraie vie: vous faites certaines choses ou vous ne le faites pas quand une condition spécifique est remplie ou non, par exemple, vous allez vous promener si le temps est bon, ou restez à la maison s'il fait humide et froid.

Pour prendre de telles décisions, Python propose une instruction spéciale. En raison de sa nature et de son application, elle est appelée **instruction conditionnelle** (ou instruction conditionnelle).

Il en existe plusieurs variantes. Nous allons commencer par le plus simple, en augmentant lentement la difficulté.

La première forme d'une déclaration conditionnelle, que vous pouvez voir ci-dessous, est écrite de manière très informelle mais figurative:

```
if true_or_not:
    do_this_if_true
```

Cette déclaration conditionnelle comprend uniquement les éléments suivants, strictement nécessaires:

- le mot - clé `if`;
- un ou plusieurs espaces blancs;
- une expression (une question ou une réponse) dont la valeur sera interprétée uniquement en termes de `True`(lorsque sa valeur est non nulle) et `False`(lorsqu'elle est égale à zéro);
- un **colon** suivi d'un saut de ligne;

- une instruction en **retrait** ou un ensemble d'instructions (au moins une instruction est absolument requise); l' **indentation** peut être obtenue de deux manières - en insérant un nombre particulier d'espaces (la recommandation est d'utiliser **quatre espaces d'indentation**), ou en utilisant le caractère de *tabulation* ; note: s'il y a plus d'une instruction dans la partie en retrait, l'indentation doit être la même sur toutes les lignes; même s'il peut avoir la même apparence si vous utilisez des tabulations mélangées à des espaces, il est important que toutes les indentations soient **identiques** - Python 3 **ne permet pas de mélanger les espaces et les tabulations** pour l'indentation.

Comment fonctionne cette déclaration?

- Si l'expression `true_or_not` **représente la vérité** (c'est-à-dire que sa valeur n'est pas égale à zéro), **la ou les instructions en retrait seront exécutées** ;
- si la L'expression true_or_not **ne représente pas la vérité** (c'est-à-dire que sa valeur est égale à zéro), **les instructions en retrait seront omises** (ignorées) et la prochaine instruction exécutée sera celle qui suit le niveau d'indentation d'origine.

Dans la vraie vie, nous exprimons souvent un désir:

si le temps est bon, on se promène

ensuite, nous déjeunerons

Comme vous pouvez le voir, le déjeuner n'est **pas une activité conditionnelle** et ne dépend pas de la météo.

Sachant quelles conditions influencent notre comportement, et en supposant que nous avons les fonctions sans paramètre `go_for_a_walk()` et `have_lunch()`, nous pouvons écrire l'extrait de code suivant:

```
if the_weather_is_good:
    go_for_a_walk()
have_lunch()
```

Exécution conditionnelle: l'instruction `if`

Si un certain développeur Python sans sommeil s'endort quand il ou elle compte 120 moutons et que la procédure d'induction du sommeil peut être implémentée comme une fonction spéciale nommée `sleep_and_dream()`, le code entier prend la forme suivante:

```
if sheep_counter >= 120: # evaluate a test expression
    sleep_and_dream() # execute if test expression is True
```

Vous pouvez le lire comme: si `sheep_counter` est supérieur ou égal à `120`, alors endormez-vous et rêvez (c'est-à-dire, exécutez la fonction `sleep_and_dream`.)

Nous avons dit que les **déclarations exécutées sous condition doivent être mises en retrait** . Cela crée une structure très lisible, montrant clairement tous les chemins d'exécution possibles dans le code.

Jetez un œil au code suivant:

```
if sheep_counter >= 120:
    make_a_bed()
    take_a_shower()
    sleep_and_dream()
feed_the_sheepdogs()
```

Comme vous pouvez le voir, faire un lit, prendre une douche et s'endormir et rêver sont tous **exécutés de manière conditionnelle** - lorsqu'ils `sheep_counter`atteignent la limite souhaitée.

Cependant, l'alimentation des chiens de berger est **toujours effectuée** (c'est-à-dire que la fonction `feed_the_sheepdogs()` n'est pas en retrait et n'appartient pas au bloc `if`, ce qui signifie qu'elle est toujours exécutée.)

Nous allons maintenant discuter d'une autre variante de l'instruction conditionnelle, qui vous permet également d'effectuer une action supplémentaire lorsque la condition n'est pas remplie.

Exécution conditionnelle : l'instruction `if-else`

Nous avons commencé avec une phrase simple qui disait : *Si le temps est bon, nous irons nous promener* .

Remarque - il n'y a pas un mot sur ce qui se passera si le temps est mauvais. Nous savons seulement que nous n'irons pas à l'extérieur, mais ce que nous pourrions faire à la place n'est pas connu. Nous pouvons également vouloir planifier quelque chose en cas de mauvais temps.

Nous pouvons dire, par exemple: *si le temps est bon, nous irons nous promener, sinon nous irons au théâtre* .

Maintenant, nous savons ce que nous ferons **si les conditions sont remplies**, et nous savons ce que nous ferons **si tout ne va pas dans notre direction**. En d'autres termes, nous avons un «plan B».

Python nous permet d'exprimer de tels plans alternatifs. Cela se fait avec une deuxième forme légèrement plus complexe de l'instruction conditionnelle, l'instruction *if-else* :

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_false
```

Ainsi, il y a un nouveau mot: `else`- c'est un **mot - clé** .

La partie du code qui commence par `else`indique quoi faire si la condition spécifiée pour le `if`n'est pas remplie (notez les **deux - points** après le mot).

L'exécution *if-else se* déroule comme suit :

- si la condition est évaluée à **True** (sa valeur n'est pas égale à zéro), l'instruction `perform_if_condition_true` est exécutée et l'instruction conditionnelle prend fin;
- si la condition est évaluée à **False** (elle est égale à zéro), l'instruction `perform_if_condition_false` est exécutée et l'instruction conditionnelle prend fin.

L'instruction `if-else` : exécution plus conditionnelle

En utilisant cette forme de déclaration conditionnelle, nous pouvons décrire nos plans comme suit:

```
if the_weather_is_good:
    go_for_a_walk()
else:
    go_to_a_theater()
have_lunch()
```

S'il fait beau, nous irons nous promener. Sinon, nous irons au théâtre. Peu importe si le temps est bon ou mauvais, nous déjeunerons après (après la promenade ou après être allé au théâtre).

Tout ce que nous avons dit sur l'indentation fonctionne de la même manière dans **la branche *else*** :

```
if the_weather_is_good:
    go_for_a_walk()
    have_fun()
else:
    go_to_a_theater()
    enjoy_the_movie()
have_lunch()
```

`if-else` : Déclarations emboîtés

Voyons maintenant deux cas particuliers de l'instruction conditionnelle.

Considérons d'abord le cas où l'instruction **placée après la** `if` **est une autre** `if` .

Lisez ce que nous avons prévu pour ce dimanche. Si le temps le permet, nous irons nous promener. Si nous trouvons un bon restaurant, nous y déjeunerons. Sinon, nous mangerons un sandwich. Si le temps est mauvais, nous irons au théâtre. S'il n'y a pas de billets, nous irons faire du shopping dans le centre commercial le plus proche.

Écrivons la même chose en Python. Considérez attentivement le code ici:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Voici deux points importants:

- cette utilisation de l' `if` instruction est appelée **imbrication** ; rappelez-vous que tout se `else` réfère à ce `if` qui se trouve **au même niveau d'indentation** ; vous devez le savoir pour déterminer comment les *si* s et les *autres* s s'associent;
- examinez comment l' **indentation améliore la lisibilité** et rend le code plus facile à comprendre et à tracer.

La déclaration `elif`

Le deuxième cas spécial introduit un autre nouveau mot-clé Python: **elif** . Comme vous vous en doutez probablement, c'est une forme plus courte d'autre **chose si** .

`Elif` est utilisé pour **vérifier plus d'une seule condition** et pour **s'arrêter** lorsque la première instruction vraie est trouvée.

Notre exemple suivant ressemble à la nidification, mais les similitudes sont très légères. Encore une fois, nous changerons nos plans et les exprimerons comme suit: Si le temps le permet, nous irons nous promener, sinon si nous obtenons des billets, nous irons au théâtre, sinon s'il y a des tables gratuites au restaurant, nous irons déjeuner; si tout le reste échoue, nous rentrerons chez nous et jouerons aux échecs.

Avez-vous remarqué combien de fois nous avons utilisé le mot *autrement* ? Il s'agit de l'étape où le mot clé `elif` joue son rôle.

Écrivons le même scénario en utilisant Python :

```
if the_weather_is_good :
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

La façon d'assembler les instructions *if-elif-else suivantes* est parfois appelée **cascade** .

Remarquez à nouveau comment l'indentation améliore la lisibilité du code.

Une attention supplémentaire doit être accordée dans ce cas:

- vous **ne devez pas utiliser** `else` **sans un précédent**`if` ;
- `else`est toujours la **dernière branche de la cascade** , que vous l'utilisiez `elif`ou non;
- `else`est une partie **facultative** de la cascade et peut être omise;
- s'il y a un branche `else` dans la cascade, une seule de toutes les branches est exécutée;
- si il n'y a pas sinon, il est possible qu'aucune des branches disponibles ne soit exécutée.

Cela peut sembler un peu déroutant, mais j'espère que quelques exemples simples aideront à faire plus de lumière.

Analyse d'échantillons de code

Nous allons maintenant vous montrer quelques programmes simples mais complets. Nous ne les expliquerons pas en détail, car nous considérons les commentaires (et les noms de variables) à l'intérieur du code comme des guides suffisants.

Tous les programmes résolvent le même problème - ils **trouvent le plus grand de plusieurs nombres et l'impriment** .

Exemple 1:

Nous commencerons par le cas le plus simple - **comment identifier le plus grand des deux nombres** :

```
# read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
```

```
# choose the larger number
if number1 > number2:
    larger_number = number1
else:
    larger_number = number2

# print the result
print("The larger number is:", larger_number)
```

L'extrait ci-dessus doit être clair - il lit deux valeurs entières, les compare et trouve celle qui est la plus grande.

Exemple 2:

Maintenant, nous allons vous montrer un fait fascinant. Python a une fonctionnalité intéressante, regardez le code ci-dessous:

```
# read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# choose the larger number
if number1 > number2: larger_number = number1
else: larger_number = number2

# print the result
print("The larger number is:", larger_number)
```

Remarque: si l'une des branches *if-elif-else* contient une seule instruction, vous pouvez la coder sous une forme plus complète (vous n'avez pas besoin de créer une ligne en retrait après le mot-clé, mais continuez simplement la ligne après les deux-points) .

Ce style, cependant, peut être trompeur, et nous n'allons pas l'utiliser dans nos futurs programmes, mais il vaut vraiment la peine de savoir si vous voulez lire et comprendre les programmes de quelqu'un d'autre.

Il n'y a pas d'autres différences dans le code.

Exemple 3:

Il est temps de compliquer le code - trouvons le plus grand des trois nombres. Va-t-il agrandir le code? Un peu.

Nous supposons que la première valeur est la plus grande. Ensuite, nous vérifions cette hypothèse avec les deux valeurs restantes.

Regardez le code ci-dessous:

```
# read three numbers
```

```
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# We temporarily assume that the first number
# is the largest one.
# We will verify this soon.
largest_number = number1

# we check if the second number is larger than current largest_number
# and update largest_number if needed
if number2 > largest_number:
    largest_number = number2

# we check if the third number is larger than current largest_number
# and update largest_number if needed
if number3 > largest_number:
    largest_number = number3

# print the result
print("The largest number is:", largest_number)
```

Cette méthode est beaucoup plus simple que d'essayer de trouver le plus grand nombre d'un coup, en comparant toutes les paires de nombres possibles (c'est-à-dire d'abord avec le deuxième, deuxième avec le troisième, troisième avec le premier). Essayez de reconstruire le code par vous-même.

Pseudocode et introduction aux boucles

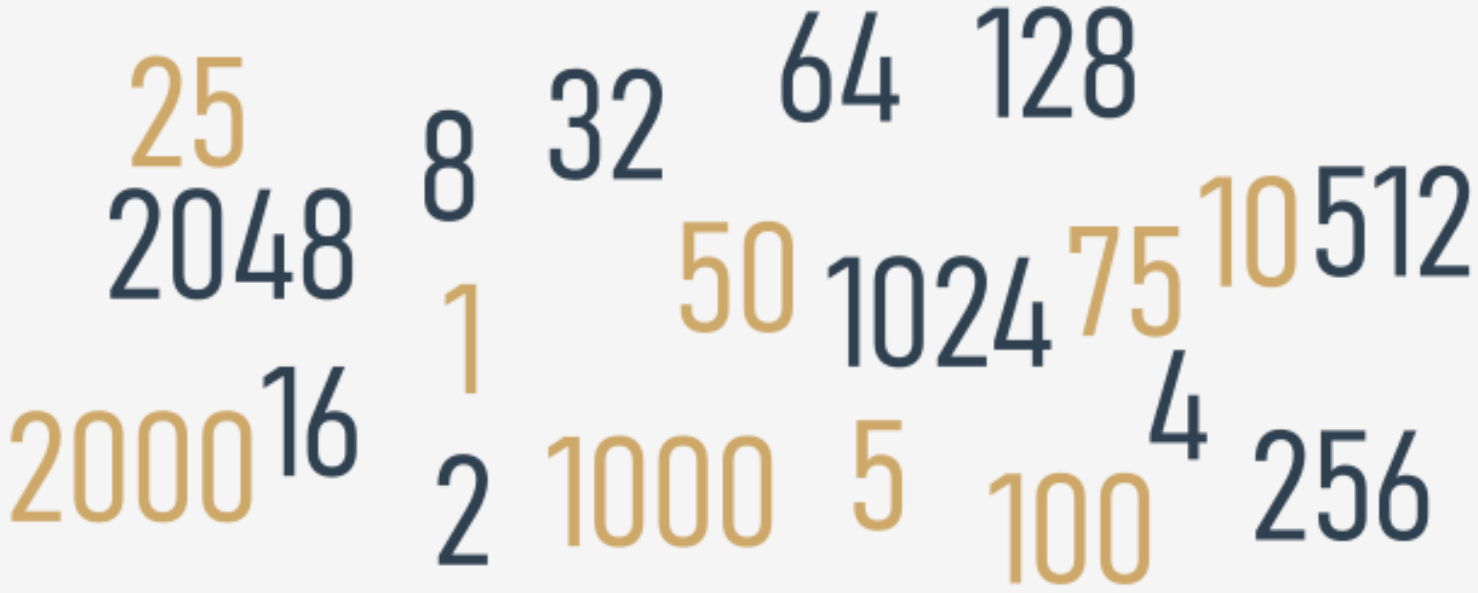
Vous devriez maintenant pouvoir écrire un programme qui trouve le plus grand de quatre, cinq, six ou même dix nombres.

Vous connaissez déjà le schéma, donc l'extension de la taille du problème ne sera pas particulièrement complexe.

Mais que se passe-t-il si nous vous demandons d'écrire un programme qui trouve le plus grand des deux cents nombres? Pouvez-vous imaginer le code?

Vous aurez besoin de deux cents variables. Si deux cents variables ne suffisent pas, essayez d'imaginer la recherche du plus grand d'un million de nombres.

Imaginez un code qui contient 199 instructions conditionnelles et deux cents appels de la `input()` fonction. Heureusement, vous n'avez pas besoin de vous en occuper. Il existe une approche plus simple.



Nous allons ignorer les exigences de la syntaxe Python pour l'instant et essayer d'analyser le problème sans penser à la vraie programmation. En d'autres termes, nous essaierons d'écrire l'algorithme , et lorsque nous en serons satisfait, nous l'implémenterons.

Dans ce cas, nous utiliserons une sorte de notation qui n'est pas un langage de programmation réel (il ne peut être ni compilé ni exécuté), mais il est formalisé, concis et lisible. Cela s'appelle un **pseudocode** .

Regardons notre pseudocode ci-dessous:

```
line 01 largest_number = -999999999
line 02 number = int(input())
line 03 if number == -1:
line 04     print(largest_number)
line 05     exit()
line 06 if number > largest_number:
line 07     largest_number = number
line 08 go to line 02
```

Que s'y passe-t-il?

Premièrement, nous pouvons simplifier le programme si, au tout début du code, nous attribuons à la variable `largestNumber` une valeur qui sera plus petite que n'importe quel nombre entré. Nous utiliserons `-999999999` à cet effet.

Deuxièmement, nous supposons que notre algorithme ne saura pas à l'avance combien de numéros seront livrés au programme. Nous nous attendons à ce que l'utilisateur saisisse autant de nombres qu'il le souhaite - l'algorithme fonctionnera bien avec cent et mille nombres. Comment fait-on cela?

Nous passons un accord avec l'utilisateur: lorsque la valeur `-1` est entrée, ce sera un signe qu'il n'y a plus de données et que le programme devrait terminer son travail.

Sinon, si la valeur entrée n'est pas égale à `-1`, le programme lira un autre nombre, etc.

L'astuce est basée sur l'hypothèse que n'importe quelle partie du code peut être exécutée plus d'une fois - précisément, autant de fois que nécessaire.

L'exécution d'une certaine partie du code plus d'une fois s'appelle une **boucle** . La signification de ce terme est probablement évidente pour vous.

Les lignes `02` traversent `08` une boucle. Nous les passerons **autant de fois que nécessaire** pour revoir toutes les valeurs saisies.

Pouvez-vous utiliser une structure similaire dans un programme écrit en Python? Oui, vous pouvez.

Informaitons supplémentaires

Python est souvent livré avec de nombreuses fonctions intégrées qui feront le travail pour vous. Par exemple, pour trouver le plus grand nombre de tous, vous pouvez utiliser une fonction intégrée Python appelée `max()`. Vous pouvez l'utiliser avec plusieurs arguments. Analysez le code ci-dessous:

```
# read three numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# check which one of the numbers is the greatest
# and pass it to the largest_number variable

largest_number = max(number1, number2, number3)

# print the result
print("The largest number is:", largest_number)
```

De la même manière, vous pouvez utiliser la `min()` fonction pour renvoyer le nombre le plus bas. Vous pouvez reconstruire le code ci-dessus et l'expérimenter dans votre EDI.

Nous allons bientôt parler de ces fonctions (et bien d'autres). Pour le moment, nous nous concentrerons sur l'exécution conditionnelle et les boucles pour vous permettre de gagner en confiance dans la programmation et vous enseigner les compétences qui vous permettront de comprendre et d'appliquer pleinement les deux concepts de votre code. Donc, pour l'instant, nous ne prenons aucun raccourci.

LABORATOIRE

Temps estimé

5-10 minutes

Niveau de difficulté

Facile

Objectifs

- se familiariser avec la fonction `input ()` ;
- se familiariser avec les opérateurs de comparaison en Python;
- se familiariser avec le concept d'exécution conditionnelle.

Scénario

Le Spathiphyllum , plus communément appelé lis de paix ou plante à voile blanche, est l'une des plantes d'intérieur les plus populaires qui filtrent les toxines nocives de l'air. Certaines des toxines qu'il neutralise comprennent le benzène, le formaldéhyde et l'ammoniac.

Imaginez que votre programme informatique aime ces plantes. Chaque fois qu'il reçoit une entrée sous la forme du mot `Spathiphyllum`, il crie involontairement à la console la chaîne suivante:`"Spathiphyllum is the best plant ever!"`

Écrivez un programme qui utilise le concept d'exécution conditionnelle, prend une chaîne en entrée et:

- `"Yes - Spathiphyllum is the best plant ever!"`affiche la phrase à l'écran si la chaîne entrée est `"Spathiphyllum"`(majuscule)
- s'affiche `"No, I want a big Spathiphyllum!"`si la chaîne entrée est `"spathiphyllum"`(en minuscules)
- imprime `"Spathiphyllum! Not [input]!"`autrement. Remarque: `[input]`est la chaîne prise en entrée.

Testez votre code à l'aide des données que nous vous avons fournies. Et procurez-vous un Spathiphyllum aussi!

Données de test

Exemple d'entrée: `spathiphyllum`

Production attendue: `No, I want a big Spathiphyllum!`

Exemple d'entrée: `pelargonium`

Production attendue: `Spathiphyllum! Not pelargonium!`

Exemple d'entrée: `Spathiphyllum`

Production attendue: `Yes - Spathiphyllum is the best plant ever!`

LABORATOIRE

Temps estimé

10-15 minutes

Niveau de difficulté

Facile/moyen

Objectifs

- Familiarisez l'étudiant avec:
- • utiliser l'instruction if-else pour dériver le chemin de contrôle;
- • créer un programme complet qui résout de simples problèmes réels.

Scénario

Il était une fois une terre - une terre de lait et de miel, habitée par des gens heureux et prospères. Les gens payaient des impôts, bien sûr - leur bonheur avait des limites. L'impôt le plus important, appelé impôt sur le revenu des personnes physiques (PIT en abrégé) devait être payé une fois par an et était évalué selon la règle suivante:

• si le revenu du citoyen n'était pas supérieur à 85 528 thalers, l'impôt était égal à 18% du revenu moins 556 thalers et 2 cents (il s'agissait de ce que l'on appelle l'allégement fiscal)

• si le revenu était supérieur à ce montant, l'impôt était égal à 14 839 thalers et 2 cents, plus 32% de l'excédent sur 85 528 thalers.

Votre tâche consiste à écrire une **calculatrice fiscale**.

• Il doit accepter une valeur à virgule flottante: le revenu.

• Ensuite, il devrait imprimer la taxe calculée, arrondie aux thalers complets. Il y a une fonction nommée `round ()` qui fera l'arrondi pour vous - vous la trouverez dans le code squelette dans l'éditeur.

Remarque: ce pays heureux ne rend jamais d'argent à ses citoyens. Si la taxe calculée est inférieure à zéro, cela signifie seulement pas de taxe du tout (la taxe est égale à zéro). Tenez-en compte lors de vos calculs.

Regardez le code dans l'éditeur - il ne lit qu'une seule valeur d'entrée et produit un résultat, vous devez donc le compléter avec quelques calculs intelligents.

```
income = float(input("Enter the annual income: "))

#

# Put your code here.

#

tax = round(tax, 0)

print("The tax is:", tax, "thalers")
```

Testez votre code en utilisant les données que nous avons fournies.

Test de données

Exemple d'entrée : `10000`

Production attendee : `The tax is: 1244.0 thalers`

Exemple d'entrée: `100000`

Production attendee: `The tax is: 19470.0 thalers`

Exemple d'entrée: `1000`

Production attendee: `The tax is: 0.0 thalers`

Exemple d'entrée : `-100`

Production attendee: `The tax is: 0.0 thalers`

LABORATOIRE

Temps estimé

10-15 minutes

Niveau de difficulté

Facile / Moyen

Objectifs

- Familiarisez l'étudiant avec:
- • utiliser l'instruction if-elif-else;
- • trouver la bonne mise en œuvre des règles définies verbalement;
- • tester le code en utilisant un échantillon d'entrée et de sortie.

Scénario

Comme vous le savez sûrement, pour des raisons astronomiques, les années peuvent être bissextiles ou communes. Les premiers durent 366 jours, tandis que les seconds durent 365 jours.

Depuis l'introduction du calendrier grégorien (en 1582), la règle suivante est utilisée pour déterminer le type d'année:

• si le numéro d'année n'est pas divisible par quatre, c'est une année commune;

• sinon, si le numéro d'année n'est pas divisible par 100, c'est une année bissextile;

• sinon, si le numéro d'année n'est pas divisible par 400, c'est une année commune;

• sinon, c'est une année bissextile.

Regardez le code dans l'éditeur - il ne lit qu'un numéro d'année et doit être complété avec les instructions de mise en œuvre du test que nous venons de décrire.

```
year = int(input("Enter a year: "))

#

# Put your code here.

#
```

Le code doit générer l'un des deux messages possibles, qui sont `Leap year` ou `Common year`, selon la valeur entrée.

Il serait bon de vérifier si l'année entrée tombe dans l'ère grégorienne et d'émettre un avertissement sinon : `Pas dans la période du calendrier grégorien`. Astuce : utilisez les opérateurs `! =` Et `%`.

Testez votre code en utilisant les données que nous avons fournies.

Testez votre code en utilisant les données que nous avons fournies.

Données de test

Exemple d’entrée : `2000`

Production attendue: `Leap year`

Exemple d’entrée : `2015`

Production attendue: `Common year`

Exemple d'entrée : `1999`

Production attendue: `Common year`

Exemple d'entrée : `1996`

Production attendue: `Leap year`

Exemple d'entrée : `1580`

Production attendee : `Not within the Gregorian calendar period`

Points clés à retenir

1. Les opérateurs de **comparaison** (ou dits *relationnels*) sont utilisés pour comparer les valeurs. Le tableau ci - dessous illustre la façon dont les opérateurs de comparaison fonctionnent, en supposant que `x = 0`, `y = 1` et `z = 0`:

Opérateur	La description	Exemple
`==`	renvoie si les valeurs des opérandes sont égales, et `False` sinon	`x == y # False` `x == z # True`
`!=`	renvoie `True` si les valeurs des opérandes ne sont pas égales, et `False` sinon	`x != y # True` `x != z # False`
`>`	`True` si la valeur de l'opérande gauche est supérieure à la valeur de l'opérande droit, et `False` sinon	`x > y # False` `y > z # True`
`<`	`True` si la valeur de l'opérande gauche est inférieure à la valeur de l'opérande droit, et `False` sinon	`x < y # True` `y < z # False`

`≥`	`True` si la valeur de l'opérande gauche est supérieure ou égale à la valeur de l'opérande droit, et `False` sinon	`x >= y # False`
		`x >= z # True`
		`y >= z # True`

`≤`	`True` si la valeur de l'opérande gauche est inférieure ou égale à la valeur de l'opérande droit, et `False` sinon	`x <= y # True`
		`x <= z # True`
		`y <= z # False`

2. Lorsque vous souhaitez exécuter du code uniquement si une certaine condition est remplie, vous pouvez utiliser une **instruction conditionnelle** :

- une seule `if` déclaration, par exemple:

```
x = 10

if x == 10: # condition
    print("x is equal to 10") # executed if the condition is True
```

- une série de `if` déclarations, par exemple:

```
x = 10

if x > 5: # condition one
    print("x is greater than 5") # executed if condition one is True

if x < 10: # condition two
    print("x is less than 10") # executed if condition two is True

if x == 10: # condition three
```

```
    print("x is equal to 10") # executed if condition three is True
```

Chaque `if` déclaration est testée séparément.

- une `if-else` déclaration, par exemple:

```
x = 10

if x < 10: # condition
    print("x is less than 10") # executed if the condition is True

else:
    print("x is greater than or equal to 10") # executed if the condition is False
```

- une série de `if` déclarations suivies d'un `else`, par exemple:

```
x = 10

if x > 5: # True
    print("x > 5")

if x > 8: # True
    print("x > 8")

if x > 10: # False
    print("x > 10")

else:
    print("else will be executed")
```

Chacun `if` est testé séparément. Le corps de `else` est exécuté si le dernier l' `if` est `False`.

- La `if-elif-else` déclaration, par exemple:

```
x = 10

if x == 10: # True
    print("x == 10")

if x > 15: # False
    print("x > 15")

elif x > 10: # False
    print("x > 10")

elif x > 5: # True
    print("x > 5")

else:
    print("else will not be executed")
```

Si la condition `if` est `False`, le programme vérifie les conditions des `elif` blocs suivants - le premier `elif` bloc qui `True` est exécuté. Si toutes les conditions sont réunies `False`, le `else` blocage sera exécuté.

- Instructions conditionnelles imbriquées, par exemple:

```
x = 10

if x > 5: # True
```

```
    if x == 6: # False
        print("nested: x == 6")
    elif x == 10: # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

Points clés à retenir: suite

Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
x = 5

y = 10

z = 8

print(x > y)

print(y > z)
```

Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
x, y, z = 5, 10, 8


print(x > z)

print((y - 5) == x)
```

Exercice 3

Quelle est la sortie de l'extrait de code suivant ?

```
x, y, z = 5, 10, 8

x, y, z = z, y, x


print(x > z)

print((y - 5) == x)
```

Exercice 4

Quelle est la sortie de l'extrait de code suivant ?

```
x = 10

if x == 10:

    print(x == 10)

if x > 5:

    print(x > 5)

if x < 10:
```

```
    print(x < 10)
else:
    print("else")
```

Exercice 5

Quelle est la sortie de l'extrait de code suivant?

```
x = "1"


if x == 1:
    print("one")
elif x == "1":
    if int(x) > 1:
        print("two")
    elif int(x) < 1:
        print("three")
    else:
        print("four")
if int(x) == 1:
    print("five")
else:
    print("six")
```

Exercice 6

Quelle est la sortie de l'extrait de code suivant ?

```
x = 1
y = 1.0
z = "1"
if x == y:
    print("one")
if y == int(z):
    print("two")
elif x == y:
    print("three")
else:
    print("four")
```

Boucler votre code avec `while`

Êtes-vous d'accord avec l'énoncé présenté ci-dessous?

```
while there is something to do
    do it
```

Notez que cet enregistrement déclare également que s'il n'y a rien à faire, rien ne se passera.

En général, en Python, une boucle peut être représentée comme suit:

```
while conditional_expression:
    instruction
```

Si vous remarquez des similitudes avec l' instruction *if* , c'est très bien. En effet, la différence syntaxique n'est qu'une: vous utilisez le mot `while`au lieu du mot `if`.

La différence sémantique est plus importante: lorsque la condition est remplie, *if* n'exécute ses instructions **qu'une seule fois** ; *tandis que* `while` **répète l'exécution tant que la condition est évaluée** `True` .

Remarque: toutes les règles concernant l' **indentation** sont également applicables ici. Nous vous le montrerons bientôt.

Regardez l'algorithme ci-dessous :

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

Il est maintenant important de se rappeler que:

- si vous voulez exécuter **plus d'une instruction à l'intérieur d'une** `while` , vous devez (comme avec `if`) mettre en **retrait** toutes les instructions de la même manière;
- une instruction ou un ensemble d'instructions exécutées à l'intérieur de la boucle `while` est appelé **corps de** la **boucle** ;
- si la condition est `False`(égale à zéro) dès lors qu'elle est testée pour la première fois, le corps n'est pas exécuté une seule fois (notez l'analogie de ne rien faire s'il n'y a rien à faire);
- le corps devrait être en mesure de modifier la valeur de la condition, car si la condition est `True` au début, le corps peut fonctionner en continu à l'infini - notez que faire une chose diminue généralement le nombre de choses à faire).

Une boucle infinie

Une boucle infinie, également appelée **boucle sans fin** , est une séquence d'instructions dans un programme qui se répète indéfiniment (boucle sans fin.)

Voici un exemple de boucle qui n'est pas en mesure de terminer son exécution :

```
while True:
    print("I'm stuck inside a loop.")
```

Cette boucle s'imprimera `"I'm stuck inside a loop."` à l' infini sur l'écran.

Si vous voulez obtenir la meilleure expérience d'apprentissage de voir comment se comporte une boucle infinie, lancez IDLE, créez un nouveau fichier, copiez-collez le code ci-dessus, enregistrez votre fichier et exécutez le programme. Ce que vous verrez est la séquence sans fin de `"I'm stuck inside a loop."` chaînes imprimée dans la fenêtre de la console Python. Pour terminer votre programme, appuyez simplement sur *Ctrl-C* (ou *Ctrl-Break* sur certains ordinateurs). Cela provoquera la soi-disant `KeyboardInterrupt` exception et laissera votre programme sortir de la boucle. Nous en parlerons plus tard dans le cours.

Revenons à l'esquisse de l'algorithme que nous vous avons montré récemment. Nous allons vous montrer comment utiliser cette boucle nouvellement apprise pour trouver le plus grand nombre à partir d'un grand ensemble de données saisies.

Analysez soigneusement le programme. Localisez le corps de la boucle et découvrez **comment le corps est sorti** :

```
# we will store the current largest number here
largest_number = -999999999

# input the first value
number = int(input("Enter a number or type -1 to stop: "))

# if the number is not equal to -1, we will continue
while number != -1:
    # is number larger than largest_number?
    if number > largest_number:
        # yes, update largest_number
        largest_number = number
    # input the next number
    number = int(input("Enter a number or type -1 to stop: "))

# print the largest number
print("The largest number is:", largest_number)
```

Vérifiez comment ce code implémente l'algorithme que nous vous avons montré précédemment.

La boucle while : plus d’exemples

Regardons un autre exemple utilisant la boucle `while`. Suivez les commentaires pour découvrir l'idée et la solution.

```
# A program that reads a sequence of numbers
# and counts how many numbers are even and how many are odd.
```

```
# The program terminates when zero is entered.


odd_numbers = 0

even_numbers = 0


# read the first number

number = int(input("Enter a number or type 0 to stop: "))


# 0 terminates execution

while number != 0:

    # check if the number is odd

    if number % 2 == 1:

        # increase the odd_numbers counter

        odd_numbers += 1

    else:

        # increase the even_numbers counter

        even_numbers += 1

    # read the next number

    number = int(input("Enter a number or type 0 to stop: "))


# print results

print("Odd numbers count:", odd_numbers)

print("Even numbers count:", even_numbers)
```

Certaines expressions peuvent être simplifiées sans changer le comportement du programme.

Essayez de vous rappeler comment Python interprète la vérité d'une condition et notez que ces deux formes sont équivalentes:

`while number != 0:` et `while number:`.

La condition qui vérifie si un nombre est impair peut également être codée sous ces formes équivalentes :

`if number % 2 == 1:` et `if number % 2:`.

Utilisation d'une variable de `compteur` pour quitter une boucle

Regardez l'extrait ci-dessous:

```
counter = 5

while counter != 0:
```

```
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Ce code est destiné à imprimer la chaîne `"Inside the loop."`et la valeur stockée dans la variable `counter` pendant une boucle donnée exactement cinq fois. Une fois que la condition n'est pas remplie (la variable `counter` atteint `0`), la boucle est quittée et le message `"Outside the loop."`ainsi que la valeur stockée dans `counter`sont imprimés.

Mais il y a une chose qui peut être écrite de manière plus compacte - l'état de la boucle `while`.

Pouvez-vous voir la différence?

```
counter = 5
while counter:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Est-il plus compact qu'auparavant? Un peu. Est-ce plus lisible? C'est discutable.

RAPPELLES TOI

Ne vous sentez pas obligé de coder vos programmes de manière toujours la plus courte et la plus compacte. La lisibilité peut être un facteur plus important. Gardez votre code prêt pour un nouveau programmeur.

LABORATOIRE

Temps estimé

15 minutes

Niveau de difficulté

Facile

Objectifs

Familiarisez l'étudiant avec:

- en utilisant le `tout` en boucle;
- reflétant des situations réelles dans le code informatique.

Scénario

Un magicien junior a choisi un numéro secret. Il l'a caché dans une variable nommée `secret_number`. Il veut que tous ceux qui gèrent son programme jouent au jeu *Devinez le numéro secret* et devinent quel numéro il a choisi pour eux. Ceux qui ne devinent pas le nombre seront coincés dans une boucle sans fin pour toujours! Malheureusement, il ne sait pas comment compléter le code.

Votre tâche est d'aider le magicien à compléter le code dans l'éditeur de manière à ce que le code:

```
secret_number = 777

print(

"""

+==============================+

| Welcome to my game, muggle!    |

| Enter an integer number        |

| and guess what number I've     |

| picked for you.                |

| So, what is the secret number? |

+==============================+

""")
```

- demandera à l'utilisateur d'entrer un nombre entier;
- utilisera une `while`boucle;
- vérifiera si le numéro entré par l'utilisateur est le même que le numéro choisi par le magicien. Si le numéro choisi par l'utilisateur est différent du numéro secret du magicien, l'utilisateur devrait voir le message `"Ha ha! You're stuck in my loop!"`et être invité à entrer à nouveau un numéro. Si le numéro entré par l'utilisateur correspond au numéro choisi par le magicien, le numéro doit être imprimé à l'écran et le magicien doit dire les mots suivants:`"Well done, muggle! You are free now."`

Le magicien compte sur vous! Ne le déçois pas.

INFORMAITONS SUPPLÉMENTAIRES

Soit dit en passant, regardez la fonction `print()`. La façon dont nous l'avons utilisé ici s'appelle l' *impression multi-lignes* . Vous pouvez utiliser des **guillemets triples** pour imprimer des chaînes sur plusieurs lignes afin de faciliter la lecture du texte, ou créer une conception textuelle spéciale. Expérimentez avec.

Boucler votre code avec `for`

Un autre type de boucle disponible en Python vient de l'observation qu'il est parfois plus important de **compter les "tours" de la boucle** que de vérifier les conditions.

Imaginez que le corps d'une boucle doit être exécuté exactement cent fois. Si vous souhaitez utiliser la boucle `while` pour le faire, cela peut ressembler à ceci:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

Ce serait bien si quelqu'un pouvait faire ce comptage ennuyeux pour vous. Est-ce possible?

Bien sûr que oui - il y a une boucle spéciale pour ce genre de tâches, et elle est nommée `for`.

En fait, la boucle `for` est conçue pour effectuer des tâches plus compliquées - **elle peut «parcourir» de grandes collections de données élément par élément** . Nous allons vous montrer comment faire cela bientôt, mais pour le moment, nous allons présenter une variante plus simple de son application.

Jetez un œil à l'extrait:

```
for i in range(100):
    # do_something()
    pass
```

Il y a de nouveaux éléments. Laissez-nous vous en parler:

- le mot-clé *for* ouvre la boucle `for`; note - il n'y a aucune condition après cela; vous n'avez pas à penser aux conditions, car elles sont vérifiées en interne, sans aucune intervention;
- toute variable après le mot clé *for* est la **variable** de **contrôle** de la boucle; il compte les tours de boucle et le fait automatiquement;
- le mot clé *in* introduit un élément de syntaxe décrivant la plage de valeurs possibles affectées à la variable de contrôle;
- la fonction `range()` (c'est une fonction très spéciale) est chargée de générer toutes les valeurs souhaitées de la variable de contrôle; dans notre exemple, la fonction créera (on peut même dire qu'elle **alimentera** la boucle avec) les valeurs suivantes de l'ensemble suivant: 0, 1, 2 .. 97, 98, 99; note: dans ce cas, la fonction `range()` démarre son travail à partir de 0 et le termine une étape (un nombre entier) avant la valeur de son argument;
- notez le mot - clé *pass* dans le corps de la boucle - il ne fait rien du tout; c'est une **instruction vide** - nous mettons ici parce que les `for` exigences de syntaxe de boucle au moins une instruction à l' intérieur du corps (en passant - `if`, `elif`, `else` et `while` exprimer la même chose)

Nos prochains exemples seront un peu plus modestes quant au nombre de répétitions de boucle.

Jetez un œil à l'extrait ci-dessous. Pouvez-vous prédire sa sortie?

```
for i in range(10):
    print("The value of i is currently", i)
```

Exécutez le code pour vérifier si vous aviez raison.

Remarque:

- la boucle a été exécutée dix fois (c'est l'argument de la fonction `range()`)
- la dernière valeur de la variable de contrôle est `9` (pas `10`, car **elle commence à partir** `0` , pas à partir de `1`)

L' `range()` invocation de fonction peut être équipée de deux arguments, pas d'un seul:

```
for i in range(2, 8):
    print("The value of i is currently", i)
```

Dans ce cas, le premier argument détermine la (première) valeur initiale de la variable de contrôle.

Le dernier argument montre la première valeur à laquelle la variable de contrôle ne sera pas affectée.

Remarque: la `range()` fonction **accepte uniquement des entiers comme arguments** et génère des séquences d'entiers.

Pouvez-vous deviner la sortie du programme? Exécutez-le pour vérifier si vous étiez en ce moment aussi.

La première valeur affichée est `2` (tirée du `range()` premier argument de.)

Le dernier est `7` (bien que le `range()` deuxième argument de l 'soit `8`).

En savoir plus sur la boucle `for` et la fonction `range ()` avec trois arguments

La `range()` fonction peut également accepter **trois arguments** - jetez un œil au code dans l'éditeur.

```
for i in range(2, 8, 3):

    print("The value of i is currently", i)
```

Le troisième argument est un **incrément** - c'est une valeur ajoutée pour contrôler la variable à chaque tour de boucle (comme vous vous en doutez, la **valeur par défaut de l'incrément est 1**).

Pouvez-vous nous dire combien de lignes apparaîtront dans la console et quelles valeurs elles contiendront ?

Exécutez le programme pour savoir si vous aviez raison.
Vous devriez pouvoir voir les lignes suivantes dans la fenêtre de la console :

```
The value of i is currently 2
```

```
The value of i is currently 5
```

Est-ce que tu sais pourquoi? Le premier argument passé à la `range()` fonction nous indique quel est le numéro de **départ** de la séquence (donc `2` dans la sortie). Le deuxième argument indique à la fonction où **arrêter** la séquence (la fonction génère des nombres jusqu'au nombre indiqué par le deuxième argument, mais ne l'inclut pas). Enfin, le troisième argument indique l' **étape** , ce qui signifie en réalité la différence entre chaque nombre dans la séquence de nombres générée par la fonction.

`2` (nombre de départ) → `5` (`2` incrément de 3 égal `5` - le nombre est compris entre 2 et 8) → `8` (`5` incrément de 3 égal `8` - le nombre n'est pas compris entre 2 et 8, car le paramètre d'arrêt n'est pas inclus dans le séquence de nombres générée par la fonction.)

Remarque: si l'ensemble généré par la `range()` fonction est vide, la boucle n'exécutera pas du tout son corps.

Tout comme ici - il n'y aura pas de sortie:

```
for i in range(1, 1):
    print("The value of i is currently", i)
```

Remarque: l'ensemble généré par le `range()` doit être trié **par ordre croissant** . Il n'y a aucun moyen de forcer le `range()` à créer un ensemble sous une forme différente. Cela signifie que le `range()` deuxième argument de doit être supérieur au premier.

Ainsi, il n'y aura pas de sortie ici non plus:

```
for i in range(2, 1):
    print("The value of i is currently", i)
```

Jetons un coup d'œil à un programme court dont la tâche est d'écrire certains des premiers pouvoirs de deux:

```
pow = 1
```

```
for exp in range(16):
```

```
    print("2 to the power of", exp, "is", pow)
    pow *= 2
```

La variable `exp` est utilisée comme variable de contrôle pour la boucle et indique la valeur actuelle de l' *exposant* . L'exponentiation elle-même est remplacée par une multiplication par deux. Puisque 2^0 est égal à 1, alors 2×1 est égal à 2^1 , 2×2^1 est égal à 2^2 , et ainsi de suite. Quel est le plus grand exposant pour lequel notre programme imprime encore le résultat?

Exécutez le code et vérifiez si la sortie correspond à vos attentes.

LABORATOIRE

Temps estimé

5 minutes

Niveau de difficulté

Très facile

Objectifs

Familiarisez l'étudiant avec:

- en utilisant la boucle `for` ;
- reflétant des situations réelles dans le code informatique.

Scénario

Savez-vous ce qu'est le Mississippi? Eh bien, c'est le nom d'un des États et des fleuves des États-Unis. Le fleuve Mississippi mesure environ 2340 milles de long, ce qui en fait le deuxième plus long fleuve des États-Unis (le plus long étant le fleuve Missouri). C'est tellement long qu'une seule goutte d'eau a besoin de 90 jours pour parcourir toute sa longueur!

Le mot *Mississippi* est également utilisé dans un but légèrement différent: *compter de manière mississippile* .

Si vous n'êtes pas familier avec la phrase, nous sommes là pour vous expliquer ce qu'elle signifie: elle est utilisée pour compter les secondes.

L'idée derrière cela est que l'ajout du mot *Mississippi* à un nombre lors du comptage des secondes les rend plus proches de l'heure de l'horloge, et donc "un Mississippi, deux Mississippi, trois Mississippi" prendra environ trois secondes de temps réel! Il est souvent utilisé par les enfants jouant à cache-cache pour s'assurer que le chercheur fait un décompte honnête.

Votre tâche est très simple ici: écrivez un programme qui utilise une `for`boucle pour "compter de manière erronée" jusqu'à cinq. Après avoir compté jusqu'à cinq, le programme devrait imprimer à l'écran le message final `"Ready or not, here I come!"`

Utilisez le squelette que nous avons fourni dans l'éditeur.

```
import time

# Write a for loop that counts to five.
```

```
    # Body of the loop - print the loop iteration number and the word "Mississippi".

    # Body of the loop - use: time.sleep(1)

# Write a print function with the final message.
```

INFORMAITONS SUPPLÉMENTAIRES

Notez que le code dans l'éditeur contient deux éléments qui peuvent ne pas être entièrement clairs pour le moment: l' `import time` instruction et la `sleep()` méthode. Nous allons en parler bientôt.

Pour le moment, nous aimerions simplement que vous sachiez que nous avons importé le `time` module et utilisé la `sleep()` méthode pour suspendre l'exécution de chaque `print()` fonction suivante à l'intérieur de la `for` boucle pendant une seconde, de sorte que le message envoyé à la console ressemble à un réel compte. Ne vous inquiétez pas, vous en apprendrez bientôt plus sur les modules et les méthodes.

Production attendue

```
1 Mississippi

2 Mississippi

3 Mississippi

4 Mississippi

5 Mississippi
```

Les déclarations `break` and `continue`

Jusqu'à présent, nous avons traité le corps de la boucle comme une séquence d'instructions indivisible et inséparable qui sont exécutées complètement à chaque tour de la boucle. Cependant, en tant que développeur, vous pourriez être confronté aux choix suivants:

- il semble qu'il ne soit pas nécessaire de continuer la boucle dans son ensemble; vous devez vous abstenir de poursuivre l'exécution du corps de la boucle et aller plus loin;
- il semble que vous devez commencer le tour suivant de la boucle sans terminer l'exécution du tour en cours.

Python fournit deux instructions spéciales pour l'implémentation de ces deux tâches. Disons pour des raisons de précision que leur existence dans le langage n'est pas nécessaire - un programmeur expérimenté est capable de coder n'importe quel algorithme sans ces instructions. De tels ajouts, qui n'améliorent pas le pouvoir expressif du langage, mais ne font que simplifier le travail du développeur, sont parfois appelés **bonbons** syntaxiques ou sucre syntaxique.

Ces deux instructions sont les suivantes:

- `break` - quitte la boucle immédiatement et met fin inconditionnellement au fonctionnement de la boucle; le programme commence à exécuter l'instruction la plus proche après le corps de la boucle;
- `continue` - se comporte comme si le programme avait soudainement atteint la fin du corps; le tour suivant est commencé et l'expression de la condition est testée immédiatement.

Ces deux mots sont des **mots clés** .

Nous allons maintenant vous montrer deux exemples simples pour illustrer le fonctionnement des deux instructions. Regardez le code dans l'éditeur. Exécutez le programme et analysez la sortie. Modifiez le code et testez.

```
# break - example
```

```
print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")
# continue - example
print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Les déclarations `break` and `continue` : plus d'exemples

Revenons à notre programme qui reconnaît le plus grand parmi les nombres saisis. Nous le convertirons deux fois, en utilisant les instructions `break`et `continue`.

Analysez le code et jugez si et comment vous utiliseriez l'un ou l'autre.

La `break`variante va ici:

```
largestNumber = -99999999
counter = 0

while True:
    number = int(input("Enter a number or type -1 to end program: "))
    if number == -1:
        break
    counter += 1
    if number > largestNumber:
        largestNumber = number

if counter != 0:
    print("The largest number is", largestNumber)
else:
    print("You haven't entered any number.")
```

Exécutez-le, testez-le et expérimentez-le.

Et maintenant la `continue`variante:

```
largestNumber = -99999999
counter = 0

number = int(input("Enter a number or type -1 to end program: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largestNumber:
        largestNumber = number
    number = int(input("Enter a number or type -1 to end program: "))

if counter:
    print("The largest number is", largestNumber)
else:
    print("You haven't entered any number.")
```

Encore une fois - exécutez-le, testez-le et expérimentez-le.

LABORATOIRE

Temps estimé

10 minutes

Niveau de difficulté

Facile

Objectifs

- Familiarisez l'étudiant avec:
- • utiliser l'instruction `break` en boucles;
- • reflétant des situations réelles dans le code informatique.

Scénario

L'instruction `break` est utilisée pour quitter / terminer une boucle.

Concevez un programme qui utilise une boucle `while` et demande continuellement à l'utilisateur d'entrer un mot à moins que l'utilisateur n'entre `"chupacabra"` comme mot de sortie secret, auquel cas le message `"Vous avez réussi à quitter la boucle"`. Doit être imprimé à l'écran et la boucle doit se terminer.

N'imprimez aucun des mots saisis par l'utilisateur. Utilisez le concept d'exécution conditionnelle et l'instruction `break`.

LABORATOIRE

Temps estimé

10-15 minutes

Niveau de difficulté

Facile

Objectifs

Familiarisez l'étudiant avec:

- utiliser l' `continue`instruction en boucles;
- reflétant des situations réelles dans le code informatique.

Scénario

L' `continue`instruction est utilisée pour ignorer le bloc actuel et passer à l'itération suivante, sans exécuter les instructions à l'intérieur de la boucle.

Il peut être utilisé avec les boucles `while` et `for` .

Votre tâche ici est très spéciale: vous devez concevoir un voyelle! Écrivez un programme qui utilise:

- une `for`boucle;
- le concept d'exécution conditionnelle (*if-elif-else*)
- la `continue`déclaration.

Votre programme doit:

- demander à l'utilisateur de saisir un mot;
- utiliser `userWord = userWord.upper()`pour convertir le mot entré par l'utilisateur en majuscules; nous parlerons **des méthodes** dites de **chaîne** et de la `upper()`méthode très bientôt - ne vous inquiétez pas;
- utiliser l'exécution conditionnelle et l' `continue`instruction pour "manger" les voyelles suivantes *A* , *E* , *I* , *O* , *U* du mot entré;
- imprimer les lettres non consommées à l'écran, chacune d'elles sur une ligne distincte.

Testez votre programme avec les données que nous vous avons fournies.

```
# Prompt the user to enter a word

# and assign it to the userWord variable.



for letter in userWord:

    # Complete the body of the for loop.
```

Données de test

Exemple d'entrée: `Gregory`

Production attendue:

```
G
R
G
R
Y
```

Exemple d'entrée: `abstemious`

Production attendue:

```
B
S
T
M
S
```

Exemple d'entrée: `IOUEA`

Production attendue:

```
LABORATOIRE
```

Temps estimé

5-10 minutes

Niveau de difficulté

Facile

Objectifs

Familiarisez l'étudiant avec:

- utiliser l' `continue`instruction en boucles;
- la modification et la mise à niveau du code existant;
- reflétant des situations réelles dans le code informatique.

Scénario

Votre tâche ici est encore plus spéciale qu'auparavant: vous devez repenser le (laid) mangeur de voyelles du laboratoire précédent (3.1.2.10) et créer un meilleur (joli) mangeur de voyelles amélioré! Écrivez un programme qui utilise:

- une `for`boucle;
- le concept d'exécution conditionnelle (*if-elif-else*)
- la `continue`déclaration.

Votre programme doit:

- demander à l'utilisateur de saisir un mot;
- utiliser `userWord = userWord.upper()`pour convertir le mot entré par l'utilisateur en majuscules; nous parlerons **des méthodes** dites de **chaîne** et de la `upper()`méthode très bientôt - ne vous inquiétez pas;
- utiliser l'exécution conditionnelle et l' `continue`instruction pour "manger" les voyelles suivantes *A* , *E* , *I* , *O* , *U* du mot entré;
- attribuez les lettres non consommées à la `wordWithoutVovels`variable et imprimez la variable à l'écran.

Regardez le code dans l'éditeur. Nous lui avons créé `wordWithoutVovels`et attribué une chaîne vide. Utilisez l'opération de concaténation pour demander à Python de combiner les lettres sélectionnées en une chaîne plus longue lors des tours de boucle suivants, et affectez-la à la `wordWithoutVovels`variable.

Testez votre programme avec les données que nous vous avons fournies.

```
wordWithoutVovels = ""


# Prompt the user to enter a word

# and assign it to the userWord variable



for letter in userWord:

    # Complete the body of the loop.


# Print the word assigned to wordWithoutVowels.
```

Données de test

Exemple d'entrée: `Gregory`

Production attendue:

`GRGRY`

Exemple d'entrée: `abstemious`

Production attendue:

`BSTMS`

Exemple d'entrée: `IOUEA`

Production attendue:

La boucle while et la branche else

Les deux boucles `while` et `for` ont une caractéristique intéressante (et rarement utilisée).

Nous allons vous montrer comment cela fonctionne - essayez de juger par vous-même si c'est utilisable et si vous pouvez vivre sans ou non.

En d'autres termes, essayez de vous convaincre si la fonctionnalité est précieuse et utile, ou s'il s'agit simplement de sucre syntaxique.

Jetez un œil à l'extrait dans l'éditeur. Il y a quelque chose d'étrange à la fin - le `else` mot - clé.

```
i = 1

while i < 5:

    print(i)

    i += 1

else:

    print("else:", i)
```

Comme vous vous en doutez, les **boucles peuvent aussi avoir la branche `else`, comme l'art `if`** .

La branche `else` de la boucle est **toujours exécutée une fois, que la boucle soit entrée dans son corps ou non** .

Pouvez-vous deviner la sortie ? Exécutez le programme pour vérifier si vous aviez raison.

Modifiez un peu l'extrait de code afin que la boucle n'ait aucune chance d'exécuter son corps une seule fois:

```
i = 5

while i < 5:

    print(i)

    i += 1

else:

    print("else:", i)
```

La `while` condition est `False` au début - pouvez-vous le voir?

Exécutez et testez le programme et vérifiez si la branche `else` a été exécutée ou non.

La boucle `for` et la branche `else`

Les boucles `for` se comportent un peu différemment - jetez un œil à l'extrait dans l'éditeur et exécutez-le.

```
for i in range(5):

    print(i)

else:
```

```
    print("else:", i)
```

La sortie peut être un peu surprenante.

La variable `i` conserve sa dernière valeur.

Modifiez un peu le code pour effectuer une autre expérience.

```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

Pouvez-vous deviner la sortie?

Le corps de la boucle ne sera pas exécuté ici du tout. Remarque : nous avons attribué la variable `i` avant la boucle.

Exécutez le programme et vérifiez sa sortie.

Lorsque le corps de la boucle n'est pas exécuté, la variable de contrôle conserve la valeur qu'elle avait avant la boucle.

Remarque : **si la variable de contrôle n'existe pas avant le début de la boucle, elle n'existera pas lorsque l'exécution atteindra la branche** `else`.

Que pensez-vous de cette variante de `else` ?

Nous allons maintenant vous parler d'autres types de variables. Nos variables actuelles ne peuvent **stocker** qu'une **seule valeur à la fois**, mais il existe des variables qui peuvent faire beaucoup plus - elles peuvent **stocker autant de valeurs que vous le souhaitez** .

LABORATOIRE

Temps estimé

20-30 minutes

Niveau de difficulté

Moyen

Objectifs

Familiarisez l'étudiant avec:

- en utilisant la `while`boucle;
- trouver la bonne mise en œuvre des règles définies verbalement;

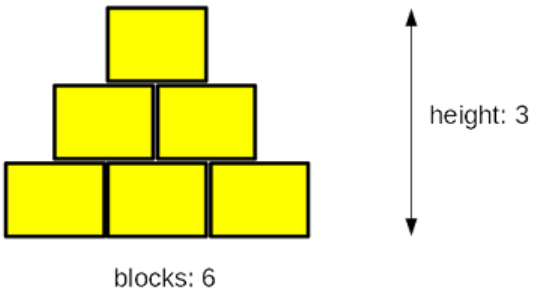
- reflétant des situations réelles dans le code informatique.

Scénario

Écoutez cette histoire: un garçon et son père, un programmeur informatique, jouent avec des blocs de bois. Ils construisent une pyramide.

Leur pyramide est un peu bizarre, car c'est en fait un mur en forme de pyramide - c'est plat. La pyramide est empilée selon un principe simple: chaque couche inférieure contient un bloc de plus que la couche supérieure.

La figure illustre la règle utilisée par les constructeurs:



Votre tâche consiste à écrire un programme qui lit le nombre de blocs dont disposent les générateurs et affiche la hauteur de la pyramide qui peut être construite à l'aide de ces blocs.

Remarque: la hauteur est mesurée par le nombre de **couches entièrement terminées** - si les constructeurs n'ont pas un nombre suffisant de blocs et ne peuvent pas terminer la couche suivante, ils terminent leur travail immédiatement.

Testez votre code à l'aide des données que nous avons fournies.

```
blocks = int(input("Enter the number of blocks: "))

#

# Write your code here.

#

print("The height of the pyramid:", height)
```

Données de test

Exemple d'entrée: `6`

Production attendue: `The height of the pyramid: 3`

Exemple d'entrée: `20`

Production attendue: `The height of the pyramid: 5`

Exemple d'entrée: `1000`

Production attendue: `The height of the pyramid: 44`

Exemple d'entrée: `2`

Production attendue: `The height of the pyramid: 1`

LABORATOIRE

Temps estimé

20 minutes

Niveau de difficulté

Moyen

Objectifs

Familiarisez l'étudiant avec:

- en utilisant la `while`boucle;
- conversion de boucles définies verbalement en code Python réel.

Scénario

En 1937, un mathématicien allemand nommé Lothar Collatz a formulé une hypothèse intrigante (elle n'est toujours pas prouvée) qui peut être décrite de la manière suivante:

1. prendre n'importe quel nombre entier non négatif et non nul et le nommer `c0`;
2. s'il est pair, évaluez un nouveau `c0`comme `c0 ÷ 2`;
3. sinon, si c'est étrange, évaluez un nouveau `c0`comme `3 × c0 + 1`;
4. si `c0 ≠ 1`, passez au point 2.

L'hypothèse dit que quelle que soit la valeur initiale de `c0`, elle ira toujours à 1.

Bien sûr, c'est une tâche extrêmement complexe d'utiliser un ordinateur afin de prouver l'hypothèse pour tout nombre naturel (cela peut même nécessiter de l'intelligence artificielle), mais vous pouvez utiliser Python pour vérifier certains nombres individuels. Peut-être trouverez-vous même celui qui réfuterait l'hypothèse.

Écrivez un programme qui lit un nombre naturel et exécute les étapes ci-dessus tant qu'il `c0`reste différent de 1. Nous voulons également que vous comptiez les étapes nécessaires pour atteindre l'objectif. Votre code devrait également afficher toutes les valeurs intermédiaires de `c0`.

Astuce: la partie la plus importante du problème est de savoir comment transformer l'idée de Collatz en `while`boucle - c'est la clé du succès.

Testez votre code à l'aide des données que nous avons fournies.

Données de test

Exemple d'entrée: `15`

Production attendue:

```
46
23
70
```

```
35
106
53
160
80
40
20
10
5
16
8
4
2
1
steps = 17
```

Exemple d'entrée: `16`

Production attendue:

```
8
4
2
1
steps = 4
```

Exemple d'entrée: `1023`

Production attendue:

```
3070
1535
4606
2303
6910
3455
10366
5183
15550
```

7775

23326

11663

34990

17495

52486

26243

78730

39365

118096

59048

29524

14762

7381

22144

11072

5536

2768

1384

692

346

173

520

260

130

65

196

98

49

148

74

37

112

56

28

14

```
7
22
11
34
17
52
26
13
40
20
10
5
16
8
4
2
1
steps = 62
```

Points clés à retenir

1. Il existe deux types de boucles en Python: `while` et `for`:
   - la `while` boucle exécute une instruction ou un ensemble d'instructions tant qu'une condition booléenne spécifiée est vraie, par exemple:

     ```
     # Example 1
     while True:
         print("Stuck in an infinite loop.")

     # Example 2
     counter = 5
     while counter > 2:
         print(counter)
         counter -= 1
     ```

   - la `for` boucle exécute plusieurs fois un ensemble d'instructions; il est utilisé pour parcourir une séquence (par exemple, une liste, un dictionnaire, un tuple ou un ensemble - vous en apprendrez bientôt plus) ou d'autres objets qui sont itérables (par exemple, des chaînes). Vous pouvez utiliser la `for` boucle pour parcourir une séquence de nombres à l'aide de la `range` fonction intégrée. Regardez les exemples ci-dessous:

     ```
     # Example 1
     word = "Python"
     for letter in word:
         print(letter, end="*")

     # Example 2
     for i in range(1, 10):
         if i % 2 == 0:
     ```

```
        print(i)
```

2. Vous pouvez utiliser les instructions `break` et `continue` pour modifier le flux d'une boucle:

- Vous utilisez `break` pour quitter une boucle, par exemple:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

- Vous utilisez `continue` pour ignorer l'itération actuelle et continuer avec l'itération suivante, par exemple:

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

3. Les boucles `while` et `for` peuvent également avoir une `else` clause en Python. La `else` clause s'exécute une fois que la boucle a terminé son exécution tant qu'elle n'a pas été terminée par `break`, par exemple:

```
n = 0

while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")

print()

for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. La `range()` fonction génère une séquence de nombres. Il accepte des entiers et renvoie des objets de plage. La syntaxe de `range()` ressemble à:, `range(start, stop, step)` où:

- `start` est un paramètre facultatif spécifiant le numéro de départ de la séquence (`0` par défaut)
- `stop` est un paramètre facultatif spécifiant la fin de la séquence générée (il n'est pas inclus),
- et `step` est un paramètre facultatif spécifiant la différence entre les nombres dans la séquence (`1` par défaut.)

Exemple de code:

```
for i in range(3):
    print(i, end=" ") # outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ") # outputs: 6, 4, 2
```

Points clés à retenir: suite

Exercice 1

Créez une `for` boucle qui compte de 0 à 10 et imprime des nombres impairs à l'écran. Utilisez le squelette ci-dessous:

```
for i in range(1, 11):
    # line of code
        # line of code
```

Vérifier

Exercice 2

Créez une `while` boucle qui compte de 0 à 10 et imprime des nombres impairs à l'écran. Utilisez le squelette ci-dessous:

```
x = 1
while x < 11:
    # line of code
        # line of code
    # line of code
```

Vérifier

Exercice 3

Créez un programme avec une `for` boucle et une `break` instruction. Le programme doit parcourir les caractères d'une adresse e-mail, quitter la boucle lorsqu'il atteint le `@` symbole et imprimer la partie précédente `@` sur une seule ligne. Utilisez le squelette ci-dessous:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        # line of code
    # line of code
```

Vérifier

Exercice 4

Créez un programme avec une `for` boucle et une `continue` instruction. Le programme doit parcourir une chaîne de chiffres, remplacer chacun `0` par `x` et imprimer la chaîne modifiée à l'écran. Utilisez le squelette ci-dessous:

```
for digit in "0165031806510":
    if digit == "0":
        # line of code
        # line of code
    # line of code
```

Vérifier

Exercice 5

Quelle est la sortie du code suivant?

```
n = 3


while n > 0:
    print(n + 1)
    n -= 1
else:
    print(n)
```

Vérifier

Exercice 6

Quelle est la sortie du code suivant?

```
n = range(4)


for num in n:
    print(num - 1)
else:
    print(num)
```

Vérifier

Exercice 7

Quelle est la sortie du code suivant?

```
for i in range(0, 6, 3):
    print(i)
```

Vérifier

Logique informatique

Avez-vous remarqué que les conditions que nous avons utilisées jusqu'à présent ont été très simples, pour ne pas dire assez primitives? Les conditions que nous utilisons dans la vie réelle sont beaucoup plus complexes. Regardons cette phrase:

Si nous avons du temps libre `and` qu'il fait beau, nous irons nous promener.

Nous avons utilisé la conjonction et, ce qui signifie que se promener dépend de la satisfaction simultanée de ces deux conditions. Dans le langage de la logique, une telle connexion de conditions est appelée **conjonction**. Et maintenant un autre exemple:

Si vous êtes au centre commercial `or` que je suis au centre commercial, l'un de nous achètera un cadeau pour maman.

L'apparition du mot ou signifie que l'achat dépend d'au moins une de ces conditions. En logique, un tel composé est appelé une **disjonction**.

Il est clair que Python doit avoir des opérateurs pour construire des conjonctions et des disjonctions. Sans eux, le pouvoir expressif de la langue serait considérablement affaibli. Ils sont appelés **opérateurs logiques**.

and

Un opérateur de conjonction logique en Python est le mot et. Il s'agit d'un **opérateur binaire avec une priorité inférieure à celle exprimée par les opérateurs de comparaison**. Il nous permet de coder des conditions complexes sans utiliser de parenthèses comme celle-ci:

```
counter > 0 and value == 100
```

Le résultat fourni par l'opérateur `and` peut être déterminé sur la base de la table de vérité.

Si nous considérons la conjonction de `A and B`, l'ensemble des valeurs possibles des arguments et des valeurs correspondantes de la conjonction se présente comme suit:

Argument A	Argument B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

or

Un opérateur de disjonction est le mot `or`. C'est un opérateur binaire avec une priorité inférieure à `and` (tout comme `+` par rapport à `*`). Sa table de vérité est la suivante:

Argument A	Argument B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

not

De plus, il existe un autre opérateur qui peut être appliqué pour construire des conditions. C'est un **opérateur unaire effectuant une négation logique**. Son fonctionnement est simple: il transforme la vérité en mensonge et le mensonge en vérité.

Cet opérateur est écrit comme le mot `not`, et **sa priorité est très élevée: la même chose que les unaires** `+` **et** `-`. Sa table de vérité est simple:

Argument	not Argument
False	True

`True`	`False`

Expressions logiques

Créons une variable nommée `var` et affectons-lui 1. Les conditions suivantes sont **équivalentes par paire**:

```
print(var > 0)
```

```
print(not (var <= 0))
print(var != 0)
```

```
print(not (var == 0))
```

Vous connaissez peut-être les lois de De Morgan. Ils disent ça:

La négation d'une conjonction est la disjonction des négations.

La négation d'une disjonction est la conjonction des négations.

Écrivons la même chose en utilisant Python:

```
not (p and q) == (not p) or (not q)
```

```
not (p or q) == (not p) and (not q)
```

Notez comment les parenthèses ont été utilisées pour coder les expressions - nous les y avons mises pour améliorer la lisibilité.

Nous devons ajouter qu'aucun de ces opérateurs à deux arguments ne peut être utilisé sous la forme abrégée appelée `op =`. Cette exception mérite d'être rappelée.

Valeurs logiques vs bits simples

Les opérateurs logiques prennent leurs arguments dans leur ensemble, quel que soit le nombre de bits qu'ils contiennent. Les opérateurs ne connaissent que la valeur: zéro (lorsque tous les bits sont réinitialisés) signifie `False`; non nul (lorsqu'au moins un bit est défini) signifie `True`.

Le résultat de leurs opérations est l'une de ces valeurs: `False` ou `True`. Cela signifie que cet extrait de code affectera la valeur `True` à la variable `j` si `i` n'est pas zéro; sinon, ce sera `False`.

```
i = 1
```

```
j = not not i
```

Opérateurs au niveau du bit

Cependant, il existe quatre opérateurs qui vous permettent de **manipuler des bits de données uniques**. Ils sont appelés **opérateurs au niveau du bit**.

Ils couvrent toutes les opérations que nous avons mentionnées précédemment dans le contexte logique, et un opérateur supplémentaire. Il s'agit de l'opérateur `xor` (comme dans **ou exclusif**), et est noté ^ (caret).

Voici tous :

- `&` (ampersand) - bitwise conjunction;
- `|` (bar) - bitwise disjunction;
- `~` (tilde) - bitwise negation;
- `^` (caret) - bitwise exclusive or (xor).

opérateurs au niveau du bit (&, |, and ^)

Arg A	Arg B	Arg B & Arg B	Arg A | Arg B	Arg A ^ Arg B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

opérateurs au niveau du bit (~)

Arg	~Arg
0	1
1	0

Rendons les choses plus faciles:

• `&` requiert exactement deux `1` pour fournir `1` comme résultat;

• `|` nécessite au moins un `1` pour fournir `1` comme résultat;

• `^` requiert exactement un `1` pour fournir `1` comme résultat.

Ajoutons une remarque importante: les arguments de ces opérateurs **doivent être des entiers**; nous ne devons pas utiliser de flotteurs ici.

La différence de fonctionnement des opérateurs logiques et binaires est importante: **les opérateurs logiques ne pénètrent pas dans le niveau binaire de son argument**. Ils ne sont intéressés que par la valeur entière finale.

Les opérateurs au niveau du bit sont plus stricts: ils traitent **chaque bit séparément**. Si nous supposons que la variable entière occupe 64 bits (ce qui est courant dans les systèmes informatiques modernes), vous pouvez imaginer l'opération au niveau du bit comme une évaluation 64 fois de l'opérateur logique pour chaque paire de bits des arguments. Cette analogie est évidemment imparfaite, car dans le monde réel, toutes ces 64 opérations sont effectuées en même temps (simultanément).

Nous allons maintenant vous montrer un exemple de la différence de fonctionnement entre les opérations logiques et binaires. Supposons que les affectations suivantes ont été effectuées:

```
i = 15
```

```
j = 22
```

Si nous supposons que les entiers sont stockés sur 32 bits, l'image bit à bit des deux variables sera la suivante:

```
i: 00000000000000000000000000001111
```

```
j: 00000000000000000000000000010110
```

La mission est donnée:

```
log = i and j
```

Nous avons affaire à une conjonction logique ici. Trouvons le cours des calculs. Les deux variables `i`et `j`ne sont pas des zéros, seront donc réputées représenter `True`. En consultant la table de vérité pour l' `and`opérateur, nous pouvons voir que le résultat sera `True`. Aucune autre opération n'est effectuée.

```
log: True
```

Maintenant, l'opération au niveau du bit - la voici:

```
bit = i & j
```

L' `&`opérateur fonctionnera avec chaque paire de bits correspondants séparément, produisant les valeurs des bits pertinents du résultat. Par conséquent, le résultat sera le suivant:

`i`	`00000000000000000000000000001111`
`j`	`00000000000000000000000000010110`
`bit = i & j`	`00000000000000000000000000000110`

Ces bits correspondent à la valeur entière de six.

Regardons maintenant les opérateurs de négation. D'abord la logique:

```
logneg = not i
```

La `logneg`variable sera définie sur `False`- rien de plus ne doit être fait.

La négation au niveau du bit se présente comme suit:

```
bitneg = ~i
```

Cela peut être un peu surprenant: la `bitneg`valeur de la variable est `-16`. Cela peut sembler étrange, mais pas du tout. Si vous souhaitez en savoir plus, vous devriez vérifier le système de nombres binaires et les règles régissant les nombres de complément à deux.

`i`	`00000000000000000000000000001111`
`bitneg = ~i`	`11111111111111111111111111110000`

Chacun de ces opérateurs à deux arguments peut être utilisé sous **forme abrégée** . Voici les exemples de leurs notations équivalentes:

`x = x & y`	X & = y
`x = x | y`	X | = y
`x = x ^ y`	X ^ = y

Comment traitons-nous les bits simples?

Nous allons maintenant vous montrer à quoi vous pouvez utiliser des opérateurs au niveau du bit. Imaginez que vous êtes un développeur obligé d'écrire un élément important d'un système d'exploitation. On vous a dit que vous êtes autorisé à utiliser une variable affectée de la manière suivante:

```
flagRegister = 0x1234
```

La variable stocke les informations sur divers aspects du fonctionnement du système. **Chaque bit de la variable stocke une valeur oui / non** . On vous a également dit qu'un seul de ces bits était le vôtre - le troisième (rappelez-vous que les bits sont numérotés à partir de zéro et que le numéro de bit zéro est le plus bas, tandis que le plus élevé est le numéro 31). Les bits restants ne sont pas autorisés à changer, car ils sont destinés à stocker d'autres données. Voici votre morceau marqué de la lettre `x`:

```
flagRegister = 00000000000000000000000000000x000
```

Vous pourriez être confronté aux tâches suivantes:

1. **Vérifiez l'état de votre bit** - vous voulez connaître la valeur de votre bit; la comparaison de la variable entière à zéro ne fera rien, car les bits restants peuvent avoir des valeurs complètement imprévisibles, mais vous pouvez utiliser la propriété de conjonction suivante:

```
x & 1 = x
x & 0 = 0
```

Si vous appliquez l' `&`opération à la `flagRegister`variable avec l'image bit suivante:

```
00000000000000000000000000001000
```

(notez la position `1`de votre bit) comme résultat, vous obtenez l'une des chaînes de bits suivantes:

- `00000000000000000000000000001000` si votre bit était réglé sur `1`
- `00000000000000000000000000000000` si votre bit a été réinitialisé `0`

Une telle séquence de zéros et de uns, dont la tâche consiste à saisir la valeur ou à modifier les bits sélectionnés, est appelée **masque de bits** .

Construisons un masque de bits pour détecter l'état de votre bit. Il devrait pointer vers **le troisième bit** . Ce bit a le poids de . Un masque approprié pourrait être créé par la déclaration suivante:$2^3 = 8$

```
theMask = 8
```

Vous pouvez également faire une séquence d'instructions en fonction de l'état de votre bit i le voici:

```
if flagRegister & theMask:
    # my bit is set
else:
```

```
    # my bit is reset
```

2. **Réinitialisez votre bit** - vous attribuez un zéro au bit tandis que tous les autres bits doivent rester inchangés; utilisons la même propriété de la conjonction qu'avant, mais utilisons un masque légèrement différent - exactement comme ci-dessous:

```
11111111111111111111111111110111
```

Notez que le masque a été créé à la suite de la négation de tous les bits de `theMask`variable. La réinitialisation du bit est simple et ressemble à ceci (choisissez celui que vous aimez le plus):

```
flagRegister = flagRegister & ~theMask
```

```
flagregister &= ~theMask
```

3. **Définissez votre bit** - vous affectez un `1`à votre bit, tandis que tous les bits restants doivent rester inchangés; utilisez la propriété de disjonction suivante:

```
x | 1 = 1
x | 0 = x
```

Vous êtes maintenant prêt à régler votre bit avec l'une des instructions suivantes:

```
flagRegister = flagRegister | theMask
```

```
flagRegister |= theMask
```

4. **Niez votre bit** - vous remplacez un `1`par un `0`et un `0`par un `1`. Vous pouvez utiliser une propriété intéressante de l' `xor`opérateur:

```
x ^ 1 = ~x
x ^ 0 = x
```

et niez votre bit avec les instructions suivantes:

```
flagRegister = flagRegister ^ theMask
```

```
flagRegister ^= theMask
```

Décalage gauche binaire et décalage droit binaire

Python propose encore une autre opération relative aux bits simples: le **décalage** . Ceci est appliqué uniquement aux valeurs **entières** , et vous ne devez pas utiliser de flottants comme arguments pour cela.

Vous appliquez déjà cette opération très souvent et assez inconsciemment. Comment multipliez-vous un nombre par dix? Regarde:

```
12345 × 10 = 123450
```

Comme vous pouvez le voir, **multiplier par dix est en fait un décalage** de tous les chiffres vers la gauche et combler le vide qui en résulte avec zéro.

Division par dix? Regarde:

`12340 ÷ 10 = 1234`

Diviser par dix n'est rien d'autre que déplacer les chiffres vers la droite.

Le même type d'opération est effectué par l'ordinateur, mais avec une différence: comme deux est la base des nombres binaires (pas 10), **décaler une valeur d'un bit vers la gauche correspond donc à la multiplier par deux** ; respectivement, **décaler un bit vers la droite revient à diviser par deux** (notez que le bit le plus à droite est perdu).

Les **opérateurs de décalage** en Python sont une paire de **digraphes** : `<<`et `>>`, suggérant clairement dans quelle direction le décalage va agir.

```
value << bits
value >> bits
```

L'argument gauche de ces opérateurs est une valeur entière dont les bits sont décalés. L'argument de droite détermine la taille du décalage.

Cela montre que cette opération n'est certainement pas commutative.

La priorité de ces opérateurs est très élevée. Vous les verrez dans le tableau des priorités mis à jour, que nous vous montrerons à la fin de cette section.

Jetez un œil aux changements dans la fenêtre de l'éditeur.

```
var = 17

varRight = var >> 1

varLeft = var << 2

print(var, varLeft, varRight)
```

L' `print()`invocation finale produit la sortie suivante:

```
17 68 8
```

Remarque:

- `17 // 2`→ `8`(le décalage d'un bit vers la droite équivaut à une division entière par deux)
- `17 * 4`→ `68`(le décalage vers la gauche de deux bits équivaut à une multiplication par quatre)

Et voici le **tableau des priorités mis à jour** , contenant tous les opérateurs introduits jusqu'à présent:

Priorité	Opérateur	
1	`~`, `+`,`-`	unaire
2	`**`	
3	`*`, `/`, `//`,`%`	
4	`+`, `-`	binaire

5	`<<`, `>>`
6	`<`, `<=`, `>`,`>=`
7	`==`, `!=`
8	`&`
9	`|`
dix	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `>>=`,`<<=`

Points clés à retenir

1. Python prend en charge les opérateurs logiques suivants:

- `and` → si les deux opérandes sont vrais, la condition est vraie, par exemple, `(True and True)` est `True`,
- `or` → si l'un des opérandes est vrai, la condition est vraie, par exemple, `(True or False)` est `True`,
- `not` → retourne faux si le résultat est vrai, et retourne vrai si le résultat est faux, par exemple, `not True` est `False`.

2. Vous pouvez utiliser des opérateurs au niveau du bit pour manipuler des bits de données uniques. Les exemples de données suivants:

- `x = 15`, qui est `0000 1111` en binaire,
- `y = 16`, qui est `0001 0000` en binaire.

sera utilisé pour illustrer la signification des opérateurs au niveau du bit en Python. Analysez les exemples ci-dessous:

- `&` fait au niveau du *bit et* , par exemple `x & y = 0`, qui est `0000 0000` en binaire,
- `|` fait un *bit ou* , par exemple `x | y = 31`, qui est `0001 1111` en binaire,
- `~` fait un *pas au niveau du bit* , par exemple `~ x = 240`, qui est `1111 0000` en binaire,
- `^` fait un *xor au niveau du bit* , par exemple `x ^ y = 31`, qui est `0001 1111` en binaire,
- `>>` fait un *décalage à droite au niveau du bit* , par exemple `y >> 1 = 8`, qui est `0000 1000` en binaire,
- `<<` fait un *décalage à gauche au niveau du bit* , par exemple `y << 3 =` , qui est `1000 0000` en binaire,

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
x = 1
y = 0

z = ((x == y) and (x == y)) or not(x == y)
print(not(z))
```

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
x = 4
y = 1

a = x & y
b = x | y
c = ~x
d = x ^ 5
e = x >> 2
f = x << 2

print(a, b, c, d, e, f)
```

Pourquoi avons-nous besoin de listes?

Il peut arriver que vous deviez lire, stocker, traiter et enfin imprimer des dizaines, voire des centaines, voire des milliers de nombres. Et alors? Avez-vous besoin de créer une variable distincte pour chaque valeur? Aurez-vous à passer de longues heures à rédiger des déclarations comme celle ci-dessous?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
:
:
```

Si vous ne pensez pas que c'est une tâche compliquée, alors prenez un morceau de papier et écrivez un programme qui:

- lit cinq nombres,
- les imprime dans l'ordre du plus petit au plus grand (NB, ce type de traitement est appelé **tri**).

Vous devriez constater que vous n'avez même pas assez de papier pour terminer la tâche.

Jusqu'à présent, vous avez appris à déclarer des variables capables de stocker exactement une valeur donnée à la fois. Ces variables sont parfois appelées **scalaires** par analogie avec les mathématiques. Toutes les variables que vous avez utilisées jusqu'à présent sont en fait des scalaires.

Pensez à la façon dont il serait commode de déclarer une variable qui pourrait **stocker plus d'une valeur** . Par exemple, cent, mille ou même dix mille. Ce serait toujours une seule et même variable, mais très large et vaste. Cela semble attrayant? Peut-être, mais comment gérerait-il un tel conteneur plein de valeurs différentes? Comment choisirait-il celui dont vous avez besoin?

Et si vous pouviez simplement les numéroter? Et puis dites: *donnez-moi la valeur numéro 2; attribuer le numéro de valeur 15; augmentez le nombre de valeur 10000* .

Nous allons vous montrer comment déclarer de telles **variables à valeurs multiples** . Nous allons le faire avec l'exemple que nous venons de suggérer. Nous allons écrire un **programme qui trie une séquence de nombres** . Nous ne serons pas particulièrement ambitieux - nous supposerons qu'il y a exactement cinq chiffres.

Créons une variable appelée `numbers`; il est attribué avec non pas un seul numéro, mais il est rempli d'une liste composée de cinq valeurs (remarque: la **liste commence par un crochet ouvert et se termine par un crochet fermé** ; l'espace entre les crochets est rempli par cinq chiffres séparés par des virgules).

```
numbers = [10, 5, 7, 2, 1]
```

Disons la même chose en utilisant une terminologie adéquate: `numbers` **est une liste composée de cinq valeurs, toutes des nombres** . Nous pouvons également dire que cette déclaration crée une liste de longueur égale à cinq (car il y a cinq éléments à l'intérieur).

Les éléments d'une liste **peuvent avoir différents types** . Certains d'entre eux peuvent être des nombres entiers, d'autres des flottants et d'autres encore peuvent être des listes.

Python a adopté une convention stipulant que les éléments d'une liste sont **toujours numérotés à partir de zéro** . Cela signifie que l'élément stocké au début de la liste aura le numéro zéro. Puisqu'il y a cinq éléments dans notre liste, le dernier d'entre eux se voit attribuer le numéro quatre. N'oublie pas ça.

Vous vous y habituerez bientôt et cela deviendra une seconde nature.

Avant d'aller plus loin dans notre discussion, nous devons déclarer ce qui suit: notre **liste est une collection d'éléments, mais chaque élément est un scalaire** .

Listes d'indexation

Comment changez-vous la valeur d'un élément choisi dans la liste?

Nous allons **attribuer une nouvelle valeur** `111` **au premier élément** dans la liste. Nous le faisons de cette façon:

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # printing original list content


numbers[0] = 111

print("New list content: ", numbers) # current list content
```

Et maintenant, nous voulons que **la valeur du cinquième élément soit copiée dans le deuxième élément** - pouvez-vous deviner comment le faire?

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # printing original list content


numbers[0] = 111

print("\nPrevious list content:", numbers) # printing previous list content


numbers[1] = numbers[4] # copying value of the fifth element to the second

print("New list content:", numbers) # printing current list content
```

La valeur entre crochets qui sélectionne un élément de la liste est appelée **index** , tandis que l'opération de sélection d'un élément dans la liste est appelée **indexation** .

Nous allons utiliser la `print()` fonction pour imprimer le contenu de la liste chaque fois que nous apportons les modifications. Cela nous aidera à suivre chaque étape plus attentivement et à voir ce qui se passe après une modification particulière de la liste.

Remarque: tous les indices utilisés jusqu'à présent sont des littéraux. Leurs valeurs sont fixes lors de l'exécution, mais **toute expression peut également être l'index** . Cela ouvre de nombreuses possibilités.

Accéder au contenu de la liste

Chacun des éléments de la liste est accessible séparément. Par exemple, il peut être imprimé:

```
print(numbers[0]) # accessing the list's first element
```

En supposant que toutes les opérations précédentes ont été effectuées avec succès, l'extrait de code sera envoyé `111` à la console.

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # printing original list content
```

```
numbers[0] = 111

print("\nPrevious list content:", numbers) # printing previous list content

numbers[1] = numbers[4] # copying value of the fifth element to the second

print("Previous list content:", numbers) # printing previous list content

print("\nList length:", len(numbers)) # printing the list's length
```

Comme vous pouvez le voir dans l'éditeur, la liste peut également être imprimée dans son ensemble - comme ici:

```
print(numbers) # printing the whole list
```

Comme vous l'avez probablement remarqué auparavant, Python décore la sortie d'une manière qui suggère que toutes les valeurs présentées forment une liste. La sortie de l'extrait d'exemple ci-dessus ressemble à ceci:

```
[111, 1, 7, 2, 1]
```

Le `len ()` fonction

La **longueur d'une liste** peut varier pendant l'exécution. De nouveaux éléments peuvent être ajoutés à la liste, tandis que d'autres peuvent en être supprimés. Cela signifie que la liste est une entité très dynamique.

Si vous souhaitez vérifier la longueur actuelle de la liste, vous pouvez utiliser une fonction nommée `len()`(son nom vient de la *longueur*).

La fonction prend le **nom de** la **liste comme argument et retourne le nombre d'éléments actuellement stockés** dans la liste (en d'autres termes - la longueur de la liste).

Regardez la dernière ligne de code dans l'éditeur, exécutez le programme et vérifiez quelle valeur il va imprimer sur la console. Peux-tu deviner?

Supprimer des éléments d'une liste

N'importe quel élément de la liste peut être **supprimé** à tout moment - cela se fait avec une instruction nommée `del` (delete). Remarque: c'est une **instruction**, pas une fonction.

Vous devez pointer l'élément à supprimer - il disparaîtra de la liste et la longueur de la liste sera réduite d'une unité.

Regardez l'extrait ci-dessous. Pouvez-vous deviner quelle sortie il produira? Exécutez le programme dans l'éditeur et vérifiez.

```
del numbers[1]
```

```
print(len(numbers))
```

```
print(numbers)
```

Vous ne pouvez pas accéder à un élément qui n'existe pas - vous ne pouvez ni obtenir sa valeur ni lui affecter une valeur. Ces deux instructions entraîneront maintenant des erreurs d'exécution:

```
print(numbers[4])
```

`numbers[4] = 1`

Ajoutez l'extrait ci-dessus après la dernière ligne de code dans l'éditeur, exécutez le programme et vérifiez ce qui se passe.

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers) # printing original list content

numbers[0] = 111

print("\nPrevious list content:", numbers) # printing previous list content

numbers[1] = numbers[4] # copying value of the fifth element to the second

print("Previous list content:", numbers) # printing previous list content

print("\nList's length:", len(numbers)) # printing previous list length

###

del numbers[1] # removing the second element from the list

print("New list's length:", len(numbers)) # printing new list length

print("\nNew list content:", numbers) # printing current list content

###
```

Remarque: nous avons supprimé l'un des éléments de la liste - il n'y a maintenant que quatre éléments dans la liste. Cela signifie que l'élément numéro quatre n'existe pas.

Les indices négatifs sont légaux

Cela peut sembler étrange, mais les indices négatifs sont légaux et peuvent être très utiles.

Un élément avec un indice égal à `-1` **est le dernier de la liste**.

`print(numbers[-1])`

L'exemple d'extrait affichera `1`. Exécutez le programme et vérifiez.

```
numbers = [111, 7, 2, 1]

print(numbers[-1])

print(numbers[-2])
```

De même, l'élément avec un indice égal à `-2` est **l'avant-dernier de la liste**.

`print(numbers[-2])`

L'extrait d'exemple sortira `2`.

Le dernier élément accessible de notre liste est le `numbers[-4]` (le premier) - n'essayez pas d'aller plus loin !

LABORATOIRE

Temps estimé

5 minutes

Niveau de difficulté

Très facile

Objectifs

Familiarisez l'étudiant avec:

- utiliser des instructions de base relatives aux listes;
- créer et modifier des listes.

Scénario

Il y avait une fois un chapeau. Le chapeau ne contenait pas de lapin, mais une liste de cinq numéros: `1`, `2`, `3`, `4`et `5`.

Votre tâche consiste à:

- écrire une ligne de code qui invite l'utilisateur à remplacer le numéro du milieu de la liste par un nombre entier entré par l'utilisateur (étape 1)
- écrire une ligne de code qui supprime le dernier élément de la liste (étape 2)
- écrire une ligne de code qui imprime la longueur de la liste existante (étape 3.)

```
hatList = [1, 2, 3, 4, 5]  # This is an existing list of numbers hidden in the hat.

# Step 1: write a line of code that prompts the user
# to replace the middle number with an integer number entered by the user.

# Step 2: write a line of code here that removes the last element from the list.

# Step 3: write a line of code here that prints the length of the existing list.

print(hatList)
```

Prêt pour ce défi?

Fonctions vs méthodes

Une **méthode est un type spécifique de fonction** - elle se comporte comme une fonction et ressemble à une fonction, mais diffère dans la façon dont elle agit et dans son style d'invocation.

Une **fonction n'appartient à aucune donnée** - elle obtient des données, elle peut créer de nouvelles données et elle (généralement) produit un résultat.

Une méthode fait tout cela, mais est également capable de **changer l'état d'une entité sélectionnée** .

Une méthode appartient aux données pour lesquelles elle fonctionne, tandis qu'une fonction appartient à tout le code .

Cela signifie également que l'invocation d'une méthode nécessite une spécification des données à partir desquelles la méthode est invoquée.

Cela peut sembler déroutant ici, mais nous y reviendrons en détail lorsque nous nous plongerons dans la programmation orientée objet.

En général, un appel de fonction typique peut ressembler à ceci:

```
result = function(arg)
```

La fonction prend un argument, fait quelque chose et renvoie un résultat.

Un appel de méthode typique ressemble généralement à ceci:

```
result = data.method(arg)
```

Remarque: le nom de la méthode est précédé du nom des données propriétaires de la méthode. Ensuite, vous ajoutez un **point** , suivi du **nom** de la **méthode** et d'une paire de **parenthèses entourant les arguments** .

La méthode se comportera comme une fonction, mais peut faire quelque chose de plus - elle peut **changer l'état interne des données à** partir desquelles elle a été invoquée.

Vous pouvez vous demander: pourquoi parlons-nous de méthodes, pas de listes?

C'est un problème essentiel en ce moment, car nous allons vous montrer comment ajouter de nouveaux éléments à une liste existante. Cela peut être fait avec des méthodes appartenant à toutes les listes, pas par des fonctions.

Ajout d'éléments à une liste: `append ()` et `insert ()`

Un nouvel élément peut être *collé* à la fin de la liste existante:

```
list.append(value)
```

Une telle opération est effectuée par une méthode nommée `append()`. Il prend la valeur de son argument et le place **à la fin de la liste** qui possède la méthode.

La longueur de la liste augmente alors d'une unité.

La `insert()`méthode est un peu plus intelligente - elle peut ajouter un nouvel élément **à n'importe quel endroit de la liste** , pas seulement à la fin.

```
list.insert(location, value)
```

Il faut deux arguments:

- le premier indique l'emplacement requis de l'élément à insérer; note: tous les éléments existants qui occupent des emplacements à droite du nouvel élément (y compris celui à la position indiquée) sont décalés vers la droite, afin de faire de la place pour le nouvel élément;
- le second est l'élément à insérer.

Regardez le code dans l'éditeur. Découvrez comment nous utilisons les méthodes `append()`et `insert()`. Faites attention à ce qui se passe après utilisation `insert()`: l'ancien premier élément est maintenant le deuxième, le deuxième le troisième, etc.

```
numbers = [111, 7, 2, 1]
print(len(numbers))
```

```
print(numbers)

###

numbers.append(4)

print(len(numbers))
print(numbers)

###

numbers.insert(0, 222)
print(len(numbers))
print(numbers)

#
```

Ajoutez l'extrait de code suivant après la dernière ligne de code dans l'éditeur:
`numbers.insert(1, 333)`
Imprimez le contenu de la liste finale à l'écran et voyez ce qui se passe. L'extrait de code ci-dessus est inséré `333` dans la liste, ce qui en fait le deuxième élément. L'ancien deuxième élément devient le troisième, le troisième le quatrième, etc.

Ajout d'éléments à une liste: suite

Vous pouvez **démarrer la vie d'une liste en la rendant vide** (cela se fait avec une paire de crochets vides), puis en lui ajoutant de nouveaux éléments si nécessaire.

Jetez un œil à l'extrait dans l'éditeur. Essayez de deviner sa sortie après l' `for` exécution de la boucle. Exécutez le programme pour vérifier si vous aviez raison.

```
myList = [] # creating an empty list

for i in range(5):

    myList.append(i + 1)

print(myList)
```

Ce sera une séquence de nombres entiers consécutifs de `1` (vous ajoutez ensuite un à toutes les valeurs ajoutées) à `5`.

Nous avons légèrement modifié l'extrait de code:

```
myList = [] # creating an empty list

for i in range(5):

    myList.insert(0, i + 1)

print(myList)
```

ce qui va se passer maintenant? Exécutez le programme et vérifiez si cette fois vous aviez raison.

Vous devriez obtenir la même séquence, mais dans **l'ordre inverse** (c'est le mérite d'utiliser la `insert()` méthode).

Utilisation des listes

La boucle `for` a une variante très spéciale qui peut **traiter les listes** très efficacement - jetons un œil à cela.

```
myList = [10, 1, 8, 3, 5]
```

```
total = 0

for i in range(len(myList)):

    total += myList[i]

print(total)
```

Supposons que vous souhaitiez **calculer la somme de toutes les valeurs stockées** dans la liste `myList`.

Vous avez besoin d'une variable dont la somme sera stockée et attribuée initialement une valeur de `0` - son nom sera `total`. (Remarque: nous n'allons pas le nommer `sum` car Python utilise le même nom pour l'une de ses fonctions intégrées - `sum ()`. L'utilisation du même nom serait généralement considérée comme une mauvaise pratique.) Ensuite, vous ajoutez à tout cela les éléments de la liste utilisant la boucle `for`. Jetez un œil à l'extrait dans l'éditeur.

Commentons cet exemple:

• la liste se voit attribuer une séquence de cinq valeurs entières;

• la variable `i` prend les valeurs `0`, `1`, `2`, `3` et `4`, puis elle indexe la liste en sélectionnant les éléments suivants: le premier, le deuxième, le troisième, le quatrième et le cinquième;

• chacun de ces éléments est additionné par l'opérateur `+ =` à la variable `total`, donnant le résultat final à la fin de la boucle;

• notez la façon dont la fonction `len()` a été utilisée - elle rend le code indépendant de tout changement possible dans le contenu de la liste.

La deuxième face de la boucle for

Mais la boucle `for` peut faire bien plus. Il peut masquer toutes les actions liées à l'indexation de la liste et fournir tous les éléments de la liste de manière pratique.

Cet extrait modifié montre comment cela fonctionne:

```
myList = [10, 1, 8, 3, 5]

total = 0


for i in myList:

    total += i


print(total)
```

Que se passe t-il ici?

• l'instruction `for` spécifie la variable utilisée pour parcourir la liste (`i` ici) suivie du mot-clé `in` et du nom de la liste en cours de traitement (ici `myList`)

• la variable `i` se voit attribuer les valeurs de tous les éléments de la liste suivante, et le processus se produit autant de fois qu'il y a d'éléments dans la liste;

• cela signifie que vous utilisez la variable `i` comme copie des valeurs des éléments et que vous n'avez pas besoin d'utiliser des indices;

• la fonction `len()` n'est pas non plus nécessaire ici.

Listes en action

Laissons les listes de côté pour un court instant et examinons un problème intrigant.

Imaginez que vous devez réorganiser les éléments d'une liste, c'est-à-dire inverser l'ordre des éléments: les premier et cinquième ainsi que les deuxième et quatrième éléments seront échangés. Le troisième restera intact.

Question: comment échanger les valeurs de deux variables?

Jetez un œil à l'extrait:

```
variable1 = 1

variable2 = 2

variable2 = variable1

variable1 = variable2
```

Si vous faites quelque chose comme ça, **vous perdriez la valeur précédemment stockée** dans `variable2`**.** Changer l'ordre des affectations n'aidera pas. Vous avez besoin d'une **troisième variable qui sert de stockage auxiliaire**.

Voici comment vous pouvez le faire:

```
variable1 = 1

variable2 = 2


auxiliary = variable1

variable1 = variable2

variable2 = auxiliary
```

Python offre un moyen plus pratique de faire l'échange - jetez un œil:

```
variable1 = 1

variable2 = 2

variable1, variable2 = variable2, variable1
```

Clair, efficace et élégant - n'est-ce pas?

Maintenant, vous pouvez facilement **échanger** les éléments de la liste pour **inverser leur ordre :**

```
myList = [10, 1, 8, 3, 5]

myList[0], myList[4] = myList[4], myList[0]
myList[1], myList[3] = myList[3], myList[1]

print(myList)
```

Exécutez l'extrait. Sa sortie devrait ressembler à ceci:

```
[5, 3, 8, 1, 10]
```

Il semble bien avec cinq éléments.

Sera-t-il toujours acceptable avec une liste contenant 100 éléments? Non, ça ne le sera pas.

Pouvez-vous utiliser la boucle `for` pour faire la même chose automatiquement, quelle que soit la longueur de la liste? Oui, vous pouvez.

Voilà comment nous l'avons fait:

```
myList = [10, 1, 8, 3, 5]
length = len(myList)

for i in range(length // 2):
    myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]

print(myList)
```

Remarque:
• nous avons attribué la variable `length` à la liste actuelle (cela rend notre code un peu plus clair et plus court)
• nous avons lancé la boucle `for` pour parcourir son corps `length// 2` fois (cela fonctionne bien pour les listes avec des longueurs paires et impaires, car lorsque la liste contient un nombre impair d'éléments, celle du milieu reste intacte)
• nous avons échangé le ième élément (depuis le début de la liste) avec celui dont l'indice est égal à (`lenght - i - 1`) (à la fin de la liste); dans notre exemple, pour `i` égal à `0`, le (`l - i - 1`) donne `4`; pour `i` égal à 1, cela donne `3` - c'est exactement ce dont nous avions besoin.
Les listes sont extrêmement utiles et vous les rencontrerez très souvent.
LABORATOIRE

Temps estimé

10-15 minutes

Niveau de difficulté

Facile

Objectifs

Familiarisez l'étudiant avec:

- créer et modifier des listes simples;
- en utilisant des méthodes pour modifier les listes.

Scénario

Les Beatles étaient l'un des groupes de musique les plus populaires des années 1960 et le groupe le plus vendu de l'histoire. Certaines personnes les considèrent comme l'acte le plus influent de l'ère du rock. En effet, ils ont été inclus dans la compilation du magazine *Time* des 100 personnes les plus influentes du 20e siècle.

Le groupe a subi de nombreux changements de line-up, culminant en 1962 avec la formation de John Lennon, Paul McCartney, George Harrison et Richard Starkey (mieux connu sous le nom de Ringo Starr).

```
# step 1

print("Step 1:", beatles)
```

```
# step 2

print("Step 2:", beatles)

# step 3

print("Step 3:", beatles)

# step 4

print("Step 4:", beatles)

# step 5

print("Step 5:", beatles)

# testing list legth

print("The Fab", len(beatles))
```

Écrivez un programme qui reflète ces changements et vous permet de vous entraîner avec le concept de listes. Votre tâche consiste à:

- étape 1: créez une liste vide nommée `beatles`;
- étape 2: utiliser la `append()` méthode pour ajouter les membres suivants de la bande à la liste: `John Lennon`, `Paul McCartney`, et `George Harrison`;
- étape 3: utilisez la `for` boucle et la `append()` méthode pour inviter l'utilisateur à ajouter les membres suivants de la bande à la liste:, `Stu Sutcliffe` et `Pete Best`;
- étape 4: utilisez l' `del` instruction pour supprimer `Stu Sutcliffe` et `Pete Best` de la liste;
- étape 5: utilisez la `insert()` méthode pour ajouter `Ringo Starr` au début de la liste.

Au fait, êtes-vous un fan des Beatles?

Points clés à retenir

1. La **liste est un type de données** en Python utilisé pour **stocker plusieurs objets** . Il s'agit d'une collection **ordonnée et modifiable** d'éléments séparés par des virgules entre crochets, par exemple:

```
myList = [1, None, True, "I am a string", 256, 0]
```

2. Les listes peuvent être **indexées et mises à jour** , par exemple:

```
myList = [1, None, True, 'I am a string', 256, 0]

print(myList[3]) # outputs: I am a string

print(myList[-1]) # outputs: 0


myList[1] = '?'

print(myList) # outputs: [1, '?', True, 'I am a string', 256, 0]


myList.insert(0, "first")

myList.append("last")
```

```
print(myList) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']
```

3. Les listes peuvent être **imbriquées** , par exemple: `myList = [1, 'a', ["list", 64, [0, 1], False]]`.

Vous en apprendrez plus sur l'imbrication dans le module 3.1.7 - pour le moment, nous voulons juste que vous soyez conscient que quelque chose comme cela est également possible.

4. Les éléments de liste et les listes peuvent être **supprimés** , par exemple:

```
myList = [1, 2, 3, 4]

del myList[2]

print(myList) # outputs: [1, 2, 4]


del myList # deletes the whole list
```

Encore une fois, vous en apprendrez plus à ce sujet dans le module 3.1.6 - ne vous inquiétez pas. Pour le moment, essayez simplement de tester le code ci-dessus et vérifiez comment sa modification affecte la sortie.

5. Les listes peuvent être **itérées** en utilisant la `for`boucle, par exemple:

```
myList = ["white", "purple", "blue", "yellow", "green"]


for color in myList:
    print(color)
```

6. La `len()` fonction peut être utilisée pour **vérifier la longueur de la liste** , par exemple:

```
myList = ["white", "purple", "blue", "yellow", "green"]

print(len(myList)) # outputs 5


del myList[2]

print(len(myList)) # outputs 4
```

7. Une typique **fonction** ressemble comme suit invocation: `result = function(arg)`, tandis qu'un typique **méthode** ressemble invocation comme ceci: `result = data.method(arg)`.

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, 2, 3, 4, 5]

lst.insert(1, 6)
```

```
del lst[0]

lst.append(1)


print(lst)
```

Vérifier

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, 2, 3, 4, 5]

lst2 = []

add = 0


for number in lst:

    add += number

    lst2.append(add)


print(lst2)
```

Vérifier

Exercice 3

Que se passe-t-il lorsque vous exécutez l'extrait de code suivant?

```
lst = []

del lst

print(lst)
```

Vérifier

Exercice 4

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))
```

Vérifier

Le tri des bulles

Maintenant que vous pouvez jongler efficacement avec les éléments des listes, il est temps d'apprendre à les **trier** . De nombreux algorithmes de tri ont été inventés jusqu'à présent, qui diffèrent beaucoup en termes de vitesse et de complexité. Nous allons vous montrer un algorithme très simple, facile à comprendre, mais malheureusement pas trop efficace non plus. Il est utilisé très rarement, et certainement pas pour les listes volumineuses et étendues.

Disons qu'une liste peut être triée de deux manières:

- croissant (ou plus précisément - non décroissant) - si dans chaque paire d'éléments adjacents, le premier élément n'est pas supérieur au second;
- décroissant (ou plus précisément - non croissant) - si dans chaque paire d'éléments adjacents, le premier élément n'est pas inférieur au second.

Dans les sections suivantes, nous allons trier la liste dans un ordre croissant, afin que les numéros soient classés du plus petit au plus grand.

Voici la liste:

8	dix	6	2	4

Nous allons essayer d'utiliser l'approche suivante: nous allons prendre le premier et le second éléments et les comparer; si nous déterminons qu'ils sont dans le mauvais ordre (c'est-à-dire que le premier est supérieur au second), nous les échangerons; si leur commande est valide, nous ne ferons rien. Un coup d'œil sur notre liste confirme cette dernière - les éléments 01 et 02 sont dans le bon ordre, comme dans `8 < 10`.

Regardez maintenant les deuxième et troisième éléments. Ils sont dans de mauvaises positions. Nous devons les échanger:

8	**6**	**dix**	2	4

Nous allons plus loin et examinons les troisième et quatrième éléments. Encore une fois, ce n'est pas ce que c'est censé être. Nous devons les échanger:

8	6	**2**	**dix**	4

Maintenant, nous vérifions les quatrième et cinquième éléments. Oui, eux aussi sont dans de mauvaises positions. Un autre échange se produit:

8	6	2	**4**	**dix**

Le premier passage dans la liste est déjà terminé. Nous sommes encore loin de terminer notre travail, mais quelque chose de curieux s'est produit entre-temps. L'élément le plus important,, `10` est déjà allé à la fin de la liste. Notez que c'est l' **endroit souhaité** pour cela. Tous les éléments restants forment un gâchis pittoresque, mais celui-ci est déjà en place.

Maintenant, pendant un moment, essayez d'imaginer la liste d'une manière légèrement différente - à savoir, comme ceci:

dix
4
2
6
8

Regardez - `10` est au sommet. On pourrait dire qu'elle flottait du bas vers la surface, tout comme la **bulle d'un verre de champagne** . La méthode de tri tire son nom de la même observation - elle s'appelle un **tri à bulles** .

Maintenant, nous commençons par le deuxième passage dans la liste. Nous examinons les premier et deuxième éléments - un échange est nécessaire:

6	**8**	2	4	dix

Il est temps pour les deuxième et troisième éléments: nous devons aussi les échanger:

6	**2**	**8**	4	dix

Maintenant, les troisième et quatrième éléments, et la deuxième passe est terminée, comme `8` c'est déjà en place:

6	2	**4**	**8**	dix

Nous commençons immédiatement le prochain passage. Regardez attentivement le premier et le deuxième éléments - un autre échange est nécessaire:

2	**6**	4	8	dix

Doit maintenant `6` se mettre en place. Nous échangeons les deuxième et troisième éléments:

2	**4**	**6**	8	dix

La liste est déjà triée. Nous n'avons plus rien à faire. C'est exactement ce que nous voulons.

Comme vous pouvez le voir, l'essence de cet algorithme est simple: **nous comparons les éléments adjacents, et en échangeant certains d'entre eux, nous atteignons notre objectif** .

Codons en Python toutes les actions effectuées lors d'un seul passage dans la liste, puis nous considérerons le nombre de passages dont nous avons réellement besoin pour l'exécuter. Nous n'avons pas expliqué cela jusqu'à présent, et nous le ferons un peu plus tard.

Tri d'une liste

De combien de passes avons-nous besoin pour trier la liste entière?

Nous résolvons ce problème de la manière suivante: **nous introduisons une autre variable** ; sa tâche est d'observer si un échange a été effectué pendant le passage ou non; s'il n'y a pas d'échange, la liste est déjà triée et il n'y a plus rien à faire. Nous créons une variable nommée `swapped` et nous lui attribuons une valeur `False` pour indiquer qu'il n'y a pas de swaps. Sinon, il sera attribué `True`.

```
myList = [8, 10, 6, 2, 4] # list to sort


for i in range(len(myList) - 1): # we need (5 - 1) comparisons
    if myList[i] > myList[i + 1]: # compare adjacent elements
        myList[i], myList[i + 1] = myList[i + 1], myList[i] # if we end up here it means that we have to swap the elements
```

Vous devriez pouvoir lire et comprendre ce programme sans aucun problème:

```
myList = [8, 10, 6, 2, 4] # list to sort

swapped = True # it's a little fake - we need it to enter the while loop


while swapped:

    swapped = False # no swaps so far

    for i in range(len(myList) - 1):

        if myList[i] > myList[i + 1]:

            swapped = True # swap occured!

            myList[i], myList[i + 1] = myList[i + 1], myList[i]


print(myList)
```

Exécutez le programme et testez-le.

Le tri des bulles - version interactive

Dans l'éditeur, vous pouvez voir un programme complet, enrichi d'une conversation avec l'utilisateur, et permettant à l'utilisateur d'entrer et d'imprimer des éléments de la liste: **Le tri à bulles - version interactive finale** .

```
myList = []

swapped = True

num = int(input("How many elements do you want to sort: "))

for i in range(num):

    val = float(input("Enter a list element: "))

    myList.append(val)

while swapped:

    swapped = False

    for i in range(len(myList) - 1):

        if myList[i] > myList[i + 1]:

            swapped = True

            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print("\nSorted:")

print(myList)
```

Python, cependant, a ses propres mécanismes de tri. Personne n'a besoin d'écrire leur propre type, car il existe un nombre suffisant d' **outils prêts à l'emploi** .

Nous vous avons expliqué ce système de tri, car il est important d'apprendre à traiter le contenu d'une liste et de vous montrer comment le vrai tri peut fonctionner.

Si vous voulez que Python trie votre liste, vous pouvez le faire comme ceci:

```
myList = [8, 10, 6, 2, 4]

myList.sort()

print(myList)
```

C'est aussi simple que ça.

La sortie de l'extrait est la suivante:

```
[2, 4, 6, 8, 10]
```

Comme vous pouvez le voir, toutes les listes ont une méthode nommée `sort()`, qui les trie le plus rapidement possible. Vous avez déjà entendu parler de certaines des méthodes de liste auparavant, et vous allez en apprendre plus sur d'autres très bientôt.

Points clés à retenir

1. Vous pouvez utiliser la `sort()` méthode pour trier les éléments d'une liste, par exemple:

```
lst = [5, 3, 1, 2, 4]

print(lst)


lst.sort()

print(lst) # outputs: [1, 2, 3, 4, 5]
```

2. Il existe également une méthode de liste appelée `reverse()`, que vous pouvez utiliser pour inverser la liste, par exemple:

```
lst = [5, 3, 1, 2, 4]

print(lst)


lst.reverse()

print(lst) # outputs: [4, 2, 1, 3, 5]
```

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
lst = ["D", "F", "A", "Z"]

lst.sort()


print(lst)
```
Vérifier

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
a = 3

b = 1

c = 2


lst = [a, c, b]

lst.sort()


print(lst)
```
Vérifier

Exercice 3

Quelle est la sortie de l'extrait de code suivant?

```
a = "A"

b = "B"

c = "C"

d = " "


lst = [a, b, c, d]

lst.reverse()


print(lst)
```
Vérifier

La vie intérieure des listes

Nous voulons maintenant vous montrer une caractéristique importante et très surprenante des listes, qui les distingue fortement des variables ordinaires.

Nous voulons que vous le mémorisiez - il peut affecter vos futurs programmes et causer de graves problèmes s'il est oublié ou négligé.

Jetez un œil à l'extrait dans l'éditeur.

```
list1 = [1]
```

```
list2 = list1

list1[0] = 2

print(list2)
```

Le programme:

- crée une liste à un élément nommée `list1`;
- l'assigne à une nouvelle liste nommée `list2`;
- change le seul élément de `list1`;
- imprime `list2`.

La partie surprenante est le fait que le programme produira: `[2]`non `[1]`, ce qui semble être la solution évidente.

Les listes (et de nombreuses autres entités Python complexes) sont stockées de différentes manières que les variables (scalaires) ordinaires.

Tu pourrais dire ça:

- le nom d'une variable ordinaire est le **nom de son contenu** ;
- le nom d'une liste est le nom d'un **emplacement mémoire où la liste est stockée** .

Lisez ces deux lignes encore une fois - la différence est essentielle pour comprendre de quoi nous allons parler ensuite.

L'affectation: `list2 = list1`copie le nom du tableau, pas son contenu. En effet, les deux noms (`list1`et `list2`) identifient le même emplacement dans la mémoire de l'ordinateur. La modification de l'un d'eux affecte l'autre, et vice versa.

Comment gérez-vous cela?

Tranches puissantes

Heureusement, la solution est à portée de main - son nom est la **tranche** .

Une tranche est un élément de la syntaxe Python qui vous permet de **faire une toute nouvelle copie d'une liste ou de parties d'une liste** .

```
# Copying the whole list

list1 = [1]

list2 = list1[:]

list1[0] = 2

print(list2)

# Copying part of the list

myList = [10, 8, 6, 4, 2]

newList = myList[1:3]

print(newList)
```

Il copie en fait le contenu de la liste, pas le nom de la liste.

C'est exactement ce dont vous avez besoin. Jetez un œil à l'extrait ci-dessous:

```
list1 = [1]

list2 = list1[:]

list1[0] = 2

print(list2)
```

Sa sortie est `[1]`.

Cette partie discrète du code décrit comme `[:]`est capable de produire une toute nouvelle liste.

L'une des formes les plus générales de la tranche se présente comme suit:

```
myList[start:end]
```

Comme vous pouvez le voir, cela ressemble à l'indexation, mais les deux points à l'intérieur font une grande différence.

Une tranche de ce formulaire **crée une nouvelle liste (cible), en prenant des éléments de la liste source - les éléments des indices du début à**`end - 1` .

Remarque: pas à `end`mais à `end - 1`. Un élément d'indice égal à `end`est le premier élément qui **ne participe pas au découpage** .

L'utilisation de valeurs négatives pour le début et la fin est possible (tout comme dans l'indexation).

Jetez un œil à l'extrait:

```
myList = [10, 8, 6, 4, 2]

newList = myList[1:3]

print(newList)
```

La `newList`liste aura `end - start`(3 - 1 = 2) éléments - ceux avec des indices égal à `1`et `2`(mais pas `3`).

La sortie de l'extrait est: `[8, 6]`

Tranches - indices négatifs

Regardez l'extrait ci-dessous:

```
myList[start:end]
```

Répéter:

- `start`est l'indice du premier élément **inclus dans la tranche** ;
- `end`est l'indice du premier élément **non inclus dans la tranche.**

Voici comment **les indices négatifs** fonctionnent avec la tranche:

```
myList = [10, 8, 6, 4, 2]
newList = myList[1:-1]
```

```
print(newList)
```

La sortie de l'extrait est: `[8, 6, 4]`.

Si le `start`spécifie un élément situé plus loin que celui décrit par le `end`(du point de vue initial de la liste), la tranche sera **vide** :

```
myList = [10, 8, 6, 4, 2]
newList = myList[-1:1]
print(newList)
```

La sortie de l'extrait est: `[]`.

Si vous omettez le `start`dans votre tranche, on suppose que vous voulez obtenir une tranche commençant à l'élément avec index `0`.

En d'autres termes, la tranche de ce formulaire:

```
myList[:end]
```

est un équivalent plus compact de:

```
myList[0:end]
```

Regardez l'extrait ci-dessous:

```
myList = [10, 8, 6, 4, 2]
newList = myList[:3]
print(newList)
```

Ceci est la raison pour laquelle sa sortie est: `[10, 8, 6]`.

De même, si vous omettez le `end`dans votre tranche, on suppose que vous voulez que la tranche se termine à l'élément avec l'index `len(myList)`.

En d'autres termes, la tranche de ce formulaire:

```
myList[start:]
```

est un équivalent plus compact de:

```
myList[start:len(myList)]
```

Regardez l'extrait de code suivant:

```
myList = [10, 8, 6, 4, 2]
newList = myList[3:]
print(newList)
```
Sa sortie est donc: `[4, 2]`.

Comme nous l'avons déjà dit, en omettant les deux `start`et en `end`faisant une **copie de la liste entière** :

```
myList = [10, 8, 6, 4, 2]
newList = myList[:]
print(newList)
```

La sortie de l'extrait est: `[10, 8, 6, 4, 2]`.

L'instruction `del` décrite précédemment est capable de **supprimer plus qu'un simple élément de liste à la fois - elle peut également supprimer des tranches** :

```
myList = [10, 8, 6, 4, 2]
del myList[1:3]
print(myList)
```

Remarque: dans ce cas, la tranche **ne produit aucune nouvelle liste** !

La sortie de l'extrait est : `[10, 4, 2]`.

La suppression de **tous les éléments** à la fois est également possible:

```
myList = [10, 8, 6, 4, 2]
del myList[:]
print(myList)
```

La liste est vide, et la sortie est: `[]`.

La suppression de la tranche du code change radicalement sa signification.

Regarde:

```
myList = [10, 8, 6, 4, 2]
del myList
print(myList)
```

L'instruction `del` **supprimera la liste elle-même, pas son contenu** .

L' invocation `print()`de fonction à partir de la dernière ligne du code provoquera alors une erreur d'exécution.

Les opérateurs `in` et `not in`

Python propose deux opérateurs très puissants, capables de **parcourir la liste afin de vérifier si une valeur spécifique est stockée dans la liste ou non** .

Ces opérateurs sont:

```
elem in myList
```

```
elem not in myList
```

Le premier d'entre eux (`in`) vérifie si un élément donné (son argument de gauche) est actuellement stocké quelque part dans la liste (l'argument de droite) - l'opérateur retourne `True`dans ce cas.

Le second (`not in`) vérifie si un élément donné (son argument de gauche) est absent dans une liste - l'opérateur retourne `True`dans ce cas.

Regardez le code dans l'éditeur.

```
myList = [0, 3, 12, 8, 2]

print(5 in myList)

print(5 not in myList)

print(12 in myList)
```

L'extrait montre les deux opérateurs en action. Pouvez-vous deviner sa sortie? Exécutez le programme pour vérifier si vous aviez raison.

Listes - quelques programmes simples

Nous voulons maintenant vous montrer quelques programmes simples utilisant des listes.

Le premier d'entre eux essaie de trouver la plus grande valeur dans la liste. Regardez le code dans l'éditeur.

Le concept est assez simple - nous supposons temporairement que le premier élément est le plus grand et vérifions l'hypothèse par rapport à tous les autres éléments de la liste.

Le code sort `17`(comme prévu).

Le code peut être réécrit pour utiliser la forme nouvellement introduite de la `for`boucle:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]

largest = myList[0]


for i in myList:

    if i > largest:

        largest = i


print(largest)
```

Le programme ci-dessus effectue une comparaison inutile, lorsque le premier élément est comparé à lui-même, mais ce n'est pas du tout un problème.

Le code sort `17`également (rien d'inhabituel).

Si vous devez économiser l'énergie de l'ordinateur, vous pouvez utiliser une tranche:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
```

```
largest = myList[0]


for i in myList[1:]:
    if i > largest:
        largest = i


print(largest)
```

La question est la suivante: laquelle de ces deux actions consomme le plus de ressources informatiques - une seule comparaison ou le découpage de presque tous les éléments d'une liste?

Listes - quelques programmes simples

Voyons maintenant l'emplacement d'un élément donné dans une liste:

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
toFind = 5
found = False

for i in range(len(myList)):
    found = myList[i] == toFind
    if found:
        break

if found:
    print("Element found at index", i)
else:
    print("absent")
```

Remarque:

- la valeur cible est stockée dans la `toFind`variable;
- l'état actuel de la recherche est stocké dans la `found`variable (`True`/ `False`)
- quand `found`devient `True`, la `for`boucle est quittée.

Supposons que vous avez choisi les numéros suivants à la loterie: `3`, `7`, `11`, `42`, `34`, `49`.

Les chiffres qui ont été tirés sont: `5`, `11`, `9`, `42`, `3`, `49`.

La question est: combien de numéros avez-vous touché?

Le programme vous donnera la réponse:

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

Remarque:

- la `drawn` liste stocke tous les numéros tirés;
- la `bets` liste stocke vos paris;
- la `hits` variable compte vos coups.

La sortie du programme est le suivant : `4`.

LABORATOIRE

Temps estimé

10-15 minutes

Niveau de difficulté

Facile

Objectifs

Familiarisez l'étudiant avec:

- indexation des listes;
- en utilisant les opérateurs `in` et `not in`.

Scénario

Imaginez une liste - pas très longue, pas très compliquée, juste une simple liste contenant des nombres entiers. Certains de ces chiffres peuvent être répétés, et c'est l'indice. Nous ne voulons pas de répétitions. Nous voulons qu'ils soient supprimés.

Votre tâche consiste à écrire un programme qui supprime toutes les répétitions de numéros de la liste. L'objectif est d'avoir une liste dans laquelle tous les chiffres n'apparaissent pas plus d'une fois.

Remarque: supposez que la liste source est codée en dur dans le code - vous n'avez pas besoin de la saisir à partir du clavier. Bien sûr, vous pouvez améliorer le code et ajouter une partie qui peut mener une conversation avec l'utilisateur et obtenir toutes les données de lui / elle.

Astuce: nous vous encourageons à créer une nouvelle liste en tant qu'espace de travail temporaire - vous n'avez pas besoin de mettre à jour la liste in situ.

Nous n'avons fourni aucune donnée de test, car ce serait trop facile. Vous pouvez utiliser notre squelette à la place.

```
myList = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]

#

# put your code here

#

print("The list with unique elements only:")

print(myList)
```

Points clés à retenir

1. Si vous avez une liste `l1`, l'affectation suivante: `l2 = l1` ne fait pas de copie de la `l1` liste, mais crée les variables `l1` et `l2` **pointe vers une seule et même liste en mémoire** . Par exemple:

```
vehiclesOne = ['car', 'bicycle', 'motor']

print(vehiclesOne) # outputs: ['car', 'bicycle', 'motor']


vehiclesTwo = vehiclesOne

del vehiclesOne[0] # deletes 'car'

print(vehiclesTwo) # outputs: ['bicycle', 'motor']
```

2. Si vous souhaitez copier une liste ou une partie de la liste, vous pouvez le faire en effectuant un **découpage** :

```
colors = ['red', 'green', 'orange']


copyWholeColors = colors[:] # copy the whole list

copyPartColors = colors[0:2] # copy part of the list
```

3. Vous pouvez également utiliser **des indices négatifs** pour effectuer des tranches. Par exemple:

```
sampleList = ["A", "B", "C", "D", "E"]

newList = sampleList[2:-1]

print(newList) # outputs: ['C', 'D']
```

4. Les paramètres `start` et `end` sont **facultatifs** lors de l'exécution d'une tranche:, `list[start:end]` par exemple:

```
myList = [1, 2, 3, 4, 5]

sliceOne = myList[2: ]

sliceTwo = myList[ :2]

sliceThree = myList[-2: ]


print(sliceOne) # outputs: [3, 4, 5]

print(sliceTwo) # outputs: [1, 2]

print(sliceThree) # outputs: [4, 5]
```

5. Vous pouvez **supprimer des tranches** à l'aide de l' `del` instruction:

```
myList = [1, 2, 3, 4, 5]

del myList[0:2]

print(myList) # outputs: [3, 4, 5]
```

```
del myList[:]

print(myList) # deletes the list content, outputs: []
```

6. Vous pouvez tester si certains éléments **existent dans une liste ou ne pas** utiliser les mots `in`- clés et `not in`, par exemple:

```
myList = ["A", "B", 1, 2]


print("A" in myList) # outputs: True

print("C" not in myList) # outputs: True

print(2 not in myList) # outputs: False
```

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]

l2 = l1

l3 = l2


del l1[0]

del l2[0]


print(l3)
```

Vérifier

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]

l2 = l1

l3 = l2


del l1[0]

del l2


print(l3)
```

Vérifier

Exercice 3

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]

l2 = l1

l3 = l2


del l1[0]

del l2[:]


print(l3)
```

Vérifier

Exercice 4

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]

l2 = l1[:]

l3 = l2[:]


del l1[0]

del l2[0]


print(l3)
```

Vérifier

Exercice 5

Insérez `in` ou `not in` au lieu de `???` pour que le code génère le résultat attendu.

```
myList = [1, 2, "in", True, "ABC"]


print(1 ??? myList) # outputs True

print("A" ??? myList) # outputs True

print(3 ??? myList) # outputs True

print(False ??? myList) # outputs False
```

Vérifier

Listes dans les listes

Les listes peuvent être constituées de scalaires (à savoir des nombres) et d'éléments d'une structure beaucoup plus complexe (vous avez déjà vu des exemples tels que des chaînes, des booléens ou même d'autres listes dans les leçons précédentes de la section Résumé). Examinons de plus près le cas où **les éléments d'** une **liste ne sont que des listes** .

Nous trouvons souvent de tels **tableaux** dans nos vies. Le meilleur exemple est probablement un **échiquier** .

Un échiquier est composé de rangées et de colonnes. Il y a huit lignes et huit colonnes. Chaque colonne est marquée par les lettres A à H. Chaque ligne est marquée par un nombre de un à huit.

L'emplacement de chaque champ est identifié par des paires lettre-chiffre. Ainsi, nous savons que le coin inférieur droit de la planche (celui avec la tour blanche) est A1, tandis que le coin opposé est H8.

Supposons que nous sommes en mesure d'utiliser les numéros sélectionnés pour représenter n'importe quelle pièce d'échecs. Nous pouvons également supposer que **chaque ligne de l'échiquier est une liste** .

Regardez le code ci-dessous:

```
row = []

for i in range(8):
    row.append(WHITE_PAWN)
```

Il construit une liste contenant huit éléments représentant la deuxième ligne de l'échiquier - celle remplie de pions (supposons que `WHITE_PAWN`c'est un **symbole prédéfini** représentant un pion blanc).

Le même effet peut être obtenu au moyen d'une **compréhension de liste** , la syntaxe spéciale utilisée par Python pour remplir des listes massives.

Une compréhension de liste est en fait une liste, mais **créée à la volée pendant l'exécution du programme, et n'est pas décrite statiquement** .

Jetez un œil à l'extrait:

```
row = [WHITE_PAWN for i in range(8)]
```

La partie du code placée entre crochets spécifie:

- les données à utiliser pour remplir la liste (`WHITE_PAWN`)
- la clause spécifiant combien de fois les données se produisent dans la liste (`for i in range(8)`)

Laissez-nous vous montrer quelques autres **exemples de compréhension de liste** :

Exemple 1:

```
squares = [x ** 2 for x in range(10)]
```

L'extrait produit une liste de dix éléments remplie de carrés de dix nombres entiers à partir de zéro (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Exemple # 2:

```
twos = [2 ** i for i in range(8)]
```

L'extrait crée un tableau à huit éléments contenant les huit premières puissances de deux (1, 2, 4, 8, 16, 32, 64, 128)

Exemple # 3:

```
odds = [x for x in squares if x % 2 != 0 ]
```

L'extrait crée une liste avec uniquement les éléments impairs de la `squares`liste.

Listes dans les listes: tableaux bidimensionnels

Supposons également qu'un **symbole prédéfini** nommé `EMPTY`désigne un champ vide sur l'échiquier.

Donc, si nous voulons créer une liste de listes représentant l'ensemble de l'échiquier, cela peut être fait de la manière suivante:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
```

Remarque:

- la partie intérieure de la boucle crée une ligne composée de huit éléments (chacun étant égal à `EMPTY`) et l'ajoute à la `board`liste;
- la partie extérieure le répète huit fois;
- au total, la `board`liste comprend 64 éléments (tous égaux à `EMPTY`)

Ce modèle imite parfaitement le véritable échiquier, qui est en fait une liste d'éléments à huit éléments, tous étant des rangées simples. Résumons nos observations:

- les éléments des lignes sont des champs, huit d'entre eux par ligne;
- les éléments de l'échiquier sont des rangées, huit d'entre eux par échiquier.

La `board`variable est maintenant un **tableau à deux dimensions** . On l'appelle aussi, par analogie avec des termes algébriques, une **matrice** .

Comme les compréhensions de liste peuvent être **imbriquées** , nous pouvons raccourcir la création du forum de la manière suivante:

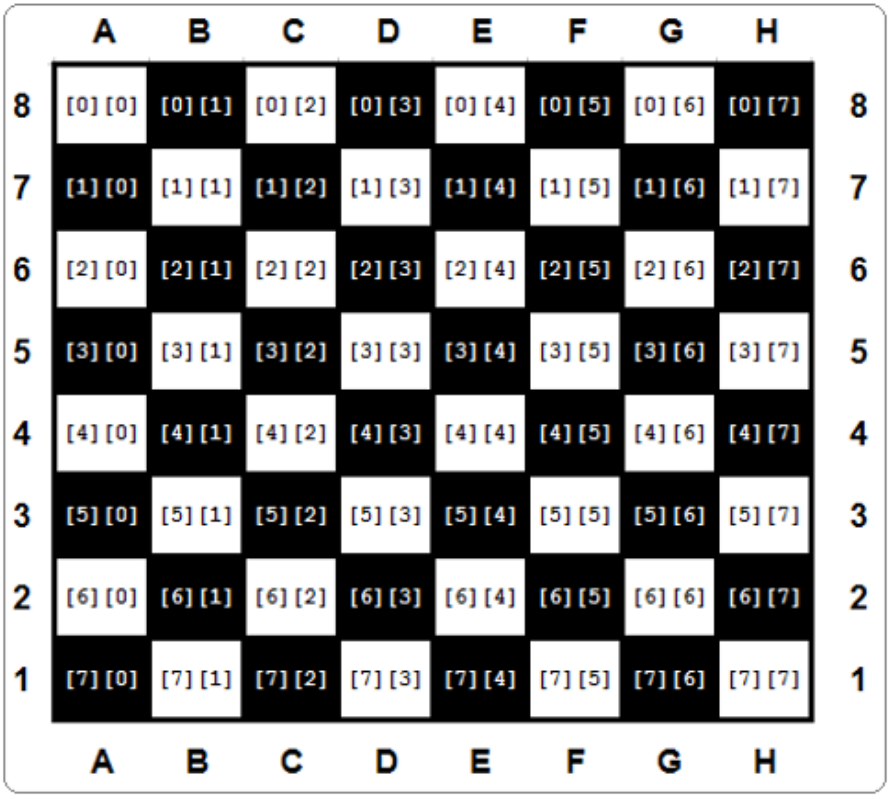
```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

La partie intérieure crée une ligne et la partie extérieure crée une liste de lignes.

Listes dans les listes: tableaux bidimensionnels - suite

L'accès au champ sélectionné du tableau nécessite deux indices - le premier sélectionne la ligne; le second - le numéro de champ à l'intérieur de la ligne, qui est de facto un numéro de colonne.

Jetez un œil à l'échiquier. Chaque champ contient une paire d'indices qui doivent être fournis pour accéder au contenu du champ:



En jetant un œil à la figure ci-dessus, posons quelques pièces d'échecs sur le plateau. Tout d'abord, ajoutons toutes les tours:

```
board[0][0] = ROOK
```

```
board[0][7] = ROOK
```

```
board[7][0] = ROOK
```

```
board[7][7] = ROOK
```

Si vous souhaitez ajouter un chevalier à C4, procédez comme suit:

```
board[4][2] = KNIGHT
```

Et maintenant un pion à E5:

```
board[3][4] = PAWN
```

Et maintenant - expérimentez le code dans l'éditeur.

```
EMPTY = "-"

ROOK = "ROOK"

board = []

for i in range(8):

    row = [EMPTY for i in range(8)]
```

```
    board.append(row)

board[0][0] = ROOK

board[0][7] = ROOK

board[7][0] = ROOK

board[7][7] = ROOK

print(board)
```

Nature multidimensionnelle des listes: applications avancées

Approfondissons la nature multidimensionnelle des listes. Pour trouver n'importe quel élément d'une liste bidimensionnelle, vous devez utiliser deux *coordonnées* :

- un vertical (numéro de ligne)
- et une horizontale (numéro de colonne).

Imaginez que vous développiez un logiciel pour une station météorologique automatique. L'appareil enregistre la température de l'air sur une base horaire et le fait tout au long du mois. Cela vous donne un total de 24 × 31 = 744 valeurs. Essayons de concevoir une liste capable de stocker tous ces résultats.

Tout d'abord, vous devez décider quel type de données conviendrait à cette application. Dans ce cas, a `float`serait préférable, car ce thermomètre est capable de mesurer la température avec une précision de 0,1 ℃.

Ensuite, vous prenez une décision arbitraire que les lignes enregistreront les lectures toutes les heures sur l'heure (donc la ligne aura 24 éléments) et chacune des lignes sera affectée à un jour du mois (supposons que chaque mois a 31 jours , vous avez donc besoin de 31 lignes). Voici la paire de compréhensions appropriée (`h`pour l'heure, `d`pour le jour):

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

La matrice entière est maintenant remplie de zéros. Vous pouvez supposer qu'il est mis à jour automatiquement à l'aide d'agents matériels spéciaux. La chose que vous devez faire est d'attendre que la matrice soit remplie de mesures.

Il est maintenant temps de déterminer la température mensuelle moyenne à midi. Additionnez les 31 lectures enregistrées à midi et divisez la somme par 31. Vous pouvez supposer que la température de minuit est enregistrée en premier. Voici le code pertinent:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
#

total = 0.0

for day in temps:
    total += day[11]

average = total / 31

print("Average temperature at noon:", average)
```

Remarque: la `day`variable utilisée par la `for`boucle n'est pas un scalaire - chaque passage dans la `temps`matrice l'affecte aux lignes suivantes de la matrice; par conséquent, c'est une liste. Il doit être indexé avec `11`pour accéder à la valeur de température mesurée à midi.

Trouvez maintenant la température la plus élevée de tout le mois - voir le code:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
#

highest = -100.0

for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("The highest temperature was:", highest)
```

Remarque:

- la `day`variable parcourt toutes les lignes de la `temps`matrice;
- la `temp`variable parcourt toutes les mesures prises en une journée.

Comptez maintenant les jours où la température à midi était d'au moins 20 ℃:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
#

hotDays = 0

for day in temps:
    if day[11] > 20.0:
        hotDays += 1

print(hotDays, "days were hot.")
```

Tableaux tridimensionnels

Python ne limite pas la profondeur de l'inclusion liste dans liste. Ici, vous pouvez voir un exemple de tableau en trois dimensions:

Imaginez un hôtel. C'est un immense hôtel composé de trois bâtiments de 15 étages chacun. Il y a 20 chambres à chaque étage. Pour cela, vous avez besoin d'un tableau qui peut collecter et traiter des informations sur les chambres occupées / libres.

Première étape - le type des éléments du tableau. Dans ce cas, une valeur booléenne (`True`/ `False`) conviendrait.

Deuxième étape - analyse calme de la situation. Résumez les informations disponibles: trois bâtiments, 15 étages, 20 chambres.

Vous pouvez maintenant créer le tableau:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

Le premier indice (`0`traversant `2`) sélectionne l'un des bâtiments; le deuxième (`0`traversant `14`) sélectionne le sol, le troisième (`0`traversant `19`) sélectionne le numéro de la pièce. Toutes les chambres sont initialement gratuites.

Maintenant, vous pouvez réserver une chambre pour deux jeunes mariés: dans le deuxième bâtiment, au dixième étage, chambre 14:

```
rooms[1][9][13] = True
```

et libérer la deuxième chambre au cinquième étage située dans le premier bâtiment:

```
rooms[0][4][1] = False
```

Vérifiez s'il y a des vacances au 15ème étage du troisième bâtiment:

```
vacancy = 0


for roomNumber in range(20):
    if not rooms[2][14][roomNumber]:
        vacancy += 1
```

La `vacancy`variable contient `0`si toutes les pièces sont occupées, ou le nombre de pièces disponibles sinon.

Toutes nos félicitations! Vous avez atteint la fin du module. Continuez votre bon travail!

Points clés à retenir

1. La **compréhension des listes** vous permet de créer de nouvelles listes à partir des listes existantes de manière concise et élégante. La syntaxe d'une compréhension de liste se présente comme suit:

```
[expression for element in list if conditional]
```

qui est en fait un équivalent du code suivant:

```
for element in list:
    if conditional:
        expression
```

Voici un exemple de compréhension de liste - le code crée une liste de cinq éléments remplie avec les cinq premiers nombres naturels élevés à la puissance de 3:

```
cubed = [num ** 3 for num in range(5)]
print(cubed) # outputs: [0, 1, 8, 27, 64]
```

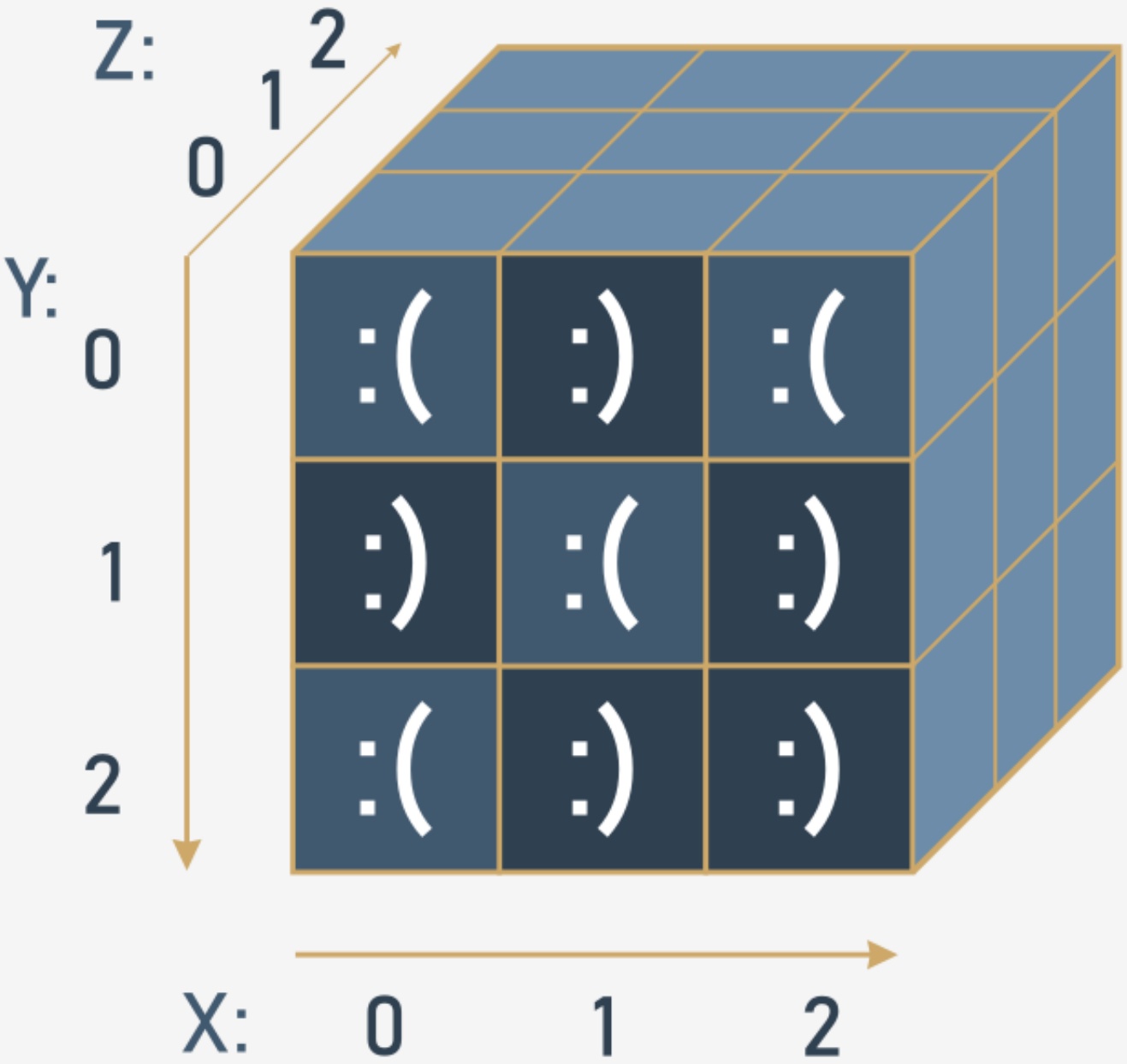
2. Vous pouvez utiliser **des listes imbriquées** en Python pour créer des **matrices** (c'est-à-dire des listes bidimensionnelles). Par exemple:


```
# A four-column/four-row table - a two dimensional array (4x4)


table = [[":(", ":)", ":(", ":)"],
         [":)", ":(", ":)", ":)"],
         [":(", ":)", ":)", ":("],
         [":)", ":)", ":)", ":("]]


print(table)
print(table[0][0]) # outputs: ':('
print(table[0][3]) # outputs: ':)'
```

3. Vous pouvez imbriquer autant de listes que vous le souhaitez dans les listes et créer ainsi des listes à n dimensions, par exemple des tableaux à trois, quatre ou même soixante-quatre dimensions. Par exemple:




```
# Cube - a three-dimensional array (3x3x3)


cube = [[[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':(', 'x', 'x']],

        [[':)', 'x', 'x'],
         [':(', 'x', 'x'],
         [':)', 'x', 'x']],

        [[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':)', 'x', 'x']]]
```

```
print(cube)

print(cube[0][0][0]) # outputs: ':('

print(cube[2][2][0]) # outputs: ':)'
```

Toutes nos félicitations! Vous avez terminé le module 3.

Bien joué! Vous avez atteint la fin du module 3 et franchi une étape importante dans votre formation en programmation Python. Voici un bref résumé des objectifs que vous avez couverts et que vous vous êtes familiarisés avec le module 3:

- Valeurs booléennes pour comparer différentes valeurs et contrôler les chemins d'exécution à l'aide des instructions `if` et `if-else`;
- l'utilisation des boucles (`while` et `for`) et comment contrôler leur comportement à l'aide des instructions `break` et `continue`;
- la différence entre les opérations logiques et au niveau du bit;
- le concept de listes et de traitement de listes, y compris l'itération fournie par la `for` boucle et le découpage;
- l'idée de tableaux multidimensionnels.

Vous êtes maintenant prêt à répondre au questionnaire du module et à tenter le dernier défi: le test du module 3, qui vous aidera à évaluer ce que vous avez appris jusqu'à présent.

Module 3 Quiz

Limite de temps: 15 minutes
Nombre de questions: 10
Points à marquer: 10
Note de passage: 70%

Début

Un opérateur capable de vérifier si deux valeurs ne sont pas égales est codé comme:

- `!=`
- `<>`
- `not ==`

Combien d'étoiles l'extrait suivant enverra-t-il à la console?

```
i = 2
while i >= 0:
    print("*")
    i -= 2
```

- Trois
- une
- deux

Combien de hachages l'extrait suivant enverra-t-il à la console?

```
for i in range(-1, 1):
    print("#")
```

- deux
- Trois
- une

Quelle valeur sera affectée à la `x` variable?

```
z = 10
y = 0
x = z > y or z == y
```

- `True`
- `False`
- `1`

Quelle est la sortie du code suivant?

```
lst = [3, 1, -1]
lst[-1] = lst[-2]
print(lst)
```

- `[3, -1, 1]`
- `[3, 1, 1]`
- `[1, 1, -1]`

La deuxième mission:

```
vals = [0, 1, 2]
vals[0], vals[1] = vals[1], vals[2]
```

- ne change pas la longueur de la liste
- étend la liste
- raccourcit la liste

Jetez un œil à l'extrait de code et choisissez l'une des affirmations suivantes, ce qui est vrai:

```
nums = []
vals = nums
vals.append(1)
```

- `nums` et `vals` sont de la même longueur
- `vals` est plus long que `nums`
- `nums` est plus long que `vals`

Combien d'éléments la `l` liste contient-elle?

```
l = [0 for i in range(1, 3)]
```

- deux
- Trois
- une

Quelle est la sortie de l'extrait de code suivant?

```
lst = [0, 1, 2, 3]
x = 1
for elem in lst:
    x *= elem
print(x)
```

- `6`
- `0`
- `1`