Introduction à la programmation avec Python

CNAM UTC503 – Paradigmes de programmation

Xavier Crégut <prenom.nom@enseeiht.fr>

Objectifs

- Savoir écrire des programmes (scripts) en Python
  - objet, nom, etc.
  - structures de contrôle
  - affection, entrées/sorties, assert
  - listes et tuples
  - exceptions
- Adopter de bonnes pratiques
  - pas de variables globales
  - raffiner
  - tester
  - ...
- C'est un prérequis pour les autres séances

Pourquoi Python ?

- **open-source**, compatible GPL et utilisations commerciales
- langage **multiplateformes**
- **bibliothèque** très riche et nombreux **modules** :
  - Cryptography, Database, Game Development, GIS (Geographic Information System), GUI, Audio / Music, ID3 Handling, Image Manipulation, Networking, Plotting, RDF Processing, Scientific, Standard Library Enhancements, Threading, Web Development, HTML Forms, HTML Parser, Workflow, XML Processing...
- importante **documentation** :
  - python.org : Tutorial, Language Reference, Library Reference, Setup and Usage
  - wiki.python.org
  - Python Enhancement Proposal (PEP)
  - The Python Package Index (PyPI)
  - stackoverflow
  - Notions de Python avancées sur Zeste de savoir
  - ...
- **outils de développement :**
  - IDE (*Integrated Development Environment*) : Idle, Spyder, Pycharm, etc.
  - Documentation : PyDOC, Sphynx, etc.
  - Tests : doctest, unittest, pytest, etc.
  - Analyse statique : pylint, pychecker, PyFlakes, mccabe, mypy, etc.
- des success stories :
  - Google, YouTube, Dropbox, Instagram,
  - Spotify, Mercurial, OpenStack, Miro, Reddit, Ubuntu...

Sommaire

La commande Unix wc (word count)

La commande wc (word count) permet de compter le nombre de lignes, mots et caractères des fichiers fournis en argument de la ligne de commande.

```
$ ls
fib.py
prime.py
secret
stats.py
tmp
wc.py
$ python wc.py * zzz
     53     134    1103 fib.py
     32     112     871 prime.py
secret: Permission denied
     40     139     855 stats.py
tmp: Is a directory
     28     124     901 wc.py
zzz: No such file or directory
    153     509    3730 total
```

Pour un tel programme, il faut :

- récupérer les arguments de la ligne de commande :
  - sys.argv
- ouvrir chaque fichier correspondant à un argument
  - open
- le lire ligne à ligne
- trouver les mots d'une ligne :
  - str.split
- conserver le cumul (nb de lignes, mots et caractères)
- traiter les erreurs :
  - fichier inexistant, pas d'accès en lecture, etc.

Le programme wc (fichier wc.py)

```
1  def wc(file_names):
2      ''' Count chars, words and lines in files (like unix command wc)...'''
3      tlc = twc = tcc = 0 # total line/word/character count
4      for fname in file_names:
5          try:
6              # count lines/words/characters in fname
7              with open(fname, 'rb') as f:
8                  lc = wc = cc = 0  # line/word/character count
9                  for line in f:
10                     lc += 1
11                     cc += len(line)
12                     wc += len(line.split())
13
14             # print counts
15             print( '%7d %7d %7d %s' % (lc, wc, cc, fname) )
16
17             # update total counts
18             tlc += lc; twc += wc; tcc += cc # to avoid!
19
20         except IOError as e:
21             print(e.filename, ': ', e.strerror, sep='')
22
23     # print total counts
24     print( '%7d %7d %7d %s' % (tlc, twc, tcc, 'total') )
25
26 if __name__ == '__main__':
27     import sys
28     wc(sys.argv[1:])
```

Les explications

1. Définir des **sous-programmes** (**fonctions**) pour **éviter les variables globales** (exemple : wc)
   - `file_names` : **paramètre formel** et `sys.argv[1:]` : **paramètre effectif**
   - pas de `return`, la valeur retournée sera `None` (procédure)
2. `tlc = twc = tcc = 0` : initialiser les variables tlc, twc et tcc sont initialisés à 0
3. `for fname in file_names:` : fname vaut successivement chaque valeur de `file_names`
4. l'indentation définit les blocs d'instructions
5. `open(fname, 'rb')` : ouvrir un fichier à partir de son nom, en lecture ('r'), binaire ('b')
6. `with` : garantir la libération d'une ressource : ici, fermer le fichier (`f.close()`, fnotation objet)
7. `for line in f:` : f étant un descripteur de fichier, line sera successivement chaque ligne de f
8. `len(s)` : le nombre d'éléments de la séquence s (chaîne, liste, tuple, etc.)
9. `line.split()` : découper une chaîne en liste de chaînes (par défaut, sur les blancs)
10. % : (string interpolation) le i^e % est remplacé par le i^e élément du tuple.
11. `try: except:` : traitement des exceptions
12. On peut mettre plusieurs instructions sur la même ligne (séparées par ' ;'). À éviter !
13. `if __name__ == '__main__':` : signifie « si le script est un programme principal »
    - les instructions de ce `if` ne sont exécutées que si le fichier est exécuté comme programme
    - elles ne le sont pas si le script est importé comme module
14. `import` : utiliser un module (ici `sys`)
15. `sys.argv` : liste des arguments de la ligne de commande (y compris le nom du script, wc.py)
16. `sys.argv[1:]` : (slice) les éléments sys.argv à partir de la position 1 jusqu'au dernier.
    - Forme générale : liste[début:fin:pas] de début inclu jusqu'à fin exclu de pas en pas.

Les raffinages

```
R0 : Count chars, words and lines in files

Exemples :
    ...

R1 : Comment « Count chars, words and lines in files » ?
    Initialize total counts                  tlc, twc, tcc: out
    for fname in command line arguments:
        count lines/words/chars in fname     fname: in ; lc, wc, cc: out
        print this file counts               fname, lc, wc, cc : in
        update total counts                  tlc, twc, tcc: in out ; lc, wc, cc: in
    print total counts                       tlc, twc, tcc: in

R2 : Comment « count lines/words/chars in fname » ?
    initialize statistical variables         lc, wc, cc: out
    open the file                            f: out
    for line in f:
        update statistical variables         lc, wc, cc: in out ; line: in
    close the file

...
```

Principe des raffinages

1. Comprendre le problème
   - Reformuler le problème : R0
   - Donner des exemples
2. Identifier une solution informelle
   - La partie déficile : comment fait-on pour résoudre le problème ,
3. Structure la solution informelle
   - Écrire le premier niveau de raffinage R1
   - Vérifier le premier niveau de raffinage R1
   - Décomposer les actions (et expressions) complexes introduites
   - On introduit un nouveau raffinage Ri pour décomposer une action compexe de niveau i-1.
   - Forme générale :

```
Ri : Comment « action complexe » ?
     ... actions complexes ou instructions
     ... liées par des structures de contrôle
```

4. Produire le programme
5. Tester le programme
   - avec les exemples identifiés : tests fonctionnels
   - et d'autres pour remplir les critères de couverture de code : tests structurels

Tester avec pytest

Une fonction... et sa fonction de test.

```
def f(x):
    '''Une fonction... comme en math.'''
    return 2 * x + 3

def test_f():
    assert f(0) == 3
    assert f(1) == 5
    assert f(.5) == 4
    assert f(-2) == 1
```

L'instruction `assert` lève une exception si l'expression à sa droite n'est pas vraie.

Est-ce que tout est correct ?

Tester avec pytest (exécuter)

pytest exécute les tests (si on ne précise rien, test_.*py et* _test.py) et en affiche le résultat :

```
$ pytest test_f.py
============================= test session starts ==============================
platform linux -- Python 3.7.3, pytest-5.2.1, py-1.8.0, pluggy-0.13.0
collected 1 item

test_f.py F                                                              [100%]

=================================== FAILURES ===================================
_________________________________ test_f _________________________________

    def test_f():
        assert f(0) == 3
        assert f(1) == 5
        assert f(.5) == 4
>       assert f(-2) == 1
E       assert -1 == 1
E        +  where -1 = f(-2)

test_f.py:9: AssertionError
============================== 1 failed in 0.04s ===============================
```

Sommaire

- Objet
- Type
- Opération
- Nom (ou Variable)
- Partage
- Égalité
- Muable et immuable

Objet

Toutes les données manipulées par Python sont des objets : **tout est objet**.

Voici quelques exemples :

```
421          # un entier (int, integer)
3.9          # un nombre réel (float)
4.5e-10      # un autre avec un exposant
3+5j         # un nombre complexe (complex)
"Bonjour"    # une chaîne de caractères (str, string)
'x'          # aussi une chaîne de caractères !
[1, 2.5, 'a' ]  # une liste (list)
```

Un objet possède :

- une **identité** : entier unique et constant durant toute la vie de l'objet.
  On l'obtient avec la fonction `id` (l'adresse en mémoire avec CPython)
- un **type** (la classe a laquelle il appartient) que l'on obtient avec la fonction `type`
  Le type d'un objet ne peut pas changer.

```
id(421)      # 140067608358512 (par exemple !)
type(421)    # <class 'int'>
type(3.9)    # <class 'float'>
type('x')    # <class 'str'>
```

- des **opérations** généralement définies au niveau de son type

Type

Un **type** définit les caractéristiques communes à un ensemble d'objets.

Parmi ces caractéristiques, on trouve les **opérations** qui permettront de manipuler les objets.

```
>>> dir(str)        ## affiche les noms de tout ce qui est défini sur `str`
[..., '__doc__', ..., 'capitalize', 'count', 'endswith', 'format', 'index',
'isalnum', 'isalpha', 'islower', 'isnumeric', 'join', 'lower', 'replace',
'split', 'startswith', 'strip', 'swapcase', ...]

>>> help('str.lower')   ## donne la description de la méthode `lower` de `str`.
str.lower = lower(...)
    S.lower() -> str

    Return a copy of the string S converted to lowercase.
```

Essayer :

```
help(str)       ## affiche la documentation de 'str'
```

Opération

Parmi les **opérations**, on peut distinguer :

- les **opérateurs** « usuels » : `+ - * / **` ...
- les **sous-programmes** (procédures [1] ou fonctions) : print, len, id, type, etc.
- les **méthodes** (sous-programmes définis sur les objets) : lower, islower, etc.

Chaque type d'opération a sa **syntaxe d'appel** :

- opérateurs : souvent infixe : `2 * 4`
- sous-programme : `print(2, 'm')` ou `len('Bonjour')`
- méthode : notation pointée : `'Bonjour'.lower()`

```
2 * 4                   ## 8
2 ** 4                  ## 16    (puissance)
print(2 ** 4)           ## 16    (une procédure)
print(2, 'm')           ## 2 m
len('Bonjour')          ## 7 : nombre de caractères de la chaîne (une fonction)
'Bonjour'.lower()       ## 'bonjour'              (une méthode)
'Bonjour'.islower()     ## False                  (une autre méthode)
```

1. En Python, tout est fonction !

Nom (ou Variable)

- Une **variable** permet de référencer un objet grâce à un **nom** (**identifiant**).
  - En Python, on parle plutôt de **nom** que de **variable**.
  - Un nom (une variable) est en fait un **accès**, une **référence**, un **pointeur** sur un objet.
  - **Règle :** Un nom est de la forme : lettre ou souligné, suivi de chiffres, lettres ou soulignés
- **Intérêt :** nommer les objets pour y accéder (et améliorer la lisibilité du code).
  $\Longrightarrow$ Toujours choisir un nom **significatif** ! (contre-exemples ci-après)

```
x = 5          ## x associé à l'entier 5
y = 'a'        ## y est associé à la chaîne 'a'
print(x, y)    ## 5 a    références aux objets associés à x et y
```

- **Convention :**
  - Un nom de variable est en minuscules.
  - Utiliser un souligné _ pour mettre en évidence les morceaux d'un nom : `prix_ttc`
  - Voir PEP 8 – Style Guide for Python Code
- **Exemples :**
  - Corrects : rayon, perimetre_cercle, prix_ttc, prix_ht, n1, n2.
  - Déconseillés : prixttc, lageducapitaine, rayon_du_cercle (trop long), r (trop court)
  - Incorrects : 2n
- **None :** Un objet particulier qui a la signification « rien »
  - Utilisé pour dire qu'une variable ne référence pas d'objet (convention)

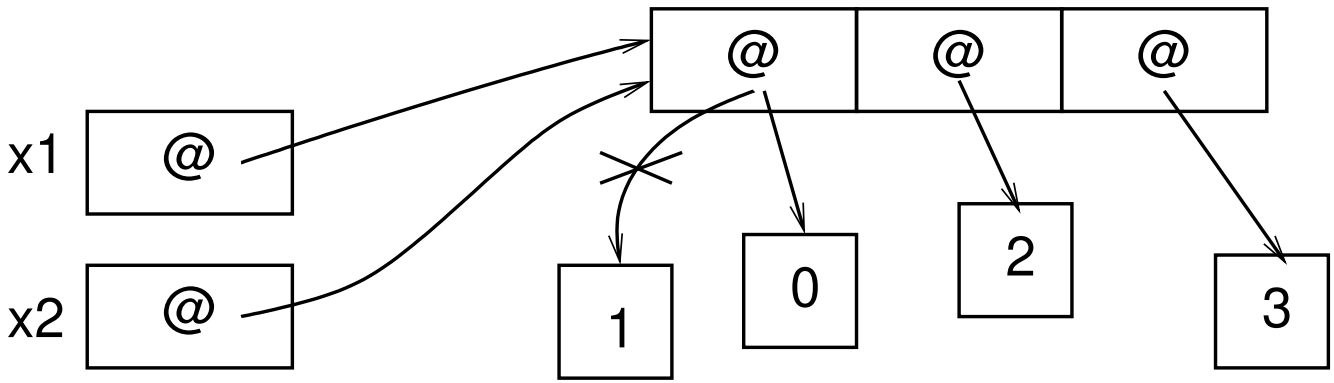
```
y = None    ## None    objet prédéfini qui signifie sans valeur associée !
y == None   ## True    Savoir si y est associé à None (on pourrait utiliser is)
```

Bien comprendre Nom et Objet : le partage

- Un objet ne peut changer ni d'identité, ni de type.
- Plusieurs noms peuvent référencer le même objet (ce sont des **alias**).
- Tout changement fait sur l'objet depuis l'un des noms est visible depuis les autres noms !

```
1 x1 = [1, 2, 3]      # une liste contenant 3 éléments 1, 2 et 3
2 x2 = x1             # un deuxième nom sur la même liste
3 x1[0] = 0           # changement du premier élément de la liste
4 print(x2)           # [0, 2, 3]
5 id(x1) == id(x2)    # True : x1 et x2 donnent accès au même objet
```

- Un nom est bien un **accès**, **pointeur**, **référence** sur un objet



Égalité physique (`is`) et égalité logique (==)

- **Égalité physique :** deux noms référencent le même objet (même identifiant)
- **Égalité logique :** deux objets (éventuellement d'identifiants différents) ont mêmes valeurs

```
1  x1 = [1, 2, 3]  # une liste contenant 3 éléments 1, 2 et 3
2  x2 = x1         # un deuxième nom sur la même liste
3  x3 = [1, 2, 3]  # une autre liste contenant 1, 2 et 3
4  x1 is x2      # True
5  x1 is x3      # False
6  x1 == x2      # True
7  x1 == x3      # True
8  x3[0] = 0
9  x1 == x3      # False
10 x1 != x3      # True, négation de ==
11 x1 is not x3 # True, négation de is
```

- L'opérateur **is** teste l'**égalité physique** : même objet
- L'opérateur == teste l'**égalité logique** : mêmes valeurs
- `n1 is not n2` est équivalent à `not (n1 is n2)`
- `n1 != n2` est équivalent à `not (n1 == n2)`
- **Normalement :** égalité physique implique égalité logique (exception : NaN, i.e. math.nan)

Remarque : `n1 is n2` est équivalent à `id(n1) == id(n2)`. Préférer `is` !

Objet muable et objet immuable

Définition :

- Objet **immuable** : objet dont l'état ne peut pas changer après sa création.
- Objet **muable** : objet dont l'état peut changer au cours de sa vie.
- Synonymes : altérable/inaltérable, modifiable/non modifiable (anglais : *mutable/imutable*)

```
1   s1 = 'bon'        # Les chaînes sont immuables :
2   s1[0] = 'B'       # TypeError: 'str' object does not support item assignment
3   del s1[0]         # TypeError: 'str' object doesn't support item deletion
4   s2 = s1.lower()   # Création d'un nouvel objet
5   s2 is s1          # False    on a bien deux objets différents
6   s2 == s1          # True     mais logiquement égaux
7
8   l1 = [ 1, 2, 3 ]     # Les listes sont muables, la preuve :
9   l1[0] = -1           # [-1, 2, 3]
10  del l1[0]            # [2, 3]
11  l1.append(4)         # [2, 3, 4]
```

Objet immuable vs objet muable :

- Un objet immuable peut être partagé sans risque car personne ne peut le modifier
- Mais chaque « modification » nécessite la création d'un nouvel objet !

Sommaire

Instructions élémentaires

pass

Instruction qui ne fait rien.

assert

Lever une exception si la condition qui suit n'est pas vraie
Exécuter avec -O supprime la vérification des assert.

Affection

```
x = y = z     # les noms x et y référencent l'objet associé à l'objet z
x, y = z, t   # 1. évaluation de z et t, puis x = z et y = t
```

Écrire

```
>>> print(1, 2, 3, sep=', ', end='...')
1, 2, 3...
```

Lire

```
reponse =  input("N ? ")     ## input retourne une chaîne de caractères
n = int(reponse)                 ## interprétation de la chaîne comme un int, ValueError si erreu
```

Conditionnelle : if, elif, else

```
if n > 0:
    resultat = 'positif'
elif n < 0:      # not (n > 0) and (n < 0)
    resultat = 'negatif'
else:            # not (n > 0) and not (n < 0)
    resultat = 'nul'
```

Répétitions

while

```
1 n = 10           # un entier
2 somme = 0    # la somme des entiers
3 i = 1        # pour parcourrir les entiers de 1 à n
4 while i <= n:    # i est à prendre en compte
5     somme = somme + i        # ou somme += i
6     i = i + 1            # ou i += 1
7 print(f'la somme des entiers de 1 à {n} est : {somme}')
```

for

```
1 n = 10           # un entier
2 somme = 0    # la somme des entiers
3 for i in range(1, n+1):
4     somme += i
5 print(f'la somme des entiers de 1 à {n} est : {somme}')
```

Mais aussi

- break : sortir de la boucle
- continue : passer à l'itération suivante
- else : si on sort normalement de la boucle

Exceptions : on les rencontre vite !

Motivation

Que peut faire :

- l'interpréteur Python si on lui demande d'interpréter un programme avec une erreur de syntaxe ou une variable non définie ?
- la méthode `index` de string si on fournit une chaîne qui n'existe pas ?

Le travail ne peut pas être fait. Une exception le signale : SyntaxError, NameError, ValueError...

Exemples

```
>>> 'bonjour'.index('j')
3
>>> 'bonjour'.index('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

```
>>> xxx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xxx' is not defined
```

Définition

Exception = moyen pour une fonction de signaler que le travail attendu ne peut pas être réalisé.

Traiter une exception : un exemple

Exception vs Conditionnelle

```
s = 'bonjour'
c = 'z' # ou 'j', 'r'...
# indice de c dans s
try:
    print(s.index(c))
except ValueError:
    print("aucun")
```

```
s = 'bonjour'
c = 'z' # ou 'j', 'r'...
# indice de c dans s
if c in s:
    print(s.index(c))
else:
    print("aucun")
```

Discussion

Avec les exceptions : approche optimiste

- on écrit une version optimiste du code (où tout se passe bien) : dans un `try`
- on traite ensuite les cas d'erreurs : dans les `except` associés
- le code nominal est plus facile à lire
- si on sait traiter l'exception, on la rècupère sinon elle continue à se propager
- Risque : mal identifier l'origine de l'exception et faire le mauvais traitement

Avec une conditionnelle : approche pessimiste

- tester explicitement tous les cas anormaux
- le code peut devenir difficile à lire
- peut être plus coûteux (ici, deux parcours de s : un pour `in`, l'autre pour index)
- Remarque : le test pourrait être fait *a posteriori* sur le résultat de la fonction

Mécanisme d'exception

Principe

Mécanisme en trois phases :

1. **Levée** de l'exception quand une anomalie est détectée : `raise`
   L'exécution du bloc est interrompue et l'exception commence à se *propager*.
2. **Propagation** de l'exception : (automatique) l'exception remonte blocs et sous-programmes
3. **Récupération** de l'exception : `except`
   fait par la portion de code qui sait traiter
   reprise de l'exécution normale
   toujours associée à un `try` : seules les exceptions levées dans ce `try` peuvent être récupérées

Intérêt

- découpler la partie du programme qui détecte une anomalie de la partie qui sait la traiter
  - plus pratique qu'un code d'erreur !
- permettre d'écrire un code optimiste (plus lisible) en regroupant après le traitement des erreurs

`raise` vs `return` dans une fonction

- `raise` et `return` arrêtent l'exécution de la fonction
- `return` rend la main à l'appelant de cette fonction
- `raise` « saute » tous les appelants jusqu'au prochain `except` correspondant à l'exception

Lever une exception

Syntaxe sur un exemple

```
raise ValueError("J'ai levé ma première exception")
```

Remarque

La lever d'une exception se fait normalement dans un sous-programme.
C'est le moyen de signaler les éventuelles anomalies à l'appelant.

Récupérer une exception

Syntaxe

```
try:
    # lignes qui peuvent lever une exception
    # normalement au travers de l'appel de sous-programmes
except NomException1:
    # traitement de l'exception NomException1
    # On n'utilise pas l'objet exception récupéré
except NomException2 as nom_exception:
    # traitement de l'exception NomException2
    # nom_exception référence l'objet exception qui se propageait
    ...
except BaseException as e: # on récupère toutes les exceptions !
    # traitement...
else:
    # Ne sera exécuté que si aucune exception n'est levée
finally:
    # Sera toujours exécuté, qu'il y ait une exception ou pas,
    # qu'elle soit récupérée ou pas
```

Explications

Dans les commentaires ci-dessus ;-)

`with ... [as ...]`

Lire et afficher le contenu d'un fichier texte

```
nom_fichier = 'exemple.txt'
with open(nom_fichier, 'r') as fichier:
    for ligne in fichier:   # pour chaque ligne de fichier
        print(ligne, end='')    # le '\n' est déjà dans `ligne`
```

Explications

- `with` garantit que l'objet créé (ici le fichier) sera bien fermé $\Longrightarrow$ **Utiliser** `with`
- `fichier.readline()` : retourne une nouvelle ligne du fichier (`''` quand plus de ligne à lire)
- `fichier.readlines()` : retourne la liste de toutes les lignes du fichier

`with ...` est transformé en `try ... finally`

```
try:
    fichier = None
    fichier = open(nom_fichier, "r")
    for ligne in fichier:
        print(ligne, end='')
finally:
    if fichier is not None:
        fichier.close()
```

Exemple : Écrire dans un fichier texte

Écrire dans un fichier

```
nom_fichier = '/tmp/exemple.txt'
with open(nom_fichier, 'w') as fichier:
    fichier.write('Première ligne\n')   # `write` n'ajoute pas de '\n'
    print('Deuxième ligne', file=fichier)
```

Remarque

- `write()` retourne le nombre de caractères effectivement écrits.

Quelques fonctions prédéfinies

- len : la taille, longueur d'un objet
- input : lire une ligne sur l'entrée standard
- print : afficher des objets
- int : convertir vers des entiers
- float : convertir vers des réels
- range : produire une séquence d'entier
- sum : la somme de nombres
- str : la chaîne de caractère qui correspond à cet objet
- id : l'identifiant d'un objet
- min, max : le minimum et le le maximum
- chr, ord : passer des caractères aux entiers et inversement
- abs : valeur absolue
- pow : la puissance