MARO 037 - Competitive Programming

Fabian Lecron - Arnaud Vandaele

Faculté Polytechnique - Services MIT et MARO

UMONS

Année académique 2024-2025

Séance 01 :

Python : introduction et lecture/écriture de fichiers

Séance 01 – Python : introduction et lecture/écriture de fichiers

Tables des matières

Plan

Python : Get Started

- Les variables sont typées mais dynamiquement et implicitement :

```
x = 42
y = 'FPMs'
```

- En une seule ligne :

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

- Conversion d'un type à l'autre :

```
x = 1 #int
y = 2.8 #float
z = 1j #complex
w = '42' #string

a = float(x) #convert from int to float
b = int(y) #convert from float to int:
c = complex(x) #convert from int to complex:
d = int(w)
```

- Importer des modules :

```
import random
print(random.randrange(1,10))
```

- Les commentaires se font à l'aide de #

Les sorties en console : les instructions `print` et `format`

- Imprimer un message à l'écran à l'aide de la méthode `print` :

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

Imprimer plus d'un objet :

```
print('Alice', 'Bob', 'Carol')
```

Imprimer plus d'un objet en spécifiant le séparateur :

```
print('Alice', 'Bob', 'Carol',sep='----')
```

- Utilisation de la méthode `format` :

```
print("La {}, bien + qu'une {}!".format('polytech', 'faculté'))
```

Un nombre entre accolades réfère à la position de l'argument souhaité :

```
print("La {1}, bien + qu'une {0}!".format('polytech', 'faculté'))
```

Des arguments nommés peuvent être utilisés :

```
print("La {par2}, bien + qu'une {par1}!".format(par1='faculté',par2='FPMs))
```

- Utilisation des `f-strings` (depuis Python 3.6) :

```
par1 = "faculté"
par2 = "FPMs"
print(f"La {par2}, bien + qu'une {par1}!")
```

Les booléens

- Un booléen ne peut prendre qu'une des deux valeurs suivantes :

  `True` `False`

- Les booléens sont considérés comme un type numérique. Il est donc possible d'appliquer des opérations arithmétiques à des booléens :
  - `True == 1` renvoie `True`
  - `False == 0` renvoie `True`
  - `True + (False / True)` renvoie `1.0`

- La fonction `bool()` permet d'évaluer n'importe quelle quantité, et renvoie `True` ou `False`.
  - `bool("Hello")`, `bool(15)`, `bool(True)` renvoient `True`
  - `bool(False)` renvoie `False`
  - `bool(None)`, `bool(0)`, `bool("")`, `bool(())`, `bool([])` renvoient `False`
  - `bool([False])`, `bool([0,0,0])` renvoient `True`
  - Presque toutes les valeurs contenant *quelque chose* sont évaluées à `True`.

Les opérateurs logiques et les conditions

Les comparaisons possibles sont `==` `!=` `>` `>=` `<` `<=`

- Opérateurs logiques `and` et `or`, exemple :

```
v = 7
v > 5 and v < 10
v < 5 or v > 100
```

Il est possible de chainer les comparateurs, exemple :

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

- Conditions à l'aide de `if`, `elif`, `else`, exemple :

```
a = 5
if a > 5:
    a = a + 1
elif a == 5:
    a = a + 1000
else:
    a = a - 1
```

Les sources d'erreurs sont souvent

- l'indentation
- le ":" après `if`, `elif`, `else`

all et any

- La méthode `all()` permet de vérifier que tous les éléments d'un *itérable* (liste, tuple, set, dictionnaire) sont des valeurs `True`.
  - `all([0,1,1])`, `all((0,1,1))` renvoient `False`
  - `all(["pomme",123,True])` renvoie `True`
- La méthode `any()` permet de vérifier qu'au moins un élément d'un *itérable* (liste, tuple, set, dictionnaire) possède une valeur `True`.
  - `any([0,1,1])`, `any((0,1,1))` renvoient `True`
  - `any(["pomme",123,True])` renvoie `True`
  - `any([0, 0, False])` renvoie `False`
- Utilisations éventuelles :
  - vérification de multiples conditions

```
maxiter, tol = 10**6, 1e-9
conditions = [k>maxiter,erreur<tol]
if all(conditions): ... #quand toutes les cond. sont vérifiées
if any(conditions): ... #quand au moins une cond. est vérifiée
```

  - en combinaison avec la **compréhension**

```
old_list = ["quelques", "Mots", "comme", "Exemples"]
check_majuscule = [x[0].isupper() for x in old_list]
print(all(check_majuscule)) #affiche False
print(any(check_majuscule)) #affiche True
```

L'instruction `range` : une particularité de Python

La fonction `range()` retourne une séquence de nombres, commençant à 0 par défaut et avec une incrémentation de 1 par défaut.

- `range(6)`

  Avec un seul argument n, la séquence 0, 1, 2, ... , $n-1$ est générée
- `range(2,6)`

  Avec ces deux arguments, la séquence 2, 3, 4, 5 est générée
- L'incrément peut être spécifié à l'aide d'un troisième argument :

  `range(0, 10, 3)`

  Avec ces trois arguments, la séquence 0, 3, 6, 9 est générée

  `range(-10, -100, -30)`

  Avec ces trois arguments, la séquence -10, -40, -70 est générée

Astuces :

- Combinaison de `range()` et `len()` :

```
phrase = ['La', 'faculté', 'polytechnique', 'de', 'Mons']
range(len(phrase))
```

- Combinaison de `range()` et `sum()` pour calculer la somme d'une séquence :

```
sum(range(99))
```

Un premier fichier .py et la fonction main

Il existe une technique spéciale pour définir un `main` dans un programme Python afin qu'il soit appelé uniquement lorsque le programme est exécuté directement (et pas lorsque le fichier est importé en tant que module).

Remarquez la différence dans l'exécution des deux scripts suivants :

- Dans un fichier `pythonmainfunction.py`, entrez les lignes suivantes :

```
print("Hello")
print("valeur de la variable __name__ : ", __name__)
if __name__ == '__main__':
    print("Je suis dans le main")
```

- Dans un fichier `pythonimport.py`, entrez les lignes suivantes :

```
import pythonmainfunction
print("Execution terminée")
```

`__main__` est le nom du scope dans lequel le code s'exécute en premier. Le nom d'un module (son `__name__`) vaut `'__main__'` lorsqu'il est exécuté en premier.
Un module peut donc découvrir s'il est exécuté dans le scope principal en vérifiant son `__name__`, ce qui permet typiquement d'exécuter du code lorsque le module est exécuté en tant que script principal mais pas lorsqu'il est importé.

La boucle `for...in`

- Boucler sur une suite d'entiers à l'aide de `range`

```
for i in range(2, 30, 3):
    print(i)
```

- Boucler à l'aide d'une liste

```
for i in [0.1, 0.2, 0.3]:
    print(i)
```

- Boucler à l'aide d'une liste !

```
phrase = ['La', 'faculté', 'polytechnique', 'de', 'Mons']
for mot in phrase:
    print(mot)
```

- Boucler sur une chainte de caractères

```
chaine = 'La faculté polytechnique de Mons'
for lettre in chaine:
    print(lettre)
```

- Double boucle

```
for x in range(1,11):
    for y in range(1,11):
        print(f"{x*y} = {x}*{y}")
```

Source d'erreurs : l'indentation, le ":", l'oubli de `in`

`for` avec autres possibilités

- Obtenir un itérateur qui parcourt une séquence à l'envers.

```
numeros = [1, 2, 3, 4]
for val in reversed(numeros):
    print(val, end=" ")
```

- Itérer sur une séquence tout en ayant un compteur automatique.

```
fruits = ["pomme", "banane", "cerise"]
for i, fruit in enumerate(fruits):
    print(i, fruit)

fruits = ["pomme", "banane", "cerise"]
for i, fruit in enumerate(fruits, start=1):
    print(i, fruit)
```

- Parcourir plusieurs itérables en parallèle.

```
noms = ["Alice", "Bob", "Carol"]
ages = [25, 30, 35]
for nom, age in zip(noms, ages):
    print(f"{nom} a {age} ans.")
```

La boucle `while`

- Utilisation standard :

```
i = 1
while i < 6:
    print(i)
    i += 1
```

(`while not` dans le cas où on veut travailler avec la négation)

- Utilisation de l'instruction `break` pour quitter la boucle
(il n'y a pas de boucle `do while` en python)

```
i = 1
while True:
    print(i)
    if i>20:
        break
    i += 1
```

- Imprimer un message lorsque la condition est fausse (idem pour `for`)

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i n'est plus inférieur à 6")
```

Les fonctions se définissent à l'aide de `def`

Le mot-clé `def` introduit une définition de fonction. Il doit être suivi du nom de la fonction et d'une série de paramètres (entre parenthèses)

- Définition d'une fonction et appel de celle-ci dans le `main` :

```
def fairesomme(a, b):
    return a + b
if __name__ == "__main__":
    x, y = 3, 2
    print(fairesomme(x,y))
```

- Utilisation de paramètres optionnels :

```
def fairesomme(a, b=7, c=1):
    return a + b + 2*c
if __name__ == "__main__":
    x, y, z = 3, 2, 20
    print(fairesomme(x))
    print(fairesomme(x,y))
    print(fairesomme(x,y,z))
    print(fairesomme(x,c=z))
```

Les fonctions, un outil puissant en Python

- Les fonctions peuvent retourner plusieurs valeurs :

```
def fairesomme(a, b=7, c=1):
    return a + b + c, a+2*b+3*c
if __name__ == "__main__":
    x, y, z = 3, 2, 20
    s1, s2 = fairesomme(x,y,z)
    print(s1,s2)
```

- Les fonctions peuvent être passées en paramètre à d'autres fonctions

```
def mettreaucarre(x):
    return x*x
def mettreaucube(x):
    return x*x*x
def calculsomme(fct,n):
    somme=0
    for i in range(n):
        somme+=fct(i)
    return somme
if __name__ == "__main__":
    print(calculsomme(mettreaucarre,10))
    print(calculsomme(mettreaucube,10))
```

Les objets : partout en Python

Cependant, les structures de données, c'est au prochain cours

Tout ce qui est manipulé en Python (listes, chaines de caractères, fichiers, etc) est objet issu d'une classe et possède des attributs et éventuellement des méthodes.

Rappel : un objet est une capsule contenant des *attributs* et des *méthodes*.

Rappel : une classe....

```
class Maison:
    nb_maisons = 0
    nb_max_maisons = 100
    def __init__(self,localisation,longitude,latitude,orientation):
        if Maison.nb_maisons >= Maison.nb_max_maisons:
            raise RuntimeError("Trop de maisons construites")
        self.loc    = localisation
        self.lng    = longitude
        self.lat    = latitude
        self.orient = orientation
        Maison.nb_maisons += 1
    def orienter_soleil(self):
        self.orient = "sud"
```

Les objets, suite

Dans la définition de classe au slide précédent, on y trouve :

- des *attributs de classe* : `nb_maisons` et `nb_max_maisons`
  (un attribut de classe est une variable définie au niveau d'une classe)
- le constructeur *__init__* qui modifie les *attributs d'instance* et les *attributs de classe*
  (un attribut d'instance est une variable définie au niveau d'un objet)
- une méthode `orienter_soleil`
  (une méthode s'écrit comme une fonction normale en Python mais est complétement définie dans le corps de la classe)
- le premier paramètre d'une méthode, `self`, est obligatoire, il représente l'objet sur lequel la méthode sera appliquée

Une fois que la classe est définie, on peut créer des objets :

```
if __name__ == "__main__":
    m1 = Maison("Mons",3.9523,50.4541,"nord")
    m2 = Maison("Ath",3.778,50.6294,"ouest")
    print(m1,m2)
    print(m1.loc,m2.loc)
    print(Maison.nb_maisons)
    print(m2.orient,m1.orient)
    m2.orienter_soleil()
    print(m2.orient,m1.orient)
```

Plan

Ouvrir des fichiers : l'ancienne et la nouvelle façon

La fonction `open()` est utilisée pour créer un *objet fichier*, que ce soit dans le but de lecture et/ ou d'écriture :

```
f = open('cheminducfichier.txt', 'r')
```

Le second paramètre est une chaîne de caractères qui spécifie le mode d'intéraction souhaité :

- `'r'` : mode lecture
- `'w'` : mode écriture
- et d'autres

Une fois que les opérations sont terminées, il est **nécessaire** de fermer le fichier :

```
f.close()
```

Peu importe le language, c'est toujours un bon exercice de s'entrainer à ouvrir et fermer un fichier. Cependant, trop souvent, le fichier ouvert n'est pas fermé, et cela conduit à des problèmes.
Introduit à partir de Python 2.5, la structure `with` permet une libération de la mémoire allouée par les opérations effectuées dans son corps, même si le code génère une exception :

```
with open('cheminducfichier.txt') as f:
    # do stuff with f
```

Quelles commandes utiliser pour lire (rapidement) les données ?

L'objet fichier retourné par `open()` possède trois méthodes principales :

`read()` `readline()` et `readlines()`

Avec l'instruction `read()` :
`read()` stocke l'entiereté du fichier comme une seule chaine de caractères :

```
with open('testlecture.txt') as f:
    texte=f.read()
    print(texte)
```

Ceci est utile pour les petits fichiers où vous souhaitez effectuer des manipulations de texte sur l'ensemble du fichier (remplacer un . par une , par exemple).
C'est votre problème si le fichier est deux fois plus volumineux que la mémoire de votre machine!
Afin de ne lire que les n premiers caractères, on peut utiliser `read(n)` :

```
with open('testlecture.txt') as f:
    texte=f.read(26)
    print(texte)
```

Attention, une ligne se termine souvent par '\n'

L'instruction `readline()`

`readline()` lit les lignes individuellement et les retourne comme chaînes de caractères :

```
with open('testlecture.txt') as f:
    ligne=f.readline()
    print(ligne)
    ligne=f.readline()
    print(ligne)
```

Il est alors possible d'itérer sur les lignes pour lire l'ensemble d'un fichier :

```
with open('testlecture.txt') as f:
    ligne = f.readline()
    nbligne = 0
    print(f"Ligne {nbligne}: {ligne}")
    while ligne:
        ligne = f.readline()
        nbligne += 1
        print(f"Ligne {nbligne}: {ligne}")
```

Remarques :

- Chaque ligne se termine par '\n'
- Une chaine de caractères vide est renvoyée lorsque la fin du fichier est rencontrée
- Avantageux lorsqu'on traite de très gros fichiers (tout n'est pas en mémoire)

L'instruction `readlines()`

`readlines()` stocke toutes les lignes restantes dans une **liste** de chaînes de caractères :

```
with open('testlecture.txt') as f:
    lignes = f.readlines()
    print(lignes)
```

La méthode `readlines()` permet donc de lire l'intégralité d'un fichier en une instruction seulement (attention à la taille du fichier).
Une boucle `for` peut alors être utilisée pour itérer sur les différentes lignes :

```
with open('testlecture.txt') as f:
    lignes = f.readlines()
    for ligne in lignes:
        print(ligne)
```

Une dernière technique.
Si on souhaite lire le fichier dès le début, un résultat équivalent peut être obtenu en itérant sur l'objet fichier lui-même :

```
with open('testlecture.txt') as f:
    for ligne in f:
        print(ligne)
```

Quelques astuces pour chaines de caractères

- L'accès à un élément se fait comme pour un tableau (et on commence en 0 !) :

```
ligne = 'la polytech de Mons\n'
result = ligne[3]
print(result)
```

- Dans le style Matlab, récupérer les éléments de 1 à 10 :

```
ligne = 'la polytech de Mons\n'
result = ligne[1:10]
print(result)
```

- Vérifier si une chaîne contient une sous-chaîne :

```
ligne = 'la polytech de Mons\n'
result = 'de M' in ligne
print(result)
```

- Concaténer des chaînes de caractère à l'aide de + :

```
ligne1 = 'la polytech'
ligne2 = ' de Mons depuis '
result = ligne1+ligne2+str(1837)
print(result)
```

Méthodes pour chaines de caractères

- La méthode `split` crée une liste en utilisant un séparateur :

```
ligne = 'la polytech de Mons\n'
result = ligne.split(' ')
print(result)
```

Le séparateur peut être changé, et en deuxième paramètre, le nombre de splits :

```
ligne = 'la,polytech,de,Mons\n'
result = ligne.split(',',1)
print(result)
```

- La méthode `replace` permet de remplacer un caractère par un autre :

```
ligne = 'la polytech de Mons\n'
result = ligne.replace('o','A')
print(result)
ligne = 'la polytech de Mons\n'
result = ligne.replace('polytech','polytechnique')
print(result)
```

- La méthode `join` joint tous les éléments d'un itérable :

```
liste = ['la','polytech','de','Mons']
separateur = 'SEP'
print(separateur.join(liste))
```

- et beaucoup d'autres : `strip`, `rstrip`, `partition`, `splitlines`, `count`, `find`, `index`, `istitle`, `endswith`, ...

Ecrire dans un fichier

La procédure est assez simple :

- utilisation du `with open` en mode `'w'`
- utilisation de la méthode `write` de l'objet fichier
- utilisation des opérations sur les chaînes de caractères

Exemple :

```
nombres = [-3, 0, 4, 5]
with open('fichieroutput.txt', 'w') as fout:
    fout.write(f"Il y a {len(nombres)} nombres\n")
    fout.write('Ces nombres sont : ')
    for nombre in nombres:
        fout.write(f"{nombre} ")
    fout.write("\n")
    fout.write(f"Leur somme vaut : {sum(nombres)}\n")
    fout.write(f"Le maximum est : {max(nombres)}\n")
    fout.write(f"Le minimum est : {min(nombres)}\n")
```

Plan