

# RÉSUMÉ 18 - INTRODUCTION AUX INDEX NUMÉRIQUES

Lien vers l'activité : [Introduction aux index numériques](#)  
www.infoforall.fr - Dernière modif. : 14 11 2020



## 18.1 STRINGS

Un **string** est une séquence ordonnée composée de caractères indexables par un numéro. Avec Python, le string est *implémenté via le type natif est str qui est immuable\**.

### 1A DÉCLARATION : GUILLEMETS SIMPLES ' ' OU DOUBLES " "

```
1 a = "bonjour"
2 b = 'bonjour !'
```

### 1B NOMBRE DE CARACTÈRES STOCKÉS

On utilise la fonction native **len**.

```
1 a = "bonjour"
2 nbr_a = len(a) # Contient 7
3 b = 'bonjour !'
4 nbr_b = len(b) # Contient 9
```

### 1C LECTURE D'UN CARACTÈRE D'INDEX PRÉCIS AVEC [INDEX]

On utilise le nom du string, un crochet d'ouverture, le numéro d'index et un crochet de fermeture.

- index 0 veut dire le premier caractère.
- index 1 veut dire le deuxième caractère .
- ..
- index -1 veut dire le dernier caractère (avec Python, ce n'est pas une généralité)
- index -2 veut dire l'avant-dernier caractère (avec Python, ce n'est pas une généralité).
- une lecture sur un index impossible déclenche une erreur.

```
1 a = "bonjour"
2 n0 = a[0] # La variable contient le 'b' initial
3 n1 = a[1] # La variable contient 'o'
4 n2 = a[2] # La variable contient 'n'
5 n6 = a[6] # La variable contient le 'r' final
6 nf = a[-1] # La variable contient le 'r' final
```

### 1D PARCOURS ET LECTURE DES ÉLÉMENTS VIA UNE BOUCLE FOR NUMÉRIQUE

Il suffit de lire les éléments de l'index 0 jusqu'à l'index "longueur-1".

```
1 a = 'bonjour'
2 longueur = len(a)
3
4 for index in range(longueur) :
5     print(a[index], end="-")
```

La variable de boucle **index** va prendre successivement les valeurs

- 0 1 2 ect et cela va provoquer l'affichage suivant :
- b-o-n-j-o-u-r-

### 1E PARCOURS ET LECTURE VIA UNE BOUCLE FOR NOMINATIVE

Plus "clair" que la précédente mais on perd la connaissance de l'index du caractère en cours.

```
1 mot = "bonjour"
2 for caractere in mot :
3     print(caractere, end='-')
```

La variable **caractere** va prendre successivement les valeurs 'b', puis 'o', puis 'n' ...

### 1F NON MUTABILITÉ DU STRING

En Python, le string est **immuable** ou **non mutable** : on ne peut pas modifier les caractères après création du string. Le code suivant **provoquerait une erreur** :

```
1 a = "bonjour"
2 a[0] = "B"
```

Si on veut **modifier** un string, on ne peut pas réellement le faire : il faut recréer un string et écraser l'ancienne

variable par une variable ayant le même nom.

Exemple avec un code qui transforme les o en X en créant **par concaténation un nouveau string**. Il ne s'agit donc pas réellement d'une modification : la variable **mot** de la ligne 7 est bien une nouvelle variable car son ID-mémoire n'est pas celui de la variable de la ligne 1.

```
1 mot = 'bonjour'
2 temp = ""
3 for caract in mot :
4     if caract == 'o' :
5         caract = 'X'
6     temp = temp + caract
7 mot = temp
```

**Une fonction ne peut donc pas modifier un string** : elle ne peut que renvoyer un nouveau string. Cette réponse pourra éventuellement écraser l'ancienne variable.

## 18.2 TABLEAUX

### 2A DÉCLARATION AVEC DES CROCHETS [ ]

Un **tableau** (statique) est une structure de données ayant les propriétés suivantes :

- nombre fixé de cases après la déclaration initiale du tableau.
- même type de données dans chaque case
- indexable par un numéro (comme les strings)

Avec Python, les **tableaux** sont implémentés via le type natif **list** qui est **mutable**.

Voici deux exemples :

```
>>> tab1 = ['a', 'b', 'c']
>>> tab2 = [45, 78, 128]
>>> tab1
['a', 'b', 'c']
```

Nous utilisons ici le type natif **list** pour implémenter les tableaux en Python.

### 2B NOMBRE D'ÉLÉMENTS STOCKÉS

On utilise la fonction native **len**.

### 2C LECTURE D'UN ÉLÉMENT D'INDEX PRÉCIS AVEC LES CROCHETS [INDEX]

On utilise le nom de la variable suivi de crochets contenant le numéro d'index voulu. Exemple :

```
>>> tab1[0]
'a'
>>> tab1[2]
'c'
```

Python : les numéros d'index négatifs font référence aux derniers éléments :

```
>>> tab1[-1]
'c'
>>> tab1[-2]
'b'
```

### 2D LECTURE SÉQUENTIELLE DES ÉLÉMENTS AVEC UN FOR NUMÉRIQUE

**Il faut utiliser cette façon de faire si vous voulez MODIFIER le tableau** ou si vous avez besoin de connaître les numéros d'index des éléments.

```
1 notes = [15, 18, 8, 10, 12, 15, 20, 5, 12, 17, 12, 10, 18, 4]
2 nbr = len(notes)
3
4 for index in range(nbr) :
5     print(notes[index])
```

On obtient successivement 15 puis 18...

### 2E LECTURE SÉQUENTIELLE DES ÉLÉMENTS AVEC UN FOR NOMINATIF

Attention : plus "clair" mais **ne permet pas la modification** des éléments du tableau.

```
1 notes = [15, 18, 8, 10, 12, 15, 20, 5, 12, 17, 12, 10, 18, 4]
2
3 for element in notes :
4     print(element)
```

### 2F MUTABILITÉ DES TABLEAUX

En Python, les tableaux sont **mutables** ou **mutables** : on peut modifier le contenu d'une case sans modifier l'ID-

mémoire du tableau.

```
>>> tabl = [45,28,89]
>>> tabl[0] = 1000
>>> tabl
[1000, 28, 89]
```

## 2H DÉCLARATION DE TABLEAU PAR COMPRÉHENSION

**Depuis un tableau de base** : on peut déclarer un **nouveau** tableau en définissant ses éléments à l'aide des éléments d'une autre structure indexable, comme un autre tableau de **base** par exemple.

```
1 base = [1, 2, 5]
2 nouveau = [element*10 for element in base]
```

Le tableau **nouveau** va alors contenir [10, 20, 50].

On peut également créer un nouveau tableau simplement à partir d'une succession numérique qu'on peut créer à l'aide d'un **in range** :

```
1 nouveau = [x*10 for x in range(5)]
```

Le tableau **nouveau** va alors contenir [0, 10, 20, 30, 40].

Enfin, on peut même utiliser des tests pour savoir si on agit sur l'élément ou pas :

```
1 base = [12, 8, 1, 3, 3, 17, 8]
2 nouveau = [note for note in base if note > 10]
```

Le tableau **nouveau** va alors contenir [12, 17].

## 18.3 P-UPLET OU TUPLE

Le p-uplet est une structure ordonnée d'éléments

- comportant p éléments
- permettant de stocker des données de natures différentes (contrairement aux strings ou aux tableaux)
- indexable par un numéro (comme les strings et les tableaux)

Dans Python, les p-uplets sont implémentés via le type natif **tuple** qui est **immutable**.

## 3A - DÉCLARATION AVEC DES PARENTHÈSES

On utilise les parenthèses comme élément de déclaration et on utilise les virgules pour séparer les éléments.

```
>>> exemple = ('Ananas' , 50)
>>> exemple
('Ananas', 50)
>>> type(exemple)
<class 'tuple'>
```

**Tuple ne comportant qu'un élément** : rajoutez bien la virgule même si vous n'avez qu'un élément.

```
>>> bon_exemple = ('Ananas' ,)
>>> bon_exemple
('Ananas',)
>>> type(bon_exemple)
<class 'tuple'>

>>> mauvais_exemple = ('Bananes')
>>> mauvais_exemple
'Bananes'
>>> type(mauvais_exemple)
<class 'str'>
```

## 3B - NOMBRE D'ÉLÉMENTS

On utilise la fonction native **len**.

## 3C - LECTURE D'UN ÉLÉMENT INDEXÉ AVEC [INDEX]

On utilise les crochets et le numéro d'index comme avec les strings et les tableaux.

```
>>> exemple = ('Ananas' , 50)
>>> exemple[0]
'Bananes'
>>> exemple[1]
50
```

## 3D ET 3E : LECTURE AVEC UNE BOUCLE FOR

Comme avec les deux structures précédentes : deux façons de faire mais on perd la valeur de l'index avec le FOR nominatif.

### 3F NON MUTABLE

Avec Python, le type natif **tuple** est **immuable / non mutable** : on ne peut pas modifier le contenu d'un tuple après création. Comme avec le string, on peut par contre écraser l'ancienne variable par une variable qui porte le même nom.

### 3G RETOUR MULTIPLE

L'un des intérêts des tuples est lié à la capacité de **renvoyer plusieurs variables via le return**. On peut se passer des parenthèses autant dans le return qu'à la réception.

Néanmoins, autant les mettre, cela montre **explicitement** la nature du tuple qu'on envoie ou qu'on reçoit.

Exemple :

```
1 def test() :
2     return (50,100)
3
4 x,y = test()
```

## 18.4 PASSAGE D'UN TYPE À L'AUTRE

- **str** tente de transformer les données fournies en une string
- **list** tente de transformer les données fournies en un tableau (list)
- **tuple** tente de transformer les données fournies en un p-uplet (tuple)

La transformation de string en tableau en utilisant un élément séparateur est possible avec la méthode des strings nommée **split**. Très pratique pour créer un tableau à partir d'un fichier texte.

```
>>> phrase = "Voici un exemple de phrase, il permet de compter les mots."
>>> mots = phrase.split()
>>> mots
['Voici', 'un', 'exemple', 'de', 'phrase,', 'il', 'permet', 'de', 'compter', 'les', 'mots.']

>>> nbr_mots = len(mots)
>>> nbr_mots
```

La version encapsulée dans une fonction : une tâche = une fonction.

Et du coup, nous pouvons en faire une fonction :

```
1 def compter_mots(phrase)
2     '''Renvoie le nombre de mots dans le string phrase'''
3     mots = phrase.split()
4     return len(mots)
```

## 18.5 APPARTENANCE

On peut tester si un élément est présent dans un string, un tableau ou un tuple avec le mot-clé **in**.

```
>>> tableau = [12, 5, 18]
>>> 5 in tableau
True
>>> 6 in tableau
False

>>> nuplet = ("alors", "ça", "va", "?")
>>> "ça" in nuplet
True
>>> "ca" in nuplet
False

>>> chaine = "Alors, ça va ?"
>>> "Alors" in chaine

>>> "Bonjour" in chaine
False
```

www.infoforall.fr

