UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en Python

Cours 6. Exceptions et fichiers texte

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- En cas d'erreur dans un programme, Python lève une exception.

SHELL

```
>>> L = [6, 4, 7, 2, 1, 8]
>>> L[6]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError:  list index out of range
>>> L.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  5 is not in list
>>> 2/0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError:  division by zero
```

- Il faut lire en détail les messages erreurs !
- La cause de l'erreur y est souvent clairement indiquée.

SCRIPT

```
1  def f(x):
2      print("Début du calcul")
3      a=g(x)
4      print("Fin du calcul")
5      return a+1
6
7  def g(x):
8      return 3%x
```

En cas d'erreur rencontrée :

- le programme s'interrompt immédiatement
- et affiche alors la trace d'exécution.

SHELL

```
>>> f(0) # f définie dans le fichier toto.py
Début du calcul
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "toto.py", line 3, in f
    a=g(x)
      ^^^^
  File "toto.py", line 8, in g
    return 3%x
           ~^~
ZeroDivisionError: integer modulo by zero
```

- Ligne 3 du fichier `toto.py` : `f` a rencontré une erreur : `a=g(x)`
- Ligne 8 du fichier `toto.py` : `g` a rencontré une erreur : `return 3%x`
- L'erreur est une division par zéro.

Éviter les exceptions

- Les exceptions peuvent être dues :
  - à une erreur de programmation.
  - à un utilisateur qui rentre de mauvaises valeurs.

SCRIPT
```
def record(chrono):
    if chrono < 9.58:
        return True
    else:
        return False
```

SHELL
```
>>> record('9s')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "sprint.py", line 2, in record
    if chrono < 9.58:
       ^^^^^^^^^^^^^

TypeError:  '<' not supported between
instances of 'str' and 'float'
```

- On peut vérifier le type :

SCRIPT
```
def record2(chrono):
    if type(chrono) == float:
        return chrono < 9.58
    else:
        return False
```

SHELL
```
>>> record2(9.0)
True
>>> record2(9)
False
>>> type(9) == float
False
```

- Mais il est facile d'oublier des cas !

Rattraper les exceptions

- Pour éviter que notre programme se termine,
  - On peut éviter les exceptions : cela peut rendre le code complexe et lourd
  - On peut les rattraper pour les traiter proprement.

SCRIPT

```
try:
    instruction # Bloc try
    instruction # Calcul qui peut
    instruction # lever des exceptions
except:
    instruction # Bloc except : Exécute
    instruction # seulement si une exception
    instruction # a été levée dans le Bloc 1
instruction # Suite du programme
```

- Dans un bloc `try`, on lance un code qui peut lever une exception.
- Si aucune exception n'est levée, on passe à la suite du programme.
- Si une exception est levée dans le bloc `try` :
  - on interrompt le bloc `try` à partir de l'exception
  - on exécute le bloc `except`
  - on passe alors à la suite du programme

Un premier exemple : la fonction `record`

SCRIPT

```
def record3(chrono):
    try:
        print('On essaie le calcul')
        résultat =  (chrono < 9.58)
        print('Tout c'est bien passé :)')
    except:
        print('Au secours une erreur !!!')
        résultat = False
    return résultat
```

- Si le test lève une exception, on considère que le record n'est pas battu.

SHELL

```
>>> record3(9.6)
On essaie le calcul
Tout c'est bien passé :)
False
>>> record3(9)
On essaie le calcul
Tout c'est bien passé :)
True
>>> record3('huit secondes et des poussières')
On essaie le calcul
Au secours une erreur !!!
False
```

Exemple : la fonction `index`

- On cherche à écrire une fonction `recherche(x,S)` :
  - qui renvoie l'indice `x` dans la séquence `S`.
  - qui renvoie `-1` si l'élèment n'est pas dans `S`.
  - cf. cours 3 page 21 et cours 5 page 11.

- Il existe une méthode `index`, mais elle lève une exception en cas d'échec.

SHELL

```
>>> ['a','b','c','b','c'].index('c')
2
>>> ['a','b','c','b','c'].index('d')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  'd' is not in list
```

- On va transformer cette « erreur » en `-1` en capturant l'exception.

Exemple : la fonction `recherche`

- On va utiliser la méthode `index`...
- ... et lorsqu'elle lance une exception...
  - on la rattrape au vol !
  - et on renvoie `-1`

SCRIPT

```
def recherche(x, L):
    try:
        res = L.index(x)
        # Pas d'erreur ? On continue.
        return res
    except:
        # Une erreur ? On exécute ce bloc.
        return -1
```

- Et cela fonctionne avec les chaînes, les listes et les tuples.

SHELL

```
>>> L=['a','b','c','b','c'] ; T=(1,2,3)
>>> recherche('c',L)
2
>>> recherche(4,T)
-1
```

Exemple : Calcul de la moyenne

- On veut calculer la moyenne d'une liste de note `moyenne(L)`.

SCRIPT
```
def moyenne(L):
    return  sum(L) / len(L) #  vous devez savoir écrire sum(…) !
```

- Si la liste est vide, on renvoie `"ABI"` *(absence injustifiée)*.
  - on va rattraper l'erreur de la division par zéro et seulement celle-ci

SCRIPT
```
def moyenne(L):
    try:
        return  sum(L) / len(L)
    except ZeroDivisionError:
        return "ABI"
```

SHELL
```
>>> moyenne([19,15,17])
17.0
>>> moyenne([])
'ABI'
>>> moyenne(12)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 3, in moyenne
TypeError:  'int' object is not iterable
```

Exemple : reconnaître un nombre dans une chaîne

- La méthode `isdigit` ne fonctionne qu'avec les entiers.
- Astuce : on essaie de convertir une chaîne avec `float`
  - si ça marche, c'est que c'était possible…
  - sinon, c'est visiblement autre chose qu'un nombre flottant !

SHELL

```
>>> '1987'.isdigit()
True
>>> 'MCMLXXXVII'.isdigit()
False
>>> '6.55957'.isdigit()
False
```

SHELL

```
>>> float('6.55957')
6.55957
>>> float('MCMLXXXVII')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: could not convert string
to float: 'MCMLXXXVII'
```

SCRIPT

```
def est_nombre(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

SHELL

```
>>> est_nombre('1987')
True
>>> est_nombre('6.55957')
True
>>> est_nombre('MCMLXXXVII')
False
```

Sommaire

Lever une exception : `assert`

- S'il y a trop peu d'erreurs dans votre code, vous pouvez en ajouter !
  - C'est une bonne pratique !
  - On ne cache pas d'erreurs mais on en informe l'utilisateur
  - Code plus court et lisible (on ne se préoccupe pas d'arguments non conformes)
- Conseil pour incarner la Zénitude !
  - L'explicite est mieux que l'implicite [...]
  - Les erreurs ne doivent jamais être passées sous silence, sauf de manière explicite
- `assert` : vérifie une propriété et lève une erreur en cas de non-respect.

SCRIPT

```
# assert
# On impose L non vide
def moyenne(L):
    assert L != []
    return sum(L)/len(L)
```

SHELL

```
>>> moyenne([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 2, in moyenne
AssertionError
```

- Rapide à mettre en place
- Permet d'être explicite sur ce qu'attend le programme.
- Message d'erreur un peu vague.

SHELL

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

SCRIPT

```
def f(L):
    """ f(L) renvoie un couple (n,b)
    n est la taille de L,  b vaut True si n est pair """
    pair=True
    for i in range(len(L)):
        pair=not(pair)
    return (i+1,pair)

assert f([2,3,4])   == (3,False) # 3 éléments : impair
assert f([2,3,4,6]) == (4,True)  # 4 éléments : pair
assert f([])        == (0,True)  # 0 élément  : pair
```

- si un test est valide, il ne se passe rien.
- sinon il y a une exception (ici l'erreur ne vient pas du `assert`)

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "tp6.py", line 9, in <module>
    assert f([]) == (0,True) # 0 élément : pair
           ^^^^^

  File "tp6.py", line 5, in f
    return (i+1,pair)
            ^

UnboundLocalError:  cannot access local variable 'i' where it is not
associated with a value
```

- Méthode 1 : `assert`
  - Très pratique pour repérer les bogues qu'on ne devrait jamais rencontrer.
  - Permet de détecter les erreurs du développeur : utile pour les tests.
  - Mais messages peu clairs en production.
  - (même si on peut ajouter un message : `assert test, "Message"`)
- Méthode 2 : `raise`
  - Permet de préciser le type d'erreur
  - par exemple : `ValueError`, `IndexError`, `ZeroDivisionError`
  - Permet d'ajouter un message clair et explicatif.

SCRIPT

```
def moyenne(L):
    if L == []:
        raise ValueError('Liste vide : non mais allo quoi !')
    else:
        return sum(L) / len(L)
```

SHELL
```
>>> raise AssertionError('Il faut lire les consignes !!!')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError:  Il faut lire les consignes !!!
```

SHELL
```
>>>     raise ZeroDivisionError("Moyenne d'une liste vide impossible")
  File "<console>", line 1
    raise ZeroDivisionError("Moyenne d'une liste vide impossible")
IndentationError:  unexpected indent
```

SHELL
```
>>> raise ValueError('L argument doit être pair' + 1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  can only concatenate str (not "int") to str
```

SHELL
```
>>> raise IndexError("Je croyais la liste plus grande" + srt(1))
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'srt' is not defined
```

- Vous pouvez utiliser `ValueError` et `TypeError`. Évitez les autres.
- On peut créer ses propres exceptions… mais nous ne verrons pas comment.

SCRIPT

```
def f():
    # déclenche une exception mais ne la rattrape pas
    2/0
    print('Ce message ne sera pas affiché.')

def g():
    f() # déclenche une exception
    print('Ce message non plus.')

def h():
    try:
        g()
        print('Ce message ne sera toujours pas affiché')
    except:
        print('Attrapée !')

h()
```

- Dans cet exemple l'exception interrompt la fonction `f()`,
- mais aussi la fonction `g()` qui appelle la fonction `f()`
- on passe directement de la ligne `2/0` à la ligne `print('Attrapée !')`

Sommaire

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

SHELL
```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```

- Version plus lisible et plus aisément modifiable sur le long terme.

SHELL
```
>>> f'{n} = {n//d} × {d} + {n%d}'
'17 = 3 × 5 + 2'
```

L'ajout d'un `f` devant la chaîne permet d'utiliser des variables ou des expressions entre accolades. On parle de `f-string`.

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

- On peut même utiliser des pourcentages :

SHELL
```
>>> f'{0.1676:.2%}    {1/2:.0%}    {4/3:.4%}'
'16.76%    50%    133.3333%'
```

- Le formatage permet aussi d'aligner des chaînes de tailles différentes.

SCRIPT

```
texte = "Où suis-je ?"
print('#'*50) # Chaîne de 50 caractères
print(f"{texte:<50} fin") # aligné à gauche sur 50 caractères
print(f"{texte:>50} fin") # aligné à droite sur 50 caractères
print(f"{texte:^50} fin") # centré sur 50 caractères
print("")
print(f"{texte:*<50} fin")
print(f"{texte:*>50} fin") # On peut même préciser
print(f"{texte:*^50} fin") # les caractères à ajouter
```

SHELL

```
##################################################
Où suis-je ?                                    fin
                                     Où suis-je ? fin
                    Où suis-je ?                 fin

Où suis-je ?********************************** fin
**********************************Où suis-je ? fin
********************Où suis-je ?******************** fin
```

- Application : écrire sur 3 chiffres (à faire chez vous) :

SHELL

```
1-11-21-31-41-51-61-71-81-91-101-avant
001-011-021-031-041-051-061-071-081-091-101-après
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>> '-'.join(['partiel','21','03','2024'])
'partiel-21-03-2024'
>>> ''.join(['partiel','21','03','2024'])
'partiel21032024'
```

Sommaire

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

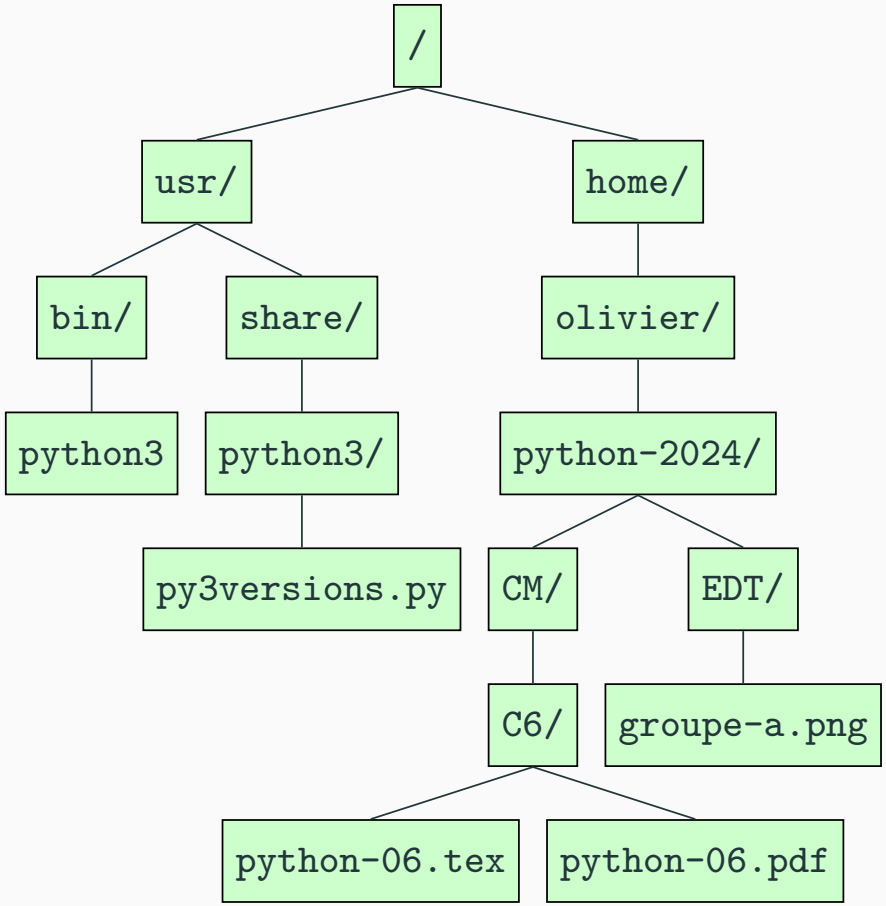
- Le fichier désignait le recueil des fiches (ou le meuble les contenant).
- Un fichier informatique est un ensemble structuré d'information
  - tableau : feuille de calcul
  - suite d'instructions : programme
  - texte (txt, html, odt).
  - données quelconques (image, vidéo, etc.)
- Les systèmes d'exploitation permettent de travailler sur des fichiers
  - stockage de grandes quantités d'informations
  - recherche, classement, modification
  - fichiers stockés sur le disque dur (ou clé USB, etc)
- Système d'exploitation : *Operating System* (OS).
  - exemples : GNU/Linux, Android, Windows, macOS, etc

- Le père de tous les systèmes d'exploitation moderne est UNIX (≈ 1975).
- Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.
  - Pour les serveurs :
    - La famille BSD : FreeBSD, NetBSD, OpenBSD (sécurité), dragonflyBSD
    - La famille propriétaire : AIX (IBM), Solaris (Oracle), HP-UX (HP),...
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
  - Pour les particuliers :
    - macOS (anciennement Mac OS X)
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
    - Les autres Linux (android)
- L'autre grande famille est composée des Windows...

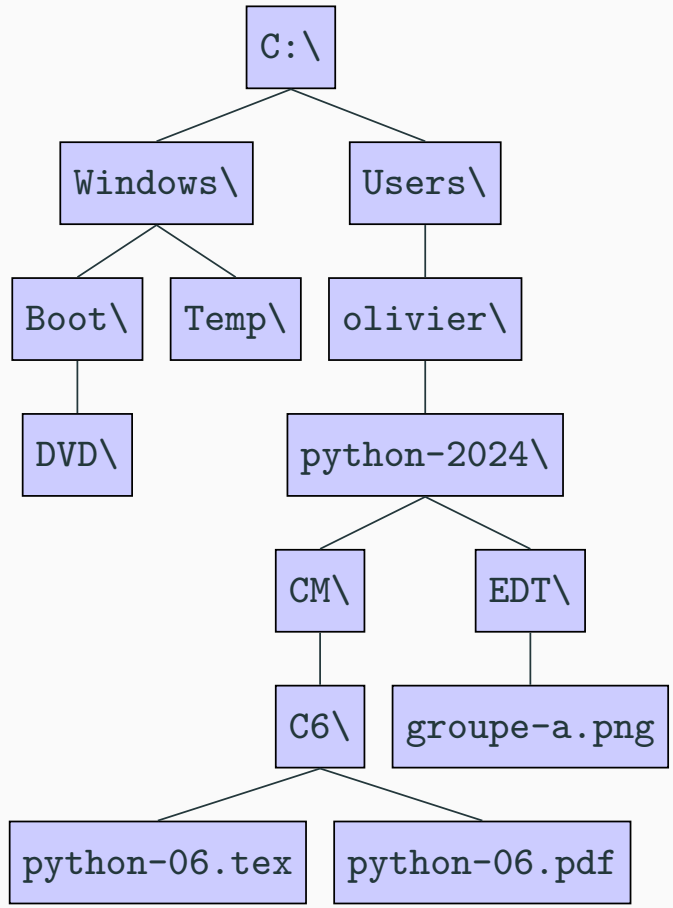
- Les fichiers sont regroupés dans une arborescence,
  - avec une ou plusieurs racines
  - les nœuds sont les répertoires (ou dossiers)
  - les feuilles sont les fichiers
  - un répertoire peut contenir des sous-répertoires et des fichiers.
- Il y a essentiellement deux systèmes d'arborescence :
  - Unix (GNU/Linux, macOS, BSD) avec des petites variantes
  - Windows
- Sur Unix, une seule racine (nommée `/`).
- Sur Windows, plusieurs racines (nommées `C:\`, `D:\`, etc).
  - En général, `C:\` représente le disque principal.

- Un répertoire est relié à la racine par un chemin

Arborescence Unix



Arborescence Windows



- Chemin :
  - Unix `/home/olivier/python-2024/EDT/groupe-a.png`
  - Windows `C:\Users\olivier\python-2024\EDT\groupe-a.png`

Chemins relatifs et absolus

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs
- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/python-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\python-2024\EDT\groupe-a.png`
- Chemins relatifs : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - `/home/olivier/python-2024/EDT/groupe-a.png`
    - `/home/olivier/python-2024/sauvegarde/EDT/groupe-a.png`
  - Cela dépend si je suis dans `sauvegarde` ou `python-2024`
- Le répertoire parent (du dessus) se note `..`
  - Si je suis dans `/home/olivier`, `..` est le chemin vers `/home`
  - La racine `/` est le seul répertoire sans parent. :'(

- Pour utiliser des chemins relatifs, il faut connaître le répertoire courant.
  - C'est le répertoire à partir duquel on lance le programme
- Le module os permet de manipuler le système de fichiers en Python.

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/python-2024/CM/C6'
```

- Pour changer de répertoire courant et se déplacer : `os.chdir(...)`

SHELL
```
>>> os.chdir('../../EDT/') # chemin relatif
>>> os.getcwd()
'/home/olivier/python-2024/EDT'
>>> os.chdir('/home/olivier/python-2024/Docs/')# chemin absolu
>>> os.getcwd()
'/home/olivier/python-2024/Docs'
>>> os.chdir('tkinter.pdf') # ne peut pas se déplacer dans un fichier
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NotADirectoryError: [Errno 20] Not a directory:
'tkinter.pdf'
```

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/python-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\python-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
'posix'
```

SHELL
```
>>> os.name #Sous Windows
'nt'
```

- On peut construire un chemin de manière portable.

SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Linux
'../QCM/q1.xml'
```

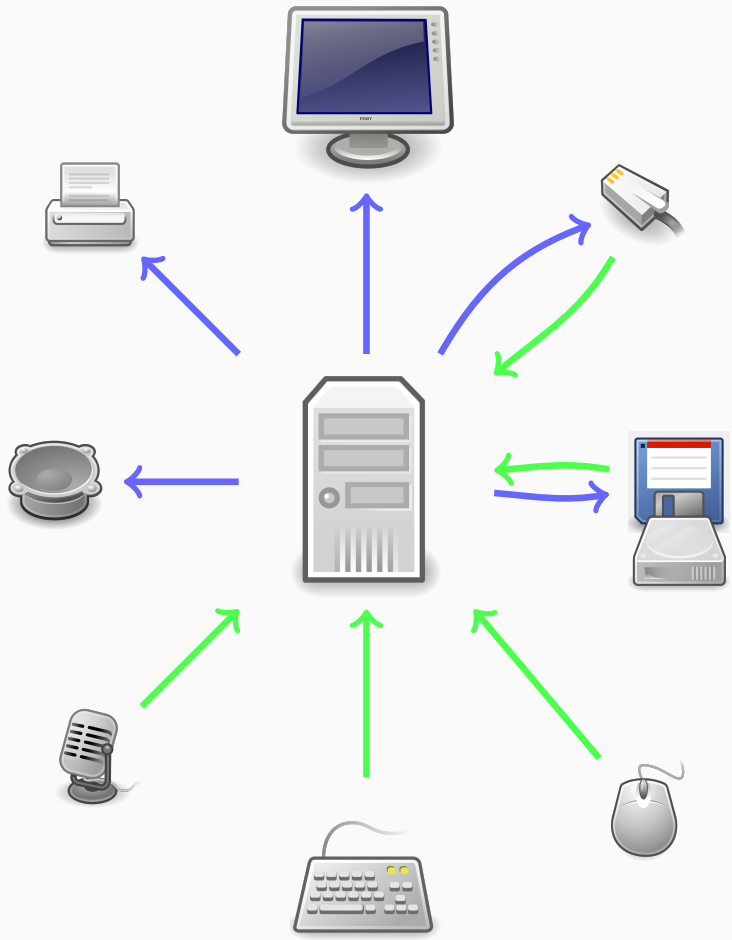
SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Windows
'..\\QCM\\q1.xml'
```

Quelques fonctions utiles du module os

`os.getcwd()`	renvoie le répertoire courant
`os.chdir(path)`	change de répertoire courant
`os.listdir(path='.')`	liste des fichiers et répertoires
`os.path.join(path1,path2,…)`	construction portable d'un chemin
`os.remove(path)`	suppression d'un fichier
`os.path.isfile(path)`	test d'existence d'un fichier
`os.path.isdir(path)`	test d'existence d'un répertoire
`os.path.split(path)`	pour extraire le fichier d'un chemin
`os.path.getsize(path)`	la taille d'un fichier

`https://docs.python.org/fr/3.7/library/os.html`

Sommaire



L'ordinateur communique avec le reste du monde

- Entrée :
  - Souris et clavier
  - Micro, caméra
  - Disque dur, clé usb, disquette
  - Câble éthernet (internet)
- Sortie :
  - Écran
  - Imprimante
  - Disque dur, clé usb, disquette
  - Enceinte

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
f_out = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>> f_out
<_io.TextIOWrapper name='test.txt' mode='w' encoding='utf-8'>
```

- La valeur de `f_out`, renvoyé par `open`, est un descripteur de fichier.
- L'encodage par défaut dépend de la machine.

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.
  - s'il existait déjà : on l'écrase et on recommence à zéro.
- On écrit dans le fichier avec la méthode `write` du descripteur de fichier.

SCRIPT
```
f_out.write('Bonjour tout le monde !\n')
f_out.write('Voici un texte écrit depuis Python !\n')
```

- Il faut déposer le caractère de retour à la ligne `'\n'`,
  - sinon, le prochain `write` prendra effet sur la même ligne.
- Les écritures sont mises en tampon ; elles ne prennent pas forcément effet immédiatement. À la fin du traitement, il faut fermer le fichier (`close`) pour que tout soit bien écrit.

SCRIPT
```
f_out.close()
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> f_out = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> f_out.write(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  write() argument must be str, not int
```

- Pour écrire un nombre, il faut d'abord le convertir en chaîne

SCRIPT
```
f_out.write(str(n))
f_out.write(" est l'année de naissance de Python.\n")
```

- ce qu'on peut aussi écrire...

SCRIPT
```
f_out.write(str(n)+" est l'année de naissance de Python.\n")
```

- ou encore...

SCRIPT
```
f_out.write(f"{n} est l'année de naissance de Python.\n")
```

Exemple : générer une table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    f_out = open('table'+str(n)+'.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        f_out.write(str(i)+'*'+ str(n)+'='+str(i*n)+'\n')
    # étape 3 : fermeture du fichier
    f_out.close()

créer_table(7)
```

- Pour lire le fichier `table7.txt`, on utilise un lecteur de fichier texte.

TABLE7.TXT

```
1*7=7
2*7=14
3*7=21
4*7=28
5*7=35
6*7=42
7*7=49
8*7=56
9*7=63
10*7=70
```

Exemple : générer une jolie table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    f_out = open(f'table{n}.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        f_out.write(f'{i:>2} × {n} = {i*n:>2}\n')
    # étape 3 : fermeture du fichier
    f_out.close()

créer_table(7)
```

- Les nombres sont correctement alignés.

TABLE7.TXT

```
 1 × 7 =  7
 2 × 7 = 14
 3 × 7 = 21
 4 × 7 = 28
 5 × 7 = 35
 6 × 7 = 42
 7 × 7 = 49
 8 × 7 = 56
 9 × 7 = 63
10 × 7 = 70
```

- Il peut être intéressant d'ajouter des lignes à un fichier. Il faut alors l'ouvrir en écriture en mode `'a'` (ajout) et non `'w'`.

SCRIPT
```
f_out = open('test.txt', 'a', encoding = 'utf-8')
```

- Pour ajouter deux lignes à la fin du fichier `table7.txt`.

SCRIPT
```
def allonger_fichier(n):
    f_out = open(f'table{n}.txt','a',encoding='utf-8')
    for i in range(11, 13):
        f_out.write(f'{i:>2} × {n} = {i*n:>2}\n')
    f_out.close()
```

- Remarque : il n'est pas possible de supprimer des lignes dans un fichier directement (mais on peut créer un nouveau fichier et détruire l'ancien).

Sommaire

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> f_in = open('table5.txt', 'r', encoding = 'utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = f_in.read() # texte contient tout le fichier !
>>> f_in.close()
>>> texte
'5 × 1 =  5\n5 × 2 = 10\n5 × 3 = 15\n5 × 4 = 20\n'
>>> print(texte)
5 × 1 =  5
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
```

- On peut aussi lire le fichier ligne à ligne avec la méthode `readline()`

SCRIPT
```
def numérote_lignes(f): # ligne par ligne
    f_in = open(f, 'r', encoding = 'utf-8')
    i = 1
    while True: # on quitte avec break
        ligne = f_in.readline() # on lit une ligne
        if ligne == '': break   # fin du fichier
        print(f'Ligne {i} : {ligne}',sep='',end='')
        i = i + 1
    f_in.close()
```

- Le fichier est fini lorsqu'on obtient une chaîne vide.
  - `''` : fin du fichier (on quitte le `while` avec `break`)
  - `'\n'` : le fichier contient une ligne vide.

SHELL
```
>>> numérote_lignes('table5.txt')
Ligne 1 : 5 × 1 =  5
Ligne 2 : 5 × 2 = 10
Ligne 3 : 5 × 3 = 15
Ligne 4 : 5 × 4 = 20
```

- On peut itérer directement sur les lignes avec une boucle `for`

SCRIPT

```
def affiche_et_compte(f):
    i=0 # pour compter le nombre de lignes
    f_in = open(f, 'r', encoding ='utf-8')
    for ligne in f_in:
        print('>' ,ligne, end = '')
        i=i+1
    f_in.close()
    return i # nombre de lignes
```

SHELL

```
>>> affiche_et_compte('table5.txt')
> 5 × 1 =  5
> 5 × 2 = 10
> 5 × 3 = 15
> 5 × 4 = 20
4
```

- `readline()` lit une ligne à la fois.
- On veut mettre les lignes lues dans une liste.

SCRIPT
```
def liste_de_lignes(f): # ligne par ligne
    f_in = open(f, 'r', encoding = 'utf-8')
    L=[]
    while True: # on quitte avec break
        ligne = f_in.readline() # on lit une ligne
        if ligne == '': break   # fin du fichier
        L.append(ligne)
    f_in.close()
    return L
```

SHELL
```
>>> liste_de_lignes('table5.txt')
['5 × 1 =  5\n', '5 × 2 = 10\n', '5 × 3 = 15\n', '5 × 4 = 20\n']
```

- Existe déjà : `readlines()` avec un `s` !

SHELL
```
>>> f_in = open('table5.txt', 'r', encoding='utf-8')
>>> lignes = f_in.readlines() ; f_in.close()
>>> lignes
['5 × 1 =  5\n', '5 × 2 = 10\n', '5 × 3 = 15\n', '5 × 4 = 20\n']
```

Lire, écrire, compter : objectifs

- Une méthode classique de travail sur fichier texte
  - on prend un fichier en entrée
  - on produit un autre fichier en sortie
- Exemple :
  - lecture d'un fichier : chaque ligne contient des nombres
  - écriture d'un fichier : chaque ligne contient la somme de ces nombres.

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

SOMME.TXT

```
246
208
324
190
269
314
307
301
```

- Pour générer le fichier aléatoire, on doit écrire 8 lignes de 5 colonnes chacune ; on utilise deux boucles imbriquées.

SCRIPT

```
def créer_aléa(nom_fichier):
    f_out = open(nom_fichier, 'w', encoding = 'utf-8')
    for ligne in range(8):    # je produis 8 lignes
        for colonne in range(5): # chaque ligne a 5 colonnes
            f_out.write(f'{randint(10,99)} ')
        f_out.write('\n') # je vais à la ligne
    f_out.close()
```

SHELL

```
>>> créer_aléa('aléa.txt')
```

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

Lire, écrire, compter : générer `somme.txt`

- Pour générer le fichier `somme.txt`,
  - on lit `aléa.txt` ligne par ligne
  - à chaque ligne lue, on écrit la somme de la ligne dans `somme.txt`

SCRIPT

```
f_out = open('somme.txt', 'w', encoding = 'utf-8')
f_in  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in f_in: # j'itère sur les lignes de f_in
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # je calcule la somme dans S2
    f_out.write(f'{S2}\n') # j'écris S2 dans somme.txt

f_in.close() ; f_out.close()
```

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

SOMME.TXT

```
246
208
324
190
269
314
307
301
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
f_out = open('somme.txt', 'w', encoding = 'utf-8')
f_in  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in f_in: # j'itère sur les lignes de f_in
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    f_out.write(f'{S2}\n') # j'écris S2 dans somme.txt

f_in.close() ; f_out.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                          # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98])                               # S2
246
>>> f'{246}\n' # Ce qu'on écrit dans le fichier
'246\n'
```

- Si le fichier n'existe pas.

SHELL
```
>>> f_in = open('complot.py', 'r', encoding = 'utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

SHELL
```
>>> f_in = open('lecture_seule.txt', 'w', encoding = 'utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

SHELL
```
>>> f_in = open('table5.txt', 'r', encoding = 'utf-8')
>>> f_in.close()
>>> f_in.read()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  I/O operation on closed file.
```

- etc.

- Ouverture en lecture (`read`) :

```
f_in = open('fichier.txt', 'r', encoding='utf-8')
```
SCRIPT

- Ouverture en écriture (`write` ou `add`) :

```
f_out = open('fichier.txt', 'w', encoding='utf-8')
```
SCRIPT

ou

```
f_out = open('fichier.txt', 'a', encoding='utf-8')
```
SCRIPT

- Lecture :
  - `f_in.read()` ♥
  - `f_in.readline()`
  - `f_in.readlines()`
  - `for ligne in f_in:` ♥
- Écriture :
  - `f_out.write(chaîne)`
- Fermeture :

```
f.close()
```
SCRIPT

Merci pour votre attention
Questions

?

Cours 6 — Exceptions et fichiers texte