Une brève introduction à Python

1 Présentation

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles. D'autre part, Python présente l'avantage d'une syntaxe très simple, combinée à des types de données évolués (listes, dictionnaires...)

2 Exécuter un programme Python

2.1 Mode interactif

Le mode interactif permet de « dialoguer » directement avec Python depuis le clavier. L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » *Linux*, ou bien dans une fenêtre *DOS* sous *Windows* : il suffit d'y taper la commande **"python"**, en se plaçant préalablement dans le répertoire contenant Python, voir Figure 1).

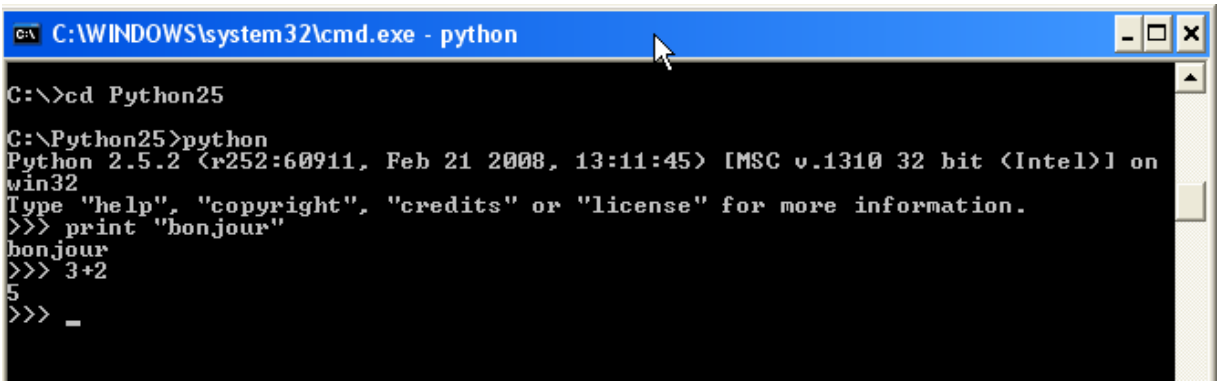

Figure 1. Mode interactif dans une fenêtre dos

Le plus pratique est d'utiliser un environnement de travail spécialisé, comme IDLE, qui permet aussi bien d'utiliser le mode interactif (cf Figure 2) que d'enregistrer et faire tourner des programmes (cf Figure 3)

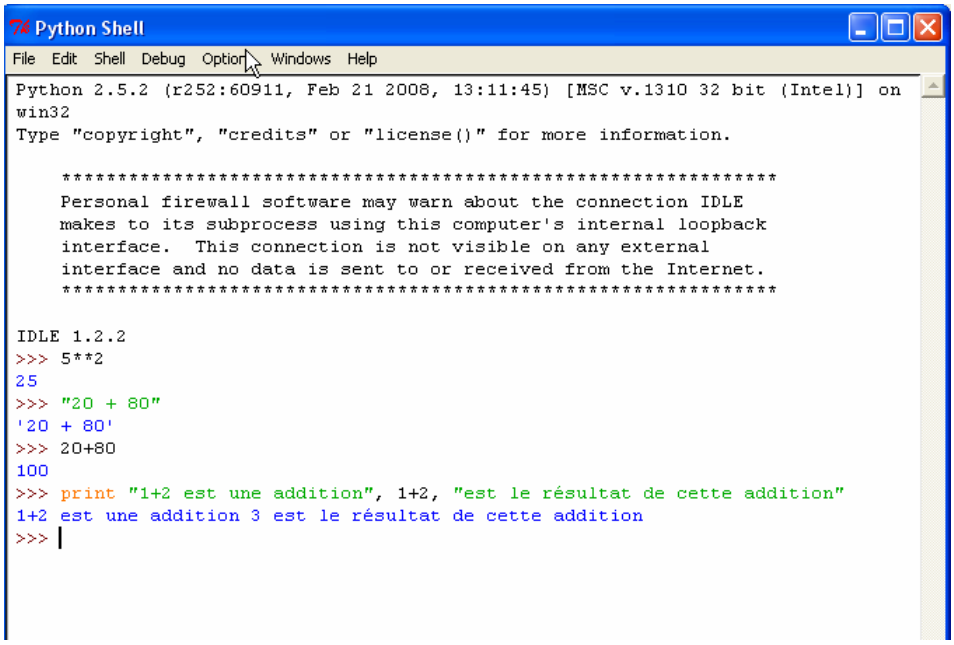

Figure 2. Interpréteur Python dans IDLE

2.2 En exécutant un fichier Python

Vous pouvez aussi enregistrer une série d'instructions dans un fichier texte (il est conseillé de l'enregistrer avec l'extension .py), puis le faire tourner. La figure 4 montre un exemple de script Python. On peut l'ouvrir dans n'importe quel éditeur de texte

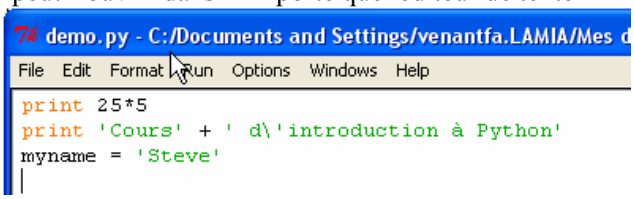

Figure 3. Exemple de script Python

La figure 5 montre ce qui se passe quand on exécute ce programme (clic sur Run ou touche F5) : les instructions sont exécutées les unes après les autres. L'instruction print provoque un affichage, l'instruction myname= 'steve' crée une variable de type string. Cette variable contient la chaîne de caractère « Steve » comme on peut le voir en interrogeant Python en mode interactif

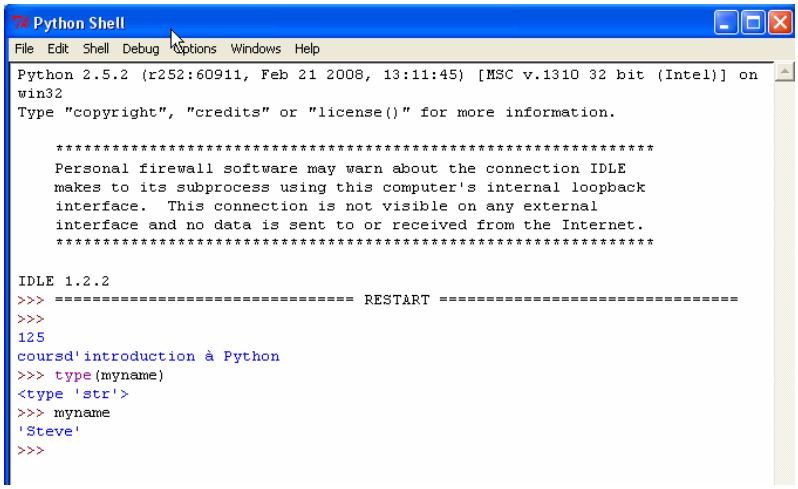

Figure 4. Exécution du script précédent

3 Données et Variables, types, chaînes de caractères, structures de contrôles

Je vous renvoie au TP1 qui détaille toutes ces notions.

4 Un objet, c'est quoi ?

L'idée de base, c'est qu'on veut définir un type de variable, par exemple « string », et en même temps toutes les opérations qu'on peut réaliser sur ce type d'objet, par exemple, les mettre en majuscule, les concaténer, les découper selon les espaces….
En programmation standard, on définirait des fonctions qui s'appliquerait à des variables de types string, par exemple mettre_en_minuscule(ch), où ch est un string.
En programmation objet, on va définir ces fonctions comme étant des propriétés communes à tous les objets string. On les appelle des méthodes. Quand on définit une classe d'objet, on définit en même temps toutes les méthodes qui vont avec.
La différence principale, à votre niveau, entre une fonction et une méthode, c'est la façon de l'utiliser.
Dans l'exemple ci-dessous, len() est une **fonction**, on l'applique à la variable ch, que l'on passe en **paramètre :**

```
>>> ch="Traitement automatique des langues"
>>> len(ch)
```

Alors que dans l'exemple suivant, split() est une méthode, que l'on appelle à partir d'une variable, et qui s'appliquera à la variable à partir de laquelle on l'a appelée (ici ch)

```
>>> ch.split()
['Traitement', 'automatique', 'des', 'langues']
```

Bien sûr ce qui complique tout, c'est que certaines méthodes prennent aussi des paramètres. C'est le cas de la fonction count() qui compte le nombre d'occurrences d'une lettre donnée dans une chaîne de caractères. Dans l'exemple ci-dessous, on appelle la méthode count, sur l'objet ch, pour compter le nombre de e dans cette chaîne de caractères :

```
>>> ch.count('e')
5
```

5 Structures de données

5.1 Les listes

Les listes ont été introduites dans le TP1. Rappelons qu'on peut définir une liste comme ***une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.***
Exemple :
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print jour
Les éléments qui constituent une liste peuvent être de types variés. Un élément d'une liste peut lui-même être une liste.
Exemple :

```
>>> jour = ['lundi', 'mardi', 'mercredi', [1800, 20, 357], 'jeudi', 'vendredi']
>>> jour
['lundi', 'mardi', 'mercredi', [1800, 20, 357], 'jeudi', 'vendredi']
>>>
```

On peut modifier une liste

```
>>> jour = ['lundi', 'mardi', 'mercredi', [1800, 20, 357], 'jeudi', 'vendredi']

>>> jour[1]='mardi'
>>> jour[5]=jour[3][2]+10
>>> jour
['lundi', 'mardi', 'mercredi', [1800, 20, 357], 'jeudi', 367]
>>>
```

Quelques méthodes des objets list:

`append`(*x*)
Ajoute un élément à la fin de la liste
Exemple d'utilisation:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.append(344)
>>> a
[66.25, 333, 333, 1, 1234.5, 344]
```

`extend`(*L*)
Ajoute une liste d'éléments à la fin de liste
Exemple d'utilisation :

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.extend([3,56,89,7])
>>> a
[66.25, 333, 333, 1, 1234.5, 3, 56, 89, 7]
```

`insert`(*i, x*)
Insère un élément en positon i (ne jamais oublier que le premier indice est 0). Le premier argument indique la position, le deuxième l'élément à insérer.
`a.insert(0, x)` insère l'élément en début de liste
`a.insert(len(a), x)` est équivalent à a.append(x)
Exemple d'utilisation:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.insert(2,543)
>>> a
[66.25, 333, 543, 333, 1, 1234.5]
```

remove(*x*)

Supprime le premier élément de la liste qui vaut x. Renvoie une erreur si aucun élément de la liste ne vaut x

Exemple d'utilisation

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.remove(333)
>>> a
[66.25, 333, 1, 1234.5]
>>> a.remove(10)

Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    a.remove(10)
ValueError: list.remove(x): x not in list
```

pop([*i*])

Supprime l'élément en position i et le renvoie (ce qui permet éventuellement de le récupérer et de le stocker dans une variable). Si aucun indice n'est spécifié, a.pop() supprime et renvoie le dernier élément de la liste. Les crochets autour du i dans la définition indiquent que le paramètre est optionnel. Vous rencontrerez fréquemment cette notation dans la documentation Python *Python Library Reference*.)

Exemple d'utilisation :

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> b=a.pop(2)
>>> b
333
>>> c=a.pop()
>>> c
1234.5
>>>
```

index(*x*)

Renvoie l'index du premier élément qui vaut x. Renvoie une erreur s'il n'y en a pas.

Exemple d'utilisation

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.index(1)
3
>>> a.index(18)

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    a.index(18)
ValueError: list.index(x): x not in list
>>>
```

count(*x*)

Compte le nombre de fois où x apparaît dans la liste.

Exemple d'utilisation :

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.count(333)
2
>>> a.count(5)
0
>>>
```

sort()

Ordonne la liste (ordre alphabétique ou croissant)

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.sort()
>>> a
[1, 66.25, 333, 333, 1234.5]
>>>
```

```
>>> a=["bonjour", "zoo", 4, "Hector", 67]
>>> a.sort()
>>> a
[4, 67, 'Hector', 'bonjour', 'zoo']
>>>
```

reverse()

Ordonne la liste en ordre décroissant

5.2 Tuples et séquences

Les listes et les chaînes de caractères ont des propriétes communes, comme le fait qu'on puisse les découper, ou que leurs éléments soient repérés par des indices. Elles sont des exemples de **séquences** de données. Il en existe d'autres comme les tuples.
Un tuple consiste en un certains nombre de valeurs séparées par des virgules, par exemple

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Quand on veut imbriquer des tuples, on les délimite par des parenthèses, qui sont optionnelles. Dans l'exemple ci-dessus, u est un tuple formé de deux sous tuples

Le tuple vide est construit par une paire de parenthèses vide. Un tuple qui ne contient qu'un seul élement (un singleton) est construit par cette valeur suivie d'une virgule. Par example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- Notez la virgule de fin
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

5.3 Ensembles

Python comprend aussi un autre type de données les ensembles (sets). Un set est une collection non ordonnés (donc pas d'index, pas de position pour repérer les éléments) sans doublon. On les utilise principalement quand on veut tester si un élément appartient ou non à une collection, ou pour éliminer des doublons. Ils permettent aussi de réaliser des opérations mathématiques comme l'union, l'intersection, la différence….

Exemple:

```
# on part d'une liste
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
# à partir de cette liste on crée un ensemble, ça permet d'éliminer les doublons
>>> fruit = set(basket)
set(['orange', 'pear', 'apple', 'banana'])
# Et maintenant on peut tester si un élément appartient ou non à la collection
>>> 'orange' in fruit
True
>>> 'crabgrass' in fruit
False

>>> # Exemple d'utilisation d'un ensemble pour comparer les lettres dans des mots
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # ensemble des lettres contenues dans a
                                        #(un seul exemplaire de chaque lettre)
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # lettres dans a mais pas dans b
set(['r', 'd', 'b'])
>>> a | b                              # lettres soit dans a soit dans b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # lettres à la fois dans a et b
set(['a', 'c'])
>>> a ^ b                              # lettres qui sont soit dans a soit dans b
                                        #mais pas dans les deux
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.4 Les dictionnaires

Un autre type de données très utiles en Python sont les dictionnaires. Contrairement aux données séquences (comme les listes ou les tuples) qui sont indexées par des nombres, les dictionnaires sont indexés par des clefs, qui peuvent être aussi bien des strings, des nombres ou des tuples (s'ils ne contiennent que des nombres et des strings)

Un dictionnaire est une liste non ordonnée de paire valeur :clé. Les clés doivent être uniques (au sein d'un même dictionnaire une clé donnée ne peut apparaître qu'une seule fois).

Pour créer un dictionnaire vide, on utiliser une paire d'accolades vide

```
>>> tel
{}
```

Pour ajouter une paire valeur : clé :

```
>>> tel={'jean':'06 26 45 65 34'}
>>> tel
{'jean': '06 26 45 65 34'}
```

```
>>> tel
{}
```

On peut entrer plusieurs paires en même temps en les séparant par une virgule

```
>>> tel={'jean':'06 26 45 65 34','Guy':'06 43 65 76 88'}
>>> tel
{'jean': '06 26 45 65 34', 'Guy': '06 43 65 76 88'}
>>>
```

On peut ensuite interroger le dictionnaire, et demander la valeur associée à un clé donnée :

```
>>> tel['Guy']
'06 43 65 76 88'
```

Si vous entrez une nouvelle valeur avec une clé déjà utilisée, l'ancienne valeur est effacée et remplacée par la nouvelle.

On peut aussi effacer une paire clé: valeur avec la fonction del()

La méthode `keys()` retourne la liste de toutes les clés utilisées dans un dictionnaire, dans un ordre arbitraire.

Exemple d'utilisation:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
>>> 'guido' in tel
True
```

On peut aussi construire un dictionnaire directement à partir d'une liste de paires clé-valeur stockée dans un tuples, grâce à la fonction dict()

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

6 Module

6.1 Pourquoi des modules

Si vous quittez l'interpréteur Python et le rouvrez un peu plus tard, toutes les variables et fonctions que vous aviez définies sont perdues. C'est pourquoi quand on a besoin de les réutiliser, on les sauve dans un script (fichier texte avec extension.py) que l'on peut ensuite soit faire tourner (run), soit importer. Si l'on veut définir une fonction réutilisable dans différents programmes, on peut aussi l'enregistrer dans un fichier et l'importer ensuite dans chaque programme où on veut s'en servir.
Tout fichier contenant des instructions Python s'appelle un module. Toutes les définitions d'un module peuvent être importées dans un autre module, ou dans le module principal (qui est l'espace de travail courant comme dans IDLE par exemple)

Python dispose de toute une bibliothèque de modules que l'on peut importer (pour peu qu'ils soient installés sur la machine sur laquelle on travaille). Dans le cadre de ce cours, nous utiliserons les modules re, pour les expressions régulières, et nltk, pour des outils spécifiques au traitement automatique des langues.
On peut importer un module de différentes façons

6.2 Import

La première façon est l'instruction import suivi du nom du module. Comme par exemple,
Import nltk
Une fois que vous avez importé un module de cette façon, vous ne pouvez pas accéder directement aux fonctions ou variables qu'il définit. Quand vous souhaitez utiliser des fonctions définies dans un module importé, vous devez inclure le nom du module.
Par exemple pour accéder à l'objet corpus inclus dans nltk, vous ne pouvez pas simplement taper
`corpus`
Vous devez préciser le nom du module,
`nltk.corpus`
Et ensuite pour accéder à un corpus précis, par exemple le corpus gutenberg
`Nltk.corpus.gutenberg`

6.3 `from` *`module`* `import`

La deuxième manière façon d'importer un module c'est d'utiliser les mots clé from et import
`from nltk import corpus`
Cela ressemble à la syntaxe `import` *`module`* que vous connaissez, mais avec une différence importante : les attributs et les méthodes du module importé sont importés directement dans l'espace de noms local, ils sont donc disponibles directement sans devoir les qualifier avec le nom du module. Vous pouvez importer des éléments précis ou utiliser `from` *`module`* `import *` pour tout importer.
En revanche si vous n'importez que les corpus par l'instruction from nltk import corpus, vous ne pourrez pas utiliser la fonction tokenize qui n'a pas été importée

7 Lire et Ecrire dans des fichiers

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un « ***objet-fichier*** » que l'on crée à l'aide de la fonction interne **open()**. Il faut donc bien comprendre que ce que cette fonction renvoie est un objet (de type file) et qu'ensuite on appelle des méthodes de la classe file sur cet objet.

open() renvoie donc un objet fiel. On l'utilise couramment avec deux arguments "open(*filename*, *mode*)".

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Le premier argument est une chaîne de caractère indiquant le nom du fichier (et le chemin pour y accéder s'il n'est pas enregistré dans le même dossier que votre script Python). Le deuxième argument est aussi une chaîne de caractère décrivant de quelle façon le fichier va être utilisé :

- 'r' indique que le fichier sera seulement lu ('read')
- 'w' indique qu'on va seulement écrire dans le fichier ('write'). Si on ouvre en écriture un fichier qui existe déjà sont contenu sera effacé et remplacé par ce qu'on va écrire . Si le fichier n'existe pas encore Python le crée.
- 'a' indique que l'on ouvre le fichier en écriture mais sans écraser le contenu existant déjà. Ce qu'on va écrire sera ajouté à la fin du fichier.
- 'r+' indique que l'on ouvre le fichier à la fois en ouverture et en écriture.

Si l'on indique rien, le mode d'ouverture par défaut sera 'r'
Méthode des objets file:
Pour le reste de ce paragraphe, on supposera qu'un objet de type file, appelé f, a été crée par la fonction open()

Méthodes de lecture:
`f.read(size)`
qui lit une certaine quantité de données et les renvoie sous forme d'une chaîne de caractères. L'argument. *size* est un optionnel. C'est un nombre qui indique le nombre maximal de bytes doivent être lus. Si on ne met rien, le fichier est lu en entier, tant pis s'il est deux fois plus gros que la mémoire de votre machine. Quand la fin du fichier a été atteinte, read() renvoie une chaîne vide (`''`)

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` lit et retourne un seule ligne du fichier. . Quand la fin du fichier a été atteinte, readline() renvoie une chaîne vide (`''`)

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

`f.readlines()()` retourne une liste contenant toutes les lignes du fichiers.`liste con` returns a list containing all the lines of data in the file.

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

Une autre façon de lire un fichier ligna par ligen est de faire une boucle qui parcourt l'objet file ::

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

The alternative approach is simpler but does not provide as fine-grained control. Since the two approaches manage line buffering differently, they should not be mixed.

`f.write(`*`string`*`)` writes the contents of *string* to the file, returning `None`.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use "`f.seek(`*`offset`*`, `*`from_what`*`)`". The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

Quand vous en avez terminé avec un fichier, appeler `f.close()` pour le fermer et libérer toutes les ressources système utilisées par le fichier ouvert..