

Python - Première partie

Introduction au langage

Joël Cavat

2017

joel.cavat@heig-vd.ch

Vue d'ensemble

Thèmes abordés dans ce chapitre :

- Introduction à Python
- Premier pas à l'aide de la console
- Structures de données
- Scripts
- Interaction h-m en mode console
- Blocs d'instructions
- Sous-programmes
- Structures de contrôles (alternatives et boucles)
- Structures de données avancées
- Modules
- Fichiers
- Exceptions

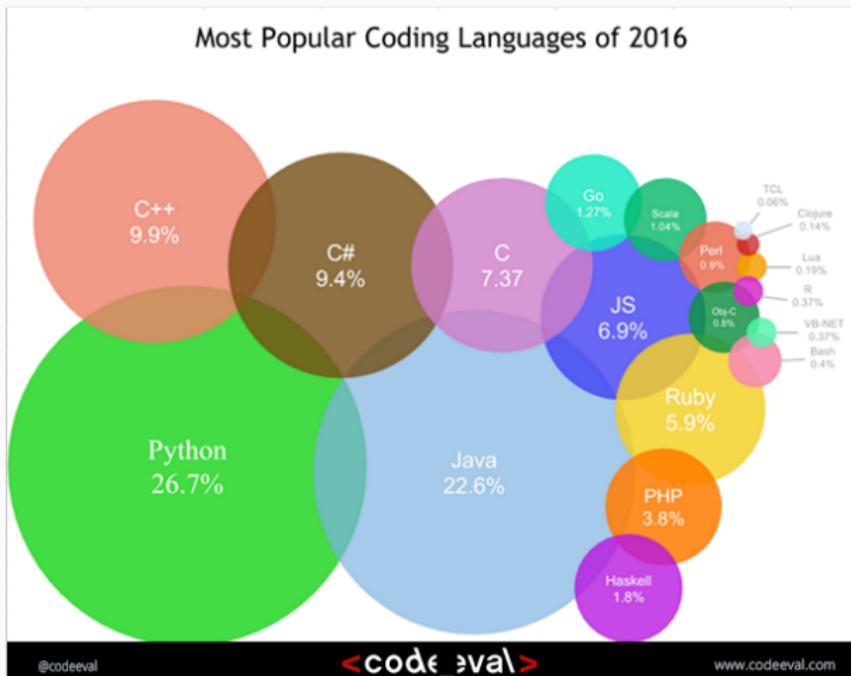
Introduction au langage Python

Pourquoi Python ?

haskell pascal
objective-c visual
coffeescript
basic tcl assembleur
simula smalltalk lua logo
powershell opa cobol bcpl
gallina go ml php nxc erlang pl/i
ocaml rust java lisp ada nqc
python sql apl
forth clips fortran perl opl
applescript vb scheme
natural pearl ruby actionscript scala
groovy eiffel bash
standard modula swift delphi
algol javascript prolog
javac common
wlangage

Pourquoi Python ?

Langages les plus populaires



Pourquoi Python ?

Langages les plus utilisés

- | | |
|-----------------|-----------------|
| 1 JavaScript | 11 Swift |
| 2 Java | 12 Shell |
| 3 Python | 12 Scala |
| 4 PHP | 14 R |
| 5 C# | 15 Go |
| 6 C++ | 15 Perl |
| 7 CSS | 17 TypeScript |
| 8 Ruby | 18 PowerShell |
| 9 C | 19 Haskell |
| 10 Objective-C | 20 CoffeeScript |

Pourquoi Python ?

Mais encore...

- Simple et facile à apprendre
- Langage très expressif
- Portable (MacOS, Linux, Windows, Android, ...)
- Bibliothèques riches
- OpenSource et gratuit
- Académique et industriel

Pourquoi Python ?

Caractéristiques

- Langage orienté objet
- Langage interprété
- Typage dynamique

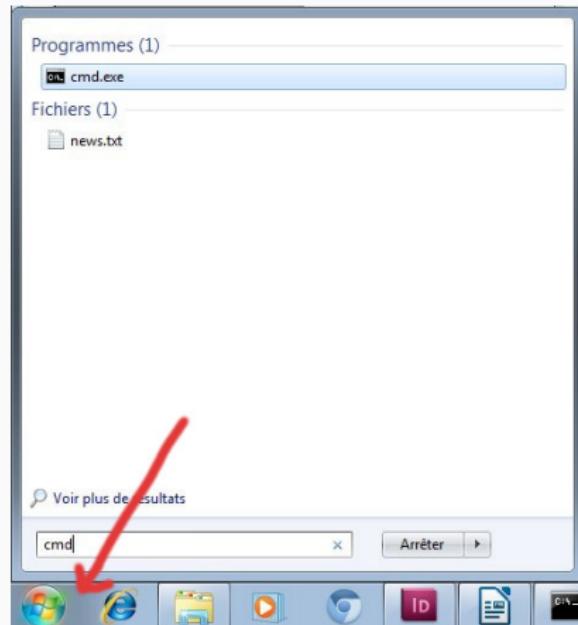
Python en mode console (REPL)

Premier pas à l'aide de la console Python (REPL)

La console Python permet

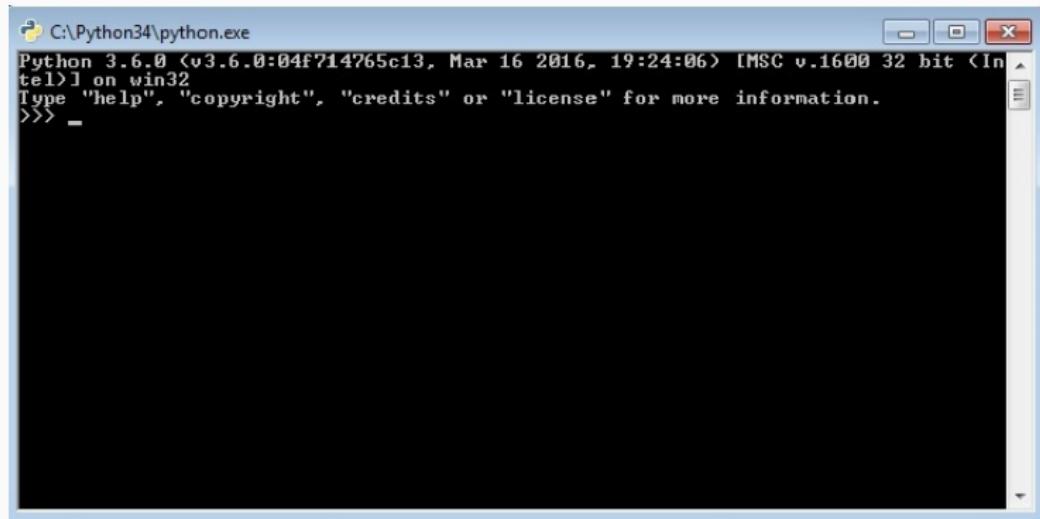
- D'utiliser Python comme calculatrice
- L'expérimentation d'algorithmes simples

Premier pas à l'aide de la console Python



Premier pas à l'aide de la console Python

Lancer la console à l'aide de la commande : py -3



Les types de base : int, float, bool et string

Types numériques

Les entiers et les floats

Un **entier** relatif (\mathbb{Z}) est représenté par le type **int** (integer).

- Exemple : 4, 10, -44

Le nombre **réel** (\mathbb{R}) est représenté par le type **float** (floating point, double précision).

- Exemple : -2.143, 5e3, 334.3094, 4.0

Le type entier et le type réel ne sont pas **équivalents** mais sont **compatibles**.

Types numériques

Préfixe et ‘_’

Il est possible de préfixer les littéraux et d'utiliser le symbole "underscore".

- 10_000_000.0
- 0b100101 (binaire)
- 0xAF102B (hexa)

Types numériques

Les opérations arithmétiques :

Opérations	Signification	Résultat
$x ** y$	exponentiation (x^y)	int / float
$x \% y$	reste de la division (modulo)	int / float
$x//y$	division entière $[x/y]$	int / float
x/y	division réelle	float
$x * y$	multiplication	int / float
$x - y$	soustraction	int / float
$x + y$	addition	int / float

La signature de ces fonctions sont (num, num) -> num

L'ordre des opérations se fait de **gauche vers la droite** sauf pour l'**exponentiation** qui va de droite à gauche

Types booléens

Le type **booléen** est représenté par le type **bool**.

- Exemple : True/False, 1 / 0

Les opérations logiques :

Opérations	Signification	Signature
x or y	ou logique	(bool, bool) -> bool
x and y	et logique	(bool, bool) -> bool
not x	négation logique	(bool) -> bool

Chaîne de caractères (string)

Une **chaîne de caractères** est représentée par le type **str**.

- Exemple : 'coucou', 'hello', "bonjour"

Les opérations courantes :

Opérations	Signification	Signature
$x + y$	concaténation	(str, str) -> str
$x * y$	répétition	(str, int) -> str (int, str) -> str

Opérations de comparaison

Les entiers, les floats, les booléens et les chaînes

Opérations	Signification
$x \text{ is } y$	tests d'identité sur les types
$x \text{ is not } y$	(?, ?) -> bool
$x \leq y$	tests d'inégalité
$x < y$	(cmp, cmp) -> bool
$x > y$	
$x \geq y$	
$x == y$	tests d'égalité et de non égalité
$x != y$	(?, ?) -> bool

Opérations de comparaison

Les entiers, les floats, les booléens et les chaînes

Exemples :

```
>>> 4 < 5  
True  
>>> 4 < 5 and 5 < 6  
True  
>>> 4 < 5 < 6  
True  
>>> 4 < 5 < 2 or 5 > 0  
True
```

```
>>> True == 1  
True  
>>> 0 < True  
True  
>>> 'a' < 'b'  
True  
>>> 'Eve' < 'Albert'  
False
```

Variables et affectations

La **variable**

- Permet de stocker, de lire et de modifier une valeur
- Opérations :
 - L'**initialisation** est la création de la variable
 - L'**affectation** est la modification de sa valeur
 - La **lecture** de la valeur permet de s'y référer.

l'initialisation



Image : <http://www.maths-cours.fr/>

Structure de données : La variable

l'affectation



Image : <http://www.maths-cours.fr/>

La lecture



Image : <http://www.maths-cours.fr/>

Structure de données : La variable

Les variables sont utilisées pour stocker des valeurs.

En Python, la création d'une variable nécessite de lui attribuer une valeur.

Python :

```
x = 5
```

Principaux attributs :

- Le **nom** sert à identifier et à s'y référer
- La **valeur** représente le contenu de la cellule mémoire
- Le **type** détermine l'ensemble des valeurs et des opérations associées
- L'**emplacement** correspond à la localisation en mémoire
- La **portée** est l'ensemble des endroits où la variable est visible
- La **durée** de vie est la portion du code où cette

Structure de données : La variable

L'affectation permet d'attribuer un nom à une valeur.
A ne pas confondre avec l'égalité

Exemples

```
poids = 68.2  
taille = 162  
imc = poids / taille ** 2
```

Structure de données : La variable

Quizz

- 1 hauteur = 10
- 2 largeur = 5
- 3 aire_triangle = hauteur * largeur / 2

Que vaut la valeur *aire_triangle* à la fin de l'algorithme ?

Structure de données : La variable

Quizz

- 1 x = 1
- 2 x = x + 1
- 3 y = 3 * x + 2
- 4 x = y

Que valent les valeurs x et y à la fin de l'algorithme ?

Structure de données : La variable

Il est possible d'effectuer des raccourcis d'écriture

```
x = x + 2  
x += 2
```

Il est également possible de réaliser une affectation multiple :

```
a = b = c = 10  
x, y = 2, 5
```

Variables et affectations

Conventions de nommage

Les variables :

```
snake_case, je_suis_une_variable, hauteur,  
poids_min, poids_de_la_pomme
```

Les constantes (conventions) :

```
JE_SUIS_UNE_CONSTANTE, GRAVITE
```

Premiers scripts

Scripts

monscript.py :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Ceci est un commentaire (non évalué)
print("Salut c't'équipe !")
```

Pour l'exécuter dans un terminal dos :

```
py -3 c :\chemin\vers\monscript.py
```

Interaction homme-machine

La procédure `print` permet d'**afficher** un message dans la console.

```
print("Bonjour")
print("Bonjour " + nom) # variables string uniquement
print("Mon nom est", nom, "et mon prénom est", prenom)
```

Signature de la procédure `print` : `(?*) -> None`

`?*` Signifie 0 ou plusieurs arguments

Scripts

Les versions récentes de Python 3 permettent

- d'utiliser la méthode `format()` pour remplacer les accolades par des valeurs :

```
print("Nom: {} Prénom: {}".format(nom, prenom))
```

- l'interpolation de chaîne de caractères avec les 'f-string' (depuis la version 3.6) :

```
prenom = "Joel"
age = 20
print(f"Mon prénom est {prenom} et j'ai {age} ans.")
```

La **lecture** au clavier se fait à l'aide de la fonction `input`. Celle-ci retourne toujours une chaîne de caractères.

```
nom = input("Entrez votre nom : ")
age = input("Entrez votre age : ")
print("En 2015, vous aurez", 2015 - int(age), "ans")
```

Signatures :

- `input : () -> str`
- `input : (str) -> str`

Bloc d'instructions

Bloc d'instructions

Indentation

L'indentation définit un bloc d'instructions, un périmètre.

```
ligne d'en-tête:  
    instruction1  
    instruction2  
    if condition:  
        instruction3  
        instruction4  
    else:  
        instruction5
```

Sous-programmes

Les sous-programmes

les fonctions et procédures

Les fonctions et procédures permettent de regrouper un ensemble d'instructions pour réaliser une fonctionnalité.

Les arguments sont facultatifs. Les procédures n'ont pas de valeur de retour.

En Python, le type des arguments et de la valeur de retour n'est pas précisé.

Les sous-programmes

les fonctions et procédures

La syntaxe

```
# définition d'une fonction
def nom_de_la_fonction(arg*):
    <instruction1>
    <instruction2>
    return <valeur>
```

```
# définition d'une procédure
def nom_de_la_procedure(arg*):
    <instruction1>
    <instruction2>
```

```
# appels
y = nom_de_la_fonction(...)
nom_de_la_procedure(...)
```

Les sous-programmes

les fonctions et procédures

La fonction mathématique $f(x) = 3 \cdot x + 2$ est représentée ainsi en Python :

```
# f: num -> num
def f(x):
    return 3*x + 2
```

Les sous-programmes

les fonctions et procédures

La fonction mathématique

$moyenne(a, b, c) = \frac{a+b+c}{3}$ est représentée ainsi en Python :

```
# moyenne: (num, num, num) -> float
def moyenne(a, b, c):
    somme = a + b + c
    return somme / 3
```

Les sous-programmes

les fonctions et procédures

Exemple de procédure affichant un message de bienvenue :

```
# bienvenue: ? -> None
def bienvenue(nom):
    print("*****")
    print("Bonjour ", nom)

bienvenue("Joel")
bienvenue("Guillaume")
```

Les deux derniers appels afficheront

```
*****
Bonjour Joel
*****
Bonjour Guillaume
```

Les sous-programmes

les fonctions et procédures

En Python, il est possible retourner plusieurs valeurs :

```
# division_eucl.: (num, num) -> (num, num)
def division_euclidienne(a, b):
    quotient = a // b
    reste = a % b
    return quotient, reste
```

Les sous-programmes

les méthodes

Une **méthode** est une fonction **rattachée** à un type.

On les appelle à l'aide de la notation-point . :

```
<valeur>.une_methode(arg*)
```

```
texte = "coucou"  
  
texte_majuscule = texte.upper()  
# texte_majuscule vaut COUCOU
```

```
texte = texte.replace("c", "ch")  
# texte vaut maintenant "chouchou"
```

Structures de contrôle

Structures de contrôle

Les structures de contrôle permettent de modifier le flux d'exécution d'une application

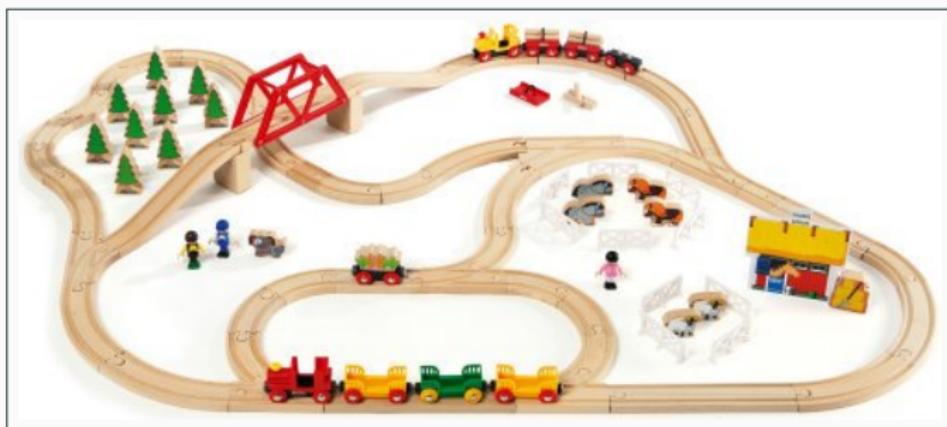


Image : brio.net

Branchement conditionnel

Structures de contrôle : branchement conditionnel

Le branchement conditionnel : l'alternative



Branchement conditionnel

if

Syntaxe générale en Python :

```
if <condition1>:  
    <instruction1>  
    <instruction2>  
elif <condition2>:  
    <instruction3>  
    <instruction4>  
elif <condition3>:  
    <instruction5>  
    <instruction6>  
else:  
    <instruction7>  
    <instruction8>  
  
<instruction9>
```

Branchement conditionnel

if

Exécution conditionnelle d'instructions

```
if <condition>:  
    <instruction1>  
    <instruction2>  
    ...
```

```
note = 5.5  
  
if note >= 4:  
    print("Vous avez réussi !")  
    print("Note obtenue :", note)
```

Branchement conditionnel

if - else

Il est possible d'effectuer des instructions par défaut si la condition n'est pas remplie

```
if <condition>:  
    <instruction1>  
    <instruction2>  
  
else:  
    <instruction3>  
    <instruction4>
```

```
note = 5.5  
if note >= 4:  
    print("Vous avez réussi !")  
    print("Note obtenue : ", note)  
  
else:  
    print("Dommage")
```

Branchement conditionnel

if - else

Il est possible d'imbriquer des *if*

```
note = 5.5

if note >= 4:
    if note >= 5.5:
        print("Félicitaiton !")
        print("Vous avez brillamment réusssi")
    else:
        print("Vous avez passé")
else:
    if note > 3.5:
        print("Dommage !")
        print("Demander une promotion exceptionnelle")
    else:
        print("Echoué")
```

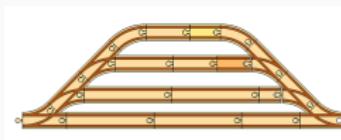
Branchement conditionnel

if - elif - else

La clause *elif* vous permet d'évaluer plusieurs conditions

```
note = 5.5

if note >= 5.5:
    print("Félicitaiton !")
    print("Vous avez brillamment réussi")
elif note >= 4:
    print("Vous avez passé")
elif note > 3.5:
    print("Dommage !")
    print("Demander une promotion exceptionnelle")
else:
    print("Echoué")
```



Opérateur ternaire

Branchement conditionnel

Opérateur ternaire

L'expression ternaire *if/else*

```
if X:  
    a = y  
else:  
    a = z
```

Il existe un raccourci permettant de représenter le code ci-dessus

```
a = <valeur si vrai> if <condition> else <val. si faux>
```

Exemple :

```
x = 3  
valeur = "impair" if x % 2 == 1 else "pair"  
# affichera 'le nombre est impair' :  
print("le nombre est ", valeur)
```

Branchement conditionnel

Opérateur ternaire

```
def max(a, b):  
    if a > b:  
        return a  
    else:  
        return b
```

```
def max(a, b):  
    return a if a > b else b
```

Les boucles

Structure de contrôle : les boucles

Les boucles permettent de la répétition d'une même suite d'opérations



La boucle while

La boucle **while** permet de répéter un ensemble d'opérations tant qu'une condition est vraie. Il s'agit d'une boucle **pré-conditionnelle**.

Syntaxe Python :

```
while <condition>:  
    <instruction1>  
    <instruction2>  
    <instruction3>  
  
<instruction4>
```

Algorithme d'Euclide i

Exemple d'une fonction factorielle :

$$n! = \prod_{1 \leq i \leq n} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot (n - 1) \cdot n$$

Variante 1

```
def factorielle(n):
    i = 1
    res = 1
    while i <= n:
        res = i * res
        i += 1
    return res
```

Variante 2

```
def factorielle(n):
    res = 1

    while n>1:
        res = res * n
        n = n - 1
    return res
```

Algorithme d'Euclide i

Exemple d'une fonction factorielle :

$$n! = \prod_{1 \leq i \leq n} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot (n - 1) \cdot n$$

Quizz Variante 3 : Récursive ?

La boucle for

La boucle **for** permet d'itérer sur une collection ou de connaître à l'avance le nombre de fois que l'on souhaite répéter la boucle.

Syntaxe Python :

```
for <var> in <iterable>:  
    <instructions>
```

Il s'agit d'une **boucle de parcours** (par opposition à une boucle d'itération)

La boucle for

Quelques exemples

```
for i in [1,2,3,4,5]:  
    print(i, end=' ')
```

Affiche : 1 2 3 4 5

```
for mot in ["Bonjour", "tout", "le", "monde", "!"]:  
    print(mot, end=' ')
```

Affiche : Bonjour tout le monde !

La boucle for

Quelques exemples

```
S = "Python"  
for lettre in S:  
    print(lettre, end=' '_')
```

Affiche : P_y_t_h_o_n_

```
for mot in ["Bonjour", "tout", "le", "monde", "!" ]:  
    for lettre in mot:  
        print(lettre, end=' . ')
```

Affiche : B.o.n.j.o.u.r.t.o.u.t.l.e.m.o.n.d.e. !.

La boucle for

Quelques fonctionnalités

La fonction `range` permet de générer des indices.

```
for i in range(10):  
    print(i, end=' ')
```

Affiche : 0 1 2 3 4 5 6 7 8 9

La boucle for

Quelques fonctionnalités

La fonction `range` peut prendre plusieurs paramètres

- 1) `range(max)` : créé les indices de `0` à `max -1`
- 2) `range(min, max)` : créé les indices de `min` à `max -1`
- 3) `range(min, max, pas)` : créé les indices de `min` à `max -1`
par `pas` de `pas`

La boucle for

Quelques fonctionnalités

Exemple :

```
for i in range(-5, 10, 3):
    print(i, end=' ')
```

Affiche : -5 -2 1 4 7

Structures de données

Thèmes abordés dans ce chapitre :

- Chaînes de caractères (Strings)
- Listes
- Les tuples
- Les ensembles (set)
- Les dictionnaires

Séquences ordonnées

Les chaînes de caractères

- Une chaîne de caractères est une **séquence ordonnée immutable** de caractères.
- Elle est délimitée par des apostrophes ' ou des guillemets "

```
>>> chaine_1 = "Bonjour"  
>>> chaine_2 = "vous"  
>>> chaine_1 + chaine_2  
'Bonjourvous'  
>>> chaine_1 + " " + chaine_2  
'Bonjour vous'  
>>> phrase = "C'est beau la vie"
```

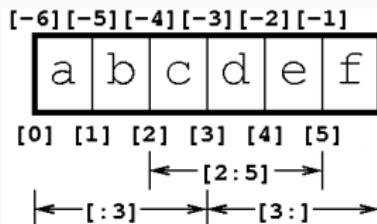
Séquences ordonnées

Les chaînes de caractères

- Il est possible d'accéder aux caractères à l'aide d'indices.
- Le premier caractère se trouve à l'indice 0.
- Exemples : `chaine[indice]`,
`chaine[debut:fin]` ou
`chaine[debut:fin:pas]`

Séquences ordonnées

Les chaînes de caractères



```
>>> ma_chaine = "abcdef"  
>>> ma_chaine[0]  
'a'  
>>> ma_chaine[-1]  
'f'  
>>> ma_chaine[2:5]  
'cde'  
>>> ma_chaine[:3]  
'abc'  
>>> ma_chaine[3:] # idem que ma_chaine[-3:]  
'def'
```

Séquences ordonnées

Les chaînes de caractères

Elle est **immutable**, c'est-à-dire qu'on ne peut pas la modifier.

```
>>> ma_chaine = 'Joel'  
>>> ma_chaine[0] = 'N'  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment
```

Séquences ordonnées

Les chaînes de caractères

Par contre, il est possible d'affecter une nouvelle valeur à une variable

```
>>> ma_chaine = 'Joel'  
>>> ma_chaine = 'N' + ma_chaine[1:]  
>>> ma_chaine  
'Noel'  
>>> ma_chaine [::2]  
'Ne'  
>>> ma_chaine [::-1]  
'leoN'
```

Séquences ordonnées

Les chaînes de caractères

Quelques fonctions et opérateurs intéressants :

- `len(chaine)` : retourne la longueur de la chaîne
- `int(c)` : retourne un entier si la chaîne c contient un entier.
Erreur sinon.
 - `int('35')` retournera 35,
 - `int("Coucou")` lèvera une exception !
- `c1 + c2` : concaténation des chaînes c1 et c2
 - `"Cou" + "cou"` retournera 'Coucou'
- `c in chaine` : vérifie si une sous-chaîne c se trouve dans une chaîne
 - `"ab" in "abcd"` retournera True

Séquences ordonnées

Les chaînes de caractères

Une **méthode** est une fonction rattachée à un type. On les appelle à l'aide de la notation-point .

Quelques méthodes intéressants :

- `c.upper()`
 - `"abcd".upper()` retournera "ABCD"
- `c.capitalize()`
 - `"joel".capitalize()` retournera "Joel"
- `c.title()`
 - `"ceci est un titre".title()` retournera "Ceci Est Un Titre"
- `c.count(sous-chaîne)`
 - `"ababbaaba".count("ab")` retournera 3

De manière générale, `help(str)` vous fourit la liste des méthodes applicables sur une chaîne de caractères

Séquences ordonnées

Les listes

- Une liste est une **séquence ordonnée mutable** d'éléments.
- Les éléments peuvent être de n'importe quels types : des caractères, des chaînes de caractères, des entiers, des listes, ...

Quelques exemples :

- [2, 3, 5, 7, 11, 13]
- []
- [2, True, "Coucou", 3.4, [1,2,3], 1, 2, 3]
- [[1, 0], [0, 1]]

Séquences ordonnées

Les listes

- Les listes sont **mutables**, c'est-à-dire qu'il est possible de les modifier (ajouter ou supprimer des éléments)
- Il est possible d'accéder aux éléments d'une liste à l'aide d'indices de la même manière que les chaînes de caractères

```
>>> l1 = [1, 2, 3, 4]
>>> l1[0] = 'One'
>>> l1
['One', 2, 3, 4]
>>> l1[2:] = [...]
>>> l1
['One', 2, ...]
>>> l1[1:2] = ['Two', 'Three', 'Four']
>>> l1
['One', 'Two', 'Three', 'Four', ...]
```

Séquences ordonnées

Les listes

Quelques fonctions et opérateurs intéressants :

- `len(liste)` retourne le nombre d'élément de la liste
- `[3] * 3 = [3,3,3]` , `["Ho"] * 3 = ["Ho", "Ho", "Ho"]`
- `|l1 += l2` ou `|l1 = l1 + l2` permet de concatener deux listes
- `element in liste` : vérifie si un élément se trouve dans une liste
- `sorted(liste)` : retourne une copie triée de la liste

Séquences ordonnées

Les listes

Quelques méthodes intéressantes :

- `l.append(element)` permet d'ajouter un élément à la liste `l`
- `l.insert(i, element)` permet d'ajouter un élément à la liste `l` à l'indice `i`
- `l.pop()` retire et retourne le dernier élément de la liste
- `l.pop(i)` retire et retourne l'élément à l'indice `i`
- `l.remove(valeur)` supprime la première occurrence de la valeur trouvée
- `l.clear()` supprimer tous les éléments de la liste
- `l.sort()` trie la liste

De manière générale, `help(list)` vous fourit la liste des méthodes applicables sur une chaîne de caractères.

Les tuples

Les **tuples** sont des séquences ordonnées immutables d'éléments. A la différence des listes, il est impossible de modifier leurs valeurs.

```
t = () # tuple vide
t = (1,) # tuple de un élément
t = (1, 2) # couple de deux éléments
t = ("Jean", 25, "Lausanne") # triplet (Nom, age, ville)
```

Les tuples

Exemple de tuples déjà utilisés :

- La valeur de retour multiple

```
def division_euclidienne(a, b):  
    quotient = a // b  
    reste = a % b  
    return quotient, reste
```

- L'affectation multiple

```
a, b, c = 1, 2, 3 # idem que (a, b, c) = (1, 2, 3)
```

Les ensembles (set)

Un **ensemble** est une collection mutable non-ordonnée d'éléments uniques. Elle permet d'étudier la théorie des ensembles en mathématique.

- Les éléments ne peuvent apparaître qu'une fois
- Il est impossible de modifier un élément d'un ensemble

Les ensembles (set)

Quelques exemples :

```
>>> {'a', 'b', 'c'}  
{'a', 'c', 'b'}  
>>> e1 = {1,2,3}  
>>> e2 = {3,4,5}  
>>> e1 & e2 # intersection  
{3}  
>>> e1 | e2 # union  
{1,2,3,4,5}  
>>> e1 - e2 # différence  
{1, 2}  
>>> e1 ^ e2 # différence symétrique  
{1,2,4,5}  
>>> {1,2} < e1 # inclusion d'ensemble  
True
```

Les ensembles (set)

Quelques méthodes intéressantes :

- `s.add(elem)` : ajoute un élément à l'ensemble `s`
- `s.remove(elem)` : supprime un élément à l'ensemble `s`
- `elem in s` : retourne `True` si `elem` se trouve dans l'ensemble `s`
- `set(...)` : transforme le paramètre en ensemble

Un **dictionnaire** est une collection mutable non-ordonnée d'éléments.

- 1) Les valeurs sont indiquées par des clés. (les listes et les tuples sont indiqués par un entier)
- 2) Chaque enregistrement d'un dictionnaire correspond à un couple (clé, valeur)
- 3) Les clés sont uniques (généralement des entiers, caractères ou chaînes de caractères)
- 4) Les valeurs sont de n'importe quel type

Exemple code AITA

Le code AITA est un identifiant à trois lettres permettant de désigner un aéroport.

Code	Aéroport
LHR	Londres Heathrow
GVA	Aéroport international de Genève
ZRH	Aéroport international de Zurich
MAD	Aéroport Adolfo-Suarez de Madrid-Barajas
CDG	Aéroport de Paris-Charles-de-Gaulle
ORY	Aéroport de Paris-Orly

- 1) Les **valeurs** sont indiqués par des **clés** : Clé = Code, Valeur = Aéroport

Les dictionnaires

Exemple code AITA

Code	Aéroport
LHR	Londres Heathrow
GVA	Aéroport international de Genève
ZRH	Aéroport international de Zurich
MAD	Aéroport Adolfo-Suarez de Madrid-Barajas
CDG	Aéroport de Paris-Charles-de-Gaulle
ORY	Aéroport de Paris-Orly

- 2) Chaque enregistrement d'un dictionnaire correspond à un couple (clé, valeur)

```
// ('Clé', 'Valeur')
('LHR', 'Londres Heathrow')
('GVA', 'Aéroport international de Genève' )
('ZRH', 'Aéroport international de Zurich')
...
```

Exemple code AITA

Un dictionnaire se présente sous cette forme :

```
dico = {'clé1': 'valeur1', 'clé2': 'valeur2', 'clé3': 'val3'}
```

```
code = {'LHR': 'Londres Heathrow', \
        'GVA': 'Aéroport international de Genève', \
        'ZRH': 'Aéroport international de Zurich', \
        'MAD': 'Aéroport Adolfo-Suarez de Madrid', \
        'XXX': "Aéroport qui n'existe pas"}
```

```
code['CDG'] = "Aéroport de Paris-Charles-de-Gaulle"
```

```
code.update({'ORY': 'Aéroport de Paris-Orly', \
            'LHR': 'Londres Heathrow'})
```

```
xxx = code.pop('XXX') # supprime et récupère l'élément
```

Exemple code AITA

Attention : si la clé n'existe pas, le programme lèvera une erreur

```
>>> print(code['GWB'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'GWB'

>>> if 'GVA' in code.keys():
    print(code['GVA'])

...
'Aéroport international de Genève'
```

Exemple code AITA

La méthode get accepte une valeur par défaut qui est retournée si la clé n'existe pas :

```
>>> code.get('ZRH', "Aucune valeur")
'Aéroport international de Zurich'
>>> code.get('XYZ', "Aucune valeur")
'Aucune valeur'
```

Exemple code AITA

Pour itérer sur un dictionnaire, la méthode items() permet de retourner un couple (cle,valeur) :

```
for cle, valeur in code.items():
    print("Code {} : {}".format(cle, valeur))
```

A l'affichage :

Code LHR : Londres Heathrow

Code ZRH : Aéroport international de Zurich

Code MAD : Aéroport Adolfo-Suarez de Madrid-Barajas

...

Module principal et autres modules

Modules

Module principal

Chaque module contient un attribut `__name__`.

- si le fichier est exécuté en tant que programme principal,
`__name__` prend la valeur `__main__`
- sinon, il prend le nom du module

Modules

Module principal

Il est dès lors possible de définir le comportement principal d'un module ou d'un programme ainsi :

```
1 # test.py
2
3 def salut():
4     print("salut")
5
6 if __name__ == "__main__":
7     print("coucou")
```

Si nous exécutons `test.py` le résultat sera :

```
"coucou"
```

Modules

Module principal

Si nous importons le module, le bloc d'exécution ne sera pas exécuté.

```
1 >>> from test import salut
2 >>> salut()
3 'salut'
```

Modules

Importation de modules

Clause import :

```
1 >>> import math  
2 >>> math.cos(0.0)  
3 1.0
```

Clause from / import :

```
1 >>> from math import cos  
2 >>> cos(0.0)  
3 1.0
```

Pour importer toutes les fonctionnalités : `from math import *`

Modules

Récupérer les arguments

Clause `import` :

```
1 # arguments.py  
2  
3 import sys  
4 print(sys.argv)
```

L'exécution suivante : `python arguments.py a b joel -c`
affichera :

```
[ 'arguments.py', 'a', 'b', 'joel', '-c' ]
```

Lecture et écriture de fichiers

Les fichiers

Lecture et écriture

Il existe deux types de fichier :

- Les fichiers binaires
- Les fichiers de type **texte**

Plusieurs opérations :

- Lecture
- Ecriture
- Modification
- Ajout

Le **format** des données est une convention utilisée pour représenter les données.

Les fichiers

Lecture et écriture

Le format personnalisé

Lecture de deux matrices carrées

1	3
2	2.2 3.4 3.2
3	1.2 3.2 4.6
4	6.4 2.3 3.5
5	8.9 5.4 3.2
6	3.6 9.3 3.7
7	1.4 5.7 7.5

- La ligne 1 indique la taille d'une matrice carrée (3 lignes sur 3 colonnes)

- Les lignes 2 à 4 correspondent à la première matrice

$$A = \begin{bmatrix} 2.2 & 3.4 & 3.2 \\ 1.2 & 3.2 & 4.6 \\ 6.4 & 2.3 & 3.5 \end{bmatrix}$$

- Les lignes 5 à 7 correspondent à la deuxième

$$B = \begin{bmatrix} 8.9 & 5.4 & 3.2 \\ 3.6 & 9.3 & 3.7 \\ 1.4 & 5.7 & 7.5 \end{bmatrix}$$

Les fichiers

Lecture et écriture

Le format personnalisé

Autre format possible

1	3
2	2.2 3.4 3.2 1.2 3.2 4.6 6.4 2.3 3.5
3	8.9 5.4 3.2 3.6 9.3 3.7 1.4 5.7 7.5

Les fichiers

Lecture et écriture

Lecture et écriture d'un fichier

```
1 with open('nom_fichier.txt', 'r') as f:  
2     instruction1()  
3     instruction2()  
4     instruction3()  
5     ...
```

Le premier argument de `open` est le nom du fichier. Le second argument est le mode d'opération.

- 'r' Lecture (read)
- 'w' Ecriture (write)
- 'r+' lecture et écriture
- 'a' Ecriture à la suite (append)

Les fichiers

Lecture et écriture

Lecture d'un fichier

- La méthode `read()` lit le fichier et retourne **tout son contenu** dans une chaîne de caractères (`str`)
- La méthode `readline()` permet de lire le fichier **ligne par ligne**. A chaque appel, la méthode retourne une chaîne de caractères contenant la ligne.

```
1 f.readline() # retourne '3'  
2 f.readline() # retourne '2.2 3.4 3.2'
```

- La méthode `readlines()` lit le fichier et retourne une liste. Chaque élément de la liste est une ligne sous forme d'une chaîne de caractères.

Les fichiers

Lecture et écriture

Lecture d'un fichier

Il est également possible d'itérer sur un fichier de cette manière :

```
1 for line in f:  
2     print(line)
```

Les fichiers

Lecture et écriture

Ecriture d'un fichier

- La méthode `write` permet d'écrire une chaîne de caractères dans un fichier texte :

```
f.write(str(33.8) + '\n')
```

- La méthode `writelines` permet d'écrire une liste contenant des chaînes de caractères dans un fichier texte :

```
f.writelines(["bonjour", " ", "tout le monde"]))
```

Les fichiers

Lecture et écriture

Quelques méthodes utiles

```
1 >>> s = "hello tout le monde"
2 >>> lst = s.split() # lst = ["hello", "tout", "le", "monde"]
3 >>> ".join(lst)
4 'hello -- tout -- le -- monde'
5 >>> print("\n".join(["hello", "world"]))
6 hello
7 world
8 >>> s.split('t')
9 ['hello ', 'ou', ' le monde']
```

Les fichiers

Lecture et écriture

Lecture d'un fichier

Exemple d'une addition de deux matrices. Ces deux dernières se trouvent dans le fichier matrice.txt :

```
1 with open(path_fichier, 'r') as f:  
2     taille_matrice = int(f.readline())  
3     matrice_1 = []  
4     matrice_2 = []  
5  
6     ...  
7     # Suite dans le fichier addition_matrices.py  
8     ...
```

Les fichiers

Lecture et écriture

Le format JSON

JSON est un format léger d'échange de données.

```
1  {
2      "id": 1023,
3      "name": {
4          "first": "Olivia",
5          "last": "Michel",
6          "title": "Madame"
7      },
8      "email": "olivia.michel@example.com",
9      "gender": "femme",
10     "localite": {
11         "ville": "Yverdon",
12         "canton": "Vaud",
13         "adresse": "Rue de la Mercerie 9",
14         "zip": 1400
15     },
16     "list_abonnement": ["Le temps", "L'Hebdo"]
17 }
```

Les fichiers

Lecture et écriture

Le format JSON

JSON se base sur deux structures :

- Une collection de couples nom/valeur. (Objet = { ... })
- Une liste de valeurs ordonnées.

Les fichiers

Lecture et écriture

On peut très facilement convertir un dictionnaire python en JSON avec la méthode `json.dump()` du module json.

```
1 import json
2 mon_dict = {'a': 1,
3             'b': [2.1, 3, True],
4             'c': {'inner_a': ['hello', 'json', '!']}}
5
6 with open('mon_fichier.json', 'w') as fp:
7     json.dump(mon_dict, fp)
```

Tout le contenu du fichier généré sera écrit sur une seule ligne. Pour avoir un formatage plus lisible, il faut définir le paramètre `indent`

```
json.dump(mon_dict, fp, indent=4)
```

Les fichiers

Lecture et écriture

La fonction `json.load()` permet de charger le contenu d'un fichier JSON dans un dictionnaire.

```
with open('mon_fichier.json', 'r') as fp:  
    mon_dict_json = json.load(fp)
```

On peut alors directement accéder aux différents éléments :

```
>>> print(mon_dict_json['b'])  
[2.1, 3, True]
```

Les fichiers

Lecture et écriture

Le format CSV

CSV est un format texte représentant des données **tabulaires** sous forme de valeurs séparées par un **caractère de séparation**.

- | | |
|---|------------------------------------|
| 1 | Sexe ; Prénom ; Année de naissance |
| 2 | M ; Alphonse ; 1932 |
| 3 | F ; Béatrice ; 1964 |
| 4 | F ; Charlotte ; 1988 |

Les fichiers

Lecture et écriture

Le format PGM

Image en niveau de gris codé à l'aide de caractères ASCII

```
1 P2
2 24 7
3 15
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 15 0
6 0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0
7 0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 11 0 0 0 15 15 15 15 15 0
8 0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 0 0 0
9 0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



Gestion des exceptions

Gestion des exceptions

Syntaxe générale

```
try:  
    ...  
except <ExceptionType> as <name>: # matching  
    ...  
except: # default  
    ...  
else: # si aucune exception n'a été levée  
    ...  
finally: # dans tous les cas  
    ...
```

Gestion des exceptions

Créer et lever une exception

```
# Création
class MonException(Exception): pass

# Lever une exception
raise MonException

raise MonException("mon message")
```

Gestion des exceptions

Exemple

```
prenom = ["Guillaume", "Jérôme", "Stéphane"]
indice = input("Indice: ")

try:
    indice = int(indice)
    value = prenom[indice]
except IndexError:
    print("Oups, cet indice n'existe pas")
except ValueError:
    print("L'indice entré n'est pas un nombre entier")
```

Exemple : exception.py

Gestion des exceptions

Clause 'with' et Context Manager

La clause `with` remplace le modèle `try ... catch ... finally`. Il s'utilise avec un `Context Manager`. Cet objet définit un contexte d'exécution.

Old style :

```
try:  
    f = open("mon_fichier.txt", "r")  
    data = f.read()  
except OSError as err:  
    print("Oups: {}".format(err))  
finally:  
    f.close()
```

Gestion des exceptions

Clause 'with' et Context Manager

Le `Context Manager` pour la lecture de fichier gère automatiquement la fermeture du fichier, même si un problème a été rencontré :

```
with open("mon_fichier.txt", "r") as f:  
    data = f.read()
```

Il est possible de créer ses propres `Context Manager`.

Motivations :

- principe DRY (Don't Repeat Yourself)
- principe SRP (Single Responsibility Principle)