es **listes** (ou **list** / **array**) en **python** sont une variable dans laquelle on peut mettre plusieurs variables.

**Créer une liste en python**

Pour créer une **liste**, rien de plus simple:

>>> liste = []

Vous pouvez voir le contenu de la **liste** en l'appelant comme ceci:

>>> liste

<type 'list'>

**Ajouter une valeur à une liste python**

Vous pouvez ajouter les valeurs que vous voulez lors de la création de la **liste python**:

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

>>> liste

[1, 2, 3]

Ou les ajouter après la création de la liste avec la méthode append (qui signifie "ajouter" en anglais):

>>> liste = []

>>> liste

[]

>>> liste.append(1)

>>> liste

[1]

>>> liste.append("ok")

>>> liste

[1, 'ok']

On voit qu'il est possible de mélanger dans une même liste des variables de type différent. On peut d'ailleurs mettre une liste dans une liste.

**Afficher un item d'une liste**

Pour lire une liste, on peut demander à voir l'index de la valeur qui nous intéresse:

>>> liste = ["a","d","m"]

>>> liste[0]

'a'

>>> liste[2]

'm'

Le premier item commence toujours avec l'index 0. Pour lire la premier item on utilise la valeur 0, le deuxième on utilise la valeur 1, etc.

Il est d'ailleurs possible de modifier une valeur avec son index

>>> liste = ["a","d","m"]

>>> liste[0]

'a'

>>> liste[2]

'm'

>>> liste[2] = "z"

>>> liste

['a', 'd', 'z']

**Supprimer une entrée avec un index**

Il est parfois nécessaire de supprimer une entrée de la liste. Pour cela vous pouvez utiliser la fonction del.

>>> liste = ["a", "b", "c"]

>>> del liste[1]

>>> liste

['a', 'c']

**Supprimer une entrée avec sa valeur**

Il est possible de supprimer une entrée d'une liste avec sa valeur avec la méthode remove.

>>> liste = ["a", "b", "c"]

>>> liste.remove("a")

>>> liste

['b', 'c']

**Inverser les valeurs d'une liste**

Vous pouvez inverser les items d'une liste avec la méthode reverse.

>>> liste = ["a", "b", "c"]

>>> liste.reverse()

>>> liste

['c', 'b', 'a']

**Compter le nombre d'items d'une liste**

Il est possible de compter le nombre d'items d'une liste avec la fonction len.

>>> liste = [1,2,3,5,10]

>>> len(liste)

5

**Compter le nombre d'occurences d'une valeur**

Pour connaitre le nombre d'occurences d'une valeur dans une liste, vous pouvez utiliser la méthode count.

>>> liste = ["a","a","a","b","c","c"]

>>> liste.count("a")

3

>>> liste.count("c")

2

**Trouver l'index d'une valeur**

La méthode index vous permet de connaitre la position de l'item cherché.

>>> liste = ["a","a","a","b","c","c"]

>>> liste.index("b")

3

**Manipuler une liste**

Voici quelques astuces pour manipuler des listes:

>>> liste = [1, 10, 100, 250, 500]

>>> liste[0]

1

>>> liste[-1] # Cherche la dernière occurence

500

>>> liste[-4:] # Affiche les 4 dernières occurrences

[500, 250, 100, 10]

>>> liste[:] # Affiche toutes les occurences

[1, 10, 100, 250, 500]

>>> liste[2:4] = [69, 70]

[1, 10, 69, 70, 500]

>>> liste[:] = [] # vide la liste

[]

**Boucler sur une liste**

Pour afficher les valeurs d'une liste, on peut utiliser une boucle:

>>> liste = ["a","d","m"]

>>> for lettre in liste:

... print lettre

...

a

d

m

Si vous voulez en plus récupérer l'index, vous pouvez utiliser la fonction enumerate.

>>> for lettre in enumerate(liste):

... print lettre

...

(0, 'a')

(1, 'd')

(2, 'm')

Les valeurs retournées par la boucle sont des **tuples.**

**Copier une liste**

Beaucoup de débutants font l'erreur de copier une liste de cette manière

>>> x = [1,2,3]

>>> y = x

Or si vous changez une valeur de la liste **y**, la liste **x** sera elle aussi affectée par cette modification:

>>> x = [1,2,3]

>>> y = x

>>> y[0] = 4

>>> x

[4, 2, 3]

En fait cette syntaxe permet de travailler sur un même élément nommé différemment

Alors comment copier une liste qui sera indépendante?

>>> x = [1,2,3]

>>> y = x[:]

>>> y[0] = 9

>>> x

[1, 2, 3]

>>> y

[9, 2, 3]

Pour des données plus complexes, vous pouvez utiliser la fonction deepcopy du module copy

>>> import copy

>>> x = [[1,2], 2]

>>> y = copy.deepcopy(x)

>>> y[1] = [1,2,3]

>>> x

[[1, 2], 2]

>>> y

[[1, 2], [1, 2, 3]]

**Transformer une string en liste**

Parfois il peut être utile de transformer une chaine de caractère en liste. Cela est possible avec la méthode split.

>>> ma\_chaine = "Olivier:ENGEL:Strasbourg"

>>> ma\_chaine.split(":")

['Olivier', 'ENGEL', 'Strasbourg']

**Transformer une liste en string**

L'inverse est possible avec la méthode "**join**".

>>> liste = ["Olivier","ENGEL","Strasbourg"]

>>> ":".join(liste)

'Olivier:ENGEL:Strasbourg'

**Trouver un item dans une liste**

Pour savoir si un élément est dans une liste, vous pouvez utiliser le mot clé **in** de cette manière:

>>> liste = [1,2,3,5,10]

>>> 3 in liste

True

>>> 11 in liste

False

**La fonction range**

La fonction range génère une liste composée d'une simple suite arithmétique.

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**Agrandir une liste par une liste**

Pour mettre bout à bout deux listes, vous pouvez utiliser la méthode extend

>>> x = [1, 2, 3, 4]

>>> y = [4, 5, 1, 0]

>>> x.extend(y)

>>> print x

[1, 2, 3, 4, 4, 5, 1, 0]

**Astuces**

Afficher les 2 premiers éléments d'une liste

>>> liste = [1,2,3,4,5]

>>> liste[:2]

[1, 2]

Afficher le dernier item d'une liste:

>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-1]

6

Afficher le 3ème élément en partant de la fin:

>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-3]

4

Afficher les 3 derniers éléments d'une liste:

>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-3:]

[4, 5, 6]

Vous pouvez additionner deux listes pour les combiner ensemble en utilisant l'opérateur +:

>>> x = [1, 2, 3]

>>> y = [4, 5, 6]

>>> x + y

[1, 2, 3, 4, 5, 6]

Vous pouvez même mutiplier une liste:

>>> x = [1, 2]

>>> x\*5

[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]

Ce qui peut être utile pour initialiser une liste:

>>> [0] \* 5

[0, 0, 0, 0, 0]

Un **tuple** est une **liste** qui ne peut plus être modifiée.

## Créer un tuple

Pour créer un **tuple**, vous pouvez utiliser la syntaxe suivante:

>>> mon\_tuple = ()

## Ajouter une valeur à un tuple

Pour créer un **tuple** avec des valeurs, vous pouvez le faire de cette façon:

>>> mon\_tuple = (1, "ok", "olivier")

Les parenthèses ne sont pas obligatoires mais facilite la lisibilité du code (rappelons que la force de python est sa simplicité de lecture):

>>> mon\_tuple = 1, 2, 3

>>> type(mon\_tuple)

<type 'tuple'>

Lorsque vous créez un tuple avec une seule valeur, n'oubliez pas d'y ajouter une virgule, sinon ce n'est pas un tuple.

>>> mon\_tuple = ("ok")

>>> type(mon\_tuple)

<type 'str'>

>>> mon\_tuple = ("ok",)

>>> type(mon\_tuple)

<type 'tuple'>

## Afficher une valeur d'un tuple

Le tuple est une sorte de liste, on peut donc utiliser la même syntaxe pour lire les données du tuple.

>>> mon\_tuple[0]

1

Et évidemment si on essaie de changer la valeur d'un index, l'interpreteur nous insulte copieusement:

>>> mon\_tuple[1] = "ok"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

## A quoi sert un tuple alors?

Le tuple permet une affectation multiple:

>>> v1, v2 = 11, 22

>>> v1

11

>>> v2

22

Il permet également de renvoyer plusieurs valeurs lors d'un appel d'une fonction:

>>> def donne\_moi\_ton\_nom():

... return ("olivier", "engel")

...

>>> donne\_moi\_ton\_nom()

('olivier', 'engel')

On utilisera un tuple pour définir des sortes de constantes qui n'ont donc pas vocation à changer.

Un **dictionnaire** en **python** est une sorte de **liste** mais au lieu d'utiliser des **index**, on utilise des **clés**, c'est à dire des valeurs autres que numériques.

## Comment créer un dictionnaire?

Pour initialiser un **dictionnaire**, on utile la syntaxe suivante:

>>> a = {}

## Comment ajouter des valeurs dans un ****dictionnaire****?

Pour ajouter des valeurs à un dictionnaire il faut indiquer une clé ainsi qu'une valeur:

>>> a = {}

>>> a["nom"] = "engel"

>>> a["prenom"] = "olivier"

>>> a

{'nom': 'engel', 'prenom': 'olivier'}

Vous pouvez utiliser des clés numériques comme dans la logique des **listes**.

## Récupérer une valeur dans un dictionnaire

La méthode get vous permet de récupérer une valeur dans un dictionnaire et si la clé est introuvable, vous pouvez donner une valeur à retourner par défaut:

>>> data = {"name": "Olivier", "age": 30}

>>> data.get("name")

'Olivier'

>>> data.get("adresse", "Adresse inconnue")

'Adresse inconnue'

## Vérifier la présence d'une clé dans un dictionnaire

Vous pouvez utiliser la méthode haskey pour vérifier la présence d'une clé que vous cherchez:

>>> a.has\_key("nom")

True

## Supprimer une entrée de dictionnaire

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes:

>>> del a["nom"]

>>> a

{'prenom': 'olivier'}

## Récupérer les clés par une boucle

Pour récupérer les clés on utilise la méthode keys

>>> fiche = {"nom":"engel","prenom":"olivier"}

>>> for cle in fiche.keys():

... print cle

...

nom

prenom

## Récupérer les valeurs par une boucle

Pour cela on utilise la méthode values

>>> fiche = {"nom":"engel","prenom":"olivier"}

>>> for valeur in fiche.values():

... print valeur

...

engel

olivier

## Récupérer les clés et les valeurs par une boucle

Pour récupérer les clés et les valeurs en même temps, on utilise la méthode items qui retourne un tuple.

>>> fiche = {"nom":"engel","prenom":"olivier"}

>>> for cle,valeur in fiche.items():

... print cle, valeur

...

nom engel

prenom olivier

## Utiliser des tuples comme clé

Une des forces de python est la combinaison tuple/dictionnaire qui fait des merveilles dans certains cas comme lors de l'utilisation de coordonnées.

>>> b = {}

>>> b[(3,2)]=12

>>> b[(4,5)]=13

>>> b

{(4, 5): 13, (3, 2): 12}

## Créer une copie indépendante d'un dictionnaire

Comme pour toute variable, vous ne pouvez pas copier un dictionnaire en faisant **dic1 = dic2**:

>>> d = {"k1":"olivier", "k2":"engel"}

>>> e = d

>>> d["k1"] = "XXX"

>>> e

{'k2': 'engel', 'k1': 'XXX'}

Pour créer une copie indépendante vous pouvez utiliser la méthode copy:

>>> d = {"k1":"olivier", "k2":"engel"}

>>> e = d.copy()

>>> d["k1"] = "XXX"

>>> e

{'k2': 'engel', 'k1': 'olivier'}

Une **fonction** (ou *function*) est une suite d'instructions que l'on peut appeler avec un nom.

**Créer ma première fonction**

Créons une **fonction** qui nous retournera un âge:

>>> def indique\_mon\_age():

... return 30;

...

>>> indique\_mon\_age()

30

Vous ne pouvez pas copier coller ce code, vous devez entrer chaque ligne à la main et appuyer sur entrée pour retourner à la ligne. Les 3 chevrons et les 3 points sont affichés par l'interpréteur python.

Tout d'abord pour indiquer à l'interpréteur que vous voulez **créer une fonction**, on utiliser le mot clé def suivi d'un nom puis de parenthèses et ensuite d'un double point.

On remarque également qu'il y a un espace entre les 3 points et le mot clé "return", il s'agit d'une **indentation**, c'est à dire un espace qui améliore non seulement la lecture de la fonction mais qui indique que nous sommes toujours dans la fonction. Lorsque l'action demandée n'est plus dans la fonction, il ne faut plus indenter le texte. Pour indenter du texte, vous devez appuyer sur la **touche TAB** de votre clavier -ou dans d'autres cas créer **4 espaces** manuellement-.

**Les paramètres**

Créons une autre fonction:

>>> def augmente\_moi(a):

... return augmente\_moi + 2

...

>>> augmente\_moi(1)

3

Cette fonction **incrémente de 2** une valeur que l'on passe en paramètre.

Il est d'ailleurs possible d'utiliser plusieurs paramètres:

>>> def augmente\_moi(a, b):

... return 30 + a + b

...

>>> augmente\_moi(1, 2)

33

Si vous avez compris les principes des fonctions, vous avez compris 80% de ce qu'est la programmation.

**Un paramètre est obligatoire**

Lorsque vous indiquez des paramètres à une fonction, ces dernièrs doivent impérativement être renseignés sans quoi une erreur apparaitra.

>>> def augmente\_moi(a, b):

... return 30 + a + b

...

>>> augmente\_moi(1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: augmente\_moi() takes exactly 2 arguments (1 given)

**L'opérateur splat**

L'opérateur splat : \* est très souvent utilisé en python.

def ma\_function(\*var)

def ma\_function(\*\*var)

ma\_function(\*var)

ma\_function(\*\*var)

**Une liste en paramètre**

On peut récupérer les valeurs renseignées via une liste:

>>> def augmente\_moi(\*param):

... return param[0] + param[1] + param[2]

...

>>> augmente\_moi(1, 2, 3)

6

>>> augmente\_moi(10, 20, 30)

60

**Rendre obligatoire uniquement certains paramètres avec une liste**

Si vous désirez rendre obligatoire uniquement certains paramètres, vous pouvez utiliser la syntaxe suivante:

>>> def ma\_fiche(prenom, nom, \*reste):

... return prenom + " " + nom

...

>>> ma\_fiche("olivier","engel")

'olivier engel'

On remarque que le paramètres "**reste**" est précédé d'une étoile \*.

**Utiliser un dictionnaire pour les paramètres**

Vous pouvez utiliser un dictionnaire en paramètres pour cela vous devez ajouter une double étoile: \*\*

>>> def ma\_fiche(\*\*parametres):

... return parametres["prenom"]

...

>>> ma\_fiche(prenom="olivier")

'olivier'

**Utilisation de splat liste au niveau des appels de fonctions**

Reprennons l'exemple de la fonction augmente\_moi:

>>> def augmente\_moi(\*param):

... return param[0] + param[1] + param[2]

...

Nous avons vu qu'il était possible de faire ceci:

>>> augmente\_moi(1, 2, 3)

6

L'utilisation de l'étoile permet de passer par une liste:

>>> data = [1, 2, 3]

>>> augmente\_moi(\*data)

6

**Utilisation de splat dictionnaire au niveau des appels de fonctions**

Prénons l'exemple de cette fonction:

>>> def test(firstname="", lastname=""):

... return "{} {}" .format(firstname, lastname)

Créons notre dictionnaire:

>>> data = {'firstname':'olivier', 'lastname':'engel'}

Et envoyons notre variable avec une étoile \*

>>> test(\*data)

'lastname firstname'

Puis avec deux étoiles \*\*

>>> test(\*\*data)

'olivier engel'

**Portée des variables (variable globale et variable locale)**

Une variable déclarée à la racine d'un module est visible dans tout ce module. On parle alors de **variable globale**.

>>> x = "hello"

>>> def test():

... print x

...

>>> test()

hello

Et une variable déclarée dans une fonction ne sera visible que dans cette fonction. On parle alors de **variable locale**.

>>> x = False

>>> def test():

... x = "hello"

...

>>> test()

>>> x

False

**Procédure et fonctions**

Pour votre culture informatique sachez qu'une fonction n'est pas obligée de renvoyer une valeur, on parlera alors dans ce cas plutôt de **procédure**.

**es fonctions prédéfinies**

Il existe des fonctions internes à python (ou **builtin**).

**abs(x)**

Retourne une valeur absolue

>>> abs(-1)

1

**all(iterable)**

Retourne **True** si tous les éléments d'un élément itérable sont **True**

>>> liste = [True,True,True,1]

>>> all(liste)

True

**any(iterable)**

Retourne **True** si au moins un élément d'un élément itérable est **True**

>>> liste = [True,False, True]

>>> any(liste)

True

**bin(x)**

Convertit un integer en chaine de caractères binaires.

>>> bin(101)

'0b1100101'

**callable(object)**

Determine si un objet est *callable*.

>>> callable("A")

False

>>> callable(int)

True

**str.capitalize()**

La méthode capitalize permet de mettre une chaine de caractères au format Xxxxx

>>> "oLIviER".capitalize()

'Olivier'

**choice([])**

Retourne une valeur d'une liste aléatoirement.

>>> import random

>>> random.choice([1,2,3,4,5])

3

>>> random.choice([1,2,3,4,5])

2

**str.count(string)**

La méthode **count** compte le nombre d'occurences de la recherche demandée.

>>> "olivier".count("i")

2

**dir(object)**

Indique les noms de la structure de l'objet.

>>> dir(int)

['\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_class\_\_', '\_\_cmp\_\_', '\_\_coerce\_\_', '\_\_delattr\_\_', '\_\_div\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_float\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_getnewargs\_\_', '\_\_hash\_\_', '\_\_hex\_\_', '\_\_index\_\_', '\_\_init\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_long\_\_', '\_\_lshift\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_nonzero\_\_', '\_\_oct\_\_', '\_\_or\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdiv\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_', '\_\_xor\_\_', 'bit\_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']

**str.endswith(str)**

La méthode endswith test si une chaine de caractères se termine par la chaine demandée

>>> a = "olivier"

>>> a.endswith("r")

True

>>> a.endswith("er")

True

>>> a.endswith("é")

False

**eval(expression,globals=None,locals=None)**

Execute une chaine de caractères.

>>> v = 101

>>> eval('v+1')

102

**str.find(string)**

La méthode **find** trouve la première occurence de la recherche demandée.

>>> "olivier".find("i")

2

**help(element)**

Cette fonction vous retourne des informations sur l'utilisation de l'élément qui vous intéresse.

>>> help(int)

Help on class int in module \_\_builtin\_\_:

class int(object)

| int(x=0) -> int or long

| int(x, base=10) -> int or long

|

| Convert a number or string to an integer, or return 0 if no arguments

| are given. If x is floating point, the conversion truncates towards zero.

| If x is outside the integer range, the function returns a long instead.

|

| If x is not a number or if base is given, then x must be a string or

| Unicode object representing an integer literal in the given base. The

| literal can be preceded by '+' or '-' and be surrounded by whitespace.

| The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to

| interpret the base from the string as an integer literal.

| >>> int('0b100', base=0)

| 4

**hex**

Convertit un nombre en valeur hexadécimale.

>>> hex(16)

'0x10'

**str.isalnum()**

Retoune True si tous les caractères sont alphanumériques et qu'il y a au moins un caractère. Sinon False.

>>> "25".isalnum()

True

>>> "25b".isalnum()

True

>>> "25bé".isalnum()

True

>>> "25bé@".isalnum()

False

>>> "-".isalnum()

False

>>> "\_".isalnum()

False

>>> "".isalnum()

False

**str.isalpha()**

Retourne True si tous les caractères sont des lettres et qu'il y a au moins un caractère. Sinon False

>>> "x".isalpha()

True

>>> "-".isalpha()

False

>>> "12".isalpha()

False

>>> "jean-claude".isalpha()

False

>>> "jean claude".isalpha()

False

>>> "élise".isalpha()

True

**str.isdigit()**

Retourne True si tous les caractères sont numériques et qu'il y a au moins un caractère. Sinon False.

>>> "1".isdigit()

True

>>> "1.5".isdigit()

False

>>> "1,5".isdigit()

False

>>> "3b".isdigit()

False

>>> " ".isdigit()

False

**str.islower()**

Retoune True si tous les caractères sont en minuscule.

>>> "olivier".islower()

True

>>> "Olivier".islower()

False

**str.isspace()**

Retoune True si il n'y a que des espaces et au moins un caractère.

>>> " ".isspace()

True

>>> "jean louis".isspace()

False

>>> " ".isspace()

True

**str.istitle()**

Retourne True si la chaine a un format titre.

>>> "Titre".istitle()

True

>>> "TitrE".istitle()

False

>>> "Titre de mon site".istitle()

False

>>> "Titre De Mon Site".istitle()

True

**str.isupper()**

Retourne True si tous les caractères sont en majucule et qu'il y a au moins un caractère.

>>> "OLIVIER".isupper()

True

>>> "Olivier".isupper()

False

>>> "OlivieR".isupper()

False

**str.join(liste)**

La méthode **join** transforme une liste en chaine de caractères.

>>> ":".join(["olivier", "engel"])

'olivier:engel'

**len(s)**

Retourne le nombre d'items d'un objet.

>>> len([1,2,3])

3

>>> len("olivier")

7

**locals()**

Retounr un dictionnaire avec les valeurs des variables en cours.

>>> locals()

{'a': 12, '\_\_builtins\_\_': , '\_\_package\_\_': None, 'i': 20, 'v': 101, 'liste': [True, False, True], '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

**str.lower()**

La méthode **lower** permet de mettre en minuscule une chaine de caractères.

>>> "OLIVIER".lower()

'olivier'

**map(function, [])**

Execute une fonction sur chaque item d'un élément itérable.

>>> def add\_one(x):

... return x + 1

...

>>> map(add\_one, [1,2,3])

[2, 3, 4]

**max() / min()**

Retourne la valeur la plus élévée pour max() et la plus basse pour min()

>>> max([1,3,2,6,99,1])

99

>>> max(1,4,6,12,1)

12

**randint()**

Retourne un int aléatoire.

>>> import random

>>> random.randint(1,11)

5

**random()**

Retourne une valeur aléatoire.

>>> import random

>>> random.random()

0.9563522652738929

**str.replace(string, string)**

La méthode **replace** remplace un segment d'une chaine de caractères par une autre:

>>> "olivier".replace("i", "a")

'olavaer'

**reverse()**

La méthode **reverse** inverse l'ordre d'une liste.

>>> x = [1,4,7]

>>> x.reverse()

>>> x

[7, 4, 1]

**reversed([])**

Retourne un itérateur inversé.

>>> list(reversed([1,2,3,4]))

[4, 3, 2, 1]

**round(number)**

Arrondi un nombre.

>>> round(1)

1.0

>>> round(1.2)

1.0

>>> round(1.5)

2.0

>>> round(1.7)

2.0

>>> round(-1.7)

-2.0

>>> round(-1.2)

-1.0

**shuffle([])**

Mélange aléatoirement une liste.

>>> import random

>>> x = [1,2,3,4,5]

>>> random.shuffle(x)

>>> x

[2, 5, 4, 1, 3]

**str.startswith(prefix[, start[, end]])**

Retourne True si la chaine commence par le préfix indiqué. Ce préfix peut être un tuple. Les paramètres start et end (optionnel) test la chaine à la position indiquée. Le test est sensible à la case.

>>> "olivier".startswith("ol")

True

>>> "olivier".startswith(("ol", "eng"))

True

>>> "olivier".startswith(("xxx", "eng"))

False

>>> "olivier".startswith("OL")

False

>>> "olivier".startswith("ol")

True

**list.sort()**

La méthode **sort** permet de trier une liste.

>>> l = [5,1,4,2,10]

>>> l.sort()

>>> l

[1, 2, 4, 5, 10]

**sorted(iterable)**

Tri un élément itérable.

>>> sorted([3,2,12,1])

[1, 2, 3, 12]

**str.split(séparateur)**

La méthode **split** transforme une chaine de caractères en liste.

>>> "olivier:engel".split(":")

['olivier', 'engel']

**str.splitlines([keepends])**

Retourne une liste des lignes de la chaine. Cette méthode utilise le saut à la ligne universel, le retour à la ligne n'est pas inclu, à moins de renseigner le paramètre **keepends** à True.

>>> "olivier\n\n\engel\n\ndéveloppeur".splitlines()

['olivier', '', '\\engel', '', 'développeur']

>>> "olivier\nengel\ndéveloppeur".splitlines()

['olivier', 'engel', 'développeur']

>>> "olivier\n\rengel\n\rdéveloppeur".splitlines()

['olivier', '', 'engel', '', 'développeur']

>>> "olivier\r\nengel\r\ndéveloppeur".splitlines()

['olivier', 'engel', 'développeur']

>>> "olivier\r\nengel\r\n\r\ndéveloppeur".splitlines()

['olivier', 'engel', '', 'développeur']

>>> "olivier\r\nengel\r\n\r\ndéveloppeur".splitlines(True)

['olivier\r\n', 'engel\r\n', '\r\n', 'développeur']

**sum(iterable [,start])**

Additionne les valeurs d'un élément itérable.

>>> sum([1,2,3])

6

**str.title()**

Transforme la chaine dans un format title.

>>> "Ceci est un titre".title()

'Ceci Est Un Titre'

**upper()**

La méthode **upper** permet de mettre en majuscule une chaine de caractères.

>>> "olivier".upper()

'OLIVIER'

**zip(\*iterables)**

Permet de regrouper sous la forme d'un tuple les items de listes.

>>> a = ["olivier", "bruce", "john"]

>>> b = ["engel", "wayne", "Wayne"]

>>> zip(a,b)

[('olivier', 'engel'), ('bruce', 'wayne'), ('john', 'Wayne')]

Cette notion est l'une des plus importante en programmation. L'idée est de dire que **si** telle variable a telle valeur **alors** faire cela **sinon** cela.

Prenon un exemple, on va donner une valeur à une variable et si cette valeur est supérieur à 5, alors on va incrémenter la valeur de 1

>>> a = 10

>>> if a > 5:

... a = a + 1

...

>>> a

11

Que se passe-t-il si la valeur était inférieure à 5?

>>> a = 3

>>> if a > 5:

... a = a + 1

...

>>> a

3

On remarque que si la condition n'est pas remplie, les instructions dans la structure conditionnelle sont ignorées.

**Condition if else**

Il est possible de donner des instructions quelque soit les choix possibles avec le mot clé else.

>>> a = 20

>>> if a > 5:

... a = a + 1

... else:

... a = a - 1

...

>>> a

21

Changeons uniquement la valeur de la variable **a**:

>>> a = 3

>>> if a > 5:

... a = a + 1

... else:

... a = a - 1

...

>>> a

2

**Condition elif**

Il est possible d'ajouter autant de conditions précises que l'on souhaite en ajoutant le mot clé elif, contraction de "else" et "if", qu'on pourrait traduire par "sinon".

>>> a = 5

>>> if a > 5:

... a = a + 1

... elif a == 5:

... a = a + 1000

... else:

... a = a - 1

...

>>> a

1005

Dans cet exemple, on a repris le même que les précédent mais nous avons ajouté la conditions "Si la valeur est égale à 5" que se passe-t-il? Et bien on ajoute 1000.

**Les comparaisons possibles**

Il est possible de comparer des éléments:

== égal à

!= différent de (fonctionne aussi avec )

> strictement supérieur à

>= supérieur ou égal à

< strictement inférieur à

<= inférieur ou égal à

**Comment fonctionne les structures conditionnelles?**

Les mots clé if, elif et else cherchent à savoir si ce qu'on leur soumet est True. En anglais *True* signifique "Vrai". Donc si c'est la valeur est True, les instructions concernant la condition seront exécutée.

Comment savoir si la valeur qu'on soumet à l'interpreteur est True? Il est possible de le voir directement dans l'interpréteur.

Demandons à python si 3 est égal à 4:

>>> 3 == 4

False

Il vous répondra gentiment que c'est False, c'est à dire que c'est faux.

Maintenant on va donner une valeur à une variable est on va lui demander si la valeur correspond bien à ce que l'on attend.

>>> a = 5

>>> a == 5

True

**AND / OR**

Il est possible d'affiner une condition avec les mots clé AND qui signifie "**ET**" et OR qui signifie "**OU**".

On veut par exemple savoir si une valeur est plus grande que 5 mais aussi plus petite que 10:

>>> v = 15

>>> v > 5 and v < 10

False

Essayons avec la valeur **7**:

>>> v = 7

>>> v > 5 and v < 10

True

Pour que le résultat soit TRUE, il faut que **les deux conditions soient remplies**

Testons maintenant la condition OR

>>> v = 11

>>> v > 5 or v > 100

True

Le résultat est TRUE parce qu'**au moins une des deux conditions est respectée**.

>>> v = 1

>>> v > 5 or v > 100

False

Dans ce cas la aucune condition n'est respectée, le résultat est donc FALSE.

**Chainer les comparateurs**

Il est également possible de chainer les comparateurs:

>>> a, b, c = 1, 10, 100

>>> a < b < c

True

>>> a > b < c

False

Une boucle ( ou **loop** ) vous permet de répéter à l'infini des instructions selon vos besoins.

## Le boucle while

En anglais "*while*" signifie "Tant que". Pour créer une **boucle**, il faut donc utiliser ce mot clé suivi d'une indication qui dit quand la boucle s'arrête.

Un exemple sera plus parlant:

On désire écrire 100 fois cette phrase:

"*Je ne dois pas poser une question sans lever la main*"

Ecrire à la main prend beaucoup de temps et beaucoup de temps x 100 s'est vraiment beaucoup de temps, et peu fiable, même pour les chanceux qui connaissent le copier-coller. Et un bon programmeur est toujours un peu *~~fainéant~~* perfectionniste, il cherchera la manière la plus élégante de ne pas répéter du code.

>>> i = 0

>>> while i < 10:

... print("Je ne dois pas poser une question sans lever la main")

... i = i +1

...

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

## La boucle for

La boucle for permet de faire des **itérations** sur un élément, comme une chaine de caractères par exemple ou une [liste](http://apprendre-python.com/page-apprendre-listes-list-tableaux-tableaux-liste-array-python-cours-debutant).

Exemple:

>>> v = "Bonjour toi"

>>> for lettre in v:

... print lettre

...

B

o

n

j

o

u

r

t

o

i

## Range

Il est possible de créer une boucle facilement avec range:

for i in range(0,100):

print i

## Stopper une boucle avec break

Pour stopper immédiatement une boucle on peut utiliser le mot clé break:

>>> liste = [1,5,10,15,20,25]

>>> for i in liste:

... if i > 15:

... print "On stoppe la boucle"

... break

... print i

...

1

5

10

15

On stoppe la boucle

La **programmation orientée objet** (POO) permet de créer des entités (objets) que l'on peut manipuler. La programmation orientée objet impose des structures solides et claires. Les objets peuvent interagir entre eux, cela facilite grandement la compréhension du code et sa maintenance. On oppose souvent la **programmation objet** à la **programmation procédurale**, la première étant plus "professionnelle" que l'autre car plus fiable et plus propre.

## Les classes

Une **classe** regroupe des fonctions et des attributs qui définissent un objet. On appelle par ailleurs les fonctions d'une classe des "**méthodes**".

Créons une classe Voiture:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

class Voiture:

def \_\_init\_\_(self):

self.nom = "Ferrari"

Notre classe Voiture est une sorte d'usine à créer des voitures.

La méthode \_\_init\_\_() est appelée lors de la création d'un objet.

self.nom est une manière de stocker une information dans la classe. On parle d'attribut de classe. Dans notre cas, on stock le nom dans l'attribut nom.

## Les objets

Un objet est une instance d'une classe. On peut créer autant d'objets que l'on désire avec une **classe**.

Créons maintenant notre voiture:

>>> ma\_voiture = Voiture()

## Les attributs de classe

Les attributs de classe permettent de stocker des informations au niveau de la classe. Elle sont similaires aux variables.

Dans notre exemple:

>>> ma\_voiture = Voiture()

>>> ma\_voiture.nom

'Ferrari'

Vous pouvez à tout moment créer un attribut pour votre objet:

>>> ma\_voiture.modele = "250"

Et le lire ainsi:

>>> ma\_voiture.modele

'250'

## Les méthodes

Les méthodes sont des fonctions définies dans une classe.

Créons une nouvelle méthode dans notre classe voiture:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

class Voiture:

def \_\_init\_\_(self):

self.nom = "Ferrari"

def donne\_moi\_le\_modele(self):

return "250"

Utilison cette méthode:

>>> ma\_voiture=Voiture()

>>> ma\_voiture.donne\_moi\_le\_modele()

'250'

## Les propriétés

Quelque soit le langage, pour la **programmation orientée objet** il est de préférable de passer par des propriétés pour changer les valeurs des attributs. Alors bien que cela ne soit pas obligatoire, il existe une convention de passer par des **getter** (ou **accesseur** en francais) et des **setter** (**mutateurs**) pour changer la valeur d'un attribut. Cela permet de garder une cohérence pour le programmeur, si je change un attribut souvent cela peut également impacter d'autres attributs et les mutateurs permettent de faire cette modification une fois pour toute.

Un exemple d'utilisation de propriétés:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

class Voiture(object):

def \_\_init\_\_(self):

self.\_roues=4

def \_get\_roues(self):

print "Récupération du nombre de roues"

return self.\_roues

def \_set\_roues(self, v):

print "Changement du nombre de roues"

self.\_roues = v

roues=property(\_get\_roues, \_set\_roues)

Quand on changera la valeur du nombre de roues, un message apparaîtra. En soi cela n'apporte rien mais au lieu de faire un simple print, vous pouvez par exemple envoyer un mail, etc.

Testons notre classe:

>>> ma\_voiture=Voiture()

>>> ma\_voiture.roues=5

Changement du nombre de roues

>>> ma\_voiture.roues

Récupération du nombre de roues

5

Il existe une autre syntaxe en passant par des décorateurs:

class Voiture(object):

def \_\_init\_\_(self):

self.\_roues=4

@property

def roues(self):

print "Récupération du nombre de roues"

return self.\_roues

@roues.setter

def roues(self, v):

print "Changement du nombre de roues"

self.\_roues = v

Le résultat sera le même, mais la lecture du code se trouve amélioré.

## La fonction dir

Parfois il est intéressant de decortiquer un objet pour résoudre à un bug ou pour comprendre un script.

La fonction dir vous donne un aperçu des méthodes de l'objet:

>>> dir(ma\_voiture)

['\_\_doc\_\_', '\_\_init\_\_', '\_\_module\_\_', 'donne\_moi\_le\_modele', 'nom']

## L'attribut spécial \_\_dict\_\_

Cet attribut spécial vous donne les valeurs des attributs de l'instance:

>>> ma\_voiture.\_\_dict\_\_

{'nom': 'Ferrari'}

## L'héritage de classe

L'héritage est un concept très utile. Cela permet de créer de nouvelles classes mais avec une base existante.

Gardons l'exemple de la voiture et créons une classe VoitureSport:

class Voiture:

roues = 4

moteur = 1

def \_\_init\_\_(self):

self.nom = "A déterminer"

class VoitureSport(Voiture):

def \_\_init\_\_(self):

self.nom = "Ferrari"

On a indiqué que VoitureSport a hérité de classe Voiture, elle recupère donc toutes ses méthodes et ses attributs.

On peut toujours instancier la classe Voiture si on le désire:

>>> ma\_voiture=Voiture()

>>> ma\_voiture.nom

'A déterminer'

>>> ma\_voiture.roues

4

Instancions maintenant la classe VoitureSport:

>>> ma\_voiture\_sport=VoitureSport()

>>> ma\_voiture\_sport.nom

'Ferrari'

>>> ma\_voiture\_sport.roues

4

On remarque tout d'abord que l'attribut roues a bien été hérité. Ensuite on remarque que la méthode \_\_init\_\_ a écrasé la méthode de la classe Voiture. On parle alors de surcharge de méthode.

## Polymorphisme / surcharge de méthode

Comme nous l'avons vu plus haut si une **classe** hérite d'une autre classe, elle hérite les méthodes de son parent.

Exemple:

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

class Voiture:

roues = 4

moteur = 1

def \_\_init\_\_(self):

self.nom = "A déterminer"

def allumer(self):

print "La voiture démarre"

class VoitureSport(Voiture):

def \_\_init\_\_(self):

self.nom = "Ferrari"

ma\_voiture\_sport = VoitureSport()

ma\_voiture\_sport.allumer()

Le résultat:

La voiture démarre

Il est cependant possible d'**écraser** la méthode de la classe parente en la redéfinissant. On parle alors de surcharger une méthode.

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

class Voiture:

roues = 4

moteur = 1

def \_\_init\_\_(self):

self.nom = "A déterminer"

def allumer(self):

print "La voiture démarre"

class VoitureSport(Voiture):

def \_\_init\_\_(self):

self.nom = "Ferrari"

def allumer(self):

print "La voiture de sport démarre"

ma\_voiture\_sport = VoitureSport()

ma\_voiture\_sport.allumer()

Le résultat:

La voiture de sport démarre

Enfin dernier point intéressant: il est possible d'appeler la méthode du parent puis de faire la spécificité de la méthode. On peut d'ailleurs appeler n'importe quelle autre méthode.

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

class Voiture:

roues = 4

moteur = 1

def \_\_init\_\_(self):

self.nom = "A déterminer"

def allumer(self):

print "La voiture démarre"

class VoitureSport(Voiture):

def \_\_init\_\_(self):

self.nom = "Ferrari"

def allumer(self):

Voiture.allumer(self)

print "La voiture de sport démarre"

ma\_voiture\_sport = VoitureSport()

ma\_voiture\_sport.allumer()

Le résultat:

La voiture démarre

La voiture de sport démarre

Les classes Voiture et VoitureSport possédent donc chacune une méthode de même nom mais ces méthodes n'éffectuent pas les mêmes tâches. On parle dans ce cas de **polymorphisme**.

## Conventions

Prenez l'habitude de nommer votre classe uniquement avec des caractères alphanumériques et commençant par une majuscule. Et à l'inverse l'instance peut être nommée sans majuscule.

voiture\_sport = VoitureSport()

Un **décorateur** est une fonction qui modifie le comportement d'autres fonctions.

Les **décorateurs** sont utiles lorsque l'on veut ajouter du même code à plusieurs fonctions existantes.

## Créer un décorateur python

Prenons l'exemple d'une fonction qui n'a le droit d'être executée uniquement si l'utilisateur est "olivier".

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

user = "olivier"

def mon\_decorateur(fonction):

def autre\_fonction():

print "Action refusée"

if user <> "olivier":

return autre\_fonction

return fonction

@mon\_decorateur

def do\_that():

print "Execution des instructions"

>>> do\_that()

Execution des instructions

Changeons la valeur de la variable user.

user = "jean-louis"

Exécutons à nouveau la fonction do\_that.

>>> do\_that()

Action refusée

Voila, nous avons réussi à interdire l'exécution de la fonction do\_that si l'utilisateur n'est pas "olivier".

## Prendre en compte les paramètres

Le décorateur peut prendre en compte les paramètres de notre fonction initiale de cette manière:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

def mon\_decorateur(fonction):

def autre\_fonction(\*param, \*\*param2):

print "Action avant .............. "

fonction(\*param, \*\*param2)

print "Action après .............."

return autre\_fonction

@mon\_decorateur

def do\_that(v):

print "Execution des instructions %s" % v

Exécutons la fonction do\_that.

>>> do\_that("delete")

Action avant ..............

Execution des instructions delete

Action après ..............

## Plusieurs décorateurs

Il est évidemment possible d'attribuer plusieurs décorateurs à une fonction:

@decorateur1

@decorateur2

def ma\_fonction():

## Un itérateur c'est quoi?

Un **itérateur** est une sorte de curseur qui a pour mission de se déplacer dans une séquence d'objets. L'itérateur permet de parcourir chaque object d'une séquence sans se préoccuper de la structure sous-jacente.

## Pourquoi utiliser un itérateur plutôt qu'une liste?

L'itérateur apporte un niveau d'abstraction plus élévé, c'est à dire qu'on ajoute une couche de code en plus pour réaliser une action. Si un jour on devait changer des éléments du code -exemple d'une mise à jour d'une lib ou changement de base de données par exemple- on n'a pas besoin de changer tout notre code puisque cette couche d'abstraction en plus permet de ne changer que ce qui est nécessaire au niveau de la classe et non plus du code. Et cerise sur le macdo, l'itérateur est un objet peu coûteux en utilisation de mémoire.

## Créer son propre itérateur

Une liste est un itérateur:

>>> liste = [1,2,3,4,5,6,7,8,9,10]

>>> for x in liste:

... print x

...

1

2

3

4

5

6

7

8

9

10

Créons notre propre classe iterateur:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

class MonIter():

current=0

def \_\_init\_\_(self,stop):

self.stop=stop

def \_\_iter\_\_(self) :

return self

def next(self) :

self.current+= 1

if self.current>self.stop:

raise StopIteration

if self.current == 5:

print "Quoi déjà 5eme tour?"

return self.current

Nous pouvons maintenant boucler sur cet élément:

>>> for i in MonIter(10):

... print i

...

1

2

3

4

Quoi déjà 5eme tour?

5

6

7

8

9

10

## Les générateurs

Les générateurs permettent de créer plus facilement des itérateurs.

Créons un générateur de base:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

def generateur():

yield "a"

yield "b"

yield "c"

Ensuite créons un itérateur:

>>> i=generateur()

>>> for v in i:

... print v

...

a

b

c

Alors c'est quoi ce bordel on n'y comprend rien. Tout d'abord on remaque qu'il existe un nouveau mot clé, spécialement crée pour les générateurs: yield. Ce mot clé est un peu similaire au return des fonctions sauf qu'il ne signifie pas la fin de l'exécution de la fonction mais une mise en pause et à la prochaine itération la fonction recherchera le prochain yield.

Créons le même itérateur que dans le chapitre précédent:

#!/usr/bin/python2.7

#-\*- coding: utf-8 -\*-

def generateur(n):

for i in range(n):

if i == 5:

print "Quoi déjà 5eme tour?"

yield i+1

Bouclons maintenant sur notre iterable:

>>> i=generateur(10)

>>> for v in i:

... print v

...

1

2

3

4

5

Quoi déjà 5eme tour?

6

7

8

9

10

Voila, on arrive au même résultat mais avec beaucoup moins de lignes de code.