



Option « Programmation en Python »

***Containers et autres fonctions***

# Les séquences ou *containers*

- ▶ En plus des types fondamentaux, Python propose nativement un ensemble d'objets à accès séquentiel dont :
  - ▶ les chaînes de caractère
  - ▶ les listes & *tuples*
  - ▶ les dictionnaires

# Les chaînes de caractères

```
In [1]: citation = "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

```
In [2]: type(citation)
```

```
Out[2]: str
```

```
In [3]: len(citation)
```

```
Out[3]: 57
```



La fonction `len()` comme la fonction `type()` sont toutes deux des fonctions intégrées au langage Python

# Les chaînes de caractères

👉 Apostrophe, guillemets & triple guillemets

## ► Chaîne de caractères délimitée par des guillemets

```
In [1]: "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

## ► Chaîne de caractères délimitée par de simples apostrophes

```
In [1]: 'Une noisette, j'la casse entre mes fesses tu vois... JCVD'
```

```
File "<ipython-input-17-39c8b67fd376>", line 1
```

```
'Une noisette, j'la casse entre mes fesses tu vois... JCVD'
```

```
SyntaxError: invalid syntax
```

```
In [2]: 'Une noisette, j\'la casse entre mes fesses tu vois... JCVD'
```

# Les chaînes de caractères

👉 Apostrophe, guillemets & triple guillemets

## ► Chaîne de caractères délimitée par des guillemets

```
In [1]: "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

## ► Chaîne de caractères délimitée par de simples apostrophes

```
In [1]: 'Une noisette, j'la casse entre mes fesses tu vois... JCVD'  
File "<ipython-input-17-39c8b67fd376>", line 1  
    'Une noisette, j'la casse entre mes fesses tu vois... JCVD'
```

**SyntaxError:** invalid syntax

```
In [2]: 'Une noisette, j\'la casse entre mes fesses tu vois... JCVD'
```

# Les chaînes de caractères

👉 Apostrophe, guillemets & triple guillemets

- Chaîne de caractères délimitée par des guillemets

```
In [1]: "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

- Chaîne de caractères délimitée par de simples apostrophes

```
In [2]: 'Une noisette, j\'la casse entre mes fesses tu vois... JCVD'
```

- Chaîne de caractères délimitée par des triples guillemets → **documentation de fonctions**

```
In [3]: """Une noisette,  
j'la casse entre mes fesses tu vois...  
JCVD"""
```

# Les chaînes de caractères

👉 Les méthodes associées

- Les méthodes ou fonctions membres associées aux objets de type str sont accessibles par le biais de **l'opérateur .**

```
In [1]: question = "Qu'est-ce qu'un chat qui travaille à la SNCF ?"
```

```
In [2]: question.upper()
```

```
Out[2]: "QU'EST-CE QU'UN CHAT QUI TRAVAILLE À LA SNCF ?"
```

```
In [3]: reponse = "un cheminou"
```

```
In [4]: reponse.capitalize().center(20)
```

```
Out[4]: '      Un cheminou      '
```

```
In [5]: reponse.capitalize().center(20).strip()
```

```
Out[5]: 'Un cheminou'
```

- L'ensemble de ces méthodes sont accessibles *via* l'aide en ligne de ipython i.e. `help(str)` ou en usant de la complétion soit `reponse.<TAB>`

# Les chaînes de caractères

👉 Les méthodes associées

- Les méthodes ou fonctions membres associées aux objets de type str sont accessibles par le biais de **l'opérateur** .

```
In [1]: question = "Qu'est-ce qu'un chat qui travaille à la SNCF ?"
```

```
In [2]: question.upper()
```

```
Out[2]: "QU'EST-CE QU'UN CHAT QUI TRAVAILLE À LA SNCF ?"
```

```
In [3]: reponse = "un cheminou"
```

```
In [4]: reponse.capitalize().center(20)
```

```
Out[4]: '    Un cheminou    '
```

```
In [5]: reponse.capitalize().center(20).strip()
```

```
Out[5]: 'Un cheminou'
```

- L'ensemble de ces méthodes sont accessibles *via* l'aide en ligne de ipython i.e. `help(str)` ou en usant de la complétion soit `reponse.<TAB>`



# Les chaînes de caractères

👉 Les méthodes associées

- Les méthodes ou fonctions membres associées aux objets de type str sont accessibles par le biais de **l'opérateur** .

```
In [1]: question = "Qu'est-ce qu'un chat qui travaille à la SNCF ?"
```

```
In [2]: question.upper()
```

```
Out[2]: "QU'EST-CE QU'UN CHAT QUI TRAVAILLE À LA SNCF ?"
```

```
In [3]: reponse = "un cheminou"
```

```
In [4]: reponse.capitalize().center(20)
```

```
Out[4]: '    Un cheminou    '
```

```
In [5]: reponse.capitalize().center(20).strip()
```

```
Out[5]: 'Un cheminou'
```

- L'ensemble de ces méthodes sont accessibles *via* l'aide en ligne de ipython i.e. `help(str)` ou en usant de la complétion soit `reponse.<TAB>`

# Les chaînes de caractères

👉 Les méthodes associées

- Les méthodes ou fonctions membres associées aux objets de type str sont accessibles par le biais de **l'opérateur** .

```
In [1]: question = "Qu'est-ce qu'un chat qui travaille à la SNCF ?"
```

```
In [2]: question.upper()
```

```
Out[2]: "QU'EST-CE QU'UN CHAT QUI TRAVAILLE À LA SNCF ?"
```

```
In [3]: reponse = "un cheminou"
```

```
In [4]: reponse.capitalize().center(20)
```

```
Out[4]: '    Un cheminou    '
```

```
In [5]: reponse.capitalize().center(20).strip()
```

```
Out[5]: 'Un cheminou'
```

- L'ensemble de ces méthodes sont accessibles *via* l'aide en ligne de ipython *i.e.* `help(str)` ou en usant de la complétion soit `reponse.<TAB>`

## ► Parcours par indice : `str[indice]`

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."
```

```
In [2]: citation[0]
```

```
Out[2]: 'C'
```

```
In [3]: citation[2]
```

```
Out[3]: 'u'
```

```
In [4]: citation[-1]
```

```
Out[4]: '.'
```

## ► Sélection de sous-chaînes : `str[début:fin:pas]`

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."
```

```
In [2]: citation[0:5]
```

```
Out[2]: 'Chuck'
```

```
In [3]: citation[2:4]
```

```
Out[3]: 'uc'
```

```
In [4]: citation[:5]
```

```
Out[4]: 'Chuck'
```

```
In [5]: citation[5:]
```

```
Out[5]: " Norris a déjà compté jusqu'à l'infini. Deux fois."
```

```
In [6]: citation[::-2]
```

```
Out[6]: "CukNri éàcmt uq' 'nii exfi."
```

```
In [7]: citation[::-1]
```

```
Out[7]: ".siof xueD .inifni'l à'uqsuj étpmoc àjéd a sirroN kcuHC"
```

# Les chaînes de caractères

👉 Parcours & sélection de chaînes

## ► Remplacement de sous-chaînes :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."
```

```
In [2]: citation[6] = "D"
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-53-7080d03086cc> in <module>()  
----> 1 citation[6] = "D"
```

```
TypeError: 'str' object does not support item assignment
```



Une chaîne de caractères est un objet *non mutable* : on ne peut modifier l'objet qu'à la condition de créer une nouvelle référence en mémoire !

# Les chaînes de caractères

👉 Parcours & sélection de chaînes

## ► Remplacement de sous-chaînes :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."
```

```
In [2]: citation[6] = "D"
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-53-7080d03086cc> in <module>()  
----> 1 citation[6] = "D"
```

```
TypeError: 'str' object does not support item assignment
```



Une chaîne de caractères est un **objet non mutable** : on ne peut modifier l'objet qu'à la condition de créer une nouvelle référence en mémoire !

## ► Remplacement de sous-chaînes :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: id(citation)  
Out[2]: 139717789098984  
  
In [3]: citation = citation[0:6] + "D" + citation[7:]  
In [4]: id(citation)  
Out[4]: 139717714061872
```

## ► Pour le remplacement de sous-chaînes, on tirera profit de la méthode `replace` associée aux méthodes `index` et `find`

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: citation = citation.replace("Chuck Norris", "Patrick Puzo")
```

## ► Remplacement de sous-chaînes :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: id(citation)  
Out[2]: 139717789098984  
  
In [3]: citation = citation[0:6] + "D" + citation[7:]  
In [4]: id(citation)  
Out[4]: 139717714061872
```

## ► Pour le remplacement de sous-chaînes, on tirera profit de la méthode `replace` associée aux méthodes `index` et `find`

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: citation = citation.replace("Chuck Norris", "Patrick Puzo")
```



## ► Vérification de présence :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: "Chuck" in citation  
Out[2]: True  
  
In [3]: "Patrick" not in citation  
Out[3]: True
```

## ► Concaténation :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: citation + "Mais seulement " + str(1) + " fois jusqu'à moins l'infini."  
  
In [3]: citation*2  
Out[3]: "Chuck Norris a déjà [...].Chuck Norris a déjà [...]"
```

## ► Vérification de présence :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: "Chuck" in citation  
Out[2]: True  
  
In [3]: "Patrick" not in citation  
Out[3]: True
```

## ► Concaténation :

```
In [1]: citation = "Chuck Norris a déjà compté jusqu'à l'infini. Deux fois."  
In [2]: citation + "Mais seulement " + str(1) + " fois jusqu'à moins l'infini."  
  
In [3]: citation*2  
Out[3]: "Chuck Norris a déjà [...].Chuck Norris a déjà [...]"
```

## ► Utilisation des formats de données du C<sup>†</sup>

```
In [1]: "Un entier: %i; un nombre flottant: %f; une chaîne : %s" % (1, 0.1, "toto")
Out[1]: 'Un entier: 1; un nombre flottant: 0.100000; une chaîne : toto'

In [2]: i = 2
In [3]: filename = "processing_of_dataset_%03d.txt" % i
In [4]: filename
Out[4]: 'processing_of_dataset_002.txt'
```

<sup>†</sup> cf. C-style format [🔗](#)

## ► Utilisation de la méthode `format`<sup>†</sup>

```
In [1]: canevas = "Nom: {}, prénom: {}, date de naissance: {}"
```

```
In [2]: canevas.format("Van Rossum", "Guido", "31/01/1956")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "Nom: {nom}, prénom: {prenom}, date de naissance: {date}"
```

```
In [2]: canevas.format(date="31/01/1956", nom="Van Rossum", prenom="Guido")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "L'année {0:d} s'écrit {0:b} en binaire et {0:x} en hexadécimal"
```

```
In [2]: canevas.format(2017)
```

```
Out[2]: "L'année 2017 s'écrit 11111100001 en binaire et 7e1 en hexadécimal"
```

<sup>†</sup> cf. Python 3 string format [🔗](#)

## ► Utilisation de la méthode format<sup>†</sup>

```
In [1]: canevas = "Nom: {}, prénom: {}, date de naissance: {}"
```

```
In [2]: canevas.format("Van Rossum", "Guido", "31/01/1956")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "Nom: {nom}, prénom: {prenom}, date de naissance: {date}"
```

```
In [2]: canevas.format(date="31/01/1956", nom="Van Rossum", prenom="Guido")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "L'année {0:d} s'écrit {0:b} en binaire et {0:x} en hexadécimal"
```

```
In [2]: canevas.format(2017)
```

```
Out[2]: "L'année 2017 s'écrit 11111100001 en binaire et 7e1 en hexadécimal"
```

<sup>†</sup> cf. Python 3 string format [🔗](#)

## ► Utilisation de la méthode `format`<sup>†</sup>

```
In [1]: canevas = "Nom: {}, prénom: {}, date de naissance: {}"
```

```
In [2]: canevas.format("Van Rossum", "Guido", "31/01/1956")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "Nom: {nom}, prénom: {prenom}, date de naissance: {date}"
```

```
In [2]: canevas.format(date="31/01/1956", nom="Van Rossum", prenom="Guido")
```

```
Out[2]: 'Nom: Van Rossum, prénom: Guido, date de naissance: 31/01/1956'
```

```
In [1]: canevas = "L'année {0:d} s'écrit {0:b} en binaire et {0:x} en hexadécimal"
```

```
In [2]: canevas.format(2017)
```

```
Out[2]: "L'année 2017 s'écrit 11111100001 en binaire et 7e1 en hexadécimal"
```

<sup>†</sup> cf. Python 3 string format [🔗](#)

- ▶ Une liste est un objet qui permet de stocker **une collection d'objets de tous types**
- ▶ Initialisation d'une liste

```
In [1]: l = []
```

```
In [2]: l = ["rouge", "vert", "bleu", "noir"]
```

```
In [3]: l = [1, 2, 3, 4]
```

```
In [4]: l = [1, 2, "bleu", 3, 4]
```

```
Out[4]: [1, 2, 'bleu', 3, 4]
```

```
In [5]: type(l)
```

```
Out[5]: list
```

- ▶ Une liste est un objet qui permet de stocker **une collection d'objets de tous types**
- ▶ Initialisation d'une liste

```
In [1]: l = []
```

```
In [2]: l = ["rouge", "vert", "bleu", "noir"]
```

```
In [3]: l = [1, 2, 3, 4]
```

```
In [4]: l = [1, 2, "bleu", 3, 4]
```

```
Out[4]: [1, 2, 'bleu', 3, 4]
```

```
In [5]: type(l)
```

```
Out[5]: list
```



## ► Conversion en liste

```
In [1]: l = list(range(4))  
In [2]: l  
Out[2]: [0, 1, 2, 3]  
  
In [3]: l = list("abcdef")  
In [4]: l  
Out[4]: ['a', 'b', 'c', 'd', 'e', 'f']
```

## ► Initialisation d'une liste "en compréhension"

```
In [1]: l = [x**2 for x in range(4)]  
In [2]: l  
Out[2]: [0, 1, 4, 9]  
  
In [3]: l = [x**2 for x in range(1,100) if x % 10 == 3]  
In [4]: l  
Out[4]: [9, 169, 529, 1089, 1849, 2809, 3969, 5329, 6889, 8649]
```

## ► Conversion en liste

```
In [1]: l = list(range(4))  
In [2]: l  
Out[2]: [0, 1, 2, 3]  
  
In [3]: l = list("abcdef")  
In [4]: l  
Out[4]: ['a', 'b', 'c', 'd', 'e', 'f']
```

## ► Initialisation d'une liste "en compréhension"

```
In [1]: l = [x**2 for x in range(4)]  
In [2]: l  
Out[2]: [0, 1, 4, 9]  
  
In [3]: l = [x**2 for x in range(1,100) if x % 10 == 3]  
In [4]: l  
Out[4]: [9, 169, 529, 1089, 1849, 2809, 3969, 5329, 6889, 8649]
```

## ► Accès par indice

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: kebab[2]
Out[2]: 'oignons'

In [3]: kebab[-1]
Out[3]: 'sauce blanche'
In [4]: kebab[-2]
Out[4]: 'oignons'

In [5]: kebab[1:3]
Out[5]: ['tomates', 'oignons']

In [6]: kebab[0] = "sans salade"
In [7]: kebab
Out[7]: ['sans salade', 'tomates', 'oignons', 'sauce blanche']
```



À la différence des chaînes de caractères, les listes sont des objets **mutables** !

## ► Accès par indice

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: kebab[2]
Out[2]: 'oignons'

In [3]: kebab[-1]
Out[3]: 'sauce blanche'
In [4]: kebab[-2]
Out[4]: 'oignons'

In [5]: kebab[1:3]
Out[5]: ['tomates', 'oignons']

In [6]: kebab[0] = "sans salade"
In [7]: kebab
Out[7]: ['sans salade', 'tomates', 'oignons', 'sauce blanche']
```



À la différence des chaînes de caractères, les listes sont des objets **mutables** !

## ► Ajout & suppression d'éléments

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: kebab.append("frites")
In [3]: kebab
Out[3]: ['salade', 'tomates', 'oignons', 'sauce blanche', 'frites']

In [4]: kebab.pop()
Out[4]: 'frites'
In [5]: kebab
Out[5]: ['salade', 'tomates', 'oignons', 'sauce blanche']

In [6]: kebab.extend(['frites', 'coca'])
In [7]: kebab
Out[7]: ['salade', 'tomates', 'oignons', 'sauce blanche', 'frites', 'coca']

In [8]: kebab.insert(3, "harissa")
In [9]: kebab
Out[9]: ['tomates', 'sauce blanche', 'salade', 'harissa', 'oignons']
```

## ► Parcourir une liste

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: for item in kebab:
...:     print(item)
salade
tomates
oignons
sauce blanche
```

## ► Parcourir une liste en conservant l'indice

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: for idx in range(0, len(kebab)):
...:     print(idx, kebab[idx])
0 salade
1 tomates
2 oignons
3 sauce blanche

In [3]: for idx, item in enumerate(kebab):
...:     print(idx, kebab[idx])
```

## ► Parcourir une liste

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: for item in kebab:
...:     print(item)
salade
tomates
oignons
sauce blanche
```

## ► Parcourir une liste en conservant l'indice

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: for idx in range(0, len(kebab)):
...:     print(idx, kebab[idx])
0 salade
1 tomates
2 oignons
3 sauce blanche

In [3]: for idx, item in enumerate(kebab):
...:     print(idx, kebab[idx])
```

## ► Tri de listes

```
In [1]: kebab = ["salade", "tomates", "oignons", "sauce blanche"]

In [2]: kebab.sort()
In [3]: kebab
Out[3]: ['oignons', 'salade', 'sauce blanche', 'tomates']

In [4]: kebab.reverse()
In [5]: kebab
Out[5]: ['tomates', 'sauce blanche', 'salade', 'oignons']
```

- Comme pour les chaînes de caractères, l'ensemble des méthodes associées aux objets de type `list` sont accessibles *via* l'aide en ligne de `ipython` *i.e.* `help(list)` ou en utilisant la complétion `kebab.<TAB>`



- Un *tuple* correspond à **une liste *immutable***

```
In [1]: kebab = ("salade", "tomates", "oignons", "sauce blanche")
```

```
In [2]: kebab
```

```
Out[2]: ('salade', 'tomates', 'oignons', 'sauce blanche')
```

```
In [3]: type(kebab)
```

```
Out[3]: tuple
```

```
In[4]: kebab[0] = "saucisson"
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-145-2c877a5b0218> in <module>()  
----> 1 kebab[0] = "saucisson"
```

```
TypeError: 'tuple' object does not support item assignment
```

- ▶ Les dictionnaires sont des structures **mutables**, non ordonnées, formées d'enregistrements du type **clé:valeur**
- ▶ Le seul moyen d'accéder à une valeur particulière est par l'intermédiaire de sa clé

```
In [1]: tel = {"emmanuelle": 5752, "sébastien": 5578}

In [2]: tel["francis"] = 5915
In [3]: tel
Out[3]: {'sébastien': 5578, 'francis': 5915, 'emmanuelle': 5752}
In [4]: tel['sébastien']
Out[4]: 5578

In [5]: tel.keys()
Out[5]: dict_keys(['emmanuelle', 'sébastien', 'francis'])

In [6]: tel.values()
Out[6]: dict_values([5752, 5578, 5915])

In [7]: "francis" in tel
Out[7]: True
```

- Les dictionnaires sont des structures **mutables**, non ordonnées, formées d'enregistrements du type **clé:valeur**
- Le seul moyen d'accéder à une valeur particulière est par l'intermédiaire de sa clé

```
In [1]: tel = {"emmanuelle": 5752, "sébastien": 5578, 'francis': 5915}
```

```
In [2]: for key, value in tel.items():  
...:     print("Clé/Valeur : {}/{}".format(key.capitalize(), value))
```

```
Clé/Valeur : Emmanuelle/5752
```

```
Clé/Valeur : Sébastien/5578
```

```
Clé/Valeur : Francis/5915
```

## INEFFECTIVE SORTS

```

DEFINE HALFHEARTEDMERGESORT(LIST):
  IF LENGTH(LIST) < 2:
    RETURN LIST
  PIVOT = INT(LENGTH(LIST) / 2)
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
  // UMMMMMM
  RETURN [A,B] // HERE. SORRY.

```

```

DEFINE FASTBOGOSORT(LIST):
  // AN OPTIMIZED BOGOSORT
  // RUNS IN O(N LOG N)
  FOR N FROM 1 TO LOG(LENGTH(LIST)):
    SHUFFLE(LIST):
    IF ISSORTED(LIST):
      RETURN LIST
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

```

```

DEFINE JOBININTERVIEWQUICKSORT(LIST):
  OK SO YOU CHOOSE A PIVOT
  THEN DIVIDE THE LIST IN HALF
  FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO WAIT, IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE EQUAL ONES GO INTO, UH
    THE SECOND LIST FROM BEFORE
  HANG ON, LET ME NAME THE LISTS
  THIS IS LIST A
  THE NEW ONE IS LIST B
  PUT THE BIG ONES INTO LIST B
  NOW TAKE THE SECOND LIST
  CALL IT LIST, UH, A2
  WHICH ONE WAS THE PIVOT IN?
  SCRATCH ALL THAT
  IT JUST RECURSIVELY CALLS ITSELF
  UNTIL BOTH LISTS ARE EMPTY
  RIGHT?
  NOT EMPTY, BUT YOU KNOW WHAT I MEAN
  AM I ALLOWED TO USE THE STANDARD LIBRARIES?

```

```

DEFINE PANICSORT(LIST):
  IF ISSORTED(LIST):
    RETURN LIST
  FOR N FROM 1 TO 10000:
    PIVOT = RANDOM(0, LENGTH(LIST))
    LIST = LIST[PIVOT:] + LIST[:PIVOT]
  IF ISSORTED(LIST):
    RETURN LIST
  IF ISSORTED(LIST):
    RETURN LIST
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING
    RETURN LIST
  IF ISSORTED(LIST): // COME ON COME ON
    RETURN LIST
  // OH JEEZ
  // I'M GONNA BE IN SO MUCH TROUBLE
  LIST = [ ]
  SYSTEM("SHUTDOWN -H +5")
  SYSTEM("RM -RF ./")
  SYSTEM("RM -RF ~/")
  SYSTEM("RM -RF /")
  SYSTEM("RD /S /Q C:\*") // PORTABILITY
  RETURN [1, 2, 3, 4, 5]

```

- ▶ Une fonction est **un bloc d'instructions** qui a reçu **un nom**
- ▶ Une fonction peut :
  1. dépendre d'un certain nombre de paramètres → **les arguments**
  2. renvoyer un résultat au moyen de l'instruction **return**
- ▶ Quelques fonctions intégrées au langage Python
  - ▶ `help` : aide sur un nom → `help(dict)`
  - ▶ `input` : entrée au clavier → `n = int(input("N ?"))`
  - ▶ `print` : affiche à l'écran → `print(n)`
  - ▶ `type`, `sum`, `range`, `min/max`, ...

- ▶ Une fonction est **un bloc d'instructions** qui a reçu **un nom**
- ▶ Une fonction peut :
  1. dépendre d'un certain nombre de paramètres → **les arguments**
  2. renvoyer un résultat au moyen de l'instruction **return**
- ▶ Quelques fonctions intégrées au langage Python
  - ▶ `help` : aide sur un nom → `help(dict)`
  - ▶ `input` : entrée au clavier → `n = int(input("N ?"))`
  - ▶ `print` : affiche à l'écran → `print(n)`
  - ▶ `type`, `sum`, `range`, `min/max`, ...

## ► Fonction sans argument et sans valeur de retour

```
In [1]: def dummy():  
...:     print("Fonction 'dummy'")  
...:
```

```
In [2]: dummy()  
Fonction 'dummy'
```



Par défaut, la valeur de retour d'une fonction est None

## ► Fonction sans argument et sans valeur de retour

```
In [1]: def dummy():  
...:     print("Fonction 'dummy'")  
...:
```

```
In [2]: dummy()  
Fonction 'dummy'
```



Par défaut, la valeur de retour d'une fonction est None



## ► Fonction avec argument et valeur de retour

```
In [1]: def aire_disque(rayon):  
...:     return 3.14 * rayon**2  
...:
```

```
In [2]: aire_disque(1.5)  
Out[2]: 7.065
```

```
In [3]: aire_disque()
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-175-daee2592ca2a> in <module>()  
----> 1 aire_disque()
```

```
TypeError: aire_disque() missing 1 required positional argument: 'rayon'
```

## ► Fonction avec argument par défaut et valeur de retour

```
In [1]: def aire_disque(rayon=10.0):  
...:     return 3.14 * rayon**2  
...:
```

```
In [2]: aire_disque(1.5)  
Out[2]: 7.065
```

```
In [3]: aire_disque()  
Out[3]: 314.0
```

```
In [4]: aire_disque(rayon=20)  
Out[4]: 1256.0
```

## ► Fonction **retournant plusieurs valeurs**

```
In [1]: def decomposer(entier, diviseur):  
...:     return entier // diviseur, entier % diviseur  
...:  
  
In [2]: partie_entiere, diviseur = decomposer(20,3)  
In [3]: partie_entiere, diviseur  
Out[3]: (6, 2)
```

- Les fonctions  $\lambda$  sont des fonctions dites **anonymes** *i.e.* sans nom pouvant être appliquée “à la volée” dans une expression

```
In [1]: f = lambda x : x**2
In [2]: f(2)
Out[2]: 4

In [3]: g = lambda x,y,z: 100*x+10*y+z
In [4]: g(1,2,3)
Out[4]: 123
```

- Les fonctions  $\lambda$  sont des fonctions dites **anonymes** *i.e.* sans nom pouvant être appliquée “à la volée” dans une expression

```
In [1]: f = lambda x : x**2
```

```
In [2]: f(2)
```

```
Out[2]: 4
```

```
In [3]: g = lambda x,y,z: 100*x+10*y+z
```

```
In [4]: g(1,2,3)
```

```
Out[4]: 123
```

```
In [1]: def dummy():
...:     """Cette fonction ne sert strictement à rien.
...:
...:     En plus détaillé, cette fonction ne sert toujours
...:     à rien mais la description est plus longue.
...:     """
```

```
In [2]: help(dummy)
1 Help on function dummy in module __main__:
2
3 dummy()
4     Cette fonction ne sert strictement à rien.
5
6     En plus détaillé, cette fonction ne sert toujours
7     à rien mais la description est plus longue.
```



Pour plus de détails sur les us et coutumes en matière de documentation  
*cf. Docstrings conventions* [↗](#)

- Les fonctions sont des objets ce qui implique qu'elles peuvent être :
  1. affectées à une variable
  2. un élément dans une séquence (liste, dictionnaires)
  3. passées comme argument à une autre fonction

```
In [1]: ad = aire_disque
```

```
In [2]: ad(2)
```

```
Out[2]: 12.56
```

```
In [3]: table = {"Calcul de l'aire d'un disque" : ad}
```

```
In [4]: table["Calcul de l'aire d'un disque"]()
```

```
Out[4]: 314.0
```

```
In [5]: decomposer(ad(), 2)
```

```
Out[5]: (157.0, 0.0)
```

- Les fonctions sont des objets ce qui implique qu'elles peuvent être :
  1. affectées à une variable
  2. un élément dans une séquence (liste, dictionnaires)
  3. passées comme argument à une autre fonction

```
In [1]: ad = aire_disque
```

```
In [2]: ad(2)
```

```
Out[2]: 12.56
```

```
In [3]: table = {"Calcul de l'aire d'un disque" : ad}
```

```
In [4]: table["Calcul de l'aire d'un disque"]()
```

```
Out[4]: 314.0
```

```
In [5]: decomposer(ad(), 2)
```

```
Out[5]: (157.0, 0.0)
```