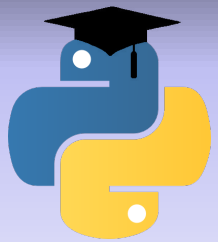
Langage et algorithmique
CR2. Fonctions et modules

A. Malek TOUMI

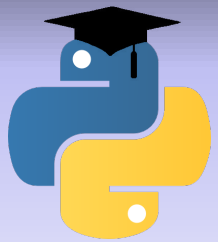
toumiab@ensta-bretagne.fr

2017/2018

ENSTA Bretagne

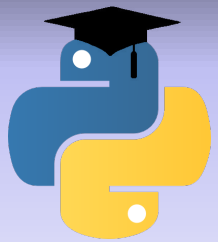


Sommaire



Typage des variables

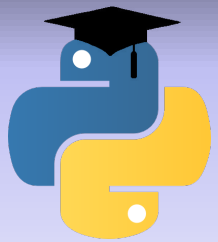
- Python : langage typé dynamiquement



Typage des variables

- Python : langage typé dynamiquement

$\Rightarrow$ typage lors de l'affectation

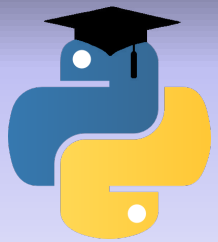


Typage des variables

- Python : langage typé dynamiquement

$\Rightarrow$ typage lors de l'affectation

- Une variable peut changer de type au cours de l'exécution

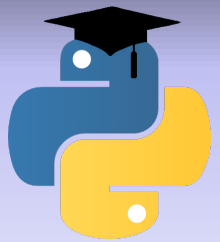


Typage des variables

- Python : langage typé dynamiquement

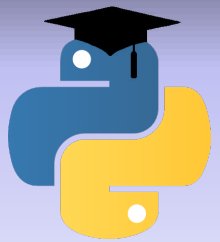
$\Rightarrow$ typage lors de l'affectation

- Une variable peut changer de type au cours de l'exécution
- Une variable n'est qu'une référence à l'objet (entier, réel, ...)



Types des variables

- **Les types intégrés de Python**
  - Les types simples :
    - Entiers signés (**int**), Réels (IEEE 754) (**float**) et Complexes (**complex**), Booléens (**bool**)
  - Les types composites (containers) :
    - **Les séquences** : Chaînes de caractères (**str**) ; Listes (**list**) et Tuples (**tuple**)
    - **Les maps** (hashs) : Dictionnaires (**dict**)
    - **Les ensembles** : le type **set** et le type **frozenset**

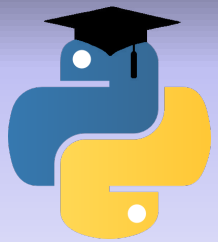


Types des variables

- **Les types intégrés de Python**
  - Les types simples :
    - Entiers signés (**int**), Réels (IEEE 754) (**float**) et Complexes (**complex**), Booléens (**bool**)
  - Les types composites (containers) :
    - **Les séquences** : Chaînes de caractères (**str**) ; Listes (**list**) et Tuples (**tuple**)
    - **Les maps** (hashs) : Dictionnaires (**dict**)
    - **Les ensembles** : le type **set** et le type **frozenset**

Remarque

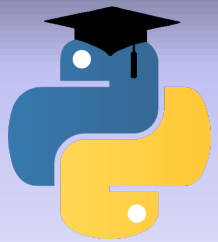
- Pour connaître le type d'une variable ou expression, on peut utiliser la fonction **type()** ou **whos** (sous ipython)



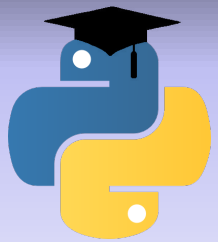
Types des variables

Exemple

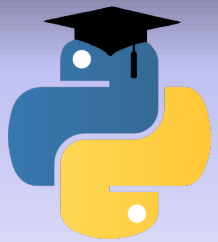
```
>>> i = 42
>>> type(i)
<class 'int'>
>>> i = 'indice'
>>> type(i)
<class 'str'>
>>> i = 42.0
>>> type(i)
<class 'float'>
>>> print(i)
42.5
>>>
```



Sommaire

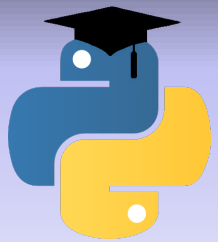


Sommaire



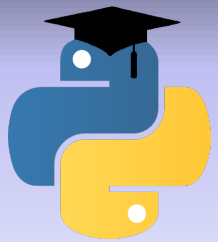
Les types numériques

- **Les entiers**
  - Leurs représentation n'est limitée que par la taille mémoire.



Les types numériques

- **Les entiers**
  - Leurs représentation n'est limitée que par la taille mémoire.
    $\Rightarrow$ pas de risque de débordement

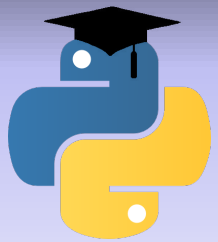


Les types numériques

- **Les entiers**
  - Leurs représentation n'est limitée que par la taille mémoire.
    $\Rightarrow$ pas de risque de débordement

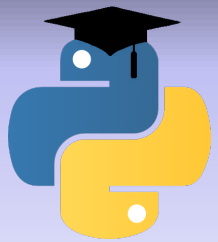
Exemple

```
>>> i = 42
>>> i
42
>>> math.factorial(i)
1405006117752879898543142606244511569936384000000000
```



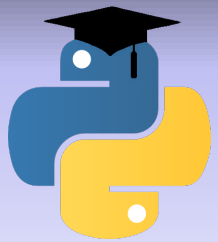
Les types numériques

- **Les entiers**
- **Les réels**
  - Sont représentés en en double précision (64 bits).
    $\Rightarrow$ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E-16$



Les types numériques

- **Les entiers**
- **Les réels**
  - Sont représentés en en double précision (64 bits).
    $\Rightarrow$ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E-16$
  - Pour utiliser la simple précision, on peut faire appel à la bibliothèque `numpy`.

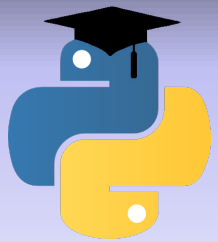


Les types numériques

- **Les entiers**
- **Les réels**
  - Sont représentés en en double précision (64 bits).
    $\Rightarrow$ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E-16$
  - Pour utiliser la simple précision, on peut faire appel à la bibliothèque `numpy`.

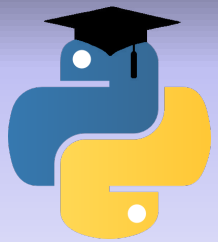
Exemple

```
>>> freq = 10e9
>>> freq
10000000000.0
```



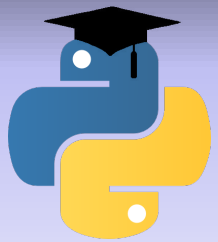
Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**
  - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire



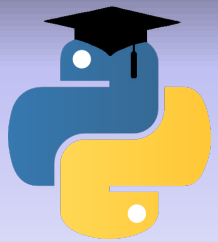
Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**
  - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire
  - Un suffixe **j** pour regrouper deux valeurs composant la partie réelle et la partie imaginaire



Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**
  - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire
  - Un suffixe **j** pour regrouper deux valeurs composant la partie réelle et la partie imaginaire
  - Un nombre complexe possède deux attributs en lecture seule **.real** et **.imag** et une méthode **.conjugate()**

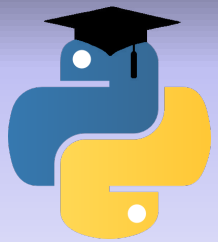


Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**

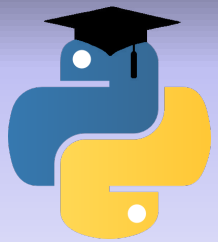
Exemple

```
>>> nb = 10 + 5j
>>> nb.real
10.0
>>> nb.conjugate()
(10-5j)
>>> print(complex(10, 5))
(10+5j)
```



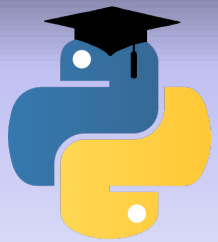
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**



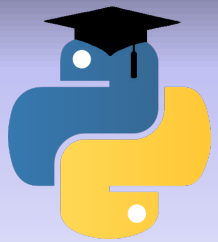
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)



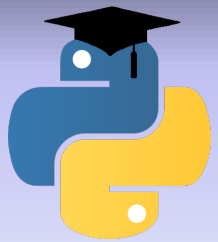
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne



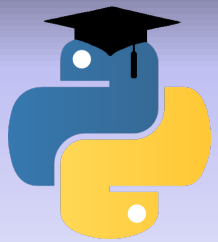
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne
- Tout ce qui n'est pas faux est vrai



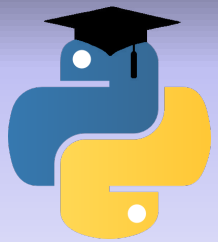
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne
- Tout ce qui n'est pas faux est vrai
    - l'objet None est faux



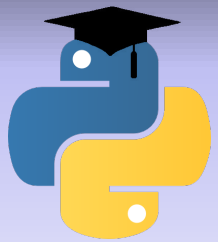
Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne
- Tout ce qui n'est pas faux est vrai
  - l'objet None est faux
  - toutes les valeurs numériques non nulles sont vraies (0, 0.0, 0.0+0.0j sont fausses)



Les booléens

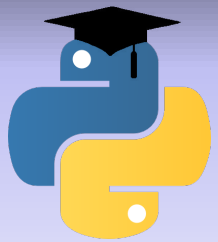
- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne
- Tout ce qui n'est pas faux est vrai
  - l'objet None est faux
  - toutes les valeurs numériques non nulles sont vraies (0, 0.0, 0.0+0.0j sont fausses)
  - tous les agrégats (séquences, dictionnaires, ensembles) sont vrais s'ils contiennent au moins un élément, faux sinon



Les types simples

Remarques

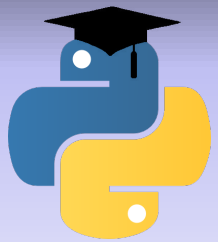
- Les types simples (scalaires, atomiques) permettent de stocker une seule donnée par variable, contrairement aux types composites (containers)
- Les types simples sont immutables (non modifiables)



Le type None

Remarque (type `NoneType`)

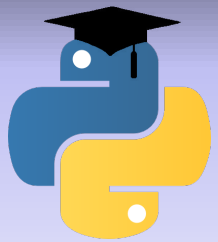
- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.



Le type None

Remarque (type `NoneType`)

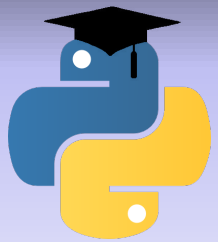
- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme



Le type None

Remarque (type `NoneType`)

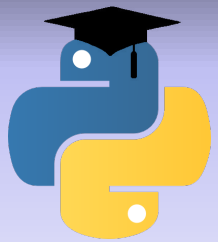
- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur



Le type None

Remarque (type `NoneType`)

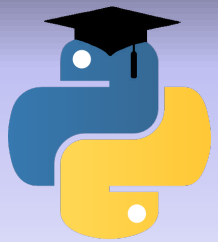
- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur
- Un fonction qui ne renvoie pas explicitement une valeur, renvoie `None`



Le type None

Remarque (type `NoneType`)

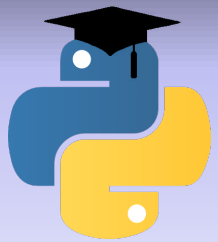
- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur
- Un fonction qui ne renvoie pas explicitement une valeur, renvoie `None`
- En mode interactif, l'affichage d'une variable (sans passer par **print()**) qui vaut *None* ne donne rien.



Le type None

Exemple

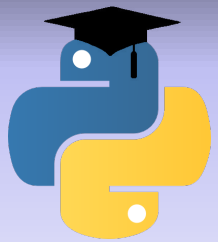
```
>>> a = None
>>> a
>>>              # Rien ne s'affiche
```



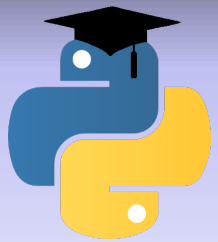
Le type None

Exemple

```
>>> a = None
>>> a
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
>>>
```



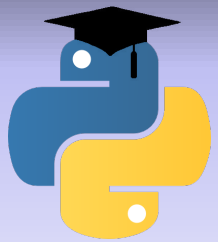
Sommaire



Les séquences

Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)

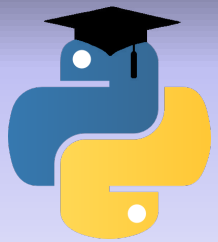


Les séquences

Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)

- On distingue deux familles de séquences
  1. Les séquences immutables (non modifiables, non-mutables) : Les chaînes de caractères (**str**) et les tuples (**tuple**)
  2. Les séquences mutables (modifiables) : Les listes



Les séquences

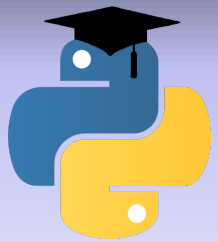
Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)

- On distingue deux familles de séquences
  1. Les séquences immutables (non modifiables, non-mutables) : Les chaînes de caractères (**str**) et les tuples (**tuple**)
  2. Les séquences mutables (modifiables) : Les listes

Remarque

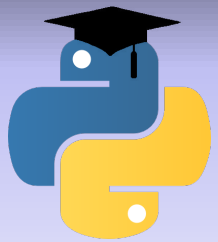
Toutes les opérations sur les séquences immutables sont disponibles sur les séquences mutables.



Les séquences

Indexation des séquences :

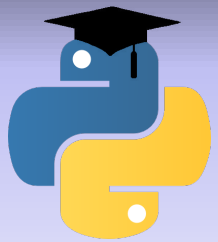
- Les séquences sont indexées en commençant par 0



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index

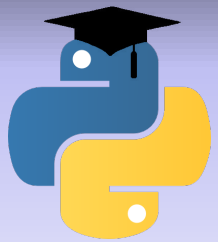


Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```



Les séquences

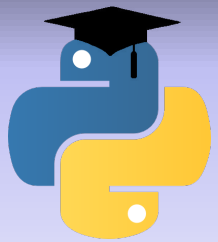
Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j-1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```



Les séquences

Indexation des séquences :

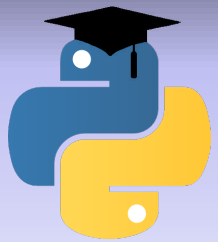
- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j-1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

  - `[:j]` : désigne tout ce qui précède j (j^e élément exclu)



Les séquences

Indexation des séquences :

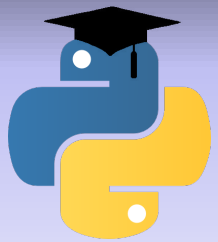
- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j-1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

  - `[:j]` : désigne tout ce qui précède j (j^e élément exclu)
  - `[i:]` : désigne tout ce qui suit i (i^e élément inclu)



Les séquences

Indexation des séquences :

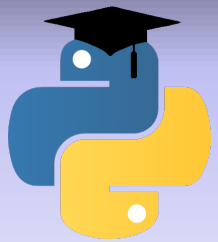
- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j-1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

  - `[:j]` : désigne tout ce qui précède j (j^e élément exclu)
  - `[i:]` : désigne tout ce qui suit i (i^e élément inclu)
  - `[:]` : tous les éléments (utilisé pour réaliser une copie *superficielle*)



Les séquences

Indexation des séquences :

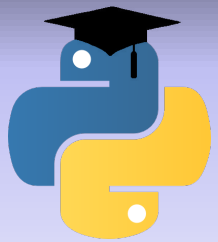
- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j-1)^e$ indice

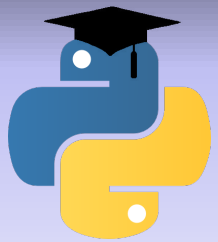
```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

  - `[:j]` : désigne tout ce qui précède j (j^e élément exclu)
  - `[i:]` : désigne tout ce qui suit i (i^e élément inclu)
  - `[:]` : tous les éléments (utilisé pour réaliser une copie *superficielle*)
  - `[i:j:pas]` : pour réaliser une sélection avec un pas donné



Chaîne de caractères

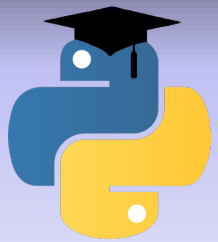
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """



Chaîne de caractères

- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """

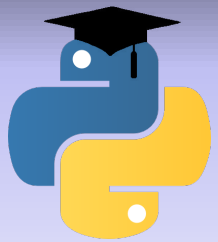
```
>>> s = 'l\'or'
```



Chaîne de caractères

- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """

```
>>> s = 'l\'or'
>>> s = "l'or"
```



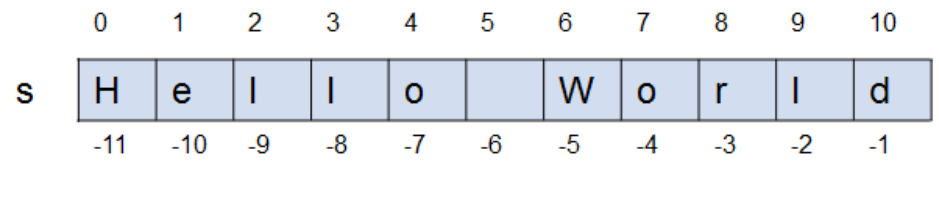
Chaîne de caractères

- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """

```
>>> s = 'l\'or'
>>> s = "l'or"
>>> s = """ c'est une phrase
en plusieurs ligne"""
```

Chaîne de caractères

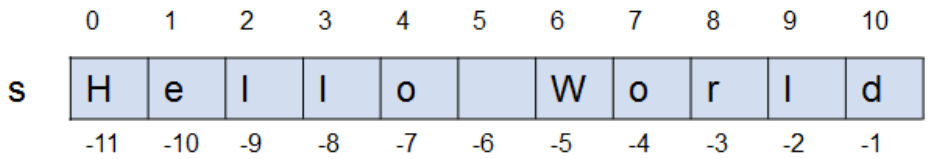
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """




```
>>> s = "Hello World"
```

Chaîne de caractères

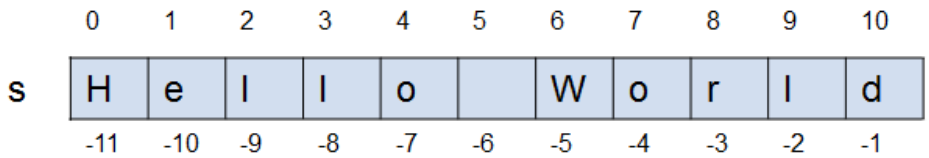
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """




```
>>> s = "Hello World"
>>> print(s[0] , s[-1], s[2:5], s [:-1])
```

Chaîne de caractères

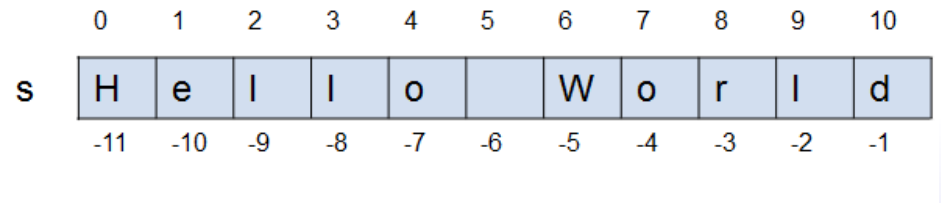
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """



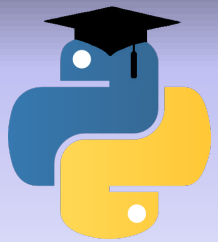

```
>>> s = "Hello World"
>>> print(s[0] , s[-1], s[2:5], s [:-1])
H d llo Hello Worl
```

Chaîne de caractères

- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.

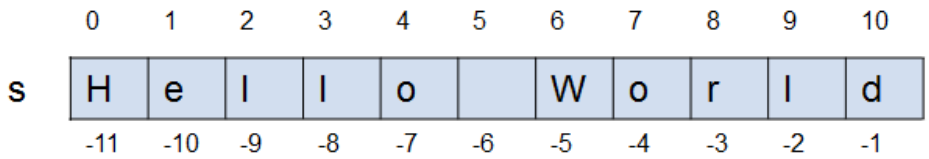



```
>>>s[0]='h'
```



Chaîne de caractères

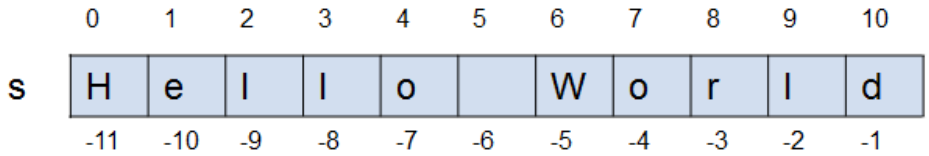
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.




```
>>>s[0]='h'
Traceback (most recent call last):
File "<pyshell #14>", line 1, in <module>
TypeError : 'str ' object does not support
item ...
```

Chaîne de caractères

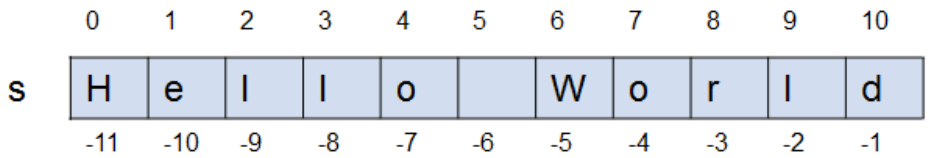
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.




```
>>>s[0]='h'
>>> ch1 = "h" + s [1:]
```

Chaîne de caractères

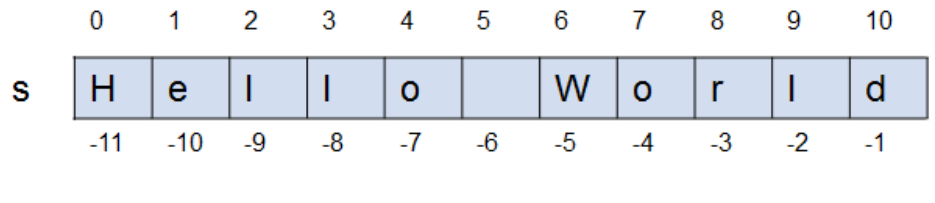
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.




```
>>> ch1 = "h" + s [1:]
>>> ch1
 'hello World'
```

Chaîne de caractères

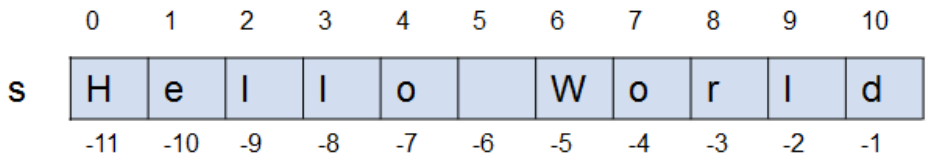
- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.
- Les opérateurs '*' et '+' réalisent la multiplication et la concaténation



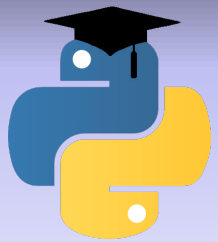

```
>>> a ='a'*2+'b'
```

Chaîne de caractères

- délimitée par des apostrophes simple' ou triple ''', des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.
- Les opérateurs '*' et '+' réalisent la multiplication et la concaténation

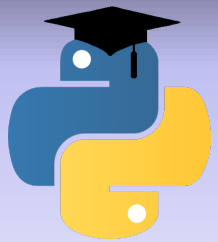



```
>>> a ='a'*2+'b'
>>> a
'aab'
```



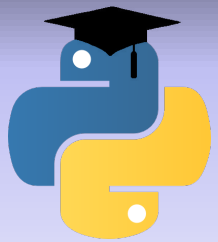
Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets [] et séparés par des virgules,



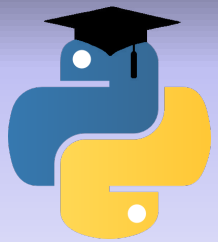
Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets [] et séparés par des virgules,
- Elles sont modifiables



Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets [] et séparés par des virgules,
- Elles sont modifiables
- Elles peuvent contenir des éléments hétérogènes (de types différents : str, list, etc)



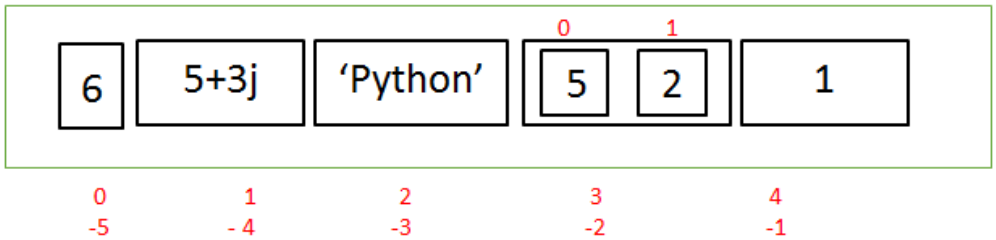
Les listes

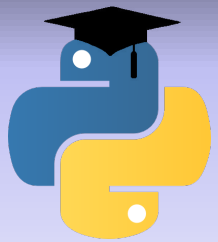
- Elles sont composées d'un ensemble d'éléments placés entre crochets [] et séparés par des virgules,
- Elles sont modifiables
- Elles peuvent contenir des éléments hétérogènes (de types différents : str, list, etc)

Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets [] et séparés par des virgules,
- Elles sont modifiables
- Elles peuvent contenir des éléments hétérogènes (de types différents : str, list, etc)

Lst = [6 , 5+3j, 'Python', [5, 2], 1]

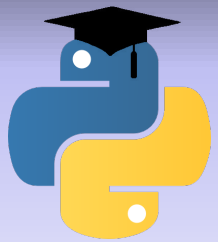




Les listes

Propriétés (Opérations)

len(s)	*taille de s*
`ls[i] = x`	*affecte x au i^e élément de* `ls`
del `ls[i]`	*supprime le i^e élément de* `ls`
del `ls[i:j]`	*supprime la tranche de i^e au $(j-1)^e$ élément*
list(seq)	*transforme la séquence seq en une nouvelle liste*
`ls1 += ls2`	*ajoute le contenu de* `ls2` *à la fin de* `ls1`
`ls3 = ls1 + ls2`	*renvoie une nouvelle liste : concaténation de* `ls1` *et* `ls2`
`ls1 = ls2 * i`	*renvoie une nouvelle liste : duplication de* `i` *fois de* `ls2`
*ls.***append(x)**	*ajouter l'élément x à la fin de la liste ls*
*ls.***insert(pos, el)**	*insérer l'élément el à l'index pos spécifié*
*ls.***remove(el)**	*supprime la première occurrence de la valeur el*
*ls.***pop(pos)**	*envoie et supprime l'élément d'index pos (le dernier sinon)*
***help(list)**	*pour plus de détails sur les propriétés de l'objet list*

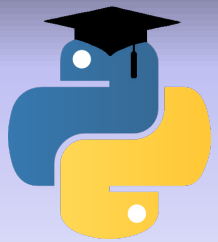


Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées $\Rightarrow$ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```



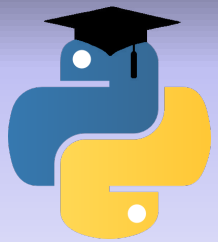
Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées ⇒ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```

- Utilisation de la bibliothèque **numpy**



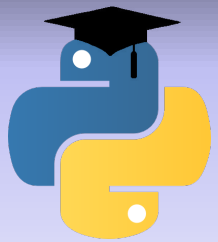
Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées ⇒ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```

- Utilisation de la bibliothèque **numpy**



Les listes

Remarque (Manipulation de matrice)

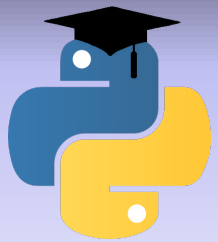
- Utilisation des listes imbriquées ⇒ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```

- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1 # n'effectue pas de copie des données`
  ⇒les 2 variables `lst1` et `lst2` référencent la même liste



Les listes

Remarque (Manipulation de matrice)

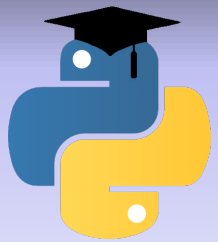
- Utilisation des listes imbriquées ⇒ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```

- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1 # n'effectue pas de copie des données`
  ⇒les 2 variables `lst1` et `lst2` référencent la même liste
- `lst2 = lst1[:] #` duplique que les éléments de 1^{er} niveau



Les listes

Remarque (Manipulation de matrice)

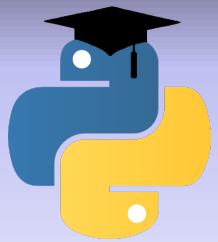
- Utilisation des listes imbriquées ⇒ moins flexible et pas efficace.

```
>>> mat_2D =[ [1,2,3], [4,5,6] ] # listes imbriquées
```

- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1 # n'effectue pas de copie des données`
  ⇒les 2 variables `lst1` et `lst2` référencent la même liste
- `lst2 = lst1[:] #` duplique que les éléments de 1^{er} niveau
- `lst2 = copy.deepcopy(lst1) #` depuis le module **copy**



Les listes

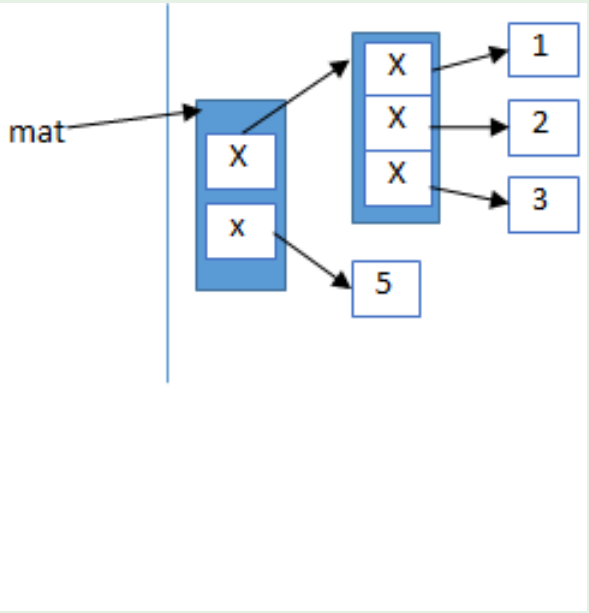
Exemple (Copie de liste)

```
>>> import copy # importation du module copy
>>> mat = [[1, 2, 3], 5]
>>> m1 = mat[:]
>>> m2 = copy.deepcopy(mat)
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3],  100]
>>> m1
[[1, 20, 3], 5]
>>> m2
[[1, 2, 3], 5]
>>>
```

Les listes

Exemple (Copie de liste)

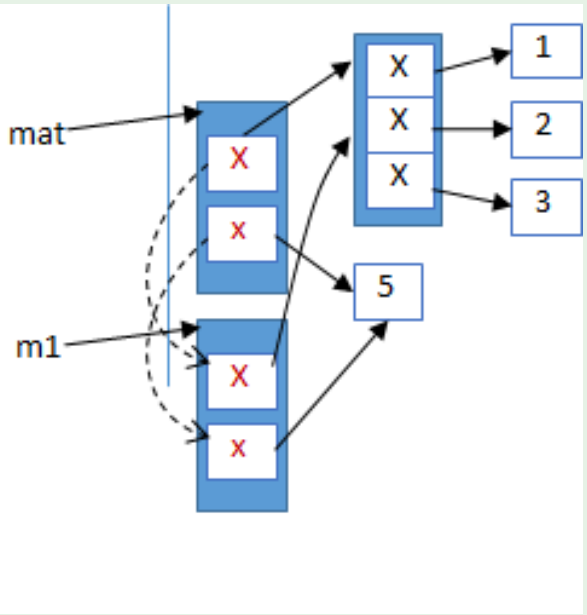
```
>>> mat = [[1, 2, 3], 5]
```



Les listes

Exemple (Copie de liste)

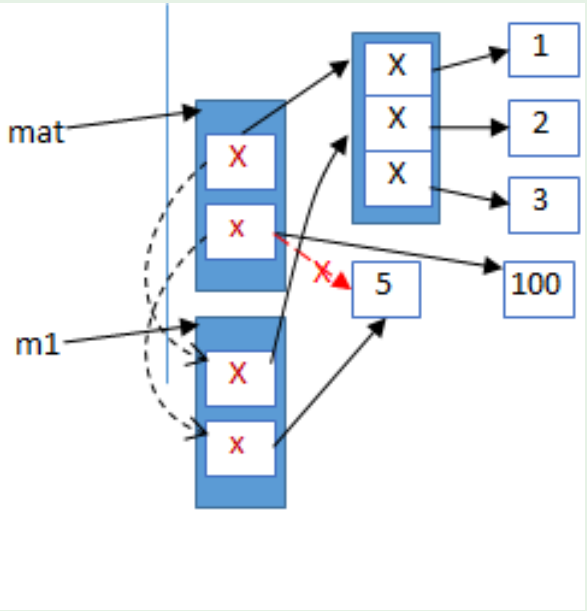
```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
```



Les listes

Exemple (Copie de liste)

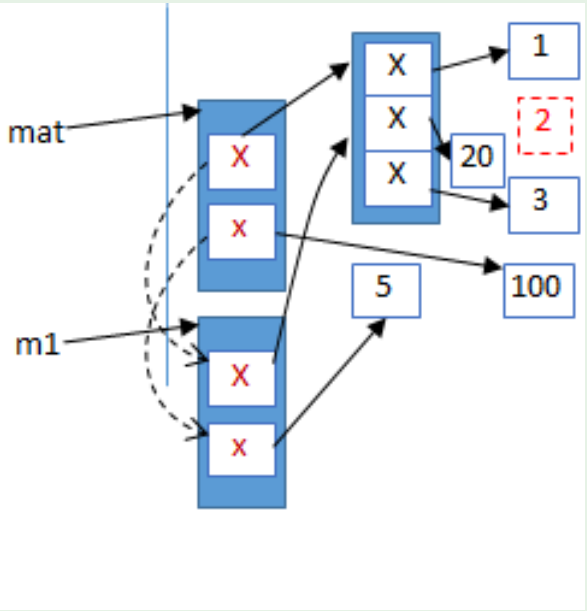
```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
>>> mat[1] = 100
```



Les listes

Exemple (Copie de liste)

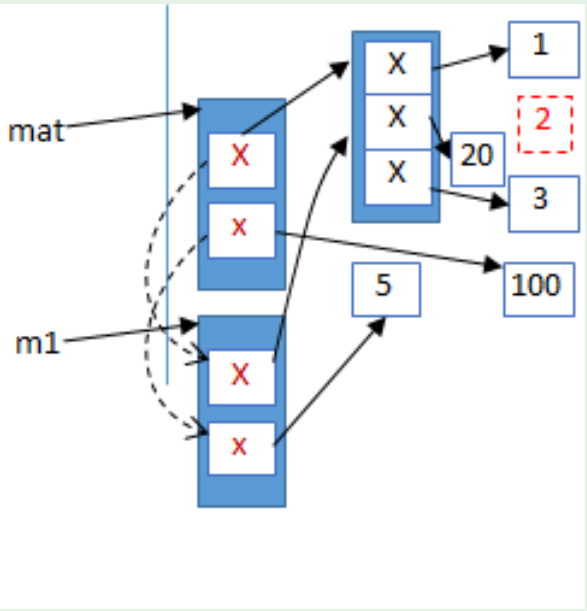
```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
>>> mat[1] = 100
>>> mat[0][1] = 20
>>>
```



Les listes

Exemple (Copie de liste)

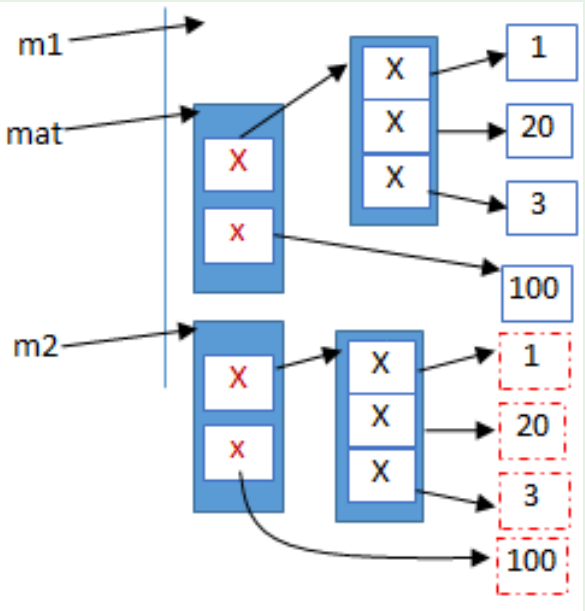
```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3], 100]
>>> m1
[[1, 20, 3], 5]
>>>
```



Les listes

Exemple (Copie de liste)

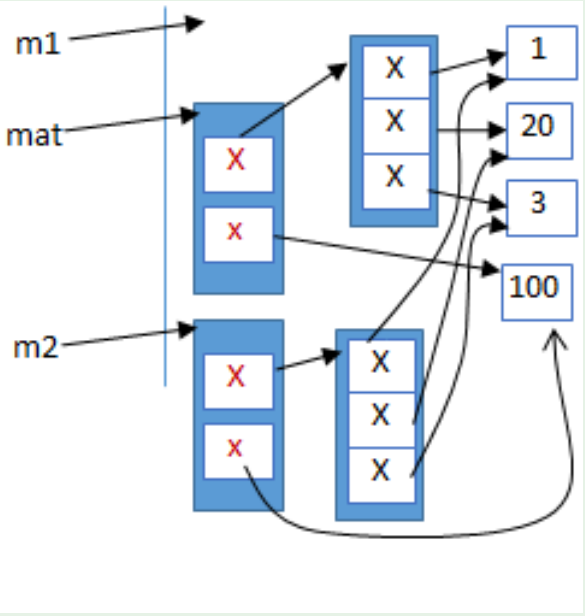
```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3], 100]
>>> m1
[[1, 20, 3], 5]
>>> m2 = copy.deepcopy(mat)
```

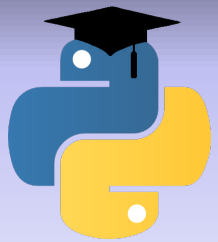


Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]
>>> m1 =mat[:]
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3], 100]
>>> m1
[[1, 20, 3], 5]
>>> m2 = copy.deepcopy(mat)
```

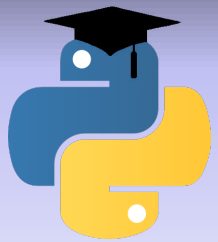




N-Uplets (Les tuples)

Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)

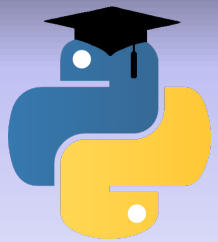


N-Uplets (Les tuples)

Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)

- Ils sont notés entre parenthèses **()**, ou une suite d'éléments séparés par des virgules
- l'accès aux éléments du tuple est réalisé via les indices placés entre crochets **[ ]** (comme pour les listes).



N-Uplets (Les tuples)

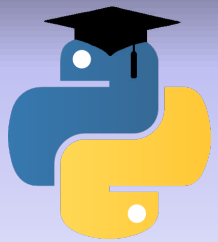
Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)

- Ils sont notés entre parenthèses **()**, ou une suite d'éléments séparés par des virgules
- l'accès aux éléments du tuple est réalisé via les indices placés entre crochets **[ ]** (comme pour les listes).

Remarques

- Si un seul élément est dans un tuple, le faire suivre d'une virgule.
- Le tuple possède les mêmes **méthodes** que la liste, à l'exception de celles permettant une modification.



N-Uplets (Les tuples)

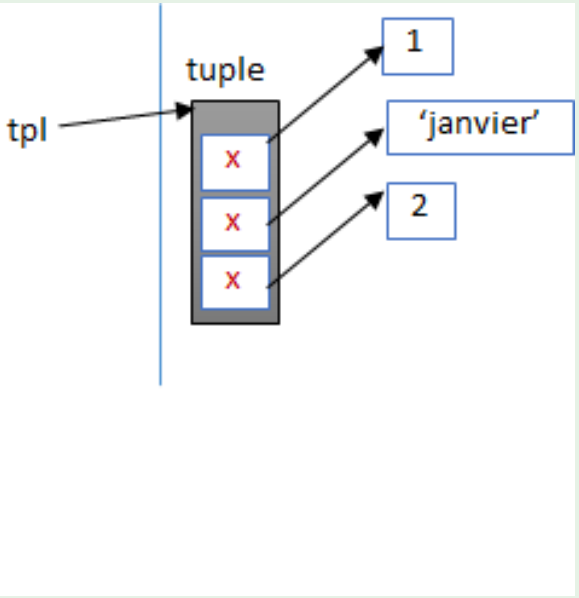
Exemple (Tuple)

```
>>> nb = (1) ; ch = ('ab')       # => entier & str !
>>> tpl = (1,)                   # ou  tpl =1, => tuple
>>> tpl = (1, 'janvier', 2)      # on peut omettre ( )
>>> len(tpl)                     # => 3 (nombre d'éléments)
>>> tpl[0:2]                    # => le tuple (1, 'janvier')
>>> tpl[2] = 3                   # => erreur (tuple non modifiable)
>>> tpl +=  5, 'mars'            # crée un nouv objet tuple
>>> 'mars' in tpl                # => True
>>> tpl2 = tuple([5,6,12]) # copie liste => nouv tuple (5,6,12)
>>> tpl3 = tuple('hello')  # copie chaine => tuple
                           #    => ('h','e','l','l','o')
>>> lst2= list(tpl2)             # copie tuple vers liste
                           #  => [5,6,12]
```

N-Uplets (Les tuples)

Exemple (Tuple)

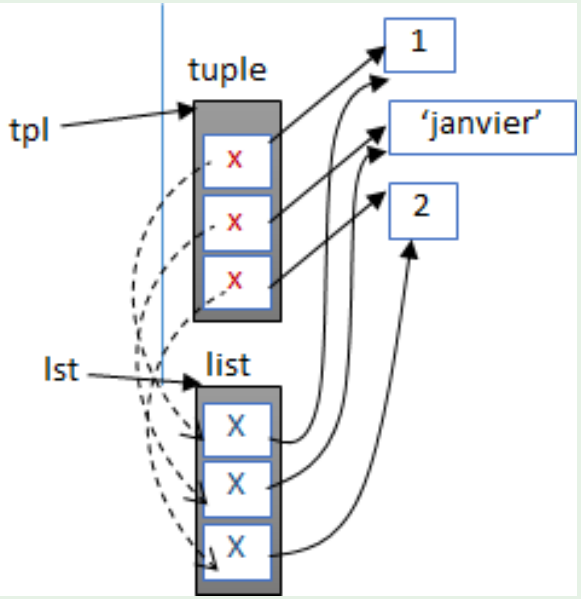
```
>>> tpl = 1, 'janvier', 2
>>>
```

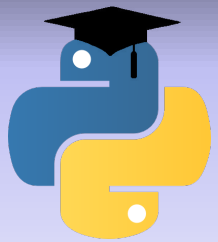


N-Uplets (Les tuples)

Exemple (Tuple)

```
>>> tpl = 1, 'janvier', 2
>>> lst = list(tpl)
>>>
```

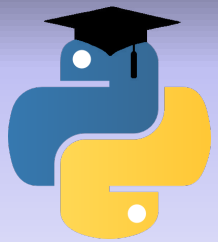




Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.

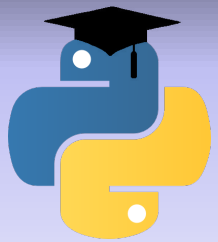


Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.

- Il est noté entre accolades $\{\}$, ou créé à partir d'une liste de couples (clé, valeur) passée en argument à la fonction **dict()**.



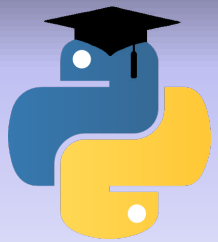
Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.

- Il est noté entre accolades {}, ou créé à partir d'une liste de couples (clé, valeur) passée en argument à la fonction **dict()**.
  Exemple :

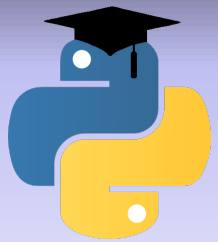
```
dic = {1: 'a', 2: 'b'}
# ou élément après élément :
dic={}; dic[1] = 'a' ; dic[2] = 'b'
# en utilisant la fonction dict()
dic = dict([(1, 'a'), (2, 'b')])
dic = dict(zip((1, 2), ('a', 'b')))
len(dic) # => 2 paires clé:valeur
1 in dic # test existence clé => True
5 in dic # => False, car 5 n'est pas une clé
```



Dictionnaire

Propriétés

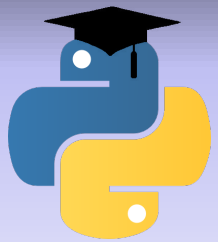
- *Les* **clés** *doivent être de type non modifiable (exemple : les types simples) et uniques.*
- *Les valeurs peuvent être de n'importe quel type*
- *Les valeurs sont accessibles par leurs clés*
- *Pas de notion d'ordre*



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
  - Récupération d'une valeur

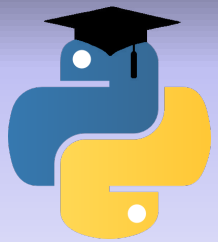
```
dic[1] # => 'a'
dic['cx'] # retourne erreur KeyError
dic.get(2) # => 'b'
dic.get('cx','erreur blabla') # => 'erreur blabla'
```



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
  - Récupération d'une valeur
  - Ajout, modification et suppression des éléments

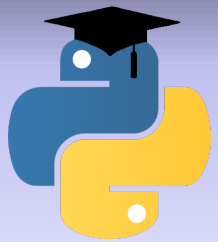
```
dic['c'] = 3 # => {1: 'a', 'c': 3, 2: 'b'}
dic[2] = 10 # => {1: 'a', 'c' : 3, 2: 10 }
del dic[1] # supp. de l'élément 1: 'a'
val = dic.pop(2) # => 'b' et supp. de l'élément 2:'b'
```



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
  - Récupération d'une valeur
  - Ajout, modification et suppression des éléments
  - Fusion de dictionnaires et mise à jour

```
dic.update({2: 20, 'c': 30}) # dic.update(dict)
# => dic = {1: 'a', 2: 20, 'c': 30}
```

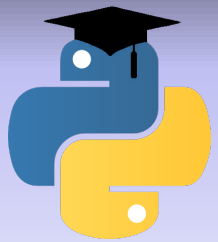


Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
  - Récupération d'une valeur
  - Ajout, modification et suppression des éléments
  - Fusion de dictionnaires et mise à jour
  - Parcourir un dictionnaire (via des objets itérables par une boucle **for**)

```
dic.keys() # => dict_keys([1, 2])
dic.values()  # => dict_values(['a', 'b'])
dic.items() # => dict_items([(1, 'a'), (2, 'b')])

# copie sur liste ou tuple
list(dic.keys()) # => [1, 2]
tuple(dic.values()) # => ('a', 'b')
```



Dictionnaire

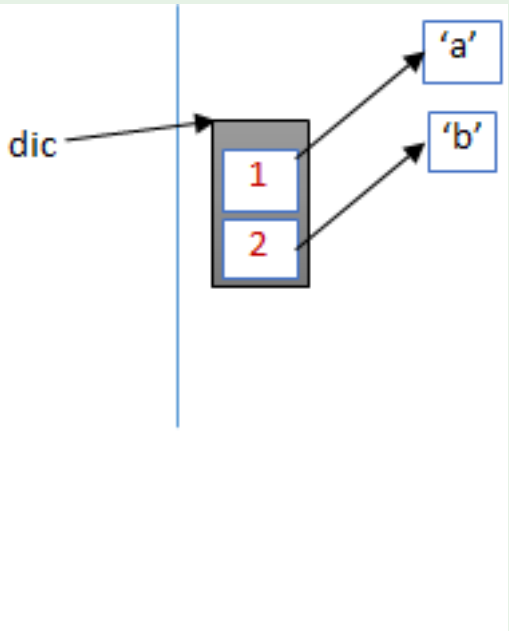
- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
  - Récupération d'une valeur
  - Ajout, modification et suppression des éléments
  - Fusion de dictionnaires et mise à jour
  - Parcourir un dictionnaire (via des objets itérables par une boucle **for**)
  - Copie d'un dictionnaire

```
dict1 = dic.copy() # => copie superficielle
# import copy
dic2 = copy.deepcopy(dic) # => copie profonde
```

Dictionnaire

Exemple

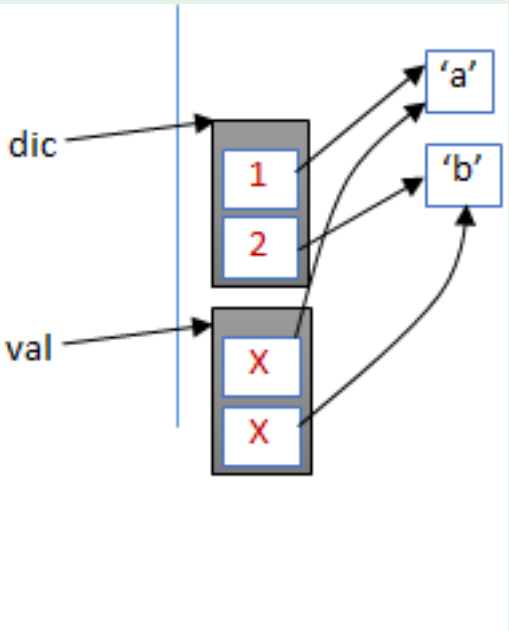
```
>>> dic = {1: 'a', 2 : 'b'}
>>>
```



Dictionnaire

Exemple

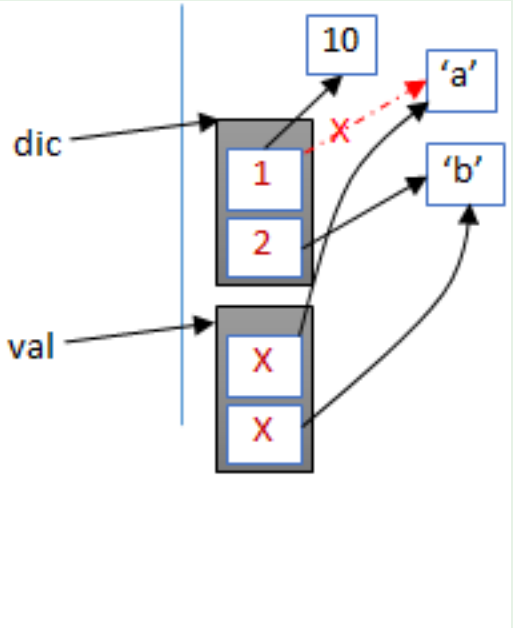
```
>>> dic = {1: 'a', 2 : 'b'}
>>> val = list(dic.values())
```



Dictionnaire

Exemple

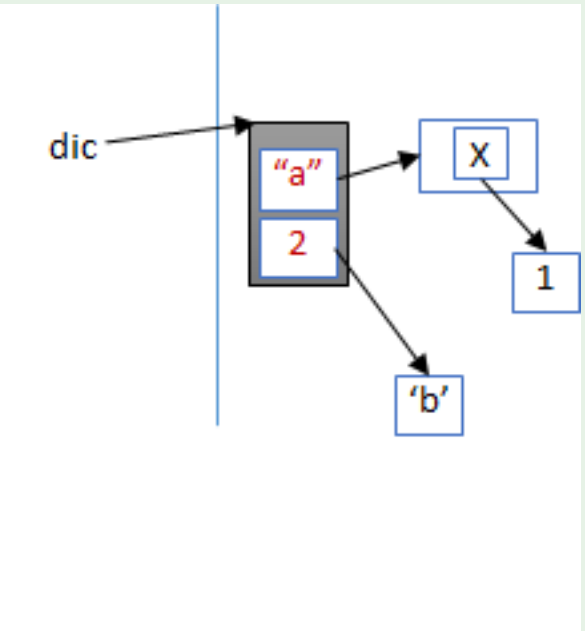
```
>>> dic = {1: 'a', 2 : 'b'}
>>> val = list(dic.values())
>>> dic[1] = 10
>>> # dic==?  Val==?
```



Dictionnaire

Exemple

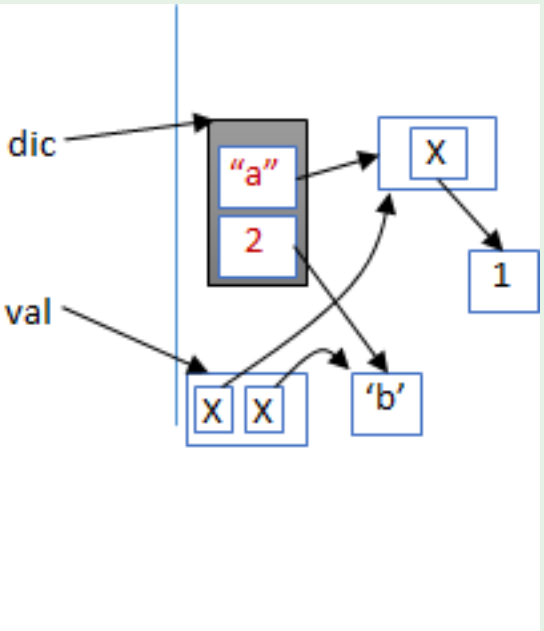
```
>>> dic = {'a' : [1] , 2 : 'b'}
```



Dictionnaire

Exemple

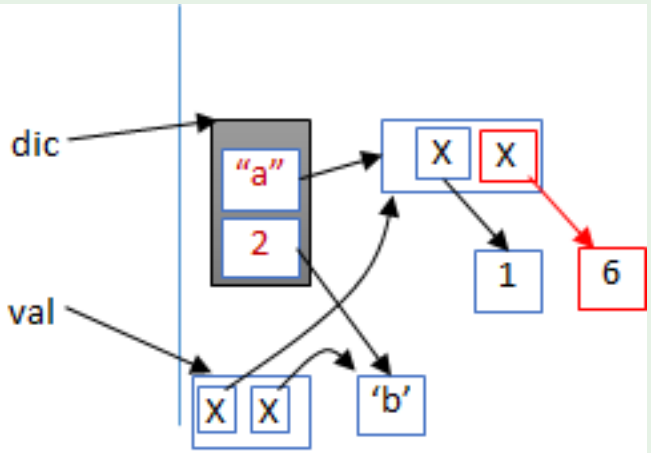
```
>>> dic = {'a' : [1] , 2 : 'b'}
>>> val = list(dic.values())
>>> # val == ?
```



Dictionnaire

Exemple

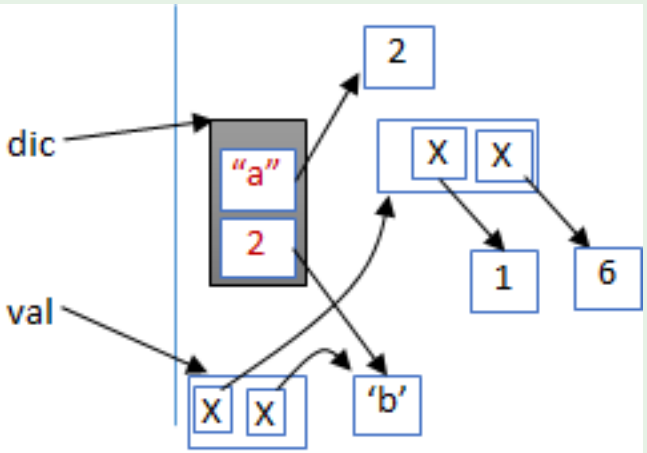
```
>>> dic = {'a' : [1] , 2 : 'b'}
>>> val = list(dic.values())
>>> # val == [ 'b', [1]]
>>> dic['a'].append(6)
>>> # val ==?
```

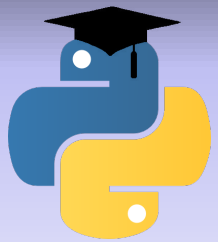


Dictionnaire

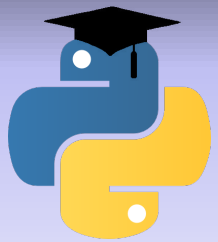
Exemple

```
>>> dic = {'a' : [1] , 2 : 'b'}
>>> val = list(dic.values())
>>> # val == [ 'b', [1]]
>>> dic['a'].append(6)
>>> # val ==['b', [1, 6]]
>>>  dic['a'] = 2
>>> # val == ?
```



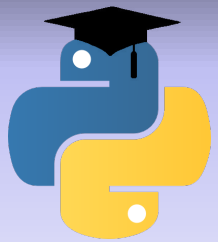


Sommaire



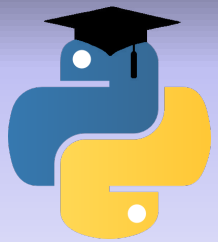
Transtypage / cast

- Objectif : convertir d'un type vers un autre.
- Exemple : `b = 23; ch = str(b)`



Transtypage / cast

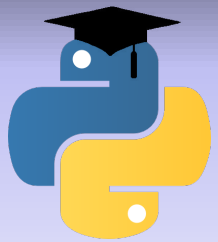
- Objectif : convertir d'un type vers un autre.
- Exemple : `b = 23; ch = str(b)`
- Utilisation des fonctions : int(), float(), bool(), str(), complex(), list(), tuple()



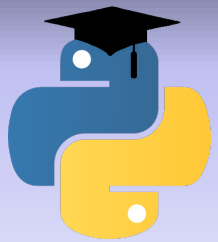
Transtypage / cast

Exemple (Transtypage)

```
>>> int(12.5)   # => 12
>>> int('125') # => 125
>>> float(125)  # => 125.0
>>> int("1000110" ,2) # =>  70
>>> complex("12+.5j") # => (12+0.5j)
>>> int(True) # => 1
>>> bool(0) # => False
>>> bool(124) # => True
>>> str(12+3j) # => '(12+3j)'
```



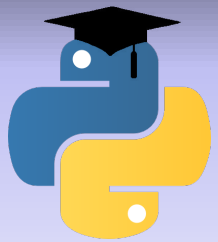
Sommaire



Affectation

Définition (Affectation (assignation))

Affectation : donner une ou plusieurs valeurs à une ou à plusieurs variables réalisée par l'opérateur "=". Elle peut être simple ou multiple.



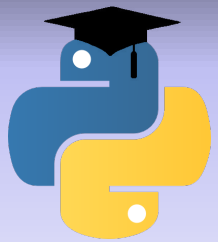
Affectation

Définition (Affectation (assignation))

Affectation : donner une ou plusieurs valeurs à une ou à plusieurs variables réalisée par l'opérateur"=". Elle peut être simple ou multiple.

Exemple (Affectations)

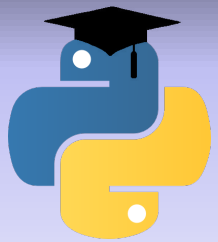
```
# affectation simple
a = 2
# affectation multiple
a = b = 5 # => a=5 et b=5
a, b = b, a # échange le contenu de a et b
c, d, e = 1, 5, 6 # => c=1; e=5 et e=6
a, *reste = [1, 2, 3] # => a=1 et reste=[2, 3]
```



Affectation

Exemple (Forme condensée)

```
a = b = 2
a += 2
b *= 3
```

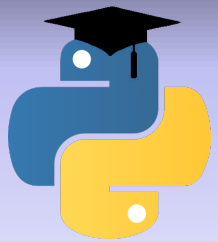


Affectation

Exemple (Forme condensée)

```
a = b = 2
a += 2
b *= 3
```

- Affectation très utile : i += 1 (ou i = i+1)



Affectation

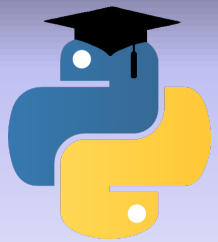
Exemple (Forme condensée)

```
a = b = 2
a += 2
b *= 3
```

- Affectation très utile : i += 1 (ou i = i+1)
- Autres formes condensées avec : +=, -=, *=, /=, //=, %=, **=, >>=, <<=, &=, ^=, |=.

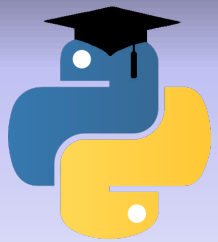
Attention

Python n'accepte pas les expressions : `i++`, `i--`, `++i` et `--i`



Comparaisons

Opérateur	Exemple	Signification
`>`	`a > 10`	strictement supérieur
`<`	`a < 10`	strictement inférieur
`>=`	`a >= 10`	supérieur ou égal
`<=`	`a <= 10`	inférieur ou égal
`==`	`a == 10`	égal à
`=!`	`a = 10!`	différent de
`is`	`a is b`	a et b représentent le même objet
`is not`	`a is not b`	a et b ne représentent pas le même objet

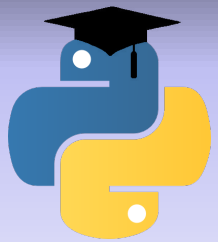


Comparaisons

Opérateur	Exemple	Signification
`>`	`a > 10`	strictement supérieur
`<`	`a < 10`	strictement inférieur
`>=`	`a >= 10`	supérieur ou égal
`<=`	`a <= 10`	inférieur ou égal
`==`	`a == 10`	égal à
`=!`	`a = 10!`	différent de
`is`	`a is b`	a et b représentent le même objet
`is not`	`a is not b`	a et b ne représentent pas le même objet

Remarque

- Le test d'égalité `==` concerne le contenu,
- le test `is` concerne la référence



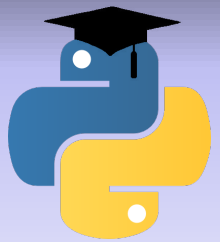
Comparaisons

Remarque

- Le test d'égalité == concerne le contenu,
- le test `is` concerne la référence

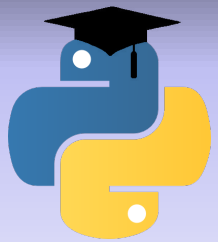
Exemple (comparaison)

```
>>> 2 is 2
True
>>> 2+2 == 4
True
>>> [2+2] is [4]
False
>>> [2+2] == [4]
True
```



Opérations logiques

Opérateur	Exemple	Signification
`not`	`not a`	NON logique
`and`	`a and b`	ET logique : évaluation paresseuse
`or`	`a or b`	OU logique : évaluation paresseuse
`&`	`a & b`	ET logique
`^`	`a ^ b`	OU exclusif logique
`|`	`a | b`	OU logique

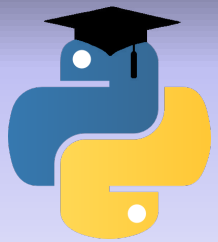


Opérations logiques

Opérateur	Exemple	Signification
`not`	`not a`	NON logique
`and`	`a and b`	ET logique : évaluation paresseuse
`or`	`a or b`	OU logique : évaluation paresseuse
`&`	`a & b`	ET logique
`^`	`a ^ b`	OU exclusif logique
`|`	`a | b`	OU logique

Remarque : évaluation paresseuse

- `a and b` évalue et retourne a si a est False, sinon évalue et retourne b.
- `a or b` évalue et retourne a si a est True, sinon évalue et retourne b



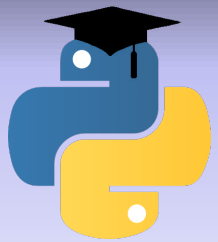
Opérations logiques

Opérateur	Exemple	Signification
`not`	`not a`	NON logique
`and`	`a and b`	ET logique : évaluation paresseuse
`or`	`a or b`	OU logique : évaluation paresseuse
`&`	`a & b`	ET logique
`^`	`a ^ b`	OU exclusif logique
`|`	`a | b`	OU logique

Remarque : évaluation paresseuse

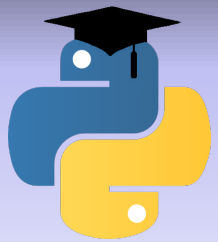
- `a and b` évalue et retourne a si a est False, sinon évalue et retourne b.
- `a or b` évalue et retourne a si a est True, sinon évalue et retourne b

```
>>> 5 and 6 # => retourne 6
>>> 0 and 5 # => retourne 0
>>> 5 or 0 # => retourne 5
>>> 0 or 5 # => retourne 5
```



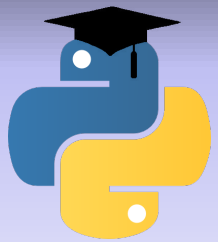
Opérateurs sur les bits

Opérateur	Exemple	Signification
`<<`	`a << n`	décalage à gauche de *n* bits
`>>`	`a >> n`	décalage à droite de *n* bits
`>>>`	`a >>> n`	décalage à droite non signé de *n* bits
`~`	`~a`	NON binaire
`&`	`a & b`	ET binaire
`^`	`a ^ b`	OU exclusif binaire
`|`	`a | b`	OU binaire

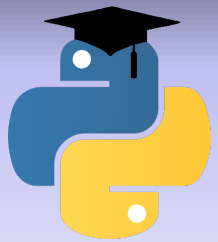


Priorité des opérateurs

Nom de l'opérateur	Symbole
Parenthèses, crochets et accolades	`( )`, `[ ]`, `{ }`
Accès à des éléments d'une séquence	`x[i]`, `x[i:j]`, `x[i:j:k]`
Puissance	`**`
Plus, moins et NON unaire	`+x`, `-x`, `~x`
Multiplication, division, division entière, et modulo	`*`, `//`, `/`, `%`
Addition et soustraction	`+`, `-`
Décalage	`<<`, `>>`
ET binaire	`&`
OU EXCLUSIF binaire	`^`
OU binaire	`|`
Comparaison	`<`, `>`, `<=`, `>=`, `==`, `=!`, `in`, `not in`, `is`, `is not`
NON booléen	`not`
ET booléen	`and`
OU booléen	`or`
Les opérateurs d'affectation	`=` `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `>>=`, `<<=`, `&=`, `^=`, `|=`

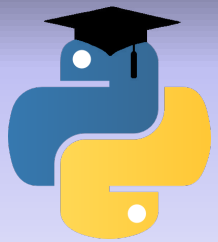


Sommaire



Règles de base

- Python est sensible à la casse

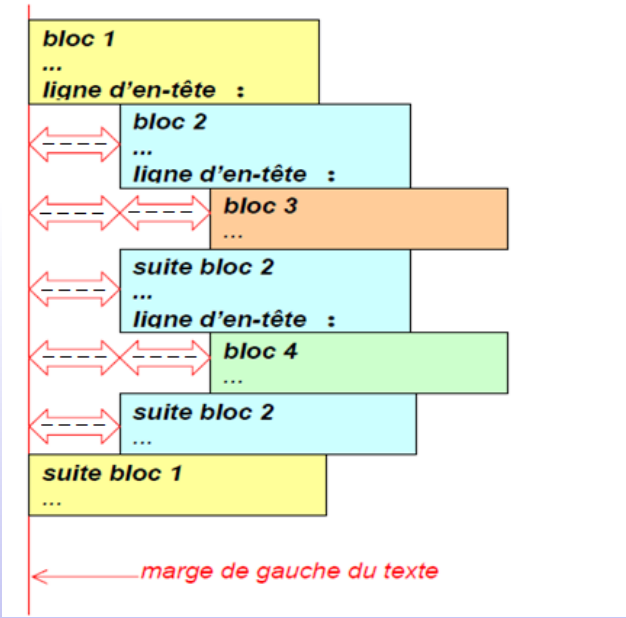


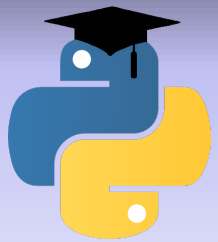
Règles de base

- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes

Règles de base

- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes
- Un bloc de code commence par " : " et est indenté plus à droite pour le bloc contenant
  Exemple :



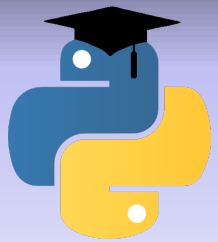


Règles de base

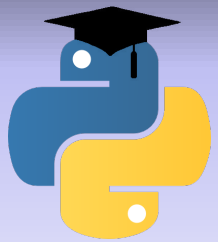
- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes
- Un bloc de code commence par " **:** " et est indenté plus à droite pour le bloc contenant

Remarques

- Il est recommandé d'utiliser 4 <espace> pour un niveau d'indentation
- Un bloc de code doit contenir au minimum une instruction. S'il n'en a pas, on peut utiliser l'instruction **pass**

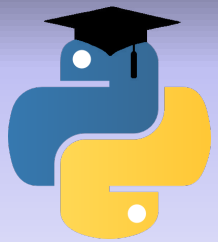


Sommaire



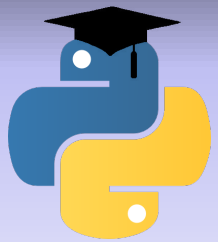
Commentaires

- Commentaires : remarques en langage naturel



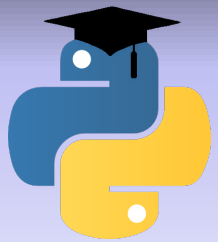
Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par le compilateur



Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par le compilateur
- Deux types de commentaires :

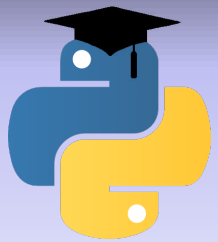


Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par le compilateur
- Deux types de commentaires :

Exemple (Commentaire sur une ligne)

```
# Un division réelle
a = 12/5
b = a // 3 # division entière
```

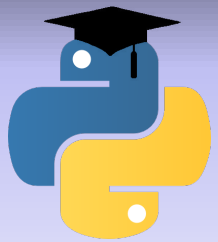


Commentaires

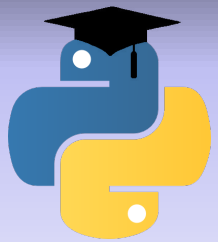
- Commentaires : remarques en langage naturel
- Ignorés par le compilateur
- Deux types de commentaires :

Exemple (Documentation automatique)

```
def somProd(var1, var2):
    """Fonction calculant somme et produit
    de var1 et var2
    Retour :
    un tuple (var1+var2, var1*var2)"""
    return (var1+var2, var1*var2)
```



Sommaire

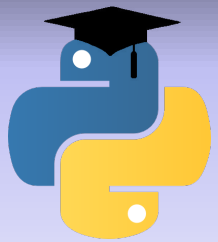


Types de littéraux

- Entiers : base 10

Exemple (Littéral)

42, -7

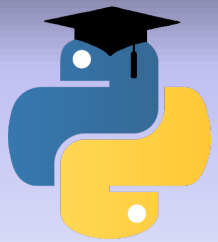


Types de littéraux

- Entiers : base 10 , base 16

Exemple (Littéral)

0x9AE3

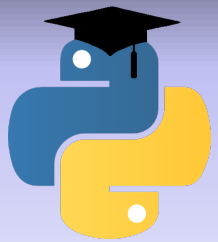


Types de littéraux

- Entiers : base 10 , base 16 , base 8

Exemple (Littéral)

0o20 (= 16)

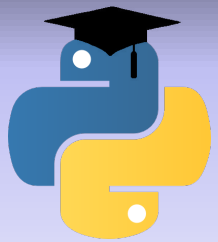


Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2

Exemple (Littéral)

0b101010 (= 42)

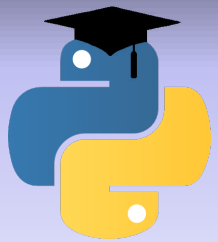


Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision

Exemple (Littéral)

3.14, 6.02E23 ($= 6.02.10^{23}$)

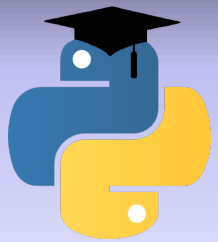


Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens

Exemple (Littéral)

True, False

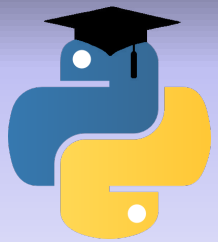


Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens
- Complexes

Exemple (Littéral)

1j, (-1)**0.5, 3**0.5/2+0.5j

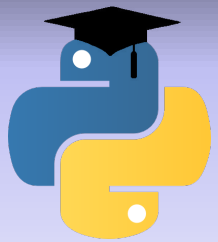


Types de littéraux

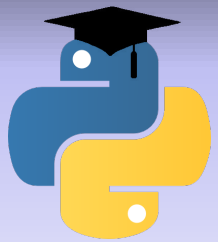
- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens
- Complexes
- Chaînes de caractères

Exemple (Littéral)

"Une chaîne de caractères", 'l\'expression "dauphine" aussi' 'Ex\u0065llent travail' (='Excellent travail')

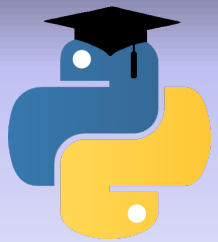


Sommaire

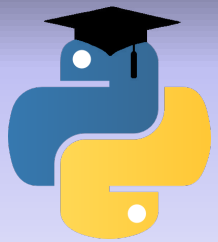


Les 33 mots clés

and	as	assert	break	class	continue
def	del	elif	else	except	False
finally	for	from	global	if	import
in	is	lambda	None	nonlocal	not
or	pass	raise	return	True	try
while	with	yield			

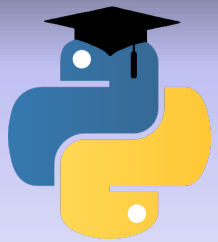


Sommaire



if, elif, else

- Teste une condition booléenne
- Si (`if`) **vrai**, exécute une partie du code
- Sinon (`else`), exécute autre partie

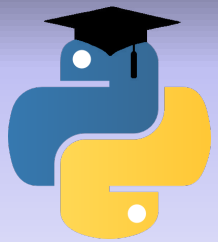


if, elif, else

- Teste une condition booléenne
- Si (`if`) **vrai**, exécute une partie du code
- Sinon (`else`), exécute autre partie

Exemple (if, else)

```
if expressionBooleenne1:
    action1
else:
    if expressionBooleenne2:
        action2
    else:
        action3
```



if, elif, else

- Teste une condition booléenne
- Si (`if`) **vrai**, exécute une partie du code
- Sinon (`else`), exécute autre partie

Exemple (if, elif, else)

```
if expressionBooleenne1:
    action1
elif expressionBooleenne2:
    action2
else:
    action3
```

Remarque

- **if** peut s'écrire sans **else**

Remarque

- **if** peut s'écrire sans **else**

Correct mais à proscrire

```
if condition:
    pass
else:
    action
```

Remarque

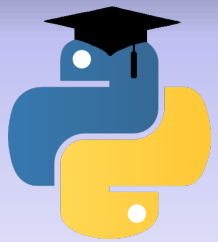
- **if** peut s'écrire sans **else**

Correct mais à proscrire

```
if condition:
    pass
else:
    action
```

Forme préférée

```
if not condition :
    action
```



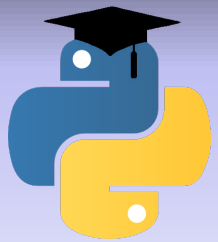
Conditions multiples, et fonctions logiques

Exemple (conditions multiples)

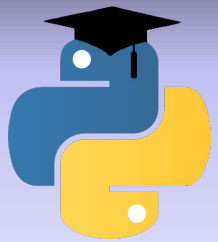
```
if 0 <= a < 10:
# équivalence
if (0 <= a) and (a < 10):
```

Remarques (fonctions logiques : any, all)

- Tester des séquences et des objets itérables
  - `any(objet)` : retourne `True` si au moins un des éléments est True
    ```
    any(range(100)) # => True
    ```
  - `all(objet)` : retourne True si tous les éléments sont True
    ```
    all(range(100)) # => False
    ```

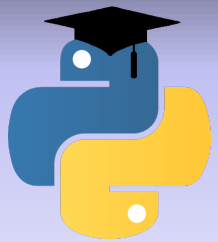


Sommaire



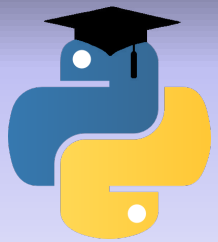
Principe

- Boucle = structure de contrôle
- But : exécuter certaines opérations plusieurs fois
- Il existe deux types de boucles en Python 3 : for, while
- Les deux boucles sont équivalentes



while

```
while condition :
    actions
```

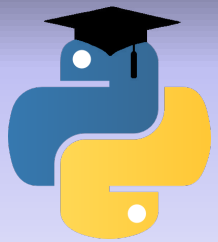


while

```
while condition :
      actions
```

Exemple (boucle while)

```
i=0
while i<10: # boucle tant que i<10
      print(i) # affiche la valeur de i
      i = i+1 # incrémente i
# maintenant i vaut 10
```

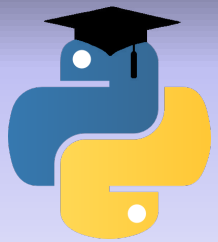


while : Sortie de boucle

- Un bloc **else :** peut être ajouté à la fin de la boucle
- Le mot clé **break** permet de sortir de la boucle
- Le mot clé **continue** permet de passer immédiatement à l'itération suivante

Exemple (do ... until)

```
while True :
    # corps de la boucle
    if exitcondition :
        break
```



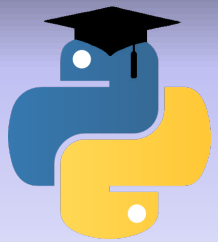
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Exemple

```
s = ['septembre', 'octobre']
for n in s:
    print(n, end=' ') # => septembre octobre
```



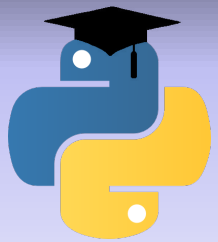
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Exemple

```
s = "Python"
for n in s:
    print(n, end=' ') # => P y t h o n
```



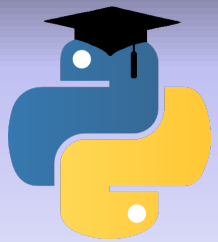
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Exemple

```
s = 1,2,3,4
for n in s:
    print(n, end=' ')  # => 1 2 3 4
```



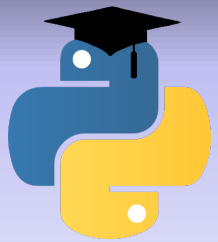
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Exemple

```
s = {'a':10, 'b':'deux'}
for n in s: # s.values(), s.keys(), s.items()
    print(n, end=' ')  # => a b
```



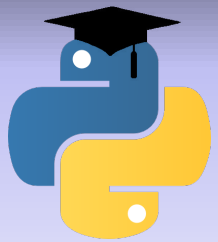
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Exemple

```
s = {'a':10, 'b':'deux'}
for n, val in s.items():
    print(n, end=' ')  # => a b
```



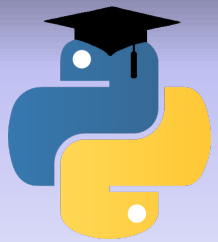
for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable
Syntaxe :

```
for variable in collection:
    actions # corps de la boucle
```

Remarque (for : sortie de boucle)

Possibilité de modifier l'exécution d'une boucle **for** avec **break** et **continue** comme dans le cas de la boucle while.



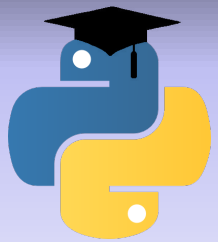
Itérateur

Remarque

- L'itérateur **range(deb,fin, inc)** permet d'itérer sur une suite de nombres entiers,
- La fonction **enumerate(sequence)** retourne un objet permettant d'itérer sur l'indice et la valeur d'une séquence

Exemple

```
s = ['septembre', 'octobre']
for i in range(2):
      print(s[i], end=' ') # => septembre octobre
```



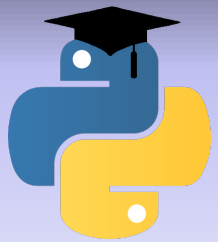
Itérateur

Remarque

- L'itérateur **range(deb,fin, inc)** permet d'itérer sur une suite de nombres entiers,
- La fonction **enumerate(sequence)** retourne un objet permettant d'itérer sur l'indice et la valeur d'une séquence

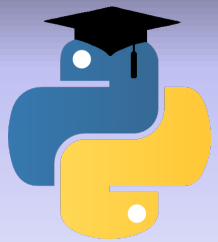
Exemple

```
s = ['septembre', 'octobre']
for ind, val in enumerate(s):
     print(ind, val, end = ' ')
     # => 0 septembre 1 octobre
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

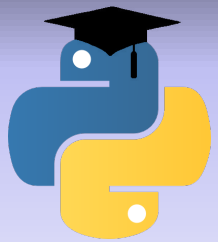


for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
dict = {expr1:expr2 for expr in iterable if cond}
```



for : forme condensée

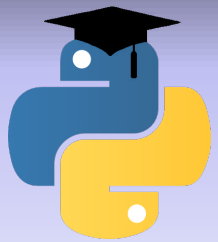
Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]
# => [4, 16, 36, 64, 100]
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

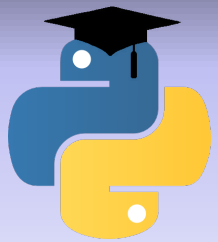
```
liste = [expression for expr in iterable if cond]
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]
# => [4, 16, 36, 64, 100]
```

Exemple 2 :

```
dic = {'a': 12, 'b': 1.5, 'c': 3}
dic1 = {cle:val for cle,val in dic.items() if
val**2>10} # => dic1 =
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

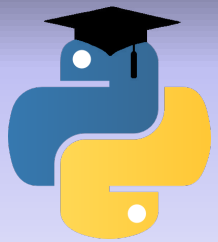
```
liste = [expression for expr in iterable if cond]
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

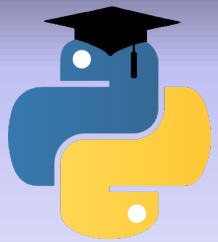
```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]
# => [4, 16, 36, 64, 100]
```

Exemple 2 :

```
dic = {'a': 12, 'b': 1.5, 'c': 3}
dic1 = {cle:val for cle,val in dic.items() if
val**2>10} # => dic1 = {'a': 12}
```



Sommaire



Hello World

hello.py

```
# affiche 'hello World"
""" Ceci est mon premier programme
avec un commentaire fait sur plusieurs lignes """
print("Hello World!")
```

- Exécution : **python3 hello.py ou python3 hello**
- Passage de paramètres par l'attribut **sys.argv**
- Ex : **python3 hello 1234 xyz**
  - sys.argv[0] ← hello
  - sys.argv[1] ← 1234
  - sys.argv[2] ← xyz
- Passage de paramètres par la fonction **input()**
- Ex : `nb = input("Donner votre paramètre :")`