Bases de l'informatique 1

Cours 3. Séquences et boucle for

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

Compléments sur le `print`

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT

```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL

```
>>>
```

Compléments sur le `print`

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT

```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL

```
>>> test()
```

Compléments sur le `print`

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT

```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL

```
>>> test()
12 34 56
bon jour
```

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT
```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL
```
>>> test()
12 34 56
bon jour
```

- On peut modifier ce comportement grâce aux options `sep` et `end`

SCRIPT
```
def test2():
    print('12','34','56',sep=' - ',end=':')
    print('bon','jour',sep='')
```

SHELL
```
>>>
```

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT
```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL
```
>>> test()
12 34 56
bon jour
```

- On peut modifier ce comportement grâce aux options `sep` et `end`

SCRIPT
```
def test2():
    print('12','34','56',sep=' - ',end=':')
    print('bon','jour',sep='')
```

SHELL
```
>>> test2()
```

Compléments sur le `print`

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT
```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL
```
>>> test()
12 34 56
bon jour
```

- On peut modifier ce comportement grâce aux options `sep` et `end`

SCRIPT
```
def test2():
    print('12','34','56',sep=' - ',end=':')
    print('bon','jour',sep='')
```

SHELL
```
>>> test2()
12 - 34 - 56:bonjour
>>>
```

- `print` ajoute automatiquement des espaces et un retour à la ligne.

SCRIPT
```
def test():
    print('12','34','56')
    print('bon','jour')
```

SHELL
```
>>> test()
12 34 56
bon jour
```

- On peut modifier ce comportement grâce aux options `sep` et `end`

SCRIPT
```
def test2():
    print('12','34','56',sep=' - ',end=':')
    print('bon','jour',sep='')
```

SHELL
```
>>> test2()
12 - 34 - 56:bonjour
>>> # Il y a bien eu nouvelle ligne après bonjour
```

- Il y a 3 délimiteurs de chaînes : `'toto' == "toto" == """toto"""`

Documenter une fonction

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.

Documenter une fonction

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

- N'intervient pas dans l'exécution, mais permet d'obtenir de l'aide.

SHELL

```
>>>
```

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

- N'intervient pas dans l'exécution, mais permet d'obtenir de l'aide.

SHELL

```
>>> somme_chiffre(3041) # 3 + 0 + 4 + 1
```

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

- N'intervient pas dans l'exécution, mais permet d'obtenir de l'aide.

SHELL

```
>>> somme_chiffre(3041) # 3 + 0 + 4 + 1
8
>>>
```

Documenter une fonction

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

- N'intervient pas dans l'exécution, mais permet d'obtenir de l'aide.

SHELL

```
>>> somme_chiffre(3041) # 3 + 0 + 4 + 1
8
>>> help(somme_chiffre)
```

Documenter une fonction

- Il y a 3 délimiteurs de chaînes : `'toto'` == `"toto"` == `"""toto"""`
  - Les guillemets triples permettent de documenter une fonction.
  - Cette `docstring` peut tenir sur plusieurs lignes.

SCRIPT

```
def somme_chiffre(n):
    """Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9"""
    res=0
    while n>0:
        res=res+n%10
        n=n//10
    return res
```

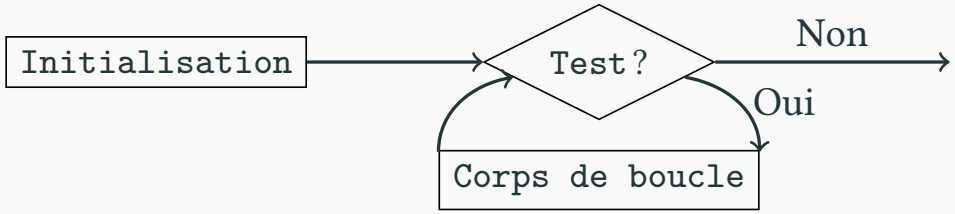
- N'intervient pas dans l'exécution, mais permet d'obtenir de l'aide.

SHELL

```
>>> somme_chiffre(3041) # 3 + 0 + 4 + 1
8
>>> help(somme_chiffre)
Help on function somme_chiffre:

somme_chiffre(n)
    Retourne la somme des chiffres de n en base 10
    Exemple 621 -> 6+2+1 -> 9
```

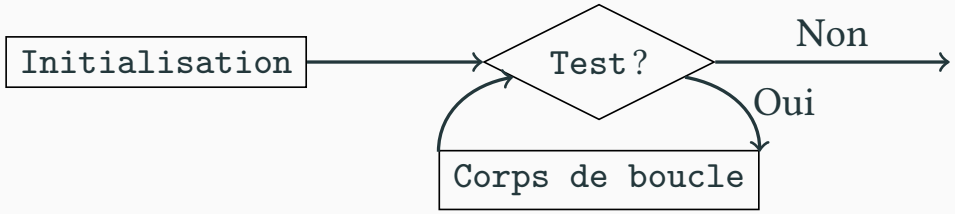
Sommaire



- De nombreuses boucles `while` sont de la forme :

SCRIPT

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

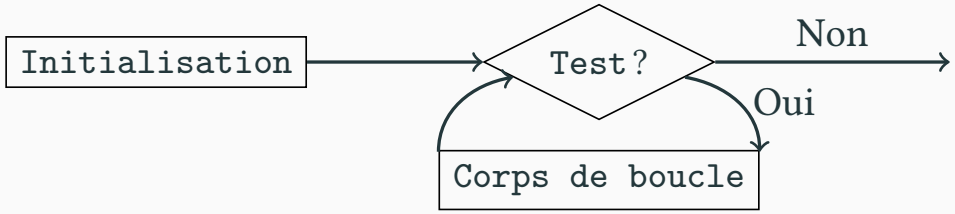


- De nombreuses boucles `while` sont de la forme :

SCRIPT

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

  - L'initialisation et le test sont toujours les mêmes quelque soit `n`.

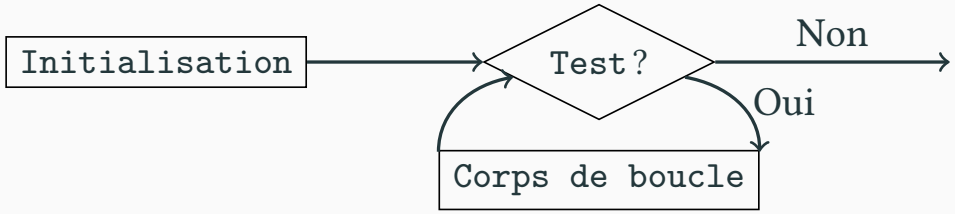


- De nombreuses boucles `while` sont de la forme :

SCRIPT

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

  - L'initialisation et le test sont toujours les mêmes quelque soit `n`.
  - Dans la boucle, la variable `i` varie de `0` à `n-1`



- De nombreuses boucles `while` sont de la forme :

SCRIPT

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

  - L'initialisation et le test sont toujours les mêmes quelque soit `n`.
  - Dans la boucle, la variable `i` varie de `0` à `n-1`
  - Il est facile de voir que pour `n>0`, un telle boucle termine toujours.

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

- Plus besoin d'initialiser `i`

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.
- Et on a la garantie que la boucle se termine après `n` passages !

La boucle `for` : principe

SCRIPT
```
i=0
while i<n:
    ...
    ...
    i=i+1
```

SCRIPT
```
for i in range(n):
    ...
    ...
```

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.
- Et on a la garantie que la boucle se termine après `n` passages !

SCRIPT
```
def f():
    for i in range(10):
        print(i,end='')
    print(' ')
```

SHELL
```
>>>
```

La boucle `for` : principe

SCRIPT
```
i=0
while i<n:
    ...
    ...
    i=i+1
```

SCRIPT
```
for i in range(n):
    ...
    ...
```

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.
- Et on a la garantie que la boucle se termine après `n` passages !

SCRIPT
```
def f():
    for i in range(10):
        print(i,end='')
    print(' ')
```

SHELL
```
>>> f()
```

La boucle `for` : principe

SCRIPT

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

SCRIPT

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

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.
- Et on a la garantie que la boucle se termine après `n` passages !

SCRIPT

```
def f():
    for i in range(10):
        print(i,end='')
    print(' ')
```

SHELL

```
>>> f()
0123456789
```

SCRIPT
```
i=0
while i<n:
    ...
    ...
    i=i+1
```

SCRIPT
```
for i in range(n):
    ...
    ...
```

- Plus besoin d'initialiser `i`
- Plus besoin d'incrémenter (`i=i+1`) la variable `i`
- Plus besoin d'écrire le test.
- Et on a la garantie que la boucle se termine après `n` passages !

SCRIPT
```
def f():
    for i in range(10):
        print(i,end='')
    print(' ')
```

SHELL
```
>>> f()
0123456789
```

- Attention : `range(10)` correspond à l'ensemble de 10 éléments `0,1,...,9`

- On peut préciser le début :

SCRIPT

```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL

```
>>>
```

- On peut préciser le début :

SCRIPT

```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL

```
>>> f()
```

- On peut préciser le début :

SCRIPT

```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL

```
>>> f()
123456789
```

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

La boucle `for` : début et pas

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>>
```

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>> f()
```

La boucle `for` : début et pas

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>> f()
147
```

La boucle `for` : début et pas

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>> f()
147
```

  - On part de `i=1`

La boucle `for` : début et pas

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>> f()
147
```

  - On part de `i=1`
  - On incrémente avec `i=i+3`

La boucle `for` : début et pas

- On peut préciser le début :

SCRIPT
```
def f():
    for i in range(1,10):
        print(i,end='')
```

SHELL
```
>>> f()
123456789
```

- Ainsi que le début et le pas

SCRIPT
```
def f():
    for i in range(1,10,3):
        print(i,end='')
```

SHELL
```
>>> f()
147
```

  - On part de `i=1`
  - On incrémente avec `i=i+3`
  - $1 \to 4 \to 7 \to$ STOP car `i` doit toujours vérifier `i<10`

Boucle `for` : synthèse

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`$\neq$`0`) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`≠0) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

SCRIPT
```
# début=0 par défaut
# pas=1 par défaut
for i in range(début,fin,pas):
    ...
    ...
```

SCRIPT
```
i=début
while i < fin:
    ...
    ...
    i=i+pas
```

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`≠0) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

SCRIPT
```
# début=0 par défaut
# pas=1 par défaut
for i in range(début,fin,pas):
    ...
    ...
```

SCRIPT
```
i=début
while i < fin:
    ...
    ...
    i=i+pas
```

- Cette équivalence est vérifiée si :

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`≠0) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

SCRIPT
```
# début=0 par défaut
# pas=1 par défaut
for i in range(début,fin,pas):
    ...
    ...
```

SCRIPT
```
i=début
while i < fin:
    ...
    ...
    i=i+pas
```

- Cette équivalence est vérifiée si :
  - On ne modifie pas `i` dans le corps de la boucle

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`≠0) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

SCRIPT
```
# début=0 par défaut
# pas=1 par défaut
for i in range(début,fin,pas):
    ...
    ...
```

SCRIPT
```
i=début
while i < fin:
    ...
    ...
    i=i+pas
```

- Cette équivalence est vérifiée si :
  - On ne modifie pas `i` dans le corps de la boucle
  - On n'utilise pas `i` après la boucle

- Une boucle `for` est équivalent à une boucle `while`
  - `début`, `fin` et `pas` sont des entiers (`pas`≠0) ;
  - dans le cas où `pas` est négatif la condition est `i>fin`.

SCRIPT
```
# début=0 par défaut
# pas=1 par défaut
for i in range(début,fin,pas):
    ...
    ...
```

SCRIPT
```
i=début
while i < fin:
    ...
    ...
    i=i+pas
```

- Cette équivalence est vérifiée si :
  - On ne modifie pas `i` dans le corps de la boucle
  - On n'utilise pas `i` après la boucle
- De toutes façons, ne pas respecter ces usages est une mauvaise pratique !

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>>
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>>
```

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
>>> while i < 2:
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
>>> while i < 2:
...     print(i)
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>>
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL

```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>>
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>>
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>>
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
...     i=i+100
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
...     i=i+100
...     print(i)
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
...     i=i+100
...     print(i)
30;130
29;129
28;128
27;127
26;126
>>>
```

Différence entre `while` et `for`

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
...     i=i+100
...     print(i)
30;130
29;129
28;128
27;127
26;126
>>> i
```

- Que vaut `i` à la fin de la boucle ?

SHELL
```
>>> i=0
>>> while i < 2:
...     print(i)
...     i=i+1
0
1
>>> i
2
```

SHELL
```
>>> for i in range(2):
...     print(i)
0
1
>>> i
1
```

Que s'est-il passé ?

- La variable de boucle prend la valeur prévue au début de chaque boucle.
- Elle garde sa dernière valeur en sortant de la boucle.

SHELL
```
>>> # Comportement étrange
>>> for i in range(30,25,-1):
...     print(i,end=';')
...     i=i+100
...     print(i)
30;130
29;129
28;128
27;127
26;126
>>> i
126
```

- Afficher les entiers de 9 à 0 : méthode universelle

SCRIPT

```
for i in range(10):
    print(9-i, end=' -> ')
```

- Afficher les entiers de 9 à 0 : méthode universelle

SCRIPT
```
for i in range(10):
    print(9-i, end=' -> ')
```

SHELL
```
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 ->
```

- Afficher les entiers de 9 à 0 : méthode universelle

SCRIPT
```
for i in range(10):
    print(9-i, end=' -> ')
```

SHELL
```
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 ->
```

- Afficher les entiers de 9 à 0 : méthode plus directe

SCRIPT
```
for i in range(9,-1,-1):# le test est i>-1
    print(i,end=' -> ')
```

- Afficher les entiers de 9 à 0 : méthode universelle

SCRIPT
```
for i in range(10):
    print(9-i, end=' -> ')
```

SHELL
```
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 ->
```

- Afficher les entiers de 9 à 0 : méthode plus directe

SCRIPT
```
for i in range(9,-1,-1):# le test est i>-1
    print(i,end=' -> ')
```

SHELL
```
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 ->
```

- Les pas flottants (`i=i+0.1`) sont interdits.

- Afficher les suites ci-dessous avec des boucles `for`.

- Afficher les suites ci-dessous avec des boucles `for`.
  - `0.0 -> 0.1 -> 0.2 -> ... -> 0.9 -> 1.0`

- Afficher les suites ci-dessous avec des boucles `for`.
  - `0.0 -> 0.1 -> 0.2 -> ... -> 0.9 -> 1.0`
  - `0.0 -> 0.5 -> 1.0 -> 1.5 ... -> 19.5 -> 20.0`

- Afficher les suites ci-dessous avec des boucles `for`.
  - `0.0 -> 0.1 -> 0.2 -> ... -> 0.9 -> 1.0`
  - `0.0 -> 0.5 -> 1.0 -> 1.5 ... -> 19.5 -> 20.0`
  - `10 -> 8 -> 6 -> ... -> -8 -> -10`

- Afficher les suites ci-dessous avec des boucles `for`.
  - `0.0 -> 0.1 -> 0.2 -> ... -> 0.9 -> 1.0`
  - `0.0 -> 0.5 -> 1.0 -> 1.5 ... -> 19.5 -> 20.0`
  - `10 -> 8 -> 6 -> ... -> -8 -> -10`
- Même question avec des boucles `while`.

Sommaire

- On peut faire une boucle `for` dans une autre boucle `for`.

SCRIPT

```
for i in range(5):
    for j in range(i):
        print('*',end='')
    print(': i=',i,sep='')
```

- On peut faire une boucle `for` dans une autre boucle `for`.

SCRIPT

```
for i in range(5):
    for j in range(i):
        print('*',end='')
    print(': i=',i,sep='')
```

- La boucle du `j` affiche des étoiles.

- On peut faire une boucle `for` dans une autre boucle `for`.

SCRIPT

```
for i in range(5):
    for j in range(i):
        print('*',end='')
    print(': i=',i,sep='')
```

- La boucle du `j` affiche des étoiles.
- La boucle du `i` fait varier le nombre d'étoiles.

- On peut faire une boucle `for` dans une autre boucle `for`.

SCRIPT
```
for i in range(5):
    for j in range(i):
        print('*',end='')
    print(': i=',i,sep='')
```

- La boucle du `j` affiche des étoiles.
- La boucle du `i` fait varier le nombre d'étoiles.

SHELL
```
: i=0
*: i=1
**: i=2
***: i=3
****: i=4
```

- On peut faire une boucle `for` dans une autre boucle `for`.

SCRIPT
```
for i in range(5):
    for j in range(i):
        print('*',end='')
    print(': i=',i,sep='')
```

- La boucle du `j` affiche des étoiles.
- La boucle du `i` fait varier le nombre d'étoiles.

SHELL
```
: i=0
*: i=1
**: i=2
***: i=3
****: i=4
```

- `for i in range(0)` correspond à une boucle vide.

- `return` interrompt la fonction, donc toutes les boucles.

SCRIPT

```
def plus_petit_div(n):
    for i in range(2,n):
        if n%i == 0:
            return i
    return n
```

- `return` interrompt la fonction, donc toutes les boucles.

SCRIPT

```
def plus_petit_div(n):
    for i in range(2,n):
        if n%i == 0:
            return i
    return n
```

- `break` interrompt seulement la boucle en cours

- `return` interrompt la fonction, donc toutes les boucles.

SCRIPT
```
def plus_petit_div(n):
    for i in range(2,n):
        if n%i == 0:
            return i
    return n
```

- `break` interrompt seulement la boucle en cours

SCRIPT
```
for i in range(9,14):
    for j in range(2, i+1):
        print(j, end = '-')
        if i%j == 0:
            break
    print('>',i, 'est divisible par', j)
```

- `return` interrompt la fonction, donc toutes les boucles.

SCRIPT
```
def plus_petit_div(n):
    for i in range(2,n):
        if n%i == 0:
            return i
    return n
```

- `break` interrompt seulement la boucle en cours

SCRIPT
```
for i in range(9,14):
    for j in range(2, i+1):
        print(j, end = '-')
        if i%j == 0:
            break
    print('>',i, 'est divisible par', j)
```

SHELL
```
2-3-> 9 est divisible par 3
2-> 10 est divisible par 2
2-3-4-5-6-7-8-9-10-11-> 11 est divisible par 11
2-> 12 est divisible par 2
2-3-4-5-6-7-8-9-10-11-12-13-> 13 est divisible par 13
```

SCRIPT

```
for i in range(10):      # dizaines
    for j in range(10): # unités
        print(i,j, sep='', end=' ')
    print('') # retour à la ligne
```

☕ Afficher les entiers de 0 à 99

SCRIPT

```
for i in range(10):      # dizaines
    for j in range(10): # unités
        print(i,j, sep='', end=' ')
    print('') # retour à la ligne
```

SHELL

```
00 01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

SCRIPT

```
for i in range(10):      # dizaines
    for j in range(10): # unités
        print(i,j, sep='', end=' ')
    print('') # retour à la ligne
```

SHELL

```
00 01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

- Comment pourrait-on écrire un programme équivalent mais avec une seule boucle `for` ? Essayez de le faire !

Sommaire

Définition

- Une séquence est un suite de donnée finie et indexée.

Définition

- Une séquence est un suite de donnée finie et indexée.

- En python, il y a trois types de séquence :
  - Les chaînes de caractère (`str`)
  - Les n-uplet (`tuple`)
  - les tableaux (`list`)

- Une séquence est un suite de donnée finie et indexée.

- En python, il y a trois types de séquence :
  - Les chaînes de caractère (`str`)
  - Les n-uplet (`tuple`)
  - les tableaux (`list`)

- Les séquences possèdent trois opérateurs en commun :
  - La longueur `len(séquence)`
  - L'indexage `séquence[i]`
  - La concaténation `séq1 + séq2` (et donc le produit)

De la chaîne aux caractères

- Une chaîne est une suite de caractères

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de `0` à `len(texte)-1`.

SHELL
```
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de `0` à `len(texte)-1`.

SHELL
```
>>> texte[0]
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de `0` à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
' '
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
' '
>>> texte[len(texte)-1]
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
' '
>>> texte[len(texte)-1]
')'
>>>
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
' '
>>> texte[len(texte)-1]
')'
>>> texte[len(texte)]
```

- Une chaîne est une suite de caractères
- La longueur (*length*) d'une chaîne est le nombre de caractères.

SHELL
```
>>> texte='123 nous allons au bois :-)'
>>> len(texte)
27
```

- Les caractères sont indexés de 0 à `len(texte)-1`.

SHELL
```
>>> texte[0]
'1'
>>> texte[3]
' '
>>> texte[len(texte)-1]
')'
>>> texte[len(texte)]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError:  string index out of range
```

- la boucle est quasiment toujours la même : ♥

SCRIPT

```
def numéroter_caractères(chaîne):
    print("i : chaîne[i]")
    print("-------------")
    for i in range(len(chaîne)):
        print(i, ":" ,chaîne[i])
```

- la boucle est quasiment toujours la même : ♥

SCRIPT
```
def numéroter_caractères(chaîne):
    print("i : chaîne[i]")
    print("-------------")
    for i in range(len(chaîne)):
        print(i, ":" ,chaîne[i])
```

- Attention à ne pas confondre
  - L'indice `i`
  - La valeur indéxée `chaîne[i]`

SHELL
```
>>>
```

- la boucle est quasiment toujours la même : ♥

SCRIPT
```
def numéroter_caractères(chaîne):
    print("i : chaîne[i]")
    print("-------------")
    for i in range(len(chaîne)):
        print(i, ":" ,chaîne[i])
```

- Attention à ne pas confondre
  - L'indice `i`
  - La valeur indéxée `chaîne[i]`

SHELL
```
>>> numéroter_caractères("Youpi !")
```

- la boucle est quasiment toujours la même : ♥

SCRIPT
```
def numéroter_caractères(chaîne):
    print("i : chaîne[i]")
    print("-------------")
    for i in range(len(chaîne)):
        print(i, ":" ,chaîne[i])
```

- Attention à ne pas confondre
  - L'indice `i`
  - La valeur indéxée `chaîne[i]`

SHELL
```
>>> numéroter_caractères("Youpi !")
i : chaîne[i]
-------------
0 : Y
1 : o
2 : u
3 : p
4 : i
5 :
6 : !
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

```shell
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL

```
>>> (3,4) # ceci est un tuple
(3, 4)
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL

```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL

```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
True
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
True
>>> L[2] #Mon 3ème est une chaîne
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
True
>>> L[2] #Mon 3ème est une chaîne
'Yeux'
>>>
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
True
>>> L[2] #Mon 3ème est une chaîne
'Yeux'
>>> L    #Mon tout est orange
```

- Les deux autres types de séquences sont les tuples et les tableaux (liste)

SHELL
```
>>> (3,4) # ceci est un tuple
(3, 4)
>>> [11, 23, 45, 1] # ceci est un tableau
[11, 23, 45, 1]
```

- Les chaînes sont des séquences de caractères
- Les listes et les tuples sont des séquences de tout ce que l'on veut

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(L)
3
>>> L[0] #Mon 1er est un chiffre
6
>>> L[1] #Mon 2nd est un booléen
True
>>> L[2] #Mon 3ème est une chaîne
'Yeux'
>>> L    #Mon tout est orange
[6, True, 'Yeux']
```

Sommaire

- Somme des notes dans une liste

SCRIPT
```
def somme(liste_notes):
    s = 0 # on renvoie une somme
    for i in range(len(liste_notes)):
        s = s + liste_notes[i]
    return s
```

- Nombre d'occurence du caractère `'e'` dans une chaînes

SCRIPT
```
def combien_de_e(chaînes):
    n = 0
    for i in range(len(chaîne)):
        if chaîne[i] == 'e':
            n = n + 1
    return n
```

- Plus grand éléments d'une liste

SCRIPT
```
def maximum(liste):
    """ la liste doit être non vide"""
    m = M[0]
    for i in range(len(liste)):
        if liste[i] > m:
            m = liste[i]
    return m
```

- plus longue chaîne d'une liste

SCRIPT
```
def plus_longue(liste):
    """ la liste doit être non vide"""
    m = M[0]
    for i in range(len(liste)):
        if len(liste[i]) > len(m):
            m = liste[i]
    return m
```

- Remplacer un caractère par un autre

SCRIPT
```
def replacer_e_par_euro(chaîne):
    s = ""
    for i in range(len(chaîne)):
        if chaîne[i] == 'e':
            s = s + '€'
        else:
            s = s + chaîne[i]
    return s
```

- Créer un nouveau tuple contenant les valeurs absolue d'un autre tuple.

SCRIPT
```
def valeur_absolue_tuple(nuplet):
    t = ()
    for i in range(len(nuplet)):
        t = t +(abs(nuplet[i]) ,) # Virgule obligatoire
    return t
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT

```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL

```
>>>
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT

```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL

```
>>> position('a', 'Koala')
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT

```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL

```
>>> position('a', 'Koala')
2
>>>
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT

```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL

```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT

```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL

```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

SCRIPT
```
def appartient(lettre,chaîne):
    return position(lettre,chaîne)>=0
```

SHELL
```
>>>
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

SCRIPT
```
def appartient(lettre,chaîne):
    return position(lettre,chaîne)>=0
```

SHELL
```
>>> appartient('a','Koala')
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

SCRIPT
```
def appartient(lettre,chaîne):
    return position(lettre,chaîne)>=0
```

SHELL
```
>>> appartient('a','Koala')
True
>>>
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

SCRIPT
```
def appartient(lettre,chaîne):
    return position(lettre,chaîne)>=0
```

SHELL
```
>>> appartient('a','Koala')
True
>>> appartient('A','Koala')
```

- Index de première apparition d'un caractère (-1 si absent)

SCRIPT
```
def position(lettre,chaîne):
    for i in range(len(chaîne)):
        if chaîne[i] == lettre:
            return i
    return -1
```

SHELL
```
>>> position('a', 'Koala')
2
>>> position('A', 'Koala')
-1
```

- Le caractère est-il présent dans la chaîne ?

SCRIPT
```
def appartient(lettre,chaîne):
    return position(lettre,chaîne)>=0
```

SHELL
```
>>> appartient('a','Koala')
True
>>> appartient('A','Koala')
False
```

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

```
>>>
```
SHELL

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

```shell
>>> tiret('abcdef')
```

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

SHELL
```
>>> tiret('abcdef')
a-b-c-d-e-f
```

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

SHELL
```
>>> tiret('abcdef')
a-b-c-d-e-f
```

- On fera attention à ce qu'il n'y ait pas de tiret après le dernier caractère.

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

```shell
>>> tiret('abcdef')
a-b-c-d-e-f
```

- On fera attention à ce qu'il n'y ait pas de tiret après le dernier caractère.

- Même exercice mais cette fois-ci en renvoyant la chaîne.

```shell
>>>
```

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

```
>>> tiret('abcdef')
a-b-c-d-e-f
```
SHELL

- On fera attention à ce qu'il n'y ait pas de tiret après le dernier caractère.

- Même exercice mais cette fois-ci en renvoyant la chaîne.

```
>>> renvoie_tiret('abcdef')
```
SHELL

- Écrire une fonction qui affiche une chaîne de caractères, en séparant les symboles par des tirets.

```
>>> tiret('abcdef')
a-b-c-d-e-f
```
SHELL

- On fera attention à ce qu'il n'y ait pas de tiret après le dernier caractère.

- Même exercice mais cette fois-ci en renvoyant la chaîne.

```
>>> renvoie_tiret('abcdef')
'a-b-c-d-e-f'
```
SHELL

Sommaire

- Les chaînes et les tuples ne sont pas mutable (modifiable)

```shell
>>>
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>>
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>>
```

- Les listes sont les seuls séquences que l'on peut modifier

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
```

- Les listes sont les seuls séquences que l'on peut modifier

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>>
```

- Les listes sont les seuls séquences que l'on peut modifier

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
```

- Les listes sont les seuls séquences que l'on peut modifier

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>>
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
>>>
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
>>> L[2] = 5
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
>>> L[2] = 5
>>>
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
>>> L[2] = 5
>>> L
```

- Les chaînes et les tuples ne sont pas mutable (modifiable)

SHELL
```
>>> t = (1,2,3)
>>> t[2] = 33
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

SHELL
```
>>> c = "Oliuier"
>>> c[3] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

- Les listes sont les seuls séquences que l'on peut modifier

SHELL
```
>>> L = [2,3,55,7]
>>> L[2] = 5
>>> L
[2, 3, 5, 7]
```

Construire un tableau

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>>
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>> t = [-12, 23, 14, -5, 0]
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>> t = [-12, 23, 14, -5, 0]
>>>
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
>>>
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT
```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL
```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
>>> t
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT

```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL

```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
>>> t
[-12, 23, 14, -5, 0]
>>>
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT

```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL

```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
>>> t
[-12, 23, 14, -5, 0]
>>> r
```

- La concatenation est une opération très lente !
- La méthode la plus efficace
  - on crée un tableau de la bonne taille (`[0] * taille`)
  - on le remplit avec une boucle

SCRIPT

```
def valeur_absolue(tableau):
    rés = [0] * len(tableau)
    for i in range(len(tableau)):
        rés[i] = abs(tableau[i])
    return rés
```

- Surtout, on ne modifie pas la paramètre `tableau` !

SHELL

```
>>> t = [-12, 23, 14, -5, 0]
>>> r = valeur_absolue(t)
>>> t
[-12, 23, 14, -5, 0]
>>> r
[12, 23, 14, 5, 0]
```

Sommaire

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>>
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>>
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
[0, 0, 0, 0, 0]
>>>
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
[0, 0, 0, 0, 0]
>>> T[0] = 7
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
[0, 0, 0, 0, 0]
>>> T[0] = 7
>>>
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
[0, 0, 0, 0, 0]
>>> T[0] = 7
>>> T
```

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- Pour initialiser un tableau de manière efficace, on utilise le produit

SHELL
```
>>> T = 5 * [0]
>>> T
[0, 0, 0, 0, 0]
>>> T[0] = 7
>>> T
[7, 0, 0, 0, 0]
```

Déstructuration

- On peut déstructurer un tuple ou une liste.

SHELL
```
>>>
```

SHELL
```
>>>
```

Déstructuration

- On peut déstructurer un tuple ou une liste.

SHELL
```
>>> liste=[1,2,3]
```

SHELL
```
>>> uplet=(1,2,3)
```

- On peut déstructurer un tuple ou une liste.

SHELL
```
>>> liste=[1,2,3]
>>>
```

SHELL
```
>>> uplet=(1,2,3)
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>>
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>>
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>>
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>> liste=[1,2,3]
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>> liste=[1,2,3]
>>>
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>> liste=[1,2,3]
>>> [a,b] = liste
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>> liste=[1,2,3]
>>> [a,b] = liste
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  too many values to unpack (expected 2)
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>>
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>>
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
```

- On peut convertir une chaîne en tuple ou en liste.

```shell
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

```shell
>>>
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

SHELL
```
>>> list((1,2,3,4))
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

SHELL
```
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>>
```

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

SHELL
```
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> tuple([1,2,3,4])
```

- On peut convertir une chaîne en tuple ou en liste.

```shell
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

```shell
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
```

- Exercice : écrire les deux fonctions `tuple` et `list`.

Merci pour votre attention
Questions

?

Cours 3 — Séquences et boucle for

Licence

- © 2024 —Olivier Baldellon
- Ce document est publié sous licence CC-BY Attribution 4.0
  - Vous êtes autorisé à :
    - Partager —copier, distribuer et communiquer le matériel par tous moyens et sous tous formats pour toute utilisation, y compris commerciale.
    - Adapter —remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.
  - Selon les conditions suivantes :
    - Attribution —Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- `https://creativecommons.org/licenses/by/4.0/deed.fr`