Algorithmique et Programmation

Fichiers textuels et exceptions

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes
http://www.math-info.univ-paris5.fr/∽bonzon/

Fichiers textuels et exceptions

1. Gestion des exceptions
2. Fichiers textuels
3. Pour conclure

Gestion des exceptions

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
>>> 2/x
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
>>> 2/x
NameError: name 'x' is not defined
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
>>> 2/x
NameError: name 'x' is not defined
>>> 2/0
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
>>> 2/x
NameError: name 'x' is not defined
>>> 2/0
ZeroDivisionError: division by zero
```

Phénomènes exceptionnels

Il est possible de rencontrer des erreurs dans un calcul, provenant souvent de la mauvaise gestion d'un cas particulier.

```
>>> liste = [1, 2, 3, 4, 5, 6]
>>> liste[6]
IndexError: list index out of range
>>> 2/x
NameError: name 'x' is not defined
>>> 2/0
ZeroDivisionError: division by zero
```

- `IndexError`, `NameError`, `ZeroDivisionError`... sont des exceptions
- Si on ne fait rien, le programme s'arrête brutalement et affiche la trace d'exécution, c'est à dire le contexte dans lequel l'erreur a été levée

Trace d'exécution

```
1  def f(x):
2      return g(x) + 1
3
4  def g(x):
5      return 1/x
6
7  f(0)
```

Trace d'exécution

```
1  def f(x):
2      return g(x) + 1
3
4  def g(x):
5      return 1/x
6
7  f(0)
```

```
Traceback (most recent call last):
File "exple.py", line 7, in <module>
f(0)
File "exple.py", line 2, in f
return g(x) + 1
File "exple.py", line 5, in g
return 1/x
ZeroDivisionError: division by zero
```

Lever une exception

- On sait déjà comment lever une exception
- Instruction `assert`

Lever une exception

- On sait déjà comment lever une exception
- Instruction `assert`

```
def moyenne(liste) :
    """List --> Float. Calcule la moyenne de la liste"""
    assert len(liste) > 0, "Moyenne d'une liste vide"
    return somme(liste)/len(liste)
```

Lever une exception

- On sait déjà comment lever une exception
- Instruction `assert`

```
def moyenne(liste) :
    """List --> Float. Calcule la moyenne de la liste"""
    assert len(liste) > 0, "Moyenne d'une liste vide"
    return somme(liste)/len(liste)
```

```
>>> moyenne([])
AssertionError: Moyenne d'une liste vide
```

Lever une exception

- On sait déjà comment lever une exception
- Instruction `assert`

```
def moyenne(liste) :
    """List --> Float. Calcule la moyenne de la liste"""
    assert len(liste) > 0, "Moyenne d'une liste vide"
    return somme(liste)/len(liste)
```

```
>>> moyenne([])
AssertionError: Moyenne d'une liste vide
```

- Permet d'avoir des messages d'erreur plus lisibles, et de corriger son code plus facilement
- Mais ne permet pas d'**éviter** l'erreur

Comment éviter les exceptions ?

- Les erreurs ne sont pas forcément des erreurs de programmation
- Peuvent être dues à des informations incorrectes fournies par l'utilisateur

Comment éviter les exceptions ?

- Les erreurs ne sont pas forcément des erreurs de programmation
- Peuvent être dues à des informations incorrectes fournies par l'utilisateur

```
def verif_min(i, j) :
    if i < j :
        return True
    else :
        return False
```

Comment éviter les exceptions ?

- Les erreurs ne sont pas forcément des erreurs de programmation
- Peuvent être dues à des informations incorrectes fournies par l'utilisateur

```
def verif_min(i, j) :
    if i < j :
        return True
    else :
        return False
```

```
>>> min('a', 2)
Traceback (most recent call last):
TypeError: '<' not supported between instances of 'str' and 'int'
```

Comment éviter les exceptions ?

- Idée : vérifier les informations, et appeler les instructions une fois que l'on est sur de ne pas faire d'erreur

Comment éviter les exceptions ?

- Idée : vérifier les informations, et appeler les instructions une fois que l'on est sur de ne pas faire d'erreur

```
def verif_min(i, j) :
    if type(i) == type(j) :
        if i < j:
            return True
        else :
            return False
    else :
        return False
```

Comment éviter les exceptions ?

- Idée : vérifier les informations, et appeler les instructions une fois que l'on est sur de ne pas faire d'erreur

```
def verif_min(i, j) :
    if type(i) == type(j) :
        if i < j:
            return True
        else :
            return False
    else :
        return False
```

- Mais il est facile d'oublier des cas

Comment éviter les exceptions ?

- Idée : vérifier les informations, et appeler les instructions une fois que l'on est sur de ne pas faire d'erreur

```
def verif_min(i, j) :
    if type(i) == type(j) :
        if i < j:
            return True
        else :
            return False
    else :
        return False
```

- Mais il est facile d'oublier des cas

```
>>> verif_min(2, 3.0)
False
```

Gérer les exceptions

- Au lieu de chercher à éviter les instructions, il est possible de les anticiper, pour mieux les gérer

Gérer les exceptions

- Au lieu de chercher à éviter les instructions, il est possible de les anticiper, pour mieux les gérer

```
def verif_min(i, j) :
    try :
        if i < j:
            return True
        else :
            return False
    except :
        return False
```

Gérer les exceptions

- Au lieu de chercher à éviter les instructions, il est possible de les anticiper, pour mieux les gérer

```
def verif_min(i, j) :
    try :
        if i < j:
            return True
        else :
            return False
    except :
        return False
```

```
>>> verif_min(2, 3.0)
True
>>> verif_min('a', 2)
False
```

Gestion des des exceptions : syntaxe

```
try :
    <sequence_instructions1>
except :
    <sequence_instructions2>
```

Gestion des des exceptions : syntaxe

```
try :
    <sequence_instructions1>
except :
    <sequence_instructions2>
```

- Si au cours de l'exécution de la `sequence_instructions1` une exception se produit, l'exécution du bloc est abandonnée, et la `sequence_instructions2` est exécutée

Gestion des des exceptions : syntaxe

```
try :
    <sequence_instructions1>
except :
    <sequence_instructions2>
```

- Si au cours de l'exécution de la `sequence_instructions1` une exception se produit, l'exécution du bloc est abandonnée, et la `sequence_instructions2` est exécutée
- Si l'exécution de `sequence_instructions1` s'est déroulée normalement, on ne rentre pas dans le bloc `except`

Gestion des des exceptions particulières

- Il y a différents types d'exceptions : `IndexError`, `NameError`, `ZeroDivisionError`...

Gestion des des exceptions particulières

- Il y a différents types d'exceptions : `IndexError`, `NameError`, `ZeroDivisionError`...
- Il est possible de ne vouloir gérer que certaines exceptions, ou de faire des traitements différents en fonction de du type d'erreur rencontrée

Gestion des des exceptions particulières

- Il y a différents types d'exceptions : `IndexError`, `NameError`, `ZeroDivisionError`...
- Il est possible de ne vouloir gérer que certaines exceptions, ou de faire des traitements différents en fonction de du type d'erreur rencontrée

```
try :
    <sequence_instructions1> #séquence normale d'exécution
except <type_exception1> :
    <sequence_instructions2> #traitement de l'exception 1
except <type_exception2> :
    <sequence_instructions3> #traitement de l'exception 2
...
else :
    #bloc d'instruction exécuté en l'absence d'erreurs
    <sequence_instruction4>
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

```
>>>div(10,5)
pas de division par 0
la division de 10 par 5 donne 2.0
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

```
>>> div(10,5)
pas de division par 0
la division de 10 par 5 donne 2.0
>>> div(10,0)
Une division par 0 a été détectée
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

```
>>> div(10,5)
pas de division par 0
la division de 10 par 5 donne 2.0
>>> div(10,0)
Une division par 0 a été détectée
>>> div(10, 'a')
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    except :
        print("Une autre erreur a été trouvée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    except :
        print("Une autre erreur a été trouvée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

```
>>> div(10,5)
pas de division par 0
la division de 10 par 5 donne 2.0
>>> div(10,0)
Une division par 0 a été détectée
```

Gestion des exceptions : un exemple

```
def div(a, b) :
    """Int x Int --> None
    Affiche le résultat de la division de a par b si il existe"""
    try :
        a/b
    except ZeroDivisionError:
        print("Une division par 0 a été détectée")
    except :
        print("Une autre erreur a été trouvée")
    else :
        print("pas de division par 0")
        print("la division de", a, "par", b, "donne", a/b)
```

```
>>> div(10,5)
pas de division par 0
la division de 10 par 5 donne 2.0
>>> div(10,0)
Une division par 0 a été détectée
>>> div(10, 'a')
Une autre erreur a été trouvée
```

Fichiers textuels

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :
  - On suppose que tous les fichiers manipulés se trouvent dans le **répertoire courant**

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :
  - On suppose que tous les fichiers manipulés se trouvent dans le **répertoire courant**
  - On ne manipule que des **fichiers textes**

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :
  - On suppose que tous les fichiers manipulés se trouvent dans le **répertoire courant**
  - On ne manipule que des **fichiers textes**
- 2 opérations distinctes sur les fichiers :

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :
  - On suppose que tous les fichiers manipulés se trouvent dans le **répertoire courant**
  - On ne manipule que des **fichiers textes**
- 2 opérations distinctes sur les fichiers :
  - On peut vouloir **écrire** des données dans un fichier : accès en écriture

Les fichiers

Fichier

Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, ...). On y accède grâce à son nom (précédée si besoin du chemin d'accès vers ce fichier.)

- Pour simplifier, dans ce cours :
  - On suppose que tous les fichiers manipulés se trouvent dans le **répertoire courant**
  - On ne manipule que des **fichiers textes**
- 2 opérations distinctes sur les fichiers :
  - On peut vouloir **écrire** des données dans un fichier : accès en écriture
  - On peut vouloir **lire** des données à partir d'un fichier : accès en lecture

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir
  2. Le mode d'ouverture

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir
  2. Le mode d'ouverture
- L'option `'a'` permet d'ouvrir le fichier en mode ajout (*append*)

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir
  2. Le mode d'ouverture
- L'option `'a'` permet d'ouvrir le fichier en mode ajout (*append*)
  - **S'il existe un fichier du nom indiqué**, les données enregistrées sont ajoutées à la fin du fichier

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir
  2. Le mode d'ouverture
- L'option `'a'` permet d'ouvrir le fichier en mode ajout (*append*)
  - **S'il existe un fichier du nom indiqué**, les données enregistrées sont ajoutées à la fin du fichier
  - **S'il n'existe pas de fichier de ce nom**, un nouveau fichier est créé

Ouverture d'un fichier en écriture – write

```
mon_fichier = open('Monfichier.txt', 'w')
```

- L'option `'w'` permet d'ouvrir le fichier en mode écriture (*write*)

Ouverture d'un fichier en écriture – write

```
mon_fichier = open('Monfichier.txt', 'w')
```

- L'option `'w'` permet d'ouvrir le fichier en mode écriture (*write*)
  - **S'il existe un fichier du nom indiqué**, ce fichier est écrasé, et l'écriture des données commence à partir du début d'un nouveau fichier, vide.

Ouverture d'un fichier en écriture – write

```
mon_fichier = open('Monfichier.txt', 'w')
```

- L'option `'w'` permet d'ouvrir le fichier en mode écriture (*write*)
  - **S'il existe un fichier du nom indiqué**, ce fichier est écrasé, et l'écriture des données commence à partir du début d'un nouveau fichier, vide.
  - **S'il n'existe pas de fichier de ce nom**, un nouveau fichier est créé

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres
  - Si le caractère de retour à la ligne `\n` n'est pas indiqué, le prochain `write()` se fera sur la même ligne

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres
  - Si le caractère de retour à la ligne `\n` n'est pas indiqué, le prochain `write()` se fera sur la même ligne
- La méthode `close()` ferme le fichier

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres
  - Si le caractère de retour à la ligne `\n` n'est pas indiqué, le prochain `write()` se fera sur la même ligne
- La méthode `close()` ferme le fichier
  - Les écritures sont mises en tampon, elles ne prennent pas forcément effet immédiatement. Elles peuvent ne pas être enregistrées tant que le fichier n'est pas fermé !

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres
  - Si le caractère de retour à la ligne `\n` n'est pas indiqué, le prochain `write()` se fera sur la même ligne
- La méthode `close()` ferme le fichier
  - Les écritures sont mises en tampon, elles ne prennent pas forcément effet immédiatement. Elles peuvent ne pas être enregistrées tant que le fichier n'est pas fermé !
  - Ne pas oublier le `close()` donc !

Formater les écritures

- La méthode `write()` ne permet d'écrire que des chaines de caractères

Formater les écritures

- La méthode `write()` ne permet d'écrire que des chaines de caractères
- Pour écrire un nombre, il faut donc le transformer en `str` d'abord

Formater les écritures

- La méthode `write()` ne permet d'écrire que des chaines de caractères
- Pour écrire un nombre, il faut donc le transformer en `str` d'abord

```
>>>mon_fichier = open('Monfichier.txt', 'a')
>>>mon_fichier.write(1975)
```

Formater les écritures

- La méthode `write()` ne permet d'écrire que des chaines de caractères
- Pour écrire un nombre, il faut donc le transformer en `str` d'abord

```
>>>mon_fichier = open('Monfichier.txt', 'a')
>>>mon_fichier.write(1975)
TypeError: write() argument must be str, not int
```

Formater les écritures

- La méthode `write()` ne permet d'écrire que des chaines de caractères
- Pour écrire un nombre, il faut donc le transformer en `str` d'abord

```
>>>mon_fichier = open('Monfichier.txt', 'a')
>>>mon_fichier.write(1975)
TypeError: write() argument must be str, not int
>>>mon_fichier.write(str(1975))
```

Un exemple : générer un fichier contenant une table de multiplication

```
def table_fichier(n) :
  """Int --> None
  Créé un fichier contenant la table de multiplication de n"""

  #Créer le fichier
  f_table = open("table" + str(n) + ".txt", "w")

  #Ecriture dans le fichier
  for i in range(1, 11):
    f_table.write(str(i) + '*' + str(n) + '=' + str(i*n) + '\n')

  #Fermeture du fichier
  f_table.close()
```

Un exemple : générer un fichier contenant une table de multiplication

```
>>> table_fichier(n)
```

```
more 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
```

Ouverture d'un fichier en lecture

```
mon_fichier = open('Monfichier.txt', 'r')
```

Ouverture d'un fichier en lecture

```
mon_fichier = open('Monfichier.txt', 'r')
```

- L'option `'r'` permet d'ouvrir le fichier en mode lecture (*read*)

Ouverture d'un fichier en lecture

```
mon_fichier = open('Monfichier.txt', 'r')
```

- L'option `'r'` permet d'ouvrir le fichier en mode lecture (*read*)
- **S'il n'existe pas de fichier de ce nom**, une exception est levée : `FileNotFoundError`

Ouverture d'un fichier en lecture

```
mon_fichier = open('Monfichier.txt', 'r')
```

- L'option `'r'` permet d'ouvrir le fichier en mode lecture (*read*)
- **S'il n'existe pas de fichier de ce nom**, une exception est levée : `FileNotFoundError`

```
>>> ofi = open("fichier.txt", "r")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'fichier.t
```

Exception à l'ouverture d'un fichier inexistant

```
filename = input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r")
except:
    print("Le fichier", filename, "est introuvable")
else :
    <Instructions à effectuer sur le fichier>
    f.close()
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read()
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read()
>>> ofi.close()
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read()
>>> ofi.close()
>>> texte
'1*7=7\n2*7=14\n3*7=21\n4*7=28\n5*7=35\n6*7=42\n7*7=49\n
8*7=56\n9*7=63\n10*7=70\n'
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read()
>>> ofi.close()
>>> texte
'1*7=7\n2*7=14\n3*7=21\n4*7=28\n5*7=35\n6*7=42\n7*7=49\n
8*7=56\n9*7=63\n10*7=70\n'
>>> print(texte)
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
```

Lecture séquentielle d'un fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read()
>>> ofi.close()
>>> texte
'1*7=7\n2*7=14\n3*7=21\n4*7=28\n5*7=35\n6*7=42\n7*7=49\n
8*7=56\n9*7=63\n10*7=70\n'
>>> print(texte)
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
```

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read(5)
```

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read(5)
>>> print(texte)
1*7=7
```

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read(5)
>>> print(texte)
1*7=7
>>> texte = ofi.read(5)
>>> print(texte)

2*7=
```

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read(5)
>>> print(texte)
1*7=7
>>> texte = ofi.read(5)
>>> print(texte)

2*7=
#Attention, impression du caractère \n, qui compte pour 1!
```

Lecture séquentielle d'un fichier

- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier

```
>>> ofi = open("table7.txt", "r")
>>> texte = ofi.read(5)
>>> print(texte)
1*7=7
>>> texte = ofi.read(5)
>>> print(texte)

2*7=
#Attention, impression du caractère \n, qui compte pour 1!
>>> ofi.close()
```

Méthode `readline()`

- La méthode `readline()` permet de lire le fichier ligne à ligne

Méthode `readline()`

- La méthode `readline()` permet de lire le fichier ligne à ligne

```
def afficher_fichier(nom) :
  """Str --> None -- Affiche le contenu du fichier nom"""

  ofi = open(nom, 'r') #Ouvrir le fichier
  #Lecture de la première ligne du fichier
  ligne = ofi.readline()

  #Tant que le fichier n'est pas vide
  while ligne != "" :
    print(ligne, end="")
    ligne = ofi.readline()

  ofi.close() #Fermeture du fichier
```

Méthode `readline()`

```
>>> afficher_fichier("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
```

Pour conclure

Résumé du cours

Aujourd'hui, on a vu :

- Comment gérer les exceptions avec l'instruction `try ... except`
- Comment manipuler un fichier textuel, en lecture et écriture