Débuter en python - Partie 1

Présenté par Guillaume Mousnier

C'est qui le mec au tableau ?

Guillaume Mousnier

DataScientist | Développeur (Big Data) chez TimeOne

Mail: mousnier.guillaume@gmail.com

Github: https://github.com/Gmousse

Langages: Python, JavaScript, Scala, R

Prérequis

- Python3

> Installateur dispo sur https://www.python.org/ (windows, mac).
> Aussi dispo sur les gestionnaires de paquets linux.

- Un éditeur de texte de votre choix

> Atom (https://atom.io/ + https://atom.io/packages/language-python)
> VisualStudio (https://code.visualstudio.com/)

- Un terminal

Ressources gratuites

- http://pymbook.readthedocs.io/en/latest/index.html
- http://pythonvisually.com/ebook/index.html
- http://www.practicepython.org/
- http://lepython.com/

Plan

1. Introduction
2. Les variables
3. Les opérateurs
4. Les types primitifs
5. Les conditions
6. Les structures de données
7. Les boucles
8. Les fonctions
9. Les erreurs
10. Les modules et les namespaces

Introduction

Présentation du langage

Publié en 1991 par Guido van Rossum.

Langage de haut niveau interprété et dynamiquement typé.

Extrêmement polyvalent (sciences, web, ...).

Supporte plusieurs paradigmes (procédural, orienté-objet, fonctionnel...).

Dispose de fonctionnalités modernes (imports, asynchrone, parallélisation...) .

Introduction

Philosophie et Syntaxe

Le python suit une philosophie minimaliste (Zen of Python).

Se veut simple à lire et à écrire (et donc à maintenir).

Adopte un style compact, basé sur les indentations.

Favorise le EAFP (*it's easier to ask for forgiveness than permission*) plutôt que le LBYL (*look before you leap*).

Introduction

Lancer python

Lancer le cli (command-line interface) de python:

```
python
```

Lancer un programme avec python:

```
python /mon_chemin/mon_programme.py
```

```
python -m mon_module
```

Introduction

Les commentaires

En python on peut définir un commentaire (texte qui ne sera pas interprété dans votre programme) avec `#` :

```
# Ceci est un commentaire
"Ceci n'est pas un commentaire"

# Une petite opération
1 + 1  # Je crois que ça va faire 2!
```

On les utilisent pour ajouter des indications (doc) dans le code.

Introduction

Afficher un résultat en python

La fonction `print` permet d'afficher un ou plusieurs résultats dans la console.

Ainsi:

```
print("Test")
print("Un et un font ", 1 + 1, "!")
```

Affichera dans la console (ou la sortie standard):

```
Test
Un et un font 2 !
```

Introduction

Documentation

En mode console, il est possible d'obtenir la documentation d'une fonction, classe ou autres via la fonction `help` :

```
help(print)
```

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
    Prints the values to a stream, or to sys.stdout by def
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the cu
    sep:   string inserted between values, default a space
    end:   string appended after the last value, default a
    flush: whether to forcibly flush the stream.
```

Sinon, il y a la doc python: https://docs.python.org/3/

Introduction

Les lignes d'instructions

Normalement on déclare une instruction (statement) par ligne:

```
a + 4
a + 8
```

Il est possible mais non conseillé d'en déclarer plusieurs:

```
a + 4;a + 8
```

Introduction

Les lignes d'instructions (2)

Ecrire une instruction (statement) sur plusieurs lignes est prohibé:

```
a +
4
```

En effet celà provoque une erreur de parsing:

```
SyntaxError: invalid syntax
```

Introduction

Les lignes d'instructions (3)

Pour des besoins de lisibilité on peut expliciter la continuation de la ligne via `\` :

```
1 + 4 + 5 + 7 + 8 + 9 + 10 + 545784 + 6 + 9 +\
4 + 1 + 3\
+ 2
```

Lors de l'appel d'une fonction, on peut également déclarer les paramètres sur plusieurs lignes:

```
print(
    parametre1,
    parametre2, parametre3
)
```

Introduction

Les indentations

Python utilise l'indentation pour séparer les blocs d'instructions d'un programme:

```
a = 1

if a != 1:
   print("WHAAAT")

print(a)
```

Il convient d'être rigoureux:

```
IndentationError: unexpected indent
```

Introduction

Les indentations (2)

La taille de l'indentation importe peu:

```
if a == 1:
           print("OUF")
```

Mais il faut être constant et rigoureux:

```
if a == 1:
    print("OUF")
           print("THAT WAS EXPECTED")
```

```
IndentationError: unexpected indent
```

Exercice 01: Votre première commande en python

Instructions:

- Lancez le cli python.
- Affichez la phrase `Hello world !` en utilisant la fonction `print`.
- Affichez les opérations `2 + 2` et `4 - 1` en utilisant une seule fonction `print`.

Résultats attendus:

- `Hello world !`
- `4 3`

Exercice 02: Votre première programme

Instructions:

- Créez un fichier hello_world.py.
- Dans ce fichier affichez la phrase `Hello world !` en utilisant la fonction `print`.
- Executez le program hello_world.py avec python.

Résultats attendus:

- `Hello world !`

Les variables

Déclarer une variable (1)

En python, la gestion des *objets* en mémoire et leur typage est dynamique.

Chaque valeur que vous allez utilisé se voit attribué un type et un id (localisation dans la mémoire):

```
print(128, id(128), type(128))
```

```
128 139810890631904 <class 'int'>
```

Les variables

Déclarer une variable (2)

On peut affecter une valeur (e.g. 128) à un nom pour le manipuler à long terme:

```
x = 128
print(128, id(128), type(128))
print(x, id(x), type(x))
```

```
128 139810890631904 <class 'int'>
128 139810890631904 <class 'int'>
```

On vient alors de *déclarer la variable* x.

Les variables

Déclarer une variable (3)

Affecter une valeur à une variable consiste à lui donner un alias.
On conserve l'id et le type de la valeur ainsi affectée.

On peut aussi affecter une variable à une autre variable:

```
x = 128
y = x    # équivalent de x = y = 128
print(x, y, id(x), id(y))
```

```
128 128 139810890631904 139810890631904
```

y possèdera alors l'identité de x.

Les variables

Déclarer une variable (4)

On peut aussi déclarer plusieurs variables en même temps:

```
a, b = 35, 24
print(a, b)
```

```
35 24
```

Les variables

Modifier une variable

Une variable peut varier dans le temps.

On peut réattribuer la valeur assignée:

```
x = x + 10 # équivalent de x += 10
print(x, y, id(x), id(y))
```

```
138 128 139810890632224 139810890631904
```

La valeur de x change, son id aussi.

y ne change pas car il ne partage plus le même id avec x.

Les variables

Modifier une variable (2)

Il est souvent conseillé de ne pas écraser une variable mais d'en créer une nouvelle:

```
a = "Valeur"
b = a + "!"
```

Celà permet une lecture (et donc une maintenabilité) facilité du code.

Les variables

Supprimer une variable

Si une variable est inutile, on peut la supprimer (forcant ainsi python à la supprimer de la mémoire):

```
a = "Valeur"
b = a
del a # on n'a plus besoin de a
print(b); print(a)
```

Si on appelle la variable supprimée (on verra pourquoi plus tard):

```
Valeur # b n'est pas supprimé, on supprime juste a
NameError: name 'a' is not defined
```

Il faut néanmoins utiliser la suppression avec prudence!

Les variables

Quid de la constante

La constante est un nom donné à une valeur qui ne doit pas changer au cours du temps.

On ne peut pas déclarer une *vraie* constante en python.

Par convention on crée juste une variable avec le nom en majuscule:

```
MA_SUPER_CONSTANTE = 35 # DO NOT TOUCH OR JUST DIE
```

Et surtout on n'y touche pas!

Les variables

Conventions de nommage

Quand on crée une variable (mais aussi une classe ou une fonction), il convient de lui donner un nom clair.

Il est conseillé d'utiliser 1 convention parmis:

- CamelCase: `maSuperVariable = 3`
- SnakeCase: `ma_super_variable = 3`

Il faut en choisir une et s'y tenir.

Les caractères autorisés pour le nommage sont: `a...z`, `A...z`, `0...9`, `_`.

Les variables

Conventions de nommage (2)

/! Ne pas lui donner le nom d'un built-in ou d'un module:

```
print = "coucou" # on ECRASE print
print(12) # print ne fonctionne plus
```

```
TypeError: 'str' object is not callable
```

Ou le nom d'un mot clef réservé:

```
if = "coucou"
```

```
SyntaxError: invalid syntax
```

Les types primitifs

Liste des types primitifs

Python possède quelques types *primitifs*:

- `bool` : Boolean: 2 valeurs logiques `True` (1) ou `False` (0)
- `int` : Integer: nombre entier (`123`).
- `float` : Float: nombre à virgule (`3.14159`).
- `str` : String: chaîne de charactères, texte (`":D"` ou `':D'`).
- `None` : None ou null: valeur manquante ou nulle.

Détails: https://docs.python.org/3/library/stdtypes.html

Les types primitifs

Vérifier un type

La fonction `type` donne le type d'une valeur:

```
a = "coucou"
print(type(a))
```

```
<class 'str'>
```

La fonction `isinstance` vérifie qu'une variable est une *instance* d'un type (et donc d'une classe):

```
print(isinstance(a, str), isinstance(a, int))
```

```
True False
```

Les types primitifs

Déclarer un type

Python définit les types de manière dynamique.

On peut néanmoins expliciter un type dans le cadre d'un changement de typage:

```
a = "123"
b = int(a)
print(type(a), type(b))
```

```
<class 'str'> <class 'int'>
```

/! Cette conversion n'est pas toujours simple !

Les types primitifs

Le booléen

Le `bool` est un type logique simple qui peut prendre uniquement 2 valeurs: True ou False.

On l'utilise pour statuer que quelquechose est vrai ou faux:

```
print(1 > 2, 3 == 3, True == False, bool(0), bool(1))
```

```
False, True, False, False, True
```

Il est retourné par les comparaisons, et on l'utilise pour valider des conditions (if).

Les types primitifs

Les nombres

On utilise `int` pour les nombres entiers. `float` pour les nombres à virgule:

```
print(128, 3.14)
print(type(128), type(3.14))
print(float(128), int(3.14))
```

```
128 3.14
<class 'int'> <class 'float'>
128.0 3
```

PS: Il existe aussi un type `complex` que nous n'aborderons pas.

PS2: Il existait en python2 un type `long` qui a été unifié avec `int`.

Les types primitifs

Les chaîne de caractères

On utilise `str` pour le texte (ou chaîne de caractères).

On peut déclarer une string via des `"`, `'`, `"""` ou `'''`:

```
print("hello world", 'hello world')
print("""
    hello
    world
""") # Les triple quotes permettent le multiline
```

```
hello world hello world
        hello
        world
```

La classe str a de nombreuses méthodes...

Les types primitifs

Les chaîne de caractères - Méthodes (1)

Concatenez du texte:

- `"un" + " " + "mot"` -> `"un mot"`
- `"{} mot{}".format(1 + 3, "s")` -> `"4 mots"`

Formatez du texte:

- `"un MOT".capitalize()` -> `"Un MOT"`
- `"UN mot".lower()` -> `"un mot"`
- `"un mot".upper()` -> `"UN MOT`
- `"uN MOT".swapcase()` -> `"Un mot"`
- `" Un mot ".strip()` -> `"Un mot"`

Les types primitifs

Les chaîne de caractères - Méthodes (2)

Remplacez dans du texte:

- `"un mot".replace("mot", "chat")` -> `"un chat"`

Cherchez dans du texte:

- `"mot" in "un mot"` -> `True`
- `"un mot".startswith("un")` -> `True`
- `"un mot".endswith("mot")` -> `True`

Séparez du texte en plusieurs textes:

- `"un mot".split(" ")` -> `["un", "mot"]`

Les types primitifs

La valeur nulle ou None (Null)

Lorsque l'on veut signifier qu'une variable n'a pas de valeur on utilise l'objet `None` :

```
a = None # a n'a pas de valeur
print(a, type(a))
print(a is None)
```

```
None <class 'NoneType'>
True
```

Les opérateurs

Les opérateurs mathématiques

Opération	Définition
`a + b`	Somme de a et de b
`a - b`	Différence de a et de b
`a * b`	Multiplication de a par b
`a / b`	Division de a par b
`a // b`	Division, renvoie l'entier
`a % b`	Division, renvoie le reste
`a ** b` ou `pow(a, b)`	a puissance b
`-a`	a multiplié par -1

Les opérateurs

Les opérateurs mathématiques - Notations courtes

Il existe une notation plus courte pour les opérations.
Si on ajoute `=` après l'opérateur on applique l'opération à la variable à gauche de l'opérateur (`a += 2`).

Notation longue:

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

Notation courte:

```
a *= 2 # Multiplie directement a par 2
b += a # Ajoute directement a à b
```

Les opérateurs

Les opérateurs de comparaisons

Opération	Définition
`a < b`	a est inférieur à b
`a <= b`	a est inférieur ou égal à b
`a > b`	a est supérieur à b
`a >= b`	a est supérieur ou égal à b
`a == b`	a est égal à b
`a != b`	a n'est pas égal à b
`a is b`	a est identique à b
`a is not b`	a n'est pas identique à b

Les opérateurs

Les opérateurs logiques

Les booléens possèdent 3 opérateurs logiques permettant de vérifier des conditions complexes:

Opération	Définition
`a or b`	Se vérifie si l'un est vrai
`a and b`	Se vérifie si les deux sont vrais
`not a`	Donne l'inverse de a

On les utilise couramment pour combiner plusieurs comparaisons:

```
if (a < 4) and (a > 0):
    print("a est entre 0 et 4 !")
```

Les opérateurs

Les opérateurs logiques (2)

Voici un résumé des possibilités avec les opérateurs logiques:

Opération	Résultat
`True or True`	`True`
`True or False` ou `False or True`	`True`
`False or False`	`False`
`True and True`	`True`
`True and False` ou `False and True`	`False`
`not True`	`False`
`not False`	`True`

Exercice 03: Variables, types et opérations

A partir de maintenant on fera des exercices types TDD (test driven development).

Instructions:

- Rendez vous sur github: https://git.io/vFj7w
- Récupérez le fichier variables_types_operators_test.py
- Ouvrez le dans un éditeur de texte et lisez les consignes.

Temps estimé: 7 minutes

Les conditions

La structure conditionnelle - if

Parfois dans un programme, il est nécessaire d'éxecutez une partie du code si et seulement si un cas est rencontré.

On a alors besoin des conditions définis par la déclaration `if` (on parle aussi de *if statement*).

Une condition doit être associé à une expression renvoyant un booléen (comparaison, prédicat...).

Les conditions

La structure conditionnelle - if (2)

Si une condition est vrai (True), python éxecute le bloc sous le if, puis revient au reste du programme:

```
a = 4

if (a <= 4) and (a > 0):
    print("La condition est vrai !") # On passe ici
# Saut de ligne obligatoire !!!
print("Les conditions ont été vérifiées !")
```

```
La condition est vrai !
Les condition ont été vérifiées !
```

Les conditions

La structure conditionnelle - if (3)

Si une condition est fausse (False), python ignore le bloc sous le if:

```
a = 6

if (a <= 4) and (a > 0):
    print("La condition est fausse !") # On ne passe pas

print("Les conditions ont été vérifiées !")
```

```
Les condition ont été vérifiées !
```

Les conditions

La structure conditionnelle - else

Si une condition est fausse (False), on peut vouloir éxecuter un bout de code spécifique. Pour se faire on chaîne le `if` avec un `else`.

```
car_color = "yellow"
car_is_yellow = car_color == "yellow"

if car_is_yellow:
    car_price = 8000 # On passe
else:
    car_price = 10000  # On ne passe pas

print("Car price: ", car_price)
```

```
Car price: 8000
```

Les conditions

La structure conditionnelle - else (2)

Parfois, on a tendance à faire des conditions inutiles:

```
car_color = "yellow"

if car_color == "yellow":
    car_is_yellow = True
else:
    car_is_yellow = False
```

Alors que vous pouvez stocker le résultat d'une comparaison:

```
car_is_yellow = car_color == "yellow"
```

C'est idiot mais on le retrouve souvent...

La structure conditionnelle - if imbriqués

Certains cas requierent d'imbriquer les conditions:

```
car_color = "yellow"
car_carburant = "gazoil"

if car_color != "yellow": # On ne passe pas
    if car_carburant == "gazoil":
        car_price = 10000
    else:
        car_price = 9000
else: # On passe
    if car_carburant == "gazoil":
        car_price = 8000  # On passe
    else:
        car_price = 7000 # On ne passe pas
```

```
Car price: 8000
```

Les conditions

La structure conditionnelle - elif

Parfois imbriquer les conditions complique le code. Pour faciliter la lecture on peut enchaîner le `if` avec un `elif` :

```
if not car_is_yellow and car_uses_gazoil:
    car_price = 10000 # On ne passe pas
elif not car_is_yellow and not car_uses_gazoil:
    car_price = 9000  # On ne passe pas
elif car_is_yellow and car_uses_gazoil:
    car_price = 8000 # On passe
else:
    car_price = 7000 # On ne passe pas
```

Celà revient à enchaîner plusieurs `if` avec une écriture plus compact.

Les conditions

Les expressions conditionnelles

Il est aussi possible d'écrire une expression contenant une condition (simple) pour avoir un code compact.

Avec une déclaration (statement):

```
if car_is_yellow: # If statement
    car_price = 8000
else:
    car_price = 10000
```

Avec une expression:

```
# If expression
car_price = 8000 if car_is_yellow else 10000
print(8000 if car_is_yellow else 10000)
```

Exercice 04: Comparaisons et conditions

Instructions:

- Rendez vous sur github: https://git.io/vFj7w
- Récupérez le fichier comparisons_conditions_test.py
- Ouvrez le dans un éditeur de texte et lisez les consignes.

Temps estimé: 10minutes

Les structures de données

La liste

L'objet `list` est une structure permettant de stocker une suite d'éléments de tout types.

Elle est relativement simple à créer:

```
liste_vide = []
liste_vide = list()
liste_nombres = [1, 2, 3, 4, 5, 6, 875484]
liste_noms = ["jean michel", "jean simon", "jean marc",
              "jean charles", "jean jean"]
```

Les structures de données

La liste (2)

C'est une structure simple, qui a un ordre et une longueur.

La fonction `len` retourne le nombre d'élements dans la liste:

```
liste_noms = ["jean michel", "jean simon", "jean marc",
              "jean charles", "jean jean"]
print(liste_noms, type(liste_noms), len(liste_noms))
```

```
["jean michel", "jean simon", "jean marc",
              "jean charles", "jean jean"]
<class 'list'> 5
```

Les structures de données

La liste - Indexation

Chaque élément d'une liste a un index (position dans la liste).

L'index commence à 0.

On peut alors récupérer un élement par son index:

```
print(liste_noms[0]) # Le premier élément
print(liste_noms[3]) # Le quatrième élément
print(liste_noms[-1]) # Le dernier élément,liste_noms[4]
print(liste_noms[-2]) # L'avant dernier élément
```

```
jean michel
jean charles
jean jean
jean charles
```

Les structures de données

La liste - Indexation(2)

On peut aussi modifier un élement par son index:

```
liste_nombres = [1, 2, 3, 4, 5, 6, 875484]
liste_nombres[-1] = 7
print(liste_nombres)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Si on accède à un index absent, python lève une exception:

```
print(liste_nombres[7])
```

```
IndexError: list index out of range
```

La liste - Indexation (3)

On peut également sélectionné une partie de la liste (slice) en précisant un début (inclusif) et une fin (exclusif):

```
liste_noms = ["jean michel", "jean simon", "jean marc",
              "jean charles", "jean jean"]
print(liste_noms[1:4]) # 2ème à l'avant dernier (4ème)
print(liste_noms[:-1]) # tout sauf le dernier
print(liste_noms[1:]) # tout sauf le premier
print(liste_noms[::2]) # tout de 2 en 2
print(liste_noms[1:-1:2]) # 2eme au 4ème de 2 en 2
print(liste_noms[::-1]) # tout en partant de la fin
```

```
["jean simon", "jean marc", "jean charles"]
["jean michel", "jean simon", "jean marc", "jean charles"]
["jean simon", "jean marc", "jean charles", "jean jean"]
["jean michel", "jean marc", "jean jean"]
["jean simon", "jean charles"]
["jean jean", "jean charles", "jean marc", "jean simon", '
```

Les structures de données

La liste - Complétion

On peut ajouter des élements dans une liste après sa création:

```
liste_noms.append("jean jacques") # ajout à la fin
print(liste_noms[-1])
liste_noms.insert(2, "jean robert")
print(liste_noms[:4]) # ajout à la position 2
list_noms.extend(["jean charles", "jean claude"])
# liste_noms += ["jean charles", "jean claude"]
print(liste_noms[5:]) # concaténation
```

```
jean jacques
['jean michel', 'jean simon', 'jean robert', 'jean marc']
['jean jean', 'jean jacques', 'jean charles', 'jean claude
```

Appartée: La mutation

Quand on utilise append, extend, insert, on mute une liste.
L'id de la liste ne change pas, la modification se fait par référence.
C'est pareil pour beaucoup de structures (dict, classes).

Il est parfois préférable de créer une nouvelle liste lors d'une modification:

```
print(id(liste_noms))
liste_noms.extend(["jean gilles", "jean naej"])
print(id(liste_noms))
liste_noms = liste_noms + ["jean charles", "jean claude"]
# ou liste_noms += ["jean charles", "jean claude"]
print(id(liste_noms))
```

```
140342640205960
140342640205960
140342640301240 # Nouvelle liste
```

Appartée: La mutation (2)

Attention! Lorsque l'on travaille sur un objet complexe et que l'on effectue une mutation, on va impacter une variable qui partage cet objet.

```
a = ["valeur"]; b = a
a[0] = "une autre valeur"
a.append("encore une autre")
print(a, b)
print(id(a), id(b))
```

```
['une autre valeur', "encore une autre"]
['une autre valeur', "encore une autre"]
139810876069496 139810876069496
```

Une modification par référence ne change pas l'id d'une valeur !!

Les structures de données

La liste - Suppression

On peut supprimer des éléments dans une liste:

```
print(liste_noms[-1])
del liste_noms[-1] # supprime l'élement dans l'index -1
print(liste_noms[-1])
liste_noms.remove("jean charles")
print(liste_noms[-1])
```

```
jean claudes
jean charles
jean jacques
```

Les structures de données

La liste - Suppression (2)

On peut aussi prélever des éléments dans une liste:

```
nombres = [1, 2, 3, 4, 5, 6, 7]
dernier_nombre = nombres.pop() # on prélève le dernier
print(liste_nombres, dernier_nombre)
deuxieme_nombre = nombres.pop(1) # on prélève le deuxieme
print(liste_nombres, deuxieme_nombre)
```

```
[1, 2, 3, 4, 5, 6]  7
[1, 3, 4, 5, 6]  2
```

Les structures de données

La liste - Modifier l'ordre

On peut aussi modifier l'ordre des élements:

```
nombres = [1, 2, 3, 4, 5, 6, 7]
nombres.reverse() # On inverse la liste par référence
print(nombres)
nombres.sort() # On tri la liste par référence
print(nombres)
nombres = nombres[::-1] # On inverse (pas par référence)
print(nombres)
```

```
[7, 6, 5, 4, 3, 2, 1]
[1, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4, 3, 2, 1]
```

Les structures de données

La liste - Chercher dans une liste

On peut réaliser diverses recherches dans une liste:

```
voitures = ["yaris", "c4", "yaris", "crz"]
print("yaris" in voitures) # yaris est il dans voitures?
print(voitures.index("c4")) # ou est c4 dans voitures ?
print(voitures.count("yaris") # combien de yaris ?
```

```
True
1
2
```

Exercice 05: Les listes

Instructions:

- Rendez vous sur github: https://git.io/vFj7w
- Récupérez le fichier lists_test.py
- Ouvrez le dans un éditeur de texte et lisez les consignes.

Temps estimé: 10 minutes

Les structures de données

Le dictionnaire

Le `dict` est une structure simple, non ordonnée, basé fait de clefs et valeurs.

Chaque clef (`str` ou `int`) est unique et est associé à une valeur:

```
dict_vide = {}; dict_vide = dict()
students_notes = {
    "Simon": 9.68,
    "Robert": 2.00,
    "Martine": 18.4
}
students_notes = dict(Simon=9.68, Robert=2.00,
                      Martine=18.4)
students_notes = dict([
    ["Simon", 9.68], ["Robert", 2.00], ["Martine", 18.4]
])
```

Les structures de données

Le dictionnaire - Accés à la donnée

Dans un dictionnaire, on peut accéder à une donnée via sa clef:

```
print(students_notes["Simon"])
```

```
9.68
```

Si la clef n'existe pas, python lève une erreur:

```
print(students_notes["Charles"])
```

```
KeyError: 'Charles'
```

Le dictionnaire - Accés à la donnée (2)

La méthode `get` permet de ne pas lever d'erreurs si la clef n'existe pas:

```
print(students_notes.get("Simon"))
print(students_notes.get("Charles"))
```

```
9.68
None
```

On peut aussi préciser une valeur par défaut à retourner si la clef n'existe pas:

```
print(students_notes.get("Charles", 0.0))
```

```
0.0 # La clef Charles n'existe pas. Donc 0.0.
```

Le dictionnaire - Modification

On peut modifier le contenu d'un dictionnaire:

```
students_notes["Charles"] = 0.0 # Ajout d'une clef

# Modif d'une clef
students_notes["Simon"] = students_notes["Simon"] + 2.0

# Mise à jour via un autre dict
students_notes.update({"Martine": 17.4, "Gilles": 10.5})
print(students_notes)
```

```
{
    "Simon": 11.68, # Avant 9.68
    "Charles": 0.0,  # Nouveau
    "Robert": 2.00,
    "Martine": 17.4, # Avant: 18.4
    "Gilles": 10.5   # Nouveau
}
```

Les structures de données

Le dictionnaire - Suppression

On peut supprimer supprimer une clef du dictionnaire:

```
car_prices = {
    "c4": 18950,
    "crz": 27800,
    "yaris": 18912
}
del car_prices["c4"]
del car_prices["crz"]
```

```
{"yaris": 18912}
```

Les structures de données

Le dictionnaire - Recherche

On peut chercher si une clef existe:

```
print("c4" in car_prices)
print("yaris" in car_prices)
```

```
False
True
```

Les structures de données

Le dictionnaire - Transformation

On peut récupérer les valeurs ou clefs d'un dictionnaire:

```
car_prices = {
    "c4": 18950,
    "crz": 27800,
    "yaris": 18912
}
print(car_prices.keys()) # Clefs
print(car_prices.values()) # Valeurs
print(car_prices.items()) # Tout
```

```
dict_keys(['c4', 'crz', 'yaris'])
dict_values([18950, 27800, 18912])
dict_items([
  ('c4', 18950), ('crz', 27800), ('yaris', 18912)])
```

Les structures de données

Le dictionnaire - Transformation (2)

On peut tout récupérer et caster (typer) en liste:

```
print(list(car_prices)) # Clefs
print(list(car_prices.keys())) # Clefs
print(list(car_prices.values())) # Valeurs
print(list(car_prices.items())) # Tout
```

```
['c4', 'crz', 'yaris']
['c4', 'crz', 'yaris']
[18950, 27800, 18912]
[('c4', 18950), ('crz', 27800), ('yaris', 18912)]
```

Toutes les méthodes de `list` sont donc utilisables sur un `dict` !!

Les structures de données

Le tuple

Le `tuple` est une liste une `list` immutable.

Il est identique à la `list` à 4 détails prês:

- On ne peut pas y ajouter des élements
- On ne peut pas en supprimer
- On ne peut pas en modifier
- On ne peut pas modifier le tuple lui même (ordre...)

Le `tuple` est à la liste ce que la constante est à la variable.

Les structures de données

Le tuple (2)

Il y a parfois quelques subtilités dans sa création:

```
tuple_vide = (); tuple_vide = tuple()
tuple_un = (4,) # Notez la virgule !!
tuple_un = 4, # Notation non conseillée.
tuple_nombres = (1, 2, 3, 4, 5, 6, 7)
tuples_noms = tuple(liste_noms)
```

Le tuple - Immutabilité

Comme nous l'avons vu, un `tuple` ne peut être modifié:

```
tuple_nombres[-1] = 4
```

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

Ni supprimé:

```
del tuple_nombres[-1]
```

```
TypeError: 'tuple' object doesn't support item deletion
```

Ainsi le tuple ne supporte pas les méthodes mutables de la `list` :

`.extend` , `.append` , `.insert` , `.remove` , `.sort` , `.reverse` , ...

Le tuple - Point commun avec la liste

On peut accéder à la longueur d'un tuple avec `len` :

```
len(tuple_nombres[-1])
```

L'indexation et le slicing est identique à la liste:

```
tuple_nombres[-1]
tuple_nombres[0:3]
```

La concaténation (non mutable) est idenEtique:

```
tuple_nombres += (7, 8)
```

Le tuple supporte aussi les méthodes de recherches: `.count` , `.index` , `.find` , ...

On peut d'ailleurs le typer le liste: `list(tuple_nombres)`

Les structures de données

Le set

Le `set` est une séquence d'élements dédupliqués (sans doublons), sans index et non ordonnée.

Il se crée facilement:

```
set_vide = set()
set_letters = set("abcdabegfa")
print(set_letters)
set_numbers = set([1, 4, 5, 7, 8, 7, 1])
print(set_numbers)
```

```
{'c', 'a', 'e', 'b', 'g', 'f', 'd'} # Pas un dict!
{1, 4, 5, 7, 8} # Pas un dict
```

Le set - Opérations

Le `set` supporte différentes opérations (entre sets) retournant un nouveau set:

Opération	Définition
`a - b`	Difference: Donne élements exclusifs à a (not in b)
`a | b`	Union: concatène a et b
`a & b`	Intersection: élements à la fois dans a et dans b
`a ^ b`	Donne élements de exclusifs à a et exclusifs à b

```
a = set('abracadabra')
b = set('alacazam')
print(a ^ b)
```

```
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Le set - Complétion

On peut ajouter des élements dans un set:

```
a = set('abracadabra')
a.add("YO")
print(a)
b = set('alacazam')
b.update(a)
print(b)
c = b | a # Non mutable !!!!
print(c)
```

```
{'YO', 'c', 'a', 'b', 'r', 'd'}
{'YO', 'z', 'c', 'a', 'b', 'm', 'l', 'r', 'd'}
{'YO', 'z', 'c', 'a', 'b', 'm', 'l', 'r', 'd'}
```

Le set - Suppression

On peut supprimer des élements dans un set:

```
a.remove("YO")
print(a)
removed_element = a.pop()
print(a, removed_element)
b.difference_update(a)
print(b)
b.clear()
print(b)
d = c - a # Non mutable !!!!
print(d)
```

```
{'b', 'd', 'c', 'a', 'r'}
{'d', 'c', 'a', 'r'} 'b'
{'YO', 'b', 'z', 'l', 'm'}
set()
{'YO', 'b', 'z', 'l', 'm'}
```

Le set - Dédupliquer un itérable

On peut donc utiliser le set pour dédupliquer des itérables (`list` , `tuple`):

```
list_to_deduplicate = [5, 2, 4, 5, 2, 4]
deduplicated_list = list(set(list_to_deduplicate))
print(deduplicated_list)
```

```
[2, 4, 5]
```

/! Le set perd l'ordre de l'itérable d'origine !!

Les boucles

Boucles et itérations

Une boucle est un concept qui permet de réexecuter un même code plusieurs fois à la suite.

On appelle chaque passage une *itération*.

Exemple:
On a un panier rempli de **4 fruits**. On veut **nettoyer chaque fruit**.
On réalise un nettoyage, puis un autre, puis un autre et enfin un dernier.
C'est donc une **boucle** de **4 itérations** (1 nettoyage par fruit).

Il existe 2 principales espèces de boucles : le for et le while.

Appartée: Les itérables

Un *itérable* est une donnée que l'on peut parcourir (exemple: le panier à fruits). Un type itérable est juste une *collection* d'éléments (ou de valeurs).

Les principaux types itérables: `str`, `list`, `tuple`, `set`, `dict`

Les strings (`str`) sont des itérables ! Un `str` est une *collection* de lettres.

On peut vérifier si un type est itérable avec la fonction `iter`:

```
print(iter("coucou")) # Est itérable
print(iter(2)) # N'est pas itérable
```

```
<str_iterator object at 0x7f41cae259e8> # iterable
TypeError: 'int' object is not iterable
```

/! `iter` ne sert pas à ça à l'origine !

Les boucles

La boucle for

La boucle `for` parcourt un *itérable* et applique du code:

```
fruits = ["apple", "perry", "apple", "peach"]
cleaned_fruits = []

for fruit in fruits:
    print("I clean my", fruit)
    cleaned_fruits.append(fruit)

print(cleaned_fruits)
```

```
I clean my apple
I clean my perry
I clean my apple
I clean my peach
['apple', 'perry', 'apple', 'peach']
```

Les boucles

La boucle for (2)

On peut s'en servir pour modifier un itérable:

```
car_prices = {
    "c4": 18950, "crz": 27800, "yaris": 18912
}

for car_name in car_prices:
    car_price = car_prices[car_name]
    if car_prices[car_name] > 20000:
        print("BLACK FRIDAY. 5% DISCOUNT on", car_name)
        car_prices[car_name] *= 0.95

print(car_prices)
```

```
BLACK FRIDAY. 5% DISCOUNT on crz
{'c4': 18950, 'crz': 26410.0, 'yaris': 18912}
```

Appartée: enumerate et range

La fonction `enumerate` génère un *itérator* avec les valeurs et **les indexs** d'un itérable:

```
print(list(enumerate(fruits))
```

```
[(0, 'apple'), (1, 'perry'), (2, 'apple'), (3, 'peach')]
```

La fonction `range` génère un *itérator* à partir d'un départ (inclusif), d'une fin (exclusive) et d'une étape:

```
print(
    list(range(0, 5)), # 0 à 4 de 1 en 1
    list(range(0, 10, 2)) # 0 à 9 de 2 en 2
)
```

```
[0, 1, 2, 3, 4] [0, 2, 4, 6, 8]
```

Les boucles

La boucle for (3)

On peut aussi filtrer un itérable:

```
fruits = ["apple", "rotten_apple", "peach", "rotten_apple"

for index, fruit in enumerate(fruits):
    if "rotten" in fruit:
        del fruits[index]

print(fruits)
```

```
['apple', 'peach']
```

Les boucles

La boucle for (4)

Et on peut itéré un nombre de fois donné grâce à `range` :

```
fruits = ["apple", "apple", "peach", "apple"]

for x in range(1, 4, 1):
    print("I eat the fruit number", x, fruits.pop())

print(fruits)
```

```
I eat the fruit number 1 apple
I eat the fruit number 2 peach
I eat the fruit number 3 apple
["apple"]
```

Les boucles

La boucle while

Contrairement au `for` la boucle `while` n'itère pas directement sur un itérable.

Elle itère tant qu'une condition est vrai, ou tant qu'on ne la *casse* pas.

Exemple:
On a un panier rempli de **4 fruits**.
On nettoie un fruit tant qu'il reste des fruits à nettoyer.

Les boucles

La boucle while (2)

Ainsi tant qu'une condition est True on itère:

```
fruits = ["apple", "apple", "peach", "apple"]
count = 0

while len(fruits) >= 0:
    count += 1
    print("I eat the fruit number", count, fruits.pop())

print(fruits)
```

```
I eat the fruit number 1 apple
I eat the fruit number 2 peach
I eat the fruit number 3 apple
I eat the fruit number 4 apple
[]
```

Les boucles

La boucle while - Danger

La boucle while peut amener des erreurs si la condition est mal définie:

```
while len(fruits) >= -1: # Mauvaise condition.
    # La boucle s'arrête trop tard
    count += 1
    print("I eat the fruit number", count, fruits.pop())

print(fruits)
```

```
I eat the fruit number 1 apple
I eat the fruit number 2 peach
I eat the fruit number 3 apple
I eat the fruit number 4 apple
IndexError: pop from empty list
```

Les boucles

La boucle while - Danger (2)

Elle peut aussi être sans fin (boucle infinie) si mal construite:

```
while len(fruits) >= 0: # Toujours True !!
    # La boucle ne s'arrête pas
    # car len(fruits) ne change pas.
    count += 1
    print("I eat the fruit number", count)

print(fruits)
```

```
I eat the fruit number 1
I eat the fruit number 2
....
I eat the fruit number 99999999999999999999999999999
```

Les boucles

La boucle while - Contrôle

La boucle `while` infinie est intéressante si on la maîtrise.

On peut par exemple l'utiliser pour:

- réaliser une tâche régulièrement et indéfiniment
- monitorer une valeur
- réagir à des inputs (actions) de l'utilisateur

On peut la controler avec les mots clefs `continue` et `break`.

On peut également limiter sa vitesse d'itérations.

Appartée: input

La fonction `input` exige une intéraction écrite de la part de l'utilisateur:

```
mot_de_passe = input("Entrez votre mot de passe")
```

La fonction input bloque le programme tant que l'utilisateur n'a pas réaliser l'action demandée.
Une fois que l'utilisateur a terminé, son *input* arrive en `str`.

Appartée: time.sleep

La fonction `sleep` du module `time` (on verra les modules dans le dernier chapitre) permet de mettre en pause un programme pendant quelques secondes:

```
from time import sleep
print("Bonjour...")
sleep(4)
print("Monde!") # 4 secondes plus tard
```

Elle est utilisée pour temporiser un applicatif gourmand, ou attendre un résultat.

Les boucles

La boucle while - Contrôle (2)

Le mot clef `continue` force `while` à passer à l'itération suivante.

Le mot clef `break` casse `while` et met fin aux itérations.

On peut l'utiliser dans le cadre d'intéractions:

```
historique = []
while True: # Boucle infinie
    value = str(input("Entrez une valeur"))
    if value.strip() != "quit()":
        historique.append(value)
        continue
    print("Leaving the program !")
    break
print("Historique:", historique)
```

Les boucles

La boucle while - Contrôle (3)

Parfois on doit temporiser les itérations d'une boucle.

On utilise alors la fonction `sleep` du module `time` :

```
from time import sleep
from datetime import datetime

while True: # Boucle infinie
    print("Current date:", datetime.now())
    sleep(2)
```

```
Current date: 2017-11-29 00:01:16.343487
Current date: 2017-11-29 00:01:18.345577
Current date: 2017-11-29 00:01:20.347678
Current date: 2017-11-29 00:01:22.349768
....
```

Les fonctions

La fonction

La fonction est bout de programme qui ne sera éxecuté que lorsqu'il sera invoqué / appelé (*call*).

Elle peut prendre des paramètres et peut retourner une valeur.

Par exemple, la fonction print affiche une valeur (passée en paramètre) au moment où on l'appelle:

```
print # Ne fait rien car pas appellée
print(12) # Est appellée avec le paramètre 12
```

```
<built-in function print>
12
```

Les fonctions

Appeller une fonction

La fonction est appellée en ouvrant les parenthèses et en lui passant le bon nombre de paramètres (du bon type):

```
# La fonction type prend 1 paramètre,
# et retourne son type
var_type = type([1, 2, 3])
```

Si on appelle une fonction avec trop peu ou trop de paramètres python lève une erreur:

```
sum_of_vars = sum()
```

```
TypeError: sum expected at least 1 arguments, got 0
```

Les fonctions

Définir une fonction

On peut définir ses propres fonctions via le mot clef `def`.
Une fonction a besoin d'un nom, de paramètres (si besoin) et d'un code à éxecuter:

```
def print_hello_world(): # Le nom et aucun paramètres
    print("Hello world!") # Le code

print_hello_world() # Réutilisable à l'infini !!!
print_hello_world()
```

```
Hello world!
Hello world!
```

Les fonctions

Définir une fonction (2)

On peut lui spécifier des paramètres obligatoires (positionnels) qui seront utilisés pour un calcul:

```
def print_sum(x, y): # print_sum prend 2 paramètres
    print(x + y)

print_sum(10, 15) # Réutilisable
print_sum(2, 4)
print_sum(4, 2)
```

```
25
6
6
```

Définir une fonction (3)

On peut aussi lui spécifier des paramètres optionnels (ou nommés):

```
def print_bad_fruits(fruits, bad="rotten"):
# print_bad_fruits a un paramètre obligatoire: fruits
# et un paramètre optionnel: bad, "rotten" par défaut
    for fruit in fruits:
        if bad in fruit:
            print("This fruit is bad:", fruit)

print_sum(["peach", "apple", "rotten_peach"])
print_sum(
    ["peach", "bitter_peach", "bitter_apple",
    "rotten_peach"], bad="bitter"
)
```

```
This fruit is bad rotten_peach
This fruit is bad bitter_peach
This fruit is bad bitter_apple
```

Définir une fonction (4)

Une fonction peut retourner une valeur grâce au mot clef `return` :

```
def wrong_sum(x, y):
    somme = x + y # wrong_sum ne retourne rien

print(wrong_sum(4, 5))
result = wrong_sum(4, 5)
print(result) # result n'a pas valeur!

def good_sum(x, y):
    return x + y

print(good_sum(4, 5))
result2 = good_sum(4, 5)
print(result2) # result a une valeur!
```

```
None
None
9
9
```

Les fonctions

Définir une fonction (5)

`return` fait sortir de la fonction. Tout code sous un return est "mort":

```
def good_sum(x, y):
    result = x + y
    return result
    # Code inatteignable après le return
    print("Sum:", result) # code mort

print(good_sum(4, 5))
```

```
9
```

Définir une fonction (6)

On peut se servir du return pour éviter un `else` inutile:

```
def bad_car_price(car_color):
    if car_color == "yellow":
       return 8000
    else:  # Else inutile
       return 10000

def car_price(car_color):
    if car_color == "yellow":
       return 8000
    return 10000  # C'est plus élégant

print(bad_car_price("yellow"), give_car_price("green"))
print(car_price("yellow"), car_price("green"))
```

```
8000 10000
8000 10000
```

Les fonctions

Refactor grâce aux fonctions

Refactor: Réecrire une partie du programme, pour le rendre plus élégant, lisible et maintenable tout en gardant sa fonctionnalité.

Les fonctions permettent de découper son code.

On peut alors réutiliser certains bouts de code plutot que de les dupliquer.

On peut également découper son programme en plusieurs fichiers.

```
fruits = ["apple", "rotten_perry", "rotten_apple",
          "peach"]
good_fruits = []

for fruit in fruits:
    if "rotten" not in fruit:
        good_fruits.append(fruit)

vegetables = ["rotten_carrot", "tomato", "rotten_carrot",
              "rotten_carrot"]
good_vegetables = []

for vegetable in vegetables:   # Code doublon
    if "rotten" not in vegetable:
        good_vegetables.append(vegetable)

print(good_fruits, good_vegetables)
```

```
['apple', 'peach'] ['tomato']
```

```
def is_rotten(element): # Fonction prédicat
    return "rotten" in element

def list_good_elements(elements):
    good_elements = []
    for element in elements:
        if is_rotten(element):
            good_elements.append(element)
    return good_elements

fruits = ["apple", "rotten_perry", "rotten_apple",
          "peach"]
vegetables = ["rotten_carrot", "tomato", "rotten_carrot",
              "rotten_carrot"]

print(
    list_good_elements(fruits),
    list_good_elements(vegetables)
)
```

```
['apple', 'peach'] ['tomato']
```

Les erreurs

Les types d'erreurs / exceptions

Python est très expressif et n'hésite pas à lever des erreurs:

```
SyntaxError # Erreur de parsing
IndentationError # Mauvaise indentation
NameError # Variable non définie
TypeError # Erreur de typage
IndexError # Index inexistant
KeyError # Clef inexistante .....
```

Python en a beaucoup déjà définies, ce sont les built-in exceptions:

https://docs.python.org/3/library/exceptions.html

Les erreurs

Les types d'erreurs / exceptions (2)

Tout ces types d'erreurs ont un type commun: `Exception`. On dit que ces erreurs héritent de la classe ou du type `Exception`. (cf partie 2).

Il est également possible d'en créer de nouvelles, mais également des dérivés (cf partie 2).

Ainsi les modules et les librairies (modules externes) amènent leurs lots de nouveaux types d'erreurs.

Les erreurs

Gérer les erreurs

Lorsqu'une erreur est levée, elle met fin à l'éxecution du programme. Il faut les gérer pour garantir la continuité de notre programme.

On utilise alors les déclarations `try` et `except` pour les gérer.

Le `try` entoure du code à risque (susceptible de lever une exception).
Si il y a une erreur (attendue), `except` va éxecuter du code qui lui a été fourni. Le programme continue.

Les erreurs

Gérer les erreurs (2)

Programme sans try / except:

```
dict_students = {"John": 18.65, "Jacques": 10.00}
print(dict_students["Martine"])
# KeyError; Le programme s'arrête.
# Le code dessous ne s'exécute pas.
print(dict_students["John"])
```

Programme avec try / except gérant les KeyError:

```
try:
    dict_students = {"John": 18.65, "Jacques": 10.00}
    print(dict_students["Martine"])
except KeyError:
    # Le programme continue.
    print(dict_students["John"])
```

Gérer les erreurs (3)

On peut chaîner les `except` pour gérer différents types d'erreurs:

```
students_name = ["John", "Gilles", "Jacques", "Jean Charle
students_notes = {"John": 18.65, "Jacques": 10.00, "Gilles
for student in students_name:
    try:
        students_notes[student] += 1.00
    except TypeError:
        print(student, "'s score is wrong!")
        del students_notes[student]
    except KeyError:
        print(student, " doesn't have score!")

print(students_notes)
```

```
Gilles 's note is wrong!
Jean Charles  doesn't have score!
{'John': 19.65, 'Jacques': 11.0}
```

Gérer les erreurs (4)

/! Evitez d'utiliser le type Exception:

```
for student in students_name:
    try:
        students_notes[student] += 1.00
    except Exception: # Chaque erreur est une exception
         print("Il y a eu une erreur !!")
    except TypeError: # On ne passe plus ici
        print(student, "'s score is wrong!")
        del students_notes[student]
    except KeyError:  # On ne passe plus ici
        print(student, " doesn't have score!")

print(students_notes)
```

```
Il y a eu une erreur !!
Il y a eu une erreur !!
{'John': 19.65, 'Jacques': 11.0}
```

Les erreurs

Lever une erreur

Lorsque l'on écrit un programme, il peut être intéressant de lever des erreurs pour communiquer avec l'utilisateur (développeur ou autres).

On peut utiliser `raise` pour lever une erreur:

```
if value < 0:
    # on choisit de lever une erreur avec un message perso
    raise ValueError("The value can't be negative!")
```

Les modules et les namespaces

Les fonctions built-in

De nombreuses fonctions ou types sont disponibles par défaut dans le namespace (ou scope) de python, ce sont les **built-in functions**.

Par exemple, la fonction `print` est une fonction built-in.

La liste non exhaustive des built-in functions est disponible ici: https://docs.python.org/3/library/functions.html

Les modules et les namespaces

Les modules

Parfois on veut utiliser du code prêt à l'emploi autre que les built-ins. Ce sont les modules (1 module ~= 1 fichier python).

On peut ainsi importer différents modules dans son programme pour ajouter des fonctionnalités.

Certains modules sont préinstallés avec python: https://docs.python.org/3/py-modindex.html

Les modules et les namespaces

Importer un module

On peut importer un module dans son intégralité:

```
import math
print(math.pi)
```

Ou importer quelques parties du module:

```
from math import pi
print(pi)
```

Ou même tout importer d'un module: **/!**

```
from math import *
print(pi)
```

Les modules et les namespaces

Importer un module (2)

On peut aussi utiliser le système de modules pour découper son programme en plusieurs fichiers:

```
from .my_folder.my_file import my_function
```

On peut alors créer une arborescence de fichier `.py` contenant chacun une petite partie du code, que l'on importera au besoin.

Les modules et les namespaces

Comprendre les namespaces

Quand une fonction, une variable, ou un module est utilisable dans python, on dit qu'elle est présente dans le namespace (ou scope).

Les built-ins par exemple sont présents par défaut.

Pourcomprendre on utilise la fonction `dir` qui liste tout ce qui est dans le namespace:

```
print(dir())
```

```
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
```

Les modules et les namespaces

Comprendre les namespaces

Quand une fonction, une variable, ou un module est utilisable dans python, on dit qu'elle est présente dans le namespace (ou scope).

Les namespaces sont (~) une liste de noms de ce qui est accessible à un endroit donné du programme.

Les built-ins par exemple sont présents par défaut dans le namespace.

Les modules et les namespaces

Comprendre les namespaces (2)

Pour visualiser un namespace, on utilise la fonction `dir` qui liste tout ce qui est dans le namespace local:

```
print(dir())
```

```
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
```

Ainsi on voit que nos built-ins sont en effet présent dans le programme.

On peut même en voir le contenu: `print(dir(__builtins__))`

Les modules et les namespaces

Comprendre les namespaces (3)

Si on essaye d'accéder à une variable qui n'est pas présent dans le namespace local:

```
print(pi)
```

Python nous signal que le nom `pi` n'est pas présent dans le namespace local: `NameError: name 'pi' is not defined`.

Les modules et les namespaces

Comprendre les namespaces (4)

Lorsque l'on importe un module, ou que l'on déclare une variable, python l'ajoute alors au namespace:

```
from math import pi
print(pi)
print(dir())
```

Ce qui rend l'accés à une variable ou à une fonction possible:

```
3.141592653589793
[..., 'pi']
```