Notions fondamentales

Aide mémoire synthétique sur le langage Python. Les différences importantes entre la branche 2 et la branche 3 seront commentées. La différence la plus fréquente est le passage de print à print() !

Règles de base

Ces règles peuvent être testées via le mode interactif de Python (en utilisant la fenêtre “Shell” ou console de l'éditeur Idle ou Idle3 par exemple).

- Définition d'une **donnée** : suite finie de nombres binaires
- Définition d'une **variable** dans un langage de programmation : apparaît sous un nom de variable, mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive
- **Types** de variables, dont la déclaration n'est pas nécessaire. Une variable est automatiquement créée avec le type qui correspond au mieux à la valeur fournie. Les types courant sont les “entier”, “flottant”, “chaîne de caractères”, “complexe”, “liste”,...
- **Mots réservés** : on ne peut pas utiliser comme noms de variables les 29 « mots réservés » utilisés par le langage lui-même (if, elif, while, for, else, print,...).
- Instruction d'**affectation** utilisant le signe = et réalisant les opérations de :
  - créer et mémoriser un nom de variable ;
  - lui attribuer un type bien déterminé ;
  - créer et mémoriser une valeur particulière ;
  - établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante
- En mode interactif, entrer le nom d'une variable, puis <Enter> provoque l'affichage de sa valeur ; Dans un programme, on utilise print. Le nombre de chiffres affichés peut varier (essayer par exemple “1/3”. et “print (1/3.)”)
- **Typage automatique**. Attention aux nombres 1, 1. etc (différencier 20/3 et 20./3 par exemple)
- **Affectation multiple** : il est possible d'effectuer plusieurs affectations en une seule instruction, par exemple : a, b = 4, 8.33
- **Opérateurs arithmétiques** : “+”, “-”, “*”, “/”, “/ /” “* *”, “%” sont les symboles permettant d'effectuer les opérations classiques : addition, soustraction, multiplication, division (normale ou entière), puissance et opérateur modulo ou reste d'une division d'entier.
- **Expressions** = variables combinées par des opérateurs, qui donnent finalement des valeurs, pouvant aussi être interprétées logiquement (vrai-faux)
- Règle “PEMDAS” de priorité des opérations : Parenthèses, Exposant, Multiplication, Division, Addition, Soustraction. Pour deux opérateurs de même priorité, l'évaluation est effectuée dans l'ordre de gauche à droite

Scripts ou programmes Python, où les conserver, et comment les nommer :

il est utile de donner des noms de programmes signifiants, d'éviter les espaces et caractères spéciaux dans les noms, d'utiliser systématiquement l'extension “.py” et de les classer en répertoires suivant leur rôle ou utilité (exercice, exemple simple, application de calcul, utilisation graphique, interface,...). Python propose des lignes directrices sur le style d'écriture des programmes, mettant en avant la

lisibilité. Il s'agit de la pep8.

Structures conditionnelles et répétitives

Pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un algorithme. Leur compréhension ne nécessite pas un ordinateur.

Suggestions : rechercher quelques algorithmes classiques, comme :

- trouver le plus grand nombre d'une liste de valeurs
- calculer une moyenne d'une liste de valeurs
- Calculer la factorielle d'un nombre
- Dresser la suite de Fibonacci
- ...

- Un programme est la transcription dans un langage informatique d'un algorithme donné pour résoudre un problème.
- La programmation moderne utilisent des séquences d'instructions, des structures de sélection et de répétition.

Sélection ou instruction conditionnelle

Cf. la documentation officielle ou cet autre cours avec des illustrations et des exemples de code exécutable sur le site pythontutor.

En python, une indentation par tabulation (dans les faits 4 espaces) délimite le bloc d'instruction en rapport avec les conditions testées, dont la valeur est évaluée comme “vrai” (true) ou “faux” (false). Exemple :

```
a = 0
if a > 0 :
    print("a est positif")
    print("car il est strictement plus grand que zéro")
elif a < 0 :
    print("a est négatif")
else:
    print("a est nul")
```

L'utilisation de “elif” est liée à une condition chaînée. L'utilisation de “else” sous-tend une condition alternative. un simple “if” correspond à une expression conditionnelle.

- Les **Opérateurs de comparaison** sont :
  - == (“égal à”),
  - != (“différent de”), à utiliser préférentiellement à <>
  - > (“plus grand que”),
  - < (“plus petit que”),

    - >= (“plus grand ou égal à”),
    - < = (“plus petit ou égal à”)
- **Blocs d'instructions** : plusieurs instructions consécutives au sein d'une structure (conditionnelle, ou autre, cf. plus loin), et qui partagent donc le même décalage d'indentation.
- **Structures imbriquées** : une structure au sein d'une autre structure ! (aisément identifiables grâce à l'indentation).
- Les **commentaires** commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante. Idéalement, il faut aussi penser à utiliser dans les définitions (des fonctions, des classes,...) une chaîne de documentation, débutant et se terminant par les 'triples quotes' : “”“

Ces dispositions sont capitales pour assurer la compréhension d'un code source, par une autre personne, mais par le programmeur lui-même lorsqu'il reprend un code longtemps après son écriture.

Structures répétitives

L'**instruction while** (documentation officielle) Permet d'exécuter des commandes tant que la condition qui suit (éventuellement combinée) est vraie. Exemple :

```
a = 0
while a < 12 :
    a = a + 1
    print(a, a**2, a**3)
```

L'**instruction for** (documentation officielle) permet d'itérer sur une liste, ou aussi sur les caractères successifs d'une séquence “chaîne de caractères”.

```
for i in range(11):
    print(i, i**2, i**3)
```

Cf. cet autre cours avec des illustrations et des exemples de code exécutable sur le site pythontutor.

Principaux types de données

Les types de données numériques principaux :

Cf. la documentation officielle

- Le type de donnée entier (**integer**) est encodé sous la forme d'un bloc de 4 octets (ou 32 bits). Or la gamme de valeurs décimales qu'il est possible d'encoder sur 4 octets seulement s'étend de -2147483648 à + 2147483647. Au-delà de ces limites, l'encodage des entiers devient du type **long**, avec une précision quasi infinie. Sous python 3, les entiers sont d'office “long”.

> Si les opérandes sont entiers, l'opération de division avec l'opérateur ”/“ se fait en arithmétique entière, sous python 2. L'opérateur spécifique ”/ /“ effectue l'opération

de division entière. Sous python 3, la division avec "/" fournit d'office le résultat "float".

Exemple : différencier 2/3 et 2./3. Il est important d'indiquer un des nombres avec le point décimal pour forcer l'arithmétique en "float". Une confusion de type à ce niveau peut provoquer des comportements indésirables des programmes!!!

- **Float** : permet de manipuler des nombres (positifs ou négatifs) compris entre 10^-308 et 10^308 avec une précision de 12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine.
- **complex** : les nombres complexes sont utilisables aussi facilement que les autres nombres. On peut les initialiser comme ceci :

```
z1=complex(1,2)
```

ou encore comme cela :

```
z2=3+5.67j
```

Les fonctions *int()*, *float()* et *complex()* permettent de transformer le type de la variable indiquée en argument, avec en retour un code d'erreur si la transformation n'est pas possible. Des chaînes de caractères contenant des représentations valides de nombres sont permises comme arguments, la fonction *int()* appliquée à un argument "float" fournira la partie entière du nombre.

Le type de données texte ou "string" :

Cf. la documentation officielle et Dive Into Python 3

Suite quelconque de caractères Unicode délimitée soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). '/' : l'antislash permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Les chaînes de caractères sont des données composites (qui rassemble dans une seule structure un ensemble d'entités plus simples). Python considère qu'une chaîne de caractères est un objet de la catégorie des séquences immutables, lesquelles sont des collections ordonnées d'éléments. Chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un index, débutant à partir de 0. Exemple : mot[3], mot[7],...

Essayez per exemple aussi de taper ces commandes dans l'interpréteur (idle)

```
mot="chimie"
print(mot[0])
print(mot[5])
print(mot[6])
```

- Concaténation de chaînes :

```
c = "auto" + "mobile"
```

- Longueur (nombre de caractères) d'une chaîne :

```
len(c)
```

- Conversion en nombre (donnée numérique créée à partir d'une chaîne de caractères) : int(“587”), float(“3.14”)
- Concaténation d'une liste de chaîne :

```
strings = ['A', 'bac', 'cali', 'jkppl']
text = ''.join(strings)
print(text)
```

Les caractères Unicode étant considérés comme abstraits dans Python 3, leur encodage (UTF-8, UTF-16,...) n'est à prendre en considération que si on utilise la méthode .encode pour les convertir en bytes.

Références

- Handling Unicode Strings in Python

Les séquences binaires

Cf. la documentation officielle

Destinés à la manipulation de données sous forme binaire, les bytes ne peuvent pas être utilisée pour des textes, même s'il y a une correspondance pour les 127 premiers caractères (codes ASCII). Leur entrée peut se faire dans ce cas via par exemple

```
"bdata = b'ceci est un texte ASCII'
```

Depuis la version 3 de Python, la séparation des utilisations des séquences binaires et des chaînes de caractère est bien différenciée.

Les booléens (vrai ou faux)

Cf. Algèbre de Boole sur Wikipedia

- https://docs.python.org/2/library/stdtypes.html#truth-value-testing (Python 2)
- https://docs.python.org/3/library/stdtypes.html#bltin-boolean-values (Python 3)
- les opérateurs booléens
- les opérateurs de comparaison

Les listes

Cf. la documentation officielle

- Collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets. Ce

sont des collections ordonnées (séquences) d'objets, qui sont modifiables.
- len() renvoie le nombre d'éléments d'une liste
- la fonction del permet d'effacer des éléments
- Les listes sont des objets sur lesquels on peut agir à l'aide de méthodes, comme .append pour ajouter un élément, .remove pour en enlever. Voir la documentation en ligne (Library Reference - Built-in Objects - Built-in Types - Sequence types - Mutable Sequence types). Exemples :

```
a=[12,15,7,13,21,24,11,13,21,27,5]
print(a,len(a))
a.append(8)
print(a,len(a))
a.reverse()
print(a,len(a))
b=a.pop(5)
print(b)
print(a,len(a))
a.insert(5,99)
print(a,len(a))
a.sort()
print(a,len(a))
```

- *range(start, stop, step)* avec des arguments entiers renvoie une liste d'entiers commençant par *start*, incrémentés chaque fois de la valeur *step*, jusque la valeur *stop* exclue.
- La fonction enumerate permet de parcourir les éléments d'une liste en même temps que leur indice, bien plus pratiquement qu'en passant par l'utilisation de range :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
enumerate : exemples
"""
recipients = ['matras', 'erlenmeyer', 'verre à pied', 'ballon', 'bécher']
print(', '.join(recipients))
# méthode "classique"
for i in range(len(recipients)):
    print('Récipient # ', i, ' : ', recipients[i])
# méthode avec enumerate
for i,recipient in enumerate(recipients):
    print('Récipient # ', i, ' : ', recipient)
```



: piles, files/queues, pop,...

Références

- Python List Comprehension Tutorial

Les tuples

Analogues aux listes, mais utilisant les parenthèses pour leur écriture, les tuples ne peuvent pas être modifiés. De ce fait, une méthode fonction de hachage peut être appliquée aux tuples, mais pas aux listes, ce qui rend les tuples utilisables comme clés de dictionnaires ou éléments d'ensembles.

Référence :

- Python Tuples Tutorial

Les dictionnaires

Cf. la documentation officielle

Un dictionnaire permet de stocker des paires (clé : valeur), où les clés doivent être uniques, non modifiables, et où les valeurs sont n'importe quel objet. Il est délimité par des accolades {}, et les couples clé : valeur sont séparés par des virgules. Au sein de ce couple, le caractère ":" sépare la clé de la valeur. Les éléments peuvent être accédés suivant les clés. Dès qu'on doit utiliser à la fois la clé et la valeur, il est recommandé d'utiliser la fonction *".items()"*, comme ceci :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

dico = {"Nom": "Belgique",
    "Capitale":"Bruxelles",
    "Population":11239755,
    "Superficie":"30 528 km2",
    "Monnaie":"Euro",
    }

for key, val in dico.items():
    print("{} = {}".format(key, val))
```

Références diverses

- https://www.datacamp.com/community/tutorials/python-dictionary-tutorial

Les ensembles

Un ensemble (set) est une collection non ordonnée d'éléments non répétés (uniques). L'utilisation des ensembles se fait par analogie avec les propriétés et opérations de la théorie mathématique des ensembles : appartenance, cardinalité (nombre d'éléments), union, intersection, différence, ...

- https://docs.python.org/2/library/stdtypes.html#set-types-set-frozenset (Python 2)
- https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset (Python 3)
- http://en.wikibooks.org/wiki/Python_Programming/Sets
- Python Sets: What, Why and How

Les éléments d'ensembles doivent être différents (et le rester), donc posséder un caractère immuable (ou immutable en anglais), et de là hachable.

Il est donc plus efficace de tester l'appartenance d'un élément à un ensemble (éventuellement créé au départ d'une liste) plutôt que de tester la présence dans une liste (toute la liste doit être parcourue) :

```
li = [4, 8, 9, 1, 6, 8, 4]
# pas efficace :
if 8 in li:
    print("8 est dans la liste")

# efficace :
se=set(li)
if 8 in se:
    print("8 est dans la liste")
```

D'autres types

Des types “haute-performance” sont aussi intégrés à Python, via le module “collections” à importer :

- Counter → cf. cet exemple d'utilisation de Counter
- deque (double-ended queue)
- defaultdict
- namedtuple → cf. cet exemple d'utilisation de namedtuple
- OrderedDict

Consulter la documentation officielle, et ces liens (1, 2, 3, 4)

Des types non intégrés par défaut dans Python peuvent facilement être implémentés, en utilisant les types répandus. C'est pas exemple le cas des arbres (informatique, théorie des graphes) :

- One-line Tree in Python avec defaultdict
- a general tree implementation in python
- Looking for a good Python Tree data structure
- How can I implement a tree in Python? Are there any built in data structures in Python like in Java?

Python n'intègre pas par défaut des types triés (sorted list, sorted dict, sorted set). Une solution en “pure python” existe via les SortedContainers

---

À ce stade de votre apprentissage, vous pouvez utiliser l'outil Online Python tutor pour visualiser dans un navigateur web le déroulement de petits programmes proposés sur le site ou écrits par vos soins ! Voici ce que cela donne pour les deux structures de répétition while et for :

- Code "while" dans Online Python Tutor
- Code "for" dans Online Python Tutor

Pensez ensuite à essayer des modifications du code !

Fonctions prédéfinies

- input() permet d'entrer des données au clavier. raw_input entre une chaîne de caractères (on peut dans certains cas la convertir par int() ou float()...)
- Importer un module de fonctions. Exemples : from math import * –> les fonction abs, sqrt, sin,... deviennent accessibles ! Essayez par exemple ceci :

```
a = 16
print(sqrt(a))
from math import *
print(sqrt(a))
```

Veracité/fausseté d'une expression

Une expression aboutissant à une valeur logique rend soit 0 ou faux, soit tout autre valeur ou vrai. Cf les rudiments de logique booléenne ! Essayez :

```
a = "coucou"
b = "coucou"
c = "cou"
d = (a == b)
print(d)
print(int(d))
d = (a == c)
print(d)
print(int(d))
print(not(d))
...
```

Modules turtle et xturtle

pour apprendre à programmer en créant des petits dessins

Le module turtle permet énormément de possibilités de dessin, même très simple. Par exemple, demander un nombre de côté et dessiner le polygone régulier correspondant,...

Pour ceux qui voudraient aller plus loin dans l'usage de turtle, il est possible d'utiliser une amélioration, le module xturtle, et les exemples qui l'accompagnent (attention, le module xturtle.py doit être dans le même répertoire que tout programme python qui y fait appel).

Ce module est disponible ici. Vous devez décompresser son contenu (essentiellement xturtle.py) dans C:\Python27\Lib\site-packages\xturtle0.95 Si vous travaillez sur un ordinateur pour lequel vous n'avez pas les droits d'écriture, vous pouvez les placer ailleurs. Il est alors nécessaire que le module xturtle.py et tout programme qui en fait usage soient dans le même répertoire.

Fonctions originales

Définir une fonction et ses **paramètres** :

```
def nomdelafonction(liste de paramètres):
    ...
    bloc d'instructions indenté
```

Les fonctions peuvent n'avoir aucun paramètre, mais ceux-ci sont souvent très utiles pour exécuter les instructions de la fonction.

Dans une autre partie (programme principale ou autre fonction), on fait appel à la fonction, en précisant des valeurs particulières qui seront affectées aux paramètres de la fonction. On parle alors des **arguments** de la fonction, qui peuvent être des constantes ou des variables. Une fonction au sens classique du terme renvoie une valeur qui sera reprise comme un “résultat” au niveau de la ligne qui a fait appel à la fonction.

Exemple (réalisable en mode interactif) :

```
def plus(a, b):
    return a + b
...
x = plus(3,9)
print(x)
plus(x,8)
print(x)
```

- Les **paramètres** décrivent ce qui est référencé dans la définition de la fonction
- Les **arguments** décrivent ce qui est “envoyé” à la fonction, là où elle est utilisée
- Chaque appel de la fonction peut utiliser des arguments différents

Variables locales et globales

Les variables définies à l'intérieur d'une fonction ne sont compréhensibles qu'au sein et à partir de cette fonction, on les appelles des variables locales.

Python peut lire, mais pas modifier **par affectation** les variables extérieures à l'espace local. C'est le

cas des variables passées en paramètres d'une fonction. Si une variable de même nom est créée en version locale, c'est seulement celle-ci qui sera connue localement.

Si une variable extérieure (en paramètre ou non) se voit appliquer une méthode, la variable est cependant modifiée.

Par opposition, les variables globales sont définies “à l'extérieur de la fonction” mais pourront être modifiées dans une fonction. Pour cela, on déclare à Python par le mot-clé **global** que la variable à utiliser (modifier par réaffectation) dans le corps de la fonction est globale.

Il est dangereux d'utiliser dans les grands programmes des variables globales, et il est préférable de n'utiliser que des variables locales et de transmettre explicitement via les paramètres de la fonction tout ce qui est extérieur et peut lui être utile. Sinon, il y a un risque de perturber le fonctionnement de la fonction à cause d'une modification non attendue de la variable globale.

Pour obtenir un dictionnaire des variables globales ou locales à une portée donnée (scope) du code, il suffit d'utiliser les méthodes “globals()” et “locals()”.

Cf.
http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-python/portee-des-variables-et-references, http://python-textbok.readthedocs.io/en/1.0/Variables_and_Scope.html

Tester aussi ce code :

variables_locales_globales.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def chipote(in1):
    in1 = in1 + 5
    print("in1 =", in1," dans chipote")
    print("a=",a," dans chipote")
    b = 8
    print("b=",b," dans chipote")
    c = a + 10
    print("c=",c," dans chipote")
    e = b + in1
    print("e=",e," dans chipote")
    li.append(b)
    global f
    f = f + 20

li=[1, 2, 3]
b = 4
a = 5
f = 13
print("f=",f)
chipote(a)
print("a=",a)
print("f=",f)
chipote(a)
```

```
print(li)
print("f=",f)
```

Passage d'arguments par tuples et dictionnaires

- Les arguments d'une fonction peuvent être transmis via un tuple en préfixant le nom du tuple par le symbole * (on utilise en général l'identifiant “*args” pour le tuple)
- Les arguments d'une fonction peuvent être transmis via un dictionnaire dont les clés correspondent aux arguments nommés dans la définition de la fonction, en préfixant le nom du dictionnaire par les ** (on utilise en général l'identifiant ”**kwargs“ pour le dictionnaire)

Passage par tuple

fonction_args_tuple.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def liste_args(*args):
    """ imprime les arguments passée en tuple (ordonnés)
    """
    print(args)

liste_args('pommes','poires','scoubidous','apples','peaches','cherries'
)
```

Output :

```
('pommes', 'poires', 'scoubidous', 'apples', 'peaches', 'cherries')
```

Passage par dictionnaire

fonction_args_dictionnaire.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def liste_kwargs(**kwargs):
    """ imprime les arguments passée en dictionnaire
    """
    for key, value in kwargs.items():
        print(key,':',value)
```

```
# premier type d'appel
liste_kwargs(entrée='homard', plat='poularde', dessert='tarte')

# second type d'appel
dico = {'entree': 'homard', 'plat': 'poularde', 'dessert': 'tarte'}
print(dico)
liste_kwargs(**dico)
```

Output :

```
entrée : homard
dessert : tarte
plat : poularde
{'entree': 'homard', 'dessert': 'tarte', 'plat': 'poularde'}
entree : homard
dessert : tarte
plat : poularde
```

Modules de fonctions

Des fonctions ou simplement des déclarations de variables peuvent être définies et regroupées dans un fichier (.py), et ensuite renseignées pour leur utilisation dans un programme grâce à la directive d'importation.

Exemples de modules de déclarations de variables

- Constantes physiques
- ...

Directive d'importation

Il y a 2 façons essentielles d'importer toutes les fonctionnalités définies dans un module dont le nom est “nomdemodule” :

1. import nomdemodule
2. from nomdemodule import *

Dans le premier cas, les fonctions seront appelables avec des noms tels que “nomdemodule.func1”, et les concepteurs des modules proposent souvent l'utilisation d'un alias par une directive recommandée telle que “import nomdemodule ad ndm”. Les appels sont alors du type “ndm.func1”.

Dans le deuxième mode, la même fonction sera utilisable avec le nom “func1”. Du fait de l'évolution de modules, il peut devenir difficile de détecter quelle fonction devient manquante avec ce mode d'appel. Avec cette deuxième méthode, il est préférable de de n'importer qu'une seule fonction particulière (ou les quelques nécessaires) au lieu de toutes celles qui sont présentes dans le module (avec *). Exemple :

- from nomdemodule import func13

Références :

- The Definitive Guide to Python import Statements
- Absolute vs Relative Imports in Python
- ...

Test sur le programme "main"

La variable python **__name__** contient **'__main__'** si l'instruction est invoquée dans le programme “principal” appelé ou contient le nom du module si cette inscription est présente au niveau d'un module appelable (donc importé).

Le rôle de la structure conditionnelle **if __name__ == '__main__':** incluse dans de nombreux modules de fonctions est de n'exécuter la suite du code **que** si le module/programme python concerné est le programme principal. Il se peut en effet que ce fichier soit appelé en tant que module par une directive d'importation écrite dans un autre programme. Dans ce dernier cas, le code qui suit la ligne **if __name__ == '__main__':** ne sera pas lancé, mais toutes les fonctions définies seront reconnues et utilisables par le programme appelant !

> Pour bien comprendre l'utilité de ce test, décompressez les 4 fichiers python de cette archive, étudiez-les et faites les fonctionner !

Espaces de noms (namespaces)

Lorsque l'interpréteur Python exécute “import nomdemodule”, l'environnement crée un espace de noms “nomdemodule”, contenant les variables et fonctions du module nomdemodule, ce qui permet de regrouper ces fonctions et variables sous un préfixe unique et spécifique, qui explique les appels sous la forme “nomdemodule.func1”. C'est une garantie pour éviter tout conflit entre des fonctions portant des noms identiques, mais tirées de modules différents. Le prix à payer est de sans cesse devoir expliciter l'espace de nom. Celui-ci peut cependant être condensé lors de la directive d'importations :

- import nomdemodule as mdul

> - pour comprendre, essayez par exemple ces directives en mode interactif sur le module math
> - pour les programmes longs et utilisant plusieurs modules, choisissez d'utiliser les namespaces
> - ne mélangez jamais les deux techniques d'importation

Pour en savoir plus :

- http://effbot.org/zone/import-confusion.htm
- http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-python/a-la-decouverte-des-modules

Impression formatée de texte et nombres (print)

On peut créer facilement une sortie “imprimée” formatée de données en utilisant des paramètres de formatage de l'instruction print (inspirés de l'instruction printf du langage C).

Par exemple, si on veut sortir la ligne suivante en disposant de deux variables (t et nmol) :

```
Au temps = 0.6 heure, la réaction a consommé 1.23 moles de réactif.
```

On peut obtenir cette ligne par l'instruction suivante, en Python 3 (print sous forme d'une fonction) :

```
print('Au temps =%g heure, la réaction a consommé %.2f moles de réactif.' %
(t,nmol))
```

Lors de l'exécution, les symboles '%' rencontrés sont remplacés par les variables qui suivent, en utilisant les spécifications de formatage :

- %g : notation décimale ou scientifique compacte
- %.2f : notation flottante avec 2 chiffres après le point décimal.

Autres spécifications :

- %s : chaîne (string)
- %d : entier

Le code “\n” inséré dans une chaîne permet d'effectuer un retour à la ligne.

Depuis Python 2.6, le langage a introduit la méthode .format() qui est plus sophistiquée et ne présente pas les défauts des techniques utilisant les ”%“ ! (Fix Me! : écrire une page spécifique).

→ Impressions avec la méthode .print()

Fix Me! : cf. https://datawhatnow.com/things-you-are-probably-not-using-in-python-3-but-should/ (f-strings vs format)

Références :

- documentation Python : http://docs.python.org/2/library/stdtypes.html#string-formatting-operations
- http://python.developpez.com/cours/DiveIntoPython/php/frdiveintopython/native_data_types/formatting_strings.php
- http://stackoverflow.com/questions/5082452/python-string-formatting-vs-format

- http://www.python-course.eu/python3_formatted_output.php

Lire et écrire dans des fichiers

L'instruction

```
f = open(filename, mode)
```

permet de traiter un fichier du système d'exploitation “filename”, avec un “mode” pouvant prendre différentes valeurs :

- “r” : en lecture seule
- “w” : en écriture
- “a” : en écriture, mais au delà de ce qui existe déjà (append)

Une fois le fichier ouvert, l'instruction “data = f.read()” lit l'entièreté du fichier et la stocke dans la variable de caractères data. “f.readline()” effectue la lecture d'une ligne à la fois. En écriture, le contenu d'une chaîne de caractère “dataw” est écrite dans le fichier par l'instruction “f.write(dataw)”.

Une fois l'utilisation achevée du fichier, il est important de fermer le fichier par la commande “f.close()” des erreurs peuvent subvenir sur le système de fichier si ce n'est pas fait correctement pour les fichiers en écriture surtout.

Il est recommandé d'ouvrir le fichier en utilisant la commande “with”, qui garantit la fermeture du fichier après exécution du code indenté, même en cas d'erreur :

```
with open("fichier.txt", "r") as fic:
    contenu_fic = fic.read()
```

> Les données de tout type devant être transformées en chaînes de caractères, il n'est pas aisé de stocker dans des fichiers des données composites comprenant des entiers, listes, dictionnaires,... Dans ce cas, une des solutions les plus simples consiste à utiliser le module json codant et décodant les données dans le format répandu JSON.

Pour en savoir plus, consultez la page https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files.

Gestion des erreurs



- https://realpython.com/python-keyerror/
- ...

Gestion de l'encodage des caractères en Python

Le système de codage des caractères a considérablement évolué. Aux débuts de l'informatique, le codage était initialement limité à l'ASCII sur 7 bits avec 128 caractères possibles, ne permettant donc d'écrire valablement qu'en anglais, sans caractères spéciaux, sans lettres accentuées. Écrire des programmes en Python (ou dans d'autres langages) en utilisant exclusivement ces caractères est une façon simple d'éviter toute difficulté.

Cette limitation a donné lieu à des encodages parfois bricolés, propriétaires et mal conçus pendant plusieurs décennies, et dont il subsiste encore des traces dans de nombreux logiciels et documents sauvegardés. Citons pour ce qui concerne l'Europe de l'ouest les encodage cp1252, applemac et Latin1 (ou iso8859-1). Ces encodages permettent quelques caractères supplémentaires, mais ne sont pas satisfaisants.

Une solution universelle aux difficultés d'encodage a été de créer une norme définissant les caractères utilisables au niveau mondial, l'unicode, permettant de décrire des dizaines de milliers de caractères différents, et d'un encodage efficace pour les usages majoritaires (UTF-8 pour la plupart des pays utilisant des caractères latins, et UTF-16 pour les autres).

Il est recommandé de spécifier les caractères utilisés pour le codage du programme via une ligne à placer en début de programme, comme une des suivantes par exemple :

```
# -*- coding: utf-8 -*-
# -*- coding: iso-8859-1 -*-
# -*- coding: cp1252 -*-
...
```

Les mots-clés du langage n'utilisent que les caractères du jeu historique ASCII. Un programme Python peut utiliser des chaînes de caractère utilisant un codage sur plus d'un octet (unicode). Python 3 utilisant des techniques différentes de Python 2 pour cela, et vu l'obsolescence progressive de cette dernière branche, les traitements particuliers de codage/décodage ne seront pas explicités. Sinon, *cf.* cette référence.

La complexité algorithmique

Un algorithme est l'énoncé dans un langage bien défini d'une suite d'opérations permettant de résoudre par calcul un problème. Cette résolution nécessite lors de son implémentation un certain temps de calcul, une certaine quantité de mémoire. La dépendance de ces quantités en le(s) paramètre(s) qui régissent la taille d'un problème constitue la complexité algorithmique, en temps ou en mémoire. Si des problèmes à résoudre traitent un grand nombre de donnée, ou sont répétés très souvent, Il est particulièrement important de sélectionner une méthode de résolution, un algorithme, de la meilleure complexité possible.

Voici quelques complexités classiques et quelques exemples d'application :

Ecriture symbolique	Type de complexité	Exemples
O(1)	complexité indépendante de la taille de la donnée	
O(log(n))	complexité logarithmique	
O(n)	complexité linéaire	Système d'équation tridiagonal
O(nlog(n))	complexité quasi-linéaire	Transformée de Fourier rapide (FFT), utilisée en spectroscopie et compression de sons (mp3) et images (jpeg). Tri rapide (quicksort)
O(n2)	complexité quadratique	Transformée de Fourier discrète, Tris à bulle, par insertion ou par sélection Résolution d'un système d'équation linéaire triangulaire par substitution.
O(n3)	complexité cubique	Système d'équation par méthode de Gauss (triangularisation) ou Gauss-Jordan (diagonalisation). Séparation en matrices triangulaires.
O(np)	complexité polynomiale	
O(nlog(n))	complexité quasi-polynomiale	
O(2n)	complexité exponentielle	Problème des tours de Hanoï
O(n!)	complexité factorielle	Calcul d'un déterminant par la méthode des cofacteurs associés

Jusqu'à la complexité polynomiale, les algorithmes peuvent être qualifiés d'efficaces.

Les stratégies mises en oeuvre pour créer des algorithmes incluent :

- les itérations (répétitions n fois d'un bloc d'instructions)
- la récursivité. Un problème de taille n se ramène à traiter un ou plusieurs problèmes de taille (n-1).
- Diviser pour régner (divide and conquer). Un problème de taille n est décomposé en plusieurs problèmes, par exemple 2 de taille n/2.

Les données manipulées peuvent s'organiser en listes, piles, files, queues, listes chaînées, arbres (avec feuilles, noeud, branches,...), tandis que les algorithmes utilisent des relations logiques, une grammaire, des structures, des quantificateurs,...

Références : http://fr.wikipedia.org/wiki/Théorie_de_la_complexité_des_algorithmes

Introspection

Un programme Python peut s'examiner lui-même :

test-introspection.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
```

```
Chaîne de caractère multiligne mise en commentaire pour décrire un
programme.
2ème ligne
3ème ligne
fin de la documentation
"""

print("Exemple de chaîne imprimée dans le programme")

print(__name__)

print(__name__.__doc__)

print(__doc__)
```

Lignes directrices pour l'écriture de code Python

Résumé des principales recommandations de la pep8

> Les règles étant faites pour ne pas être respectées systématiquement. Surtout si leur respect pose un problème de compatibilité,...

Espaces

- Entourer les opérateurs d'espaces, sauf pour des groupements mathématiques et dans les arguments/paramètres de fonctions, et pas à l'intérieur de parenthèses, crochets ou accolades. Exemples :

```
variable = 'valeur de variable'
if a == b:
6 / 3
d = a*x + b
f = (1+x) * (1-x)
def fonction(arg='val'):
[x**2 for x in range(20)]
```

Retours à la ligne

- limiter les lignes à 79 caractères (utiliser les retours à la ligne, l'indentation, le backslash \)
- Séparer par des lignes vides les fonctions (2), classes (2) et méthodes (1)
- Utiliser une ligne par directive d'importation

caractères

- indentation par 4 espaces (pas de tabulation !)
- encodage en utf-8, à mentionner via # -*- coding: utf-8 -*-
- Utiliser des triples “quotes” ”“” pour des doctrings (commentaires en début de fonction, classe,...)

Noms de variable

- boucle, indices : utiliser un seul caractère minuscule
- utiliser uniquement des minuscules et le caractère de soulignement (underscore “_”) pour les modules, variables, fonctions et méthodes
- Pour les “constantes” : majuscules et underscore
- Noms de classe en CamelCase

Outils de vérification automatique

- pycodestyle
- pyflakes

Référence :

- Code Style Checks in Python

From:
https://dvillers.umons.ac.be/wiki/ - **Didier Villers, UMONS - wiki**

Permanent link:
https://dvillers.umons.ac.be/wiki/teaching:progappchim:notions_fondamentales?rev=1575544823

Last update: **2019/12/05 12:20**