INTRODUCTION À LA PROGRAMMATION

Christophe Saint-Jean

Transparents du cours

Code du cours

Année 2018-2019

ORGANISATION DE L'UE

L'ÉQUIPE ENSEIGNANTE

- **Christophe Saint-Jean** (Cours/TP - Resp.)
- El Hadi Zahzah (TP)
- Jordan Calandre (TP)
- Laurent Mascarilla (TP)
- Matthieu Robert (TP)
- Sanae Boutarfass (TP)

COMMUNICATION

- Questions pédagogiques : Moodle
  - Appronfondissement/Questions (Forum)
  - Organisation de l'UE/Planning (Messages privés)
- Questions administratives (Secrétariat)
  - Appartenance groupes TD/TP
  - Absences/Justifications

DISPOSITIF HORAIRE

- 5 cours de 1,5 heures (Amphithéâtre)
- 10 TPs de 1,5 heures (Sallles de TP)
- 2 créneaux de 1,5h de TEA (Salles de TP)

EVALUATION

$S_1 = \frac{1}{2}CC_1 \frac{1}{2}CC_2$

$S_2 = CC_3$

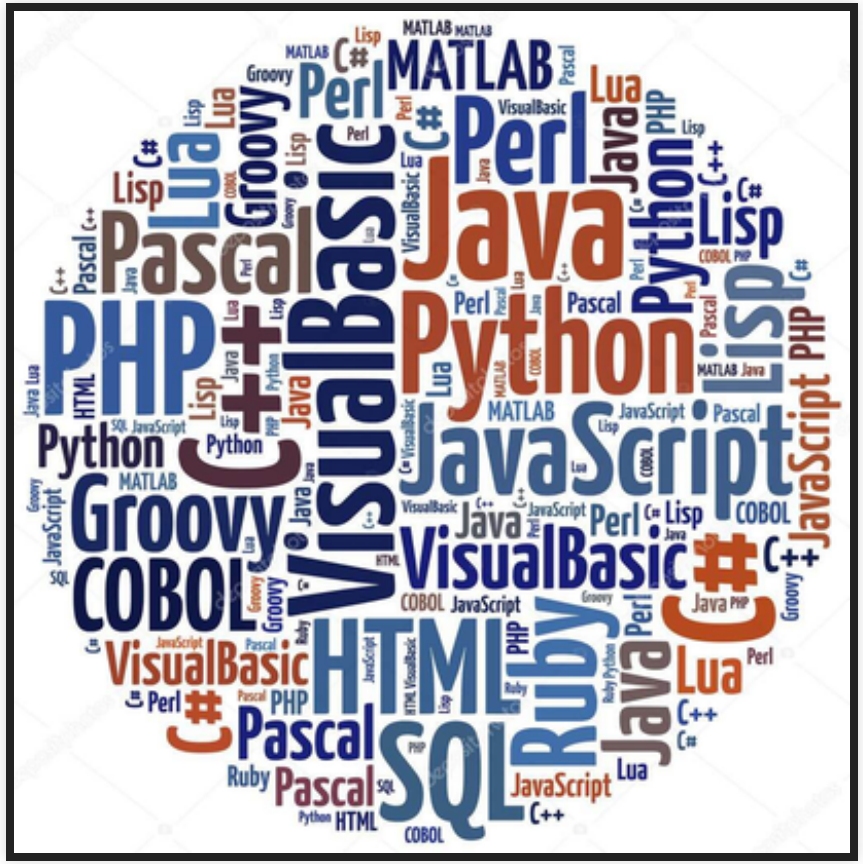
Les CC se passent en TP (5/6 et 10) sur machine:

- Une partie QCM
- Une partie évaluée par un enseignant
- Attention à la règle sur les absences

LES OBJECTIFS DE CET ENSEIGNEMENT

- Découvrir les bases de la programmation informatique
- Développer l'esprit logique par la pratique de la programmation
- Appprendre un des langages support de votre formation
- Libérer votre créativité !!

SONDAGE !


MATLAB
Perl
Lua
Groovy
Lisp
Pascal
Java
Python
PHP
C++
VisualBasic
JavaScript
C#
COBOL
HTML
SQL
Ruby

GÉNÉRALITÉS SUR LA PROGRAMMATION

LANGAGE HUMAIN

- Un certain *vocabulaire*, *une orthographe*, des *règles de grammaire* communes
- Grande expressivité et diversité
- Même si l'on commet des erreurs, nous sommes capables de comprendre "globalement" le message

LANGAGE INFORMATIQUE

- La machine traite des informations binaires: 10011001010100011O ... (même si images, sons, *programmes*, ...)
- Le vocabulaire d'instructions machine est très réduit
- Pas ou peu (encore) d'expressivité
- Pas de tolérance aux erreurs d'instructions

NIVEAU D'UN LANGAGE

Différents niveaux d'abstraction par rapport aux instructions du processeur:

- 10011001010100011O ... (Quasi-impossible)
- Langages de bas niveau (Ex. Assembleur)
- Langages de bas/haut niveau (Ex. C)
- Langages de haut niveau (Ex. Python, Java, C++, R, ...)

PROGRAMME SOURCE (OU CODE)

Un programme source est un ***texte*** qui:

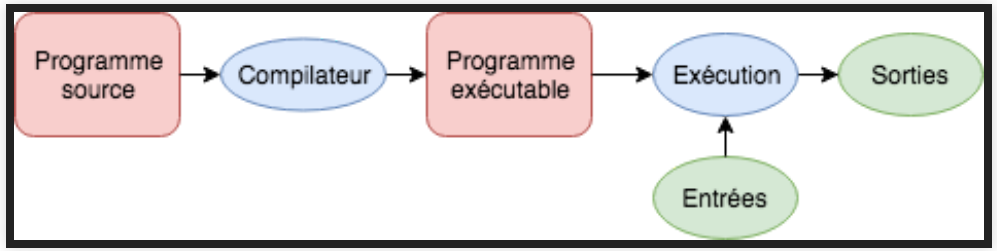
- Dépend d'un *langage de programmation*
- Utilise un certain nombre de *conventions* (nommage, opérations, ...),
- Obéit à des règles de *syntaxe*, de *grammaire*.
- En géneral, sauvegardé puis exécuté depuis un fichier.

Le mode d'exécution du programme est variable (compilation, interprétation, hybride)

LES DIFFÉRENTES SOURCES D'ERREUR

- Erreurs de syntaxe: Non respect des conventions du langage
- Erreurs d'exécution (Runtime-Error): Opération non valide lors de l'exécution
- Erreurs sémantiques: Résultat différent de celui désiré

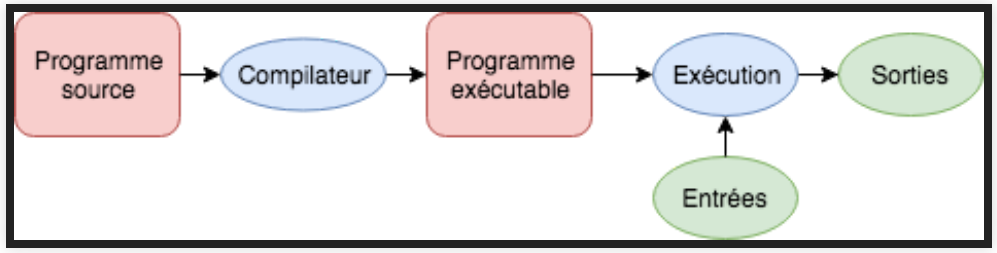
EXÉCUTION D'UN PROGRAMME (COMPILATION) 1/2



Compilation et exécution pour le compilateur gcc:

```
> gcc source.c -o prog_executable.exe
> ./prog_executable.exe
```

EXÉCUTION D'UN PROGRAMME (COMPILATION) 2/2

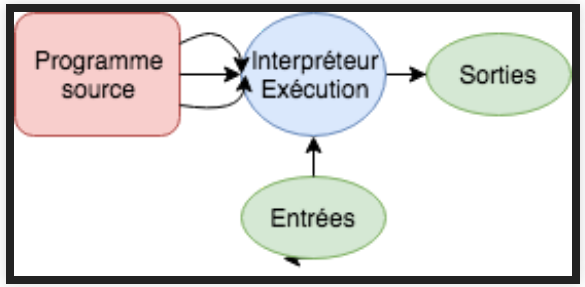


Analogie: Service de traduction intégrale à distance

Propriétés:

- Cible un type de machine (rapide)
- Vérification de la syntaxe à la compilation (Temps de compilation long)

EXÉCUTION D'UN PROGRAMME (INTERPRÉTATION) 1/2

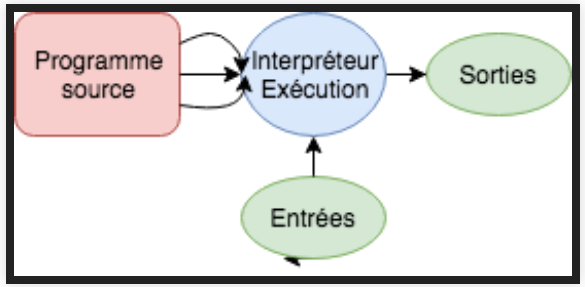


Exemple ligne de commande Unix (bash):

```
> echo $((4+5))
9
> echo $nexistepas

> [ -x  3]
-bash: [: missing `]'
```

EXÉCUTION D'UN PROGRAMME (INTERPRÉTATION) 2/2

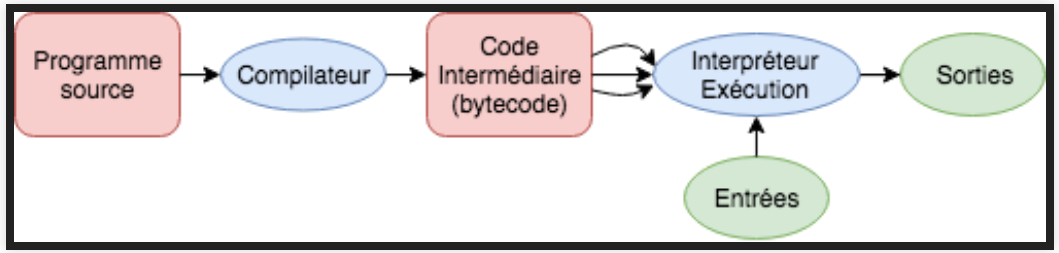


Analogie: Traduction à la volée puis exécution

Propriétés:

- Flexible car pas de cible (c'est l'interpréteur)
- Cycle Vérification + Exécution (lent)
- Découverte d'erreurs à l'exécution

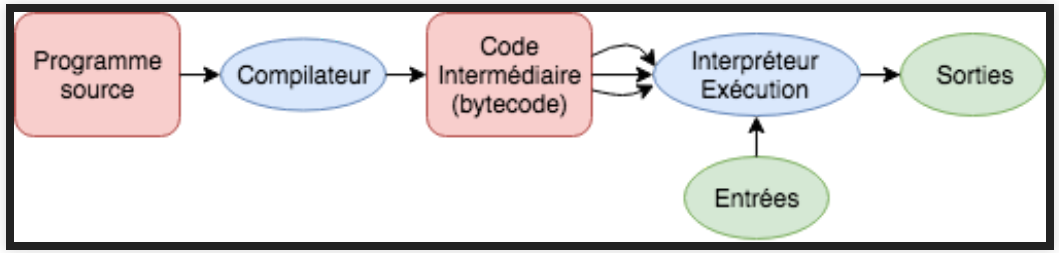
EXÉCUTION D'UN PROGRAMME (HYBRIDE) 1/2



Compilation et interprétation d'un code Java:

```
> javac HelloWorld.java
> java HelloWorld
```

EXÉCUTION D'UN PROGRAMME (HYBRIDE) 2/2



Le meilleur des deux mondes:

- Flexible
- Relativement rapide
- Multi-cible
- Supprime les erreurs de syntaxe à la compilation

MODE D'EXÉCUTION - CONCLUSION

- De nombreux langages disposent à la fois de compilateurs et d'interpréteurs.
- C'est l'usage historique qui a fait penché la balance...
- *Python* s'exécute en mode hybride même s'il pourrait croire qu'il s'agit d'un pur interpréteur:

```
> python HelloWorld.py
```

Arguments techniques - hors cadre de l'UE

LES OUTILS D'ÉDITION DU CODE 1/2

Un "bon" *environnement de développement intégré* est un programme qui:

- Facilite l'édition du code (coloration syntaxique, saisie prédictive, ...)
- Intègre les règles de bonnes pratiques d'un langage (Ex.: PEP8 pour Python)
- Détecte les erreurs de syntaxe lors de l'édition

LES OUTILS D'ÉDITION DU CODE 2/2

Un "bon" *environnement de développement intégré* est un programme qui:

- Permet de compiler/d'interpreter un code source par un clic
- Donne accès à des outils de débogage (valeurs des variables, point d'arrêt, ...)
- Gestion de projets, des versions, lancement du code à distance, ...

GÉNÉRALITÉS SUR LES LANGAGES DE PROGRAMMATION

QUELQUES PARADIGMES

On peut catégoriser les langages suivants des propriétés qui les caractérisent:

- Langages impératifs
- Langages procéduraux
- Langages à objets
- Langages déclaratifs

et bien d'autres ...

La plupart des langages sont multi-paradigmes.

LANGAGE IMPÉRATIF 1/2

Un langage est dit impératif lorsque le programme correspond à une succession d'instructions:

```
Instruction 1
Instruction 2
...
Instruction n
```

Chaque instruction tient compte de l'état du système (mémoire, E/S, ...) et peut le modifier.

LANGAGE IMPÉRATIF 2/2

On y trouve le même genre d'instructions:

- Affectations d'une valeur à une *variable*
- Le saut conditionnel "If"
- Les répétitives "Pour" et "Tant que"
- Optionnel: le saut inconditionnel "Goto"

LANGAGE PROCÉDURAL

Il s'agit simplement de pouvoir regrouper un ensemble d'instructions nommé **procédure** (ou *routine* ou *sous-routine*).

```
Procédure Quelconque:
    Instruction 1
    Instruction 2
    ...
    Instruction n
```

On parle de **programme modulaire** lorsque regroupe thématiquement des procédures dans un *module* (ou *bibliothèque* ou *paquet*)

LANGAGE ORIENTÉ OBJET 1/2

Dans ce type de langage, les programmes sont organisés autour de briques logicielles appelées *Objets* qui:

- représente un concept, une entité réelle ou non
- possède une représentation interne (*attributs* ou *slots*)
- disposent de *méthodes* ou *slots*:
  - récupérer/changer sa répresentation interne
  - exécuter des traitements
  - interagir avec d'autres objets

LANGAGE ORIENTÉ OBJET 2/2

```
class Ballon {
   double rayon;
   Ballon(double rayon) {
     this.rayon = rayon;
   }
   void gonfler() {
     this.rayon = this.rayon + 1;
   }
   void collision(Ballon autre){
     double rayon = (this.rayon + autre.rayon)/2;
     this.rayon = autre.rayon = rayon;
   }
 }
```

APARTÉ : LANGAGE DÉCLARATIF

On parle de *programmation déclarative* lorsqu'on décrit le résultat attendu, les objectifs (**quoi**) sans donner la manière de le faire (**comment**).

Exemple: page HTML5 minimaliste

```
<!DOCTYPE html>
<html>
<head><title>This is Hello World page</title></head>
<body>
<h1>Hello World</h1>
</body>
</html>
```

GÉNÉRALITÉS SUR LE LANGAGE PYTHON

CARACTÉRISTIQUES DU LANGAGE

Créé en 1991, Python est un langage multi-paradigme:

- Impératif et procédural, orienté-objet
- Fourni avec un interpréteur (usage dominant) qui:
  - infère le type des variables lors de l'exécution (typage dynamique)
  - gère la mémoire automatiquement (ramasse-miettes)
- qui dispose d'une grande bibliothèque de base (modules)
- Version actuelle: **3.7.0** (ou *2.7.15*)

OUTILS D'ÉDITION DU CODE PYTHON

- *Thonny* (cette UE)
- *Visual Code*
- *Spyder* (Calcul scientifique)
- NetBeans, *PyCharm*
- *Jupyter Notebook* (voir TEA)

Liste complète

EXEMPLE D'UN PROGRAMME PYTHON

```
print('Hello World!!')
```

Test via Thonny:

- en mode interactif
- en mode script

Par convention, les scripts Python ont pour extension ***.py***

LES VARIABLES

VARIABLE (DÉFINITION)

Une variable est une zone de la mémoire dans laquelle une valeur est stockée.

Elle est désignée par:

- un **nom** pour le programmeur
- une **adresse** pour l'ordinateur

La fonction *id* renvoie un nombre unique (~ adresse) qui qualifie une variable.

TYPE D'UNE VARIABLE

Le **type** d'une variable définit les opérations valides pour elle.

Types élémentaires:

- Les nombres entiers
- Les nombres "réels" ou "à virgule"
- Les chaînes de caractères
- Les booléens

La fonction *type* renvoie le type d'une variable.

DÉCLARATION D'UN VARIABLE

En Python, la **déclaration** et l'**initialisation** d'une variable se fait de manière simultanée.

```
In [1]: a = 2

In [2]: type(a)
Out[2]: int

In [3]: id(a)
Out[3]: 4525733248
```

REMARQUES SUR LE TYPAGE 1/2

- Le type d'une variable est donné par le type de l'expression à droite du = (*Inférence de type*)
- Toute variable a un type (*Typage fort*) (qui peut changer lors de l'éxécution (*Typage dynamique*)

REMARQUES SUR LE TYPAGE 2/2

```
In [1]: a = 2

In [2]: type(a)
Out[2]: int

In [3]: a = 3.14159

In [4]: type(a)
Out[4]: float

In [5]: type(3.14159)
Out[5]: float
```

LES TYPES DE BASE DE PYTHON

LES TYPES NUMÉRIQUES: *INT*

- entier avec précision arbitraire
- attention, le type *long* existait dans les versions précédentes

```
In [1]: type(2)
Out[1]: int

In [2]: 2**1024
Out[2]: 179769313486231590772930519078902473361797697894230657
```

LES TYPES NUMÉRIQUES: *FLOAT*

- nombre à virgule avec une précision **fixe**

$$[2.26..*10^{-308}, 1.79..*10^{308}]$$

- Tous les réels ne sont pas représentables par le type *float*

```
In [1]: 1.79e308
Out[1]: 1.79e+308

In [2]: 1.79e308*10
Out[2]: inf

In [3]: type(1.79e308*10)
Out[3]: float

In [4]: 1e20 + 1
```

AUTRES TYPES NUMÉRIQUES

Seront également évoqués si besoin en TP:

- *complex* : les nombres complexes
- *decimal* : les nombres décimaux
- *fraction* : les nombres rationnels

pour mémoire: $float \subset decimal \subset fraction \subset \mathbb{R}$

type *int* := $\mathbb{Z}$

OPÉRATIONS SUR LES TYPES NUMÉRIQUES

- Arithmétique usuelle: +, -, *, /
- Division entière: a // b (arrondi vers $-\infty$)
- Reste de la division entière: a % b
- Puissance: a ** b ou pow(a,b)
- Valeur absolue: abs(a)

LE TYPE BOOLÉEN

- Il permet de représenter les valeurs de vérité *True* ou *False*
- Opérations sur les booléens (priorité décroissante):
  - *not*: négation logique
  - *and*: "et" logique
  - *or*: "ou" logique

```
In [1]: type(True)
Out[1]: bool

In [2]: True and not False
Out[2]: True
```

LES CHAINES DE CARACTÈRES 1/2

Les chaînes de caractères sont délimitées par les simples ou doubles guillemets.

L'opération de **concaténation** de deux chaînes est effectuée par le symbole +.

LES CHAINES DE CARACTÈRES 2/2

```
In [1]: a = 'abc'

In [2]: b = 'def'

In [3]: a + b
Out[3]: 'abcdef'
```

LE TYPE NONETYPE

- *None* est une valeur spéciale pour indiquer qu'une variable *existe* mais *n'a pas de contenu connu* (valeur manquante)
- Peut indiquer également "n'a pas de sens"

```
In [1: a
...
NameError: name 'a' is not defined

In [2]: a = None

In [3]: a
```

CONVERSION IMPLICITE ENTRE TYPES NUMÉRIQUES

```
In [1]: type(1+2.)
Out[1]: float

In [2]: 2**100000 + 5.
...
OverflowError: int too large to convert to float
```

Ccl: calcul mixte int/float -> float

CONVERSION IMPLICITE ENTRE TYPES 1/2

Par **convention**, le *True* "équivaut" à 1 et *False* à 0 dans les calculs numériques.

```
In [1]: True * 2
Out[1]: 2
```

CONVERSION IMPLICITE ENTRE TYPES 2/2

A l'opposé, *0* ou *None* sont considérées comme *False*, sinon *True* dans les expressions booléennes.

```
In [1]: not 0
Out[1]: True
```

CONVERSION EXPLICITE ENTRE TYPES

On peut forcer la conversion avec la syntaxe:

```
    <type>(<expression>)
```

Exemples:

```
In [1]: int(4.7)
Out[1]: 4

In [2]: bool(0)
Out[2]: False

In [3]: str(4.7)
Out[3]: '4.7'
```

ENTRÉES-SORTIE CLAVIER/ECRAN

AFFICHAGE À L'ÉCRAN

Syntaxe sur la fonction *print*:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=Fa
```

- ... : *print* accepte une suite de valeurs converties en *str*
- sep: le séparateur
- end: le caractère de fin
- file : flux de sortie (sys.stdout est l'écran)
- flush : force à vider le flux immédiatement.

EXEMPLES D'AFFICHAGE 1/2

```
In [1]: année = 2018

In [2]: print(année)
2018

In [3: print("année", année)
année 2018
```

EXEMPLES D'AFFICHAGE 2/2

```
In [4]: print("l'année", année, end=""); print(" est un bon cr
l'année  2018 est un bon cru !!

In [5]: print(année, année*2, année / 2, type(année), sep=", "
2018, 4036, 1009.0, <class 'int'>
```

On remarque:

- Il n'a pas de "Out" (Output) pour *print*
- *Confusion usuelle*: l'affichage à l'écran n'est pas un retour de la fonction *print*.

SAISIE UTILISATEUR

Fonction *input* :

```
  input(prompt=None)
```

retourne la chaîne de caractères saisie.

```
In [1]: a = int(input('Nombre de pas ? '))
Nombre de pas ? 12

In [2]: b = float(input('pi ? '))
pi ? 3,14
...
ValueError: could not convert string to float: '3,14'
```

C'est la convention anglaise pour les *float*.

STRUCTURES CONDITIONNELLES

if, else, elseif

INSTRUCTION IF, ELSE

```
if <condition>:
  <instructions si True>
[else:
  <instructions si False>]
```

- La partie else est optionnelle
- **L'indendation est obligatoire car elle marque le début et la fin d'un bloc d'instructions**

CONDITION DANS UN IF 1/2

Une condition peut-être:

- une expression booléenne construite par:
  - un opérateur de comparaison: <=, <, >, >=
  - un opérateur d'égalité: ==, !=, <>
  - un opérateur d'identité: *is*, *is not*
  - un opérateur d'appartenance : *in*, *not in*

CONDITION DANS UN IF 2/2

Une condition peut-être aussi:

- une valeur numérique: 0 équivaut à *False* sinon *True*
- une chaine de caractères: '' équivaut à *False* sinon *True*
- *None* équivaut à *False*
- *etc*

Les conditions peuvent être combinées par *not*, *and* et *or*

EXEMPLES IF

```
In [1]: a = int(input('Nombre de pas ? '))
Nombre de pas ? 9680

In [2]: if a > 8000:
            print('Vous avez fait le nombre de pas journaliers r
        else:
            print('Faire plus d\'exercice.')
###
Vous avez fait le nombre de pas journaliers recommandé.

In [3]: if a > 8000: print('bravo')
bravo
```

INSTRUCTION ELIF

- *elif* est une contraction de *else if* qui n'oblige pas à une indentation supplémentaire
- C'est le "switch case" de Python.

```
In [1]: a = 7

In [2]: if a < 0:
            print('a est négatif')
        elif a % 2 == 0:
            print('a est pair')
        elif a % 3 == 0:
            print('a est divisible par 3')
        else:
            print('a est un nombre positif, impair et non divisi
###
a est un nombre positif, impair et non divisible par 3
```

DÉMONSTRATION !

- Conditions if imbriquées : a $\in$ [0,1]
- Année Bissextile

RÉPÉTITIVES

while, break, for, continue

RÉPÉTITIVE *"TANT QUE"*

Syntaxe:

```
while <condition>:
  <instructions>
```

La condition est évaluée **avant** chaque éxécution des instructions.

Conditions de sortie du "while":

- La condition n'est pas vérifiée.
- Une sortie explicite par *break*.
- Crash du programme...

CAS FRÉQUENTS D'UTILISATION *"TANT QUE"* 1/2

- Répéter *n* fois

```
a = 1
while a < 10:
  <instructions>
  a = a + 1
```

- Compter le nombre d'itérations

```
cpt = 0
while <condition>:
  <instructions>
  cpt += 1
```

CAS FRÉQUENTS D'UTILISATION *"TANT QUE"* 2/2

- Parcourir un intervalle de valeurs [a,b] par pas de eps

```
x = a
while x <= b:
  <instructions>
  x += eps
```

- Une boucle d'événements

```
while True:
  <instructions>
  if <événement particulier>:
    break
```

PETITS EXOS SUR "WHILE"

- Combien de fois peut on diviser un nombre par deux ?
- Compter le nombre de entiers impairs entre 1 et 1000 divisibles par 3 mais pas par 7.
- Racine carrée entière: Etant donné un entier n, déterminer le plus grand nombre entier r tel que $r^2 \leq n$.

RÉPÉTITIVE FOR

Syntaxe:

```
for <variable> in <sequence>:
   <instructions>
```

La séquence peut être:

- Une plage de valeurs avec *range*
- Une chaîne de caractères
- Une liste, un tuple (plus tard)
- personnalisée ...

Le terme anglais est *iterable*.

INSTRUCTION *RANGE*

Syntaxe:

```
range(start, stop[, step]) -> range object
```

Cas d'utilisation:

- range(i, j) -> i, i+1, i+2, ..., j-1.
- range(i) -> 0, 1, ..., i-1..

Attention, step peut être négatif.

EXEMPLES *FOR* AVEC *RANGE*

```
In [1]: for i in range(1,10):
            print(i, end=' ')
###
1 2 3 4 5 6 7 8 9
In [2]: for i in range(5):
            print(i, end=' ')
###
0 1 2 3 4
In [3]: for i in range(8, 0, -1):
            print(i, end=' ')
###
8 7 6 5 4 3 2 1
```

EXEMPLE *FOR* AVEC *STR*

```
In [1]: for c in 'Python':
            print(c, end=', ')
###
P, y, t, h, o, n, @

In [2]: cpt = 0
        for c in 'Pythonneries':
            if c == 'e':
                cpt += 1
        print('Nombre de "e": ', cpt)
###
Nombre de "e":  2
```

EXERCICE D'APPLICATION

$$\lim_{n\to+\infty} 4\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = \pi$$

Ecrire un programme basé sur cette formule qui approxime π:

```
In [1]: n = 10**6; som = 0

In [2]: for k in range(n+1):
            som = som + (-1)**k / (2*k+1)

In [3]: print(4*som)
3.1414926535900345

In [4]: import math; print(math.pi)
3.141592653589793
```

BONUS: INSTRUCTION *CONTINUE*

continue permet d'interrompre une itération:

- On retourne au test de la condition dans *while*
- Prochaine itération dans *for*

```
In [1]: for i in range(10):
            if i % 2 == 0:
              continue
            print(i, end=' ')
###
1 3 5 7 9
```

BONUS: *ELSE* DANS *WHILE* ET *FOR*

- *else* est exécuté si:
  - la condition du *while* est *False*
  - *for* a parcouru toute la séquence
- *else* n'est pas exécuté si interruption par un *break*.

```
In [2]: for i in range(10):
            print(i, end=' ')
        else:
            print('\nTerminé')
###
0 1 2 3 4 5 6 7 8 9
Terminé
```

STRUCTURES DE DONNNÉES

liste, tuple, dictionnaire

LISTE

Une **liste** est une structure de données qui contient une séquence de valeurs.

Syntaxe:

```
[<valeur_1>, <valeur_2>, ..., <valeur_n>]
```

- Les valeurs ne sont pas nécessairement de même type.
- Une liste est une séquence

EXEMPLES DE LISTES

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: Sam = [28 , 'Toronto', False, None]

In [3]: Jeff = [70, 'Cambridge', True, 25]

In [4]: People = [Sam, Jeff]

In [5]: print(People)
[[28, 'Toronto', False, None], [70, 'Cambridge', True, 25]]
```

UTILISATION D'UNE LISTE

On peut rappeler un élément particulier d'une liste par son **indice**.

Pour une liste de longueur *n*, l'indice est un entier entre **0** et **n-1**.

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: len(couleurs)   # longueur de la liste
Out[2]: 3

In [3]: couleurs[0]
Out[3]: 'rouge'

In [4]: couleurs[5]
...
IndexError: list index out of range
```

LISTE: INDIÇAGE NÉGATIF

Pour faciliter l'accès des derniers éléments d'une liste, *Python* a introduit l'indiçage négatif.

liste	**'h'**	**'e'**	**'l'**	**'l'**	**'o'**
indice positif	0	1	2	3	4
indice négatif	-5	-4	-3	-2	-1

Le dernier élément de la liste toujours l'indice -1.

EXTRACTION D'UNE SOUS-LISTE 1/3

Syntaxe (~ *range*):

```
L[<start>:<stop>:<step>]
```

Quelques cas fréquents (L est une liste):

- Eléments entre l'indice 2 (inclus) et l'indice 5 (exclus):
  L[2:5]
- Eléments à partir de l'indice 4:
  L[4:]
- Les 10 premiers éléments:
  L[:10]

EXTRACTION D'UNE SOUS-LISTE 2/3

- Duplication de la liste:
  L[:]
- Un élément sur 2:
  L[::2]

EXTRACTION D'UNE SOUS-LISTE 3/3

Quelques cas fréquents avec indice négatif:

- Les 5 derniers éléments:
  L[-5:]
- Tout sauf les derniers 3 éléments:
  L[:-3]
- Duplication de a dans l'ordre inverse:
  L[::-1]

AFFECTER UNE VALEUR À UNE LISTE EXISTANTE

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: couleurs[0] = 'jaune'

In [3]: couleurs
Out[3]: ['jaune', 'vert', 'bleu']

In [4]: couleurs[:2]= [34, 48]

In [5]: couleurs
Out[5]: [34, 48, 'bleu']
```

INSÉRER UN ÉLÉMENT EN FIN DE LISTE : *APPEND*

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: couleurs.append('cyan')

In [3]: couleurs
Out[3]: ['rouge', 'vert', 'bleu', 'cyan']

In [4]: L = []   ## liste vide !!!

In [5]: L.append(4)

In [6]: L
Out[6]: [4]
```

INSÉRER UN ÉLÉMENT : *INSERT*

Syntaxe (~ *range*):

```
    L.insert(<indice>, <element>)
```

Insère avec décalage vers la fin:

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: couleurs.insert(2, 'cyan')

In [3]: couleurs
Out[3]: ['rouge', 'vert', 'cyan', 'bleu']
```

CONCATÉNER DEUX LISTES 1/2

Rappel: Concaténer c'est mettre bout à bout deux structures de données.

Deux syntaxes:

```
    L = L1 + L2 ou L1+=L2
    L1.extend(L2)
```

```
In [1]: ['rouge', 'vert', 'bleu'] + ['r', 'v', 'b']
Out[1]: ['rouge', 'vert', 'bleu', 'r', 'v', 'b']
```

CONCATÉNER DEUX LISTES 2/2

```
In [2]: couleurs = ['rouge', 'vert', 'bleu']

In [3]: couleurs.extend(['r', 'v', 'b'])

In [4]: print(couleurs)
['rouge', 'vert', 'bleu', 'r', 'v', 'b']
```

SUPPRESSION D'ÉLÉMENTS: *DEL* OU *REMOVE*

```
    del L[3] ou del L[3:]    ## par indice
    L.remove(5)              ## la première occurrence
```

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: del couleurs[1]

In [3]: couleurs
Out[3]: ['rouge', 'bleu']

In [4]: couleurs = ['rouge', 'vert', 'bleu', 'vert', 'orange']

In [5]: couleurs.remove('vert')

In [6]: couleurs
Out[6]: ['rouge', 'bleu', 'vert', 'orange']
```

PARCOURS D'UNE LISTE PAR INDICE

On peut utiliser les indices.

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: for i in range(len(couleurs)):
            print(couleurs[i].upper(), end=', ')
###
ROUGE, VERT, BLEU,
```

On peut très bien faire aussi avec un *while*

PARCOURS D'UNE LISTE PAR ITÉRATEUR

Rappel: *for* permet d'itérer toute séquence.

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: for couleur in couleurs:
            print(couleur.upper(), end=', ')
###
ROUGE, VERT, BLEU,
```

Très simple, mais on a perdu la position dans la liste !

PARCOURS D'UNE LISTE PAR *ENUMERATE*

```
In [1]: couleurs = ['rouge', 'vert', 'bleu']

In [2]: for i, couleur in enumerate(couleurs):
            print('indice:', i, 'valeur:', couleur.upper())
###
indice: 0 valeur: ROUGE
indice: 1 valeur: VERT
indice: 2 valeur: BLEU
```

C'est le meilleur choix si l'on a besoin de l'indice en plus de la valeur.

EXEMPLE SUR LES LISTES 1/2

A partir d'une liste de noms, sélectionner ceux qui commencent ou terminent par une voyelle.

```
In [1]: voyelles = ['a', 'e', 'i', 'o', 'u', 'y']

In [2]: noms = ['mila', 'mathis', 'anne', 'myriam', 'eloan', '

In [3]: select = []

In [4]: for nom in noms:
            for voyelle in voyelles:
                if nom[0] == voyelle or nom[-1]==voyelle:
                    select.append(nom)
                    break

In [5]: select
Out[5]: ['mila', 'anne', 'eloan', 'pierre']
```

EXEMPLE SUR LES LISTES 2/2

Une version plus compacte:

```
In [1]: voyelles = 'aeiouy'

In [2]: noms = ['mila', 'mathis', 'anne', 'myriam', 'eloan', '

In [3]: select = []

In [4]: for nom in noms:
            if nom[0] in voyelles or nom[-1] in voyelles:
                select.append(nom)

In [5]: select
Out[5]: ['mila', 'anne', 'eloan', 'pierre']
```

TUPLE

Un **tuple** est une structure de données qui contient une séquence de valeurs.

Syntaxe:

```
(<valeur_1>, <valeur_2>, ..., <valeur_n>)
```

COMPARAISON *LISTE*/*TUPLE*:

	Liste	Tuple
Taille	dynamique	fixe
Ajout	oui	non
Suppression	oui	non
Parcours	oui	oui
Test 'in'	oui	oui
Rapidité	-	+
Mémoire	-	+

EXEMPLE STOCKAGE AVEC *TUPLE*

```
In [1]: t = ('a', 5000, 'c', 234)

In [2]: t[2:4]
Out[2]: ('c', 234)

In [3]: t = t[:2] + ('b', 858) + t[2:]

In [4]: t
Out[4]: ('a', 5000, 'b', 858, 'c', 234)
```

DÉPLIEMENT (UNPACKING) D'UN TUPLE

Permet de faire l'affectation multiple de valeurs

```
In [1]: (a, b, c) = (1, '2', '3.0')

In [2]: print(a, b, c)
1 2 3.0

In [3]: a, b, c = 1, '2', '3.0'  ## syntaxe usuelle identique
```

Echange de valeurs de variables

```
In [1]: a, b = b, a
```

RETOUR SUR *ENUMERATE*

```
In [1]: lettres = 'abcd'

In [2]: for el in enumerate(lettres):
            print(el)
####
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

- *enumerate* renvoie une séquence de tuples (*indice*, *valeur*) que l'on a déplié.

DICTIONNAIRE

Un **dictionnaire** est une structure de données qui contient une séquence de couples (clé, valeur).

Syntaxe:

```
    {<cle_1>: <valeur_1>, ..., <cle_n>: <valeur_n>}
```

Exemple:

```
In [1]: Sam = { 'age': 28, 'location': 'Toronto', 'active': Fa

In [2]: Sam['age']   # ou Sam.get('age')
Out[2]: 28
```

PROPRIÉTÉS D'UN DICTIONNAIRE

- Les clés sont uniques, les valeurs peuvent être mutiples.
- Les clés sont immuables (chaînes, nombres, tuples) i.e. ne sont pas modifiables.
- Les valeurs peuvent être mise à jour.

AJOUT/MISE À JOUR

```
In [1]: Sam = {'age': 28}

In [2]: Sam['affiliation'] = 'La Rochelle'

In [3]: Sam['age'] = 29

In [4]: Sam
Out[n4]: {'age': 29, 'affiliation': 'La Rochelle'}
```

MISE À JOUR/SUPPRESSION

```
In [1]: Sam = {'age': 29, 'affiliation': 'La Rochelle'}

In [2]: Sam.update({ 'age': 30, 'location': 'Toronto'})

In [3]: Sam
Out[3]: {'age': 30, 'affiliation': 'La Rochelle', 'location':

In [4]: del Sam['location']

In [5]: 'location' in Sam
Out[5]: False

In [6]: Sam['location']
KeyError: 'location'
```

PARCOURS D'UN DICTIONNAIRE

Trois façons de parcourir un dictionaire *d*:

- *d.items()*: séquence de paires *cle*:*valeur*.
- *d.keys()* : séquence des clés.
- *d.values()*: séquence des valeurs.

```
In [1]: for cle, valeur in Sam.items():
            print(cle, valeur, sep=': ', end='  ;  ')
###
age: 30  ;  affiliation: La Rochelle  ;
```

EXERCICES SUR LES DICTIONNAIRES

- Compter le nombre d'occurrences d'une lettre de l'alphabet dans un texte.
- La Famille Simpson

FONCTIONS

Les fonctions sont un moyen d'exécuter un ensemble d'instructions en les nommant.

Syntaxe de base:

```
def <nom_de_la_fonction>(<parametre_1>, ..., <parametre_n>):
    <instructions>
```

Si l'exécution de ces instructions dépend de certaines valeurs, on parlera de fonction des *paramètres*.

Les parenthèses sont obligatoires, même si la fonction n'a pas aucun paramètre.

EXEMPLE DE FONCTION

```
In [1]: def somme_n_entiers(n):
            som = 0
            for i in range(1, n+1):
              som += i
            print('La somme des', n, 'premiers entiers: ', som,

In [2]: somme_n_entiers(10)
La somme des 10 premiers entiers:  55
```

L'INSTRUCTION *RETURN* 1/2

return indique ce que renvoie la fonction.

```
In [1]: def somme_n_entiers(n):
            som = 0
            for i in range(1, n+1):
              som += i
            return som
In [2]: s = somme_n_entiers(10)

In [3]: print('La somme des', 10, 'premiers entiers: ', s, sep
La somme des 10 premiers entiers:  55
```

L'INSTRUCTION *RETURN* 2/2

Pour retourner plusieurs valeurs, on utilisera un tuple (ou une liste) ou dictionnaire:

```
In [1]: def decomposition(n, m):
            a = n // m
            b = n % m
            return (a,b)  # ou return {'quotient': a, 'reste': b

In [2]: a, b = decomposition (43, 7)

In [3]: print('43 =', a, '* 7 +',b)
43 = 6 * 7 + 1
```

UNE FONCTION SANS RETURN ? !

Une fonction sans *return* explicite "retournera" la valeur spéciale *None*

```
In [1]: def f(a, b):
            c = a + b
In [2]: res = f(4, 3)

In [3]: res   #None n'affiche rien !

In [4]: print(res)
None
```

VALEURS PAR DÉFAUT

On peut donner des valeurs par défaut à certains paramètres que l'on positionne à droite:

```
In [1]: def f(a,b=2):
            return a**b
In [2]: f(2)
Out[2]: 4

In [3]: f(4,3)
Out[3]: 64

In [4]: def f(b=2, a):
            return a**b
  File "<ipython-input-6-8d9bc7ce0f4b>", line 1
    def f(b=2, a):
          ^
SyntaxError: non-default argument follows default argument
```

TOUT PEUT ÊTRE PARAMÈTRE !

```
In [1]: def f(x):
            return x**2

In [2]: def mon_map(L,fun):
            res = []
            for l in L:
              res.append(fun(l))
            return res

In [3]: L = [1, 0, 3, 2, -3]

In [4]: print(mon_map(L, f))
[1, 0, 9, 4, 9]
```

LA RÉCURSIVITÉ

Une fonction est dite récursive si elle se calcule en faisant appel à elle même.

```
def factorielle(n):
  if n < 2:
    return 1
  return n * factorielle(n-1)
```

PORTÉE DES VARIABLES

Quand et comment mes variables sont accessibles ?

CAS SIMPLES 1/2

On a déjà vu que les variables n'existent que si elles ont été assignées (valeur ou *None*)

```
In [1]: a
NameError: name 'a' is not defined

In [2]: a = 3

In [3]: a
Out[3]: 3
```

CAS SIMPLE 2/2

Et dans un bloc:

```
In [1]: for i in range(10):
            a=3
In [2]: print(a, i)
3 9
```

On parle du niveau global ou principal.

PORTÉE DES VARIABLES: FONCTIONS 1/4

```
In [1]: def f(b):
            a = 3
In [2]: a
NameError: name 'a' is not defined

In [3]: b
NameError: name 'b' is not defined
```

Les variables **locales** et les **paramètres** n'existent pas à l'extérieur d'une fonction.

PORTÉE DES VARIABLES: FONCTIONS 2/4

```
In [1]: a = 1

In [2]: def f():
            print(a)

In [3]: f()
1
```

- Les variables du niveau supérieur sont utilisables dans la fonction
- C'est considéré comme une *mauvaise pratique* si `f` est paramétrée par a

PORTÉE DES VARIABLES: FONCTIONS 3/4

```
In [1]: a = 1

In [2]: def f():
            a = 2

In [3]: f()

In [4]: a
Out[4]: 1
1
```

L'affectation = dans une fonction ne change pas la valeur d'une variable.

PORTÉE DES VARIABLES: FONCTIONS 4/4

```
In [1]: a = 1

In [2]: def f(b):
            b = 2

In [3]: f(a)

In [4]: a
Out[4]: 1
```

Idem si c'est un paramètre.

PORTÉE DES VARIABLES: FONCTIONS ET LISTES

On peut modifier des objets en passant par des méthodes (Ex.: append pour une liste)

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

In [2]: def f(liste):
            liste.append(4)

In [3]: l
Out[3]: [1, 2, 3]

In [4]: f(l)

In [5]: l
Out[5]: [1, 2, 3, 4]
```

BONUS: PYTHON ET RÉFÉRENCES

```
In [1]: l1 = [1, 2, 3]

In [2]: l2 = l1

In [3]: l2.append(4)

In [4]: l1
Out[4]: [1, 2, 3, 4]

In [5]: id(l1), id(l2)
Out[5]: (4520736584, 4520736584)
```

- Les variables *l1* et *l2* **référencent** le même objet.
- On peut créer une copie (entre autres) par:

```
l2 = l1[:] ou l2 = l1.copy()
```

LA DOCUMENTATION

Parce ce que:

- Le programmeur n'a pas une mémoire infaillible
- On peut espérer faire du code que les autres vont lire

On peut documenter tous les niveaux du code: fonction, ~~classe~~, ~~module~~

Il s'agit de placer une chaine de caractères au bon endroit ...

Règles pour la documentation: PEP 257

CONVENTIONS POUR LA DOCUMENATION SIMPLE

- On utilise des triples guillemets pour commencer et finir la ligne.
- La première lettre est en majuscule et on termine la ligne par un point.
- Indentation identique au code.
- On décrit ce que fait la fonction.

EXEMPLE DE DOCUMENTATION SIMPLE

```
In [1]: def carre(x):
   ...:     """Calcul du carré d'un nombre."""
   ...:     return x * x
In [2]: print(carre.__doc__)
Calcul du carré d'un nombre.

In [2]: help(carre)
```

POUR UNE DOCUMENTATION PLUS LONGUE

```
"""La première ligne décrit la fonction.

   Description textuelle de la fonction.
   Description textuelle de la fonction.
   Description textuelle de la fonction.
"""
```

EXEMPLE DE DOCUMENTATION PLUS RICHE 1/2

```
def monpow(a, b):
  """Calcule a à la puissance b

    Ici une explication longue de la puissance
    d'un nombre :math:`a^b = aa..a` b fois

    :param a: la valeur
    :param b: l'exposant
    :type a: int, float,...
    :type b: int, float,...
    :returns: a**b
    :rtype: int, float
```

EXEMPLE DE DOCUMENTATION PLUS RICHE 2/2

```
""""" suite ....

    :Exemples:
    >>>nompow(2, 3)
    8
    >>>nompow(2., 2)
    4.0

    .. note:: c'est une version accélérée de la puissance par
    .. seealso:: pow
    .. warning:: a et b sont des nombres
    """
    return a**b
```

FONCTIONS EXTERNES ET MODULES

- Comment organiser son code pour le réutiliser ?
- Comment utiliser du code Python fait par d'autres ?

EXEMPLE: IMPORTER UNE FONCTION 1/2

Contenu du fichier "racine.py":

```
def racine_dicho(x):
  min, max, eps = 0, x, 1e-10
  while True:
    r = (min + max) / 2
    if abs(r*r - x) < eps:
        break
    elif r*r < a:
        min = r
    else:
        max = r
  return r
```

EXEMPLE: IMPORTER UNE FONCTION 2/2

Contenu du fichier "comp_rac.py":

```
import math
import racine

x = float(input('x ?'))
if abs(math.sqrt(x) - racine.racine_dicho(x)) < 1e-16:
  print("Les valeurs sont les mêmes")
else:
  print("Roger, on a un problème !!!")
```

DIRECTIVE IMPORT ET SES CONVENTIONS 1/2

```
import xxx

v = xxx.vvv
a = xxx.fff(4) # directement
```

- Le module xxx est un fichier nommé xxx.py
- Le module xxx à importer est dans le même répertoire que le module yyy qui importe
- Dans yyy.py, on importe xxx par la directive import

DIRECTIVE IMPORT ET SES CONVENTIONS 2/2

- On doit préciser le nom du module chaque fois à moins de faire un raccourci:

```
import xxx

f = xxx.fff # indirectement: raccourci local
a = f(4)
```

- Le module créé par l'utilisateur est prioritaire à celui fourni par Python.
- Seul, le symbole xxx est importé, fff inconnu

DIRECTIVE FROM ... IMPORT ... AS ...

- On peut décider de n'importer que certains symboles (variables, fonctions, ...)

```
from racine import racine_dicho

a = racine_dicho(4)
```

- Egalement les renommer localement

```
from racine import racine_dicho as mon_sqrt
from math import sqrt as py_sqrt, exp as py_exp

a = mon_sqrt(4)
```

DIRECTIVE FROM ... IMPORT *

- On peut décider de tout importer:

```
from racine import *

a = racine_dicho(4)
```

- Pratique déconseillée car on ne maîtrise pas totalement ce qui est importé.

  On devrait plutôt faire:

```
from turtle import forward, left, right, done
```

QUELQUES MODULES FOURNIS

- random: fonctions pour produire des nombres aléatoires
- math: opérations mathématiques basiques (cosinus,sinus,exp,etc.)
- turtle: dessin à la tortue
- os: Interagir avec le système d'exploitation
- time: La date, heure, ...
- tkinter: Créer une interface graphique

Liste complète

AUTRES MODULES POPULAIRES

- Numpy, Scipy, Pandas: Calcul scientifique.
- Matplotlib: Dessin 2d (courbes, histogrammes, *etc*).
- Django, Flask: Faire des sites par programmation.
- Pillow: Manipuler des images.

Egalement, il existe des modules pour:

- Interagir avec les réseaux sociaux.
- Analyser les pages web.
- ...

FICHIERS

- Comment charger un fichier texte.
- Comment écrire des données dans un fichier texte
- Comment charger des fichiers au format .csv.

MODES D'OUVERTURE

```
f = open('fichier.txt', mode='r')
```

- ‘r’ : Lecture seule
- ‘w’ : Lecture/Ecriture (écrase le fichier existant)
- ‘a’ : Lecture/Ecriture à partir de la fin

LECTURE D'UN FICHIER TEXTE

```
f = open('fichier.txt', mode='r')
```

Méthodes de base:

- f.read(n) : lire *n* caractères
- f.readline() : lire une ligne
- f.readlines() : tout lire

EXEMPLE : LECTURE D'UN FICHIER TEXTE

```
f = open('fichier.txt', mode='r')
lignes = f.readlines()
f.close()
```

ou encore

```
with open('fichier.txt', mode='r') as f:
    lignes = f.readlines()
```

- Fermeture automatique du fichier avec *with*
- Remarque: Chaque ligne est terminée par '\n'

ECRITURE DANS UN FICHIER TEXTE

Méthodes de base:

```
# Ecrit une chaine dans f et retourne le nombre de caractères
n = f.write(chaine)
# Autre possibilité
print(3, 'Chaine', file=f)
```

EXEMPLE : ECRITURE D'UN FICHIER TEXTE

```
f = open('fichier.txt', mode='w')
n = f.write("une première ligne\n")
print("La seconde ligne", file=f)
f.close()
```

- Remarque: On peut également utiliser un *with*

LECTURE D'UN FICHIER .CSV

CSV ("Comma-separated values") = des valeurs séparées par des virgules.

Exemple: place de parking disponibles

```
dp_id,dp_parc_id,dp_libelle,dp_place_disponible,dp_date,dp_nb_
8977068,5,VIEUX PORT OUEST,289,11-11-2018 18:39:18,420,"379378
8977069,4,ENCAN,366,11-11-2018 18:39:22,406,"379864,062986826"
8977070,17,VIEUX PORT SUD,459,11-11-2018 18:39:23,500,"379925,
8977071,16,VERDUN,424,11-11-2018 18:39:24,452,"379670,37712084
8977072,20,MAUBEC,65,11-11-2018 18:39:18,109,"380380,783303404
8977073,21,PORT NEUF,155,11-11-2018 18:39:16,172,"377155,70064
```

EXEMPLE DE LECTURE D'UN FICHIER .CSV

```
import csv

with open('fichier.csv', newline='') as f:
  lecteur = csv.reader(f, delimiter=',', quotechar='"')
  for ligne in lecteur:
    print(ligne)
```

EXEMPLE D'ÉCRITURE D'UN FICHIER .CSV

```
import csv

with open('sortie.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(lignes)
```

TEA 1/2

- Un fichier .csv à charger
- Des activités, des questions amènent à écrire du code.
- Le résultat du code est entré dans un test avec retour immédiat lorsque que cela est possible (réponse numérique, une chaine de caractère, sortie de print)
- Nombre de réponses illimité
- Assistance auprès de votre enseignant de TP en présentiel ou via moodle.

TEA 2/2

- Un code solution est donné à la fin du TEA.
- Un super entrainement au CC2...
- Participation à la note de CC2.

POUR ALLER PLUS LOIN

- Listes, dictionnaires en compréhension
- *args*, *Kwargs
- zip, itertools