Analyse et Programmation avec Python

Janvier 2020

Enseignant : Jaafar Chaaouri

Email: Jaafar.chaaouri@fsm.rnu.tn

Le langage Python [2/2]

- Opérateurs
- Structures de contrôle
- Fonctions
- Exceptions & gestionnaires de contexte
- Compréhensions de listes & expressions génératrices
- Modules
- Bonnes pratiques

Opérateurs

Arithmétiques

```
+, -, *, /, //, %, **
```

Les classiques se comportent "normalement", pas besoin d'entrer dans les détails.

Particularités de la division :

```
# Avec des nombres entiers
print(16 / 3)  # Quotient de la division euclidienne (produit un réel)
print(16 // 3) # Quotient de la division euclidienne (produit un entier)
print(16 % 3)  # Reste de la division euclidienne (produit un entier)

# Avec des nombres flottants
print(16. / 3)  # Division (produit un réel)
print(16. // 3) # Quotient de la division (produit un réel)
print(16. % 3)  # Reste de la division ou modulo (produit un réel)
```

Puissance :

```
print(2 ** 10)
# On peut aussi utiliser la fonction pow() du module math, mais celle-ci renvoie un réel...
import math
print(math.pow(2, 10))
```

Logiques

```
and, or, not
```

retournent une valeur booléenne.

```
print(True or False)
print(True and False)
print(not True)
print(not False)
```

```
True
False
False
True
```

Comparaison

`==, is, !=, is not, >, >=, <, <=`

L'évaluation de ces opérateurs retourne une valeur booléenne.

```
print(2 == 2)
print(2 != 2)
print(2 == 2.0)
print(type(2) is int)
```

```
True
False
True
True
```

On peut utiliser ces opérateurs avec des variables et des appels à des fonctions.

```
x = 3
print(1 > x)
y = [0, 1, 42, 0]
print(x <= max(y))
print(x <= min(y))
```

On peut chaîner ces opérateurs :

```
x = 3
print(2 < x <= 9) # équivalent à 2 < x and x <= 9
```

```
True
```

Attention : comparer des types non numériques peut avoir des résultats surprenants.

```
# Chaînes de caractères
print("aaa" < "abc")
print("aaa" < "aaaa")
print("22" > "3.0")
```

```
True
True
False
```

Attention : comparer des types incompatibles peut avoir des résultats surprenants.

```
# Cette cellule génère des erreurs
print('chaîne:\t', "a" < 2)
print('liste:\t', ["zogzog"] > 42)
print('vide:\t', [] > 1)
print('tuple:\t', [23, 24] >= (23, 24))
print('dict:\t', [23, 24] >= {23: True, 24: "c'est pas faux"})
```

Attention, l'égalité de valeur n'implique pas forcément que l'identité des objets comparés est la même.

```
a = []  # a est une liste vide
b = []  # b est une AUTRE liste vide

print(a == b) # test de valeur
print(a is b) # test d'identité
print('id(a) =', id(a))
print('id(b) =', id(b))
```

Mais, comme vu précédemment, des variables différentes peuvent référencer le même objet.

```
c = a
print(a == c) # test de valeur
print(a is c) # test d'identité
```

```
True
True
```

bits à bits *(bitwise)*

`|, ^, &, <<, >>, ~`

Ces opérateurs permettent de manipuler individuellement les bits d'un entier.

Ce sont des opérations bas-niveau, souvent utilisées pour piloter directement du matériel, pour implémenter des protocoles de communication binaires (par exemple réseau ou disque).

L'utilisation d'entiers comme ensemble de bits permet des encodages de données très compacts, un booléen (True, False) ne prendrait qu'un bit en mémoire, c'est a dire que l'on peut encoder 64 booléens dans un entier.

Description complète ici.

```
val = 67 # == 64 + 2 + 1 == 2**6 + 2**1 + 2**0 == 0b1000011
print(bin(val))

mask = 1 << 0 # On veut récupérer le 1er bit
print('le 1er  bit vaut', (val & mask) >> 0)

mask = 1 << 1 # On veut récupérer le 2ème bit
print('le 2ème bit vaut', (val & mask) >> 1)

mask = 1 << 2 # On veut récupérer le 3ème bit
print('le 3ème bit vaut', (val & mask) >> 2)

mask = 1 << 6 # On veut récupérer le 7ème bit
print('le 7ème bit vaut', (val & mask) >> 6)

# Si on positionne le 4ème bit a 1 (on rajoute 2**3 = 8)
newval = val | (1 << 3)
print(newval)

# Si on positionne le 6ème bit a 0 (on soustrait 2**7 = 64)
print(newval & ~(1 << 6))
```

> **Exercice :** Retournez une chaîne de caractères représentant le nombre contenu dans `var` écrit en notation binaire.
>
> Par exemple:
>
> 5 -> '101'
>
> 6 -> '110'
>
> 7 -> '111'

```
var = 7
# votre code ici


# Solution simple :

var = 7
print((var >> 0) & 1, (var >> 1) & 1, (var >> 2) & 1)

# Solution réutilisable :

def num2bin(num):
    ret = []
    while num > 0:
        ret.append(str(num & 1))
        num >>= 1
    return ''.join(reversed(ret))

print(num2bin(6))
print(num2bin(7))
print(num2bin(8))
print(num2bin(42))
```

Assignation augmentée

```
 +=, -=, /=
```

```
a = 4
a += 1  # <=> a = a + 1
print(a)
a //= 2
print(a)
a **= 3
print(a)
a %= 2
print(a)
```

Compatibilité de type, conversion de type

Python effectue certaines conversions implicites, quand cela ne perd pas d'information (par exemple d'entier vers flottant).

```
print(1 + 0b1)
print(1 + 1.0)
print(1.0 + 2 + 0b11 + 4j)
```

```
2
2.0
(6+4j)
```

Mais dans d'autres cas, la conversion doit être explicite.

```
# Cette cellule génère une erreur
print(1 + '1')
```

Exercice :

Sans enlever le '+':

1. Corrigez le code de la cellule ci dessus, afin d'afficher la chaine 11
2. Corrigez le code de la cellule ci dessus, afin d'afficher le nombre 2
3. Corrigez le code de la cellule ci dessus, afin d'afficher le nombre 11

```
# Solutions
# 1.
print('1' + '1')    # On rajoute les apostrophes manquantes

# 2.
print(1 + 1)        # On enlève les apostrophes
print(1 + int('1')) # On convertit en entier

# 3.
print(int('1' + '1')) # On fait les deux
print(10 + 1)         # On improvise
```

Priorité des opérateurs

Les opérateurs ont en python des priorités classiques.

Par exemple, dans l'ordre :

1. puissance : `**`

- multiplication, division : `*` et `/`
- addition, soustraction : `+` et `-`

etc...

Utilisez des parenthèses quand cela aide à la lisibilité et à la clarté.

Une priorité explicitée avec des parenthèses est souvent plus facile à relire que s'il n'y en a pas et qu'il faut se remémorer les règles.

Pour plus d'informations, voir ici

Structures de contrôle

- La mise en page comme syntaxe
- `pass`
- tests conditionels : `if/elif/else`
- boucles
    - `for elt in liste`
    - `for idx in range(len(liste))`
    - `while`
    - `break`
    - `continue`

La mise en page comme syntaxe

- La mise en page est importante en python, c'est une différence majeure avec les autres langages (Java, C++, etc.)
- Python utilise l'indentation du code avec des caractères blancs plutôt que des mots clés (begin/end en pascal) ou des symboles (`{}` en java et C++). Cela permet de rendre le code plus compact.
- Elle va servir a délimiter des blocs de code sur lesquels les structures de contrôle comme les boucles ou les tests de conditions vont s'appliquer.
- De toute façon, dans les autres langages, on indente aussi le code pour l'aspect visuel et la lisibilité.
- L'indentation faisant partie de la syntaxe du langage, il faut être rigoureux et suivre la règle suivante :
    - 4 espaces pour passer au niveau suivant
    - éviter les tabulations et préférer les espaces et surtout ne pas les mélanger !

```
if True:
    print("toutes")
    print("les")
    print("lignes")
    print("au même niveau d'indentation forment un bloc de code")
print('et quand on remonte, on "termine" un bloc de code')
```

> **Exercice** : changez le True en False, et observez quelles lignes de code ne sont plus exécutées.

`Pass`

En cas de besoin d'un bloc de code qui ne fait rien, on utilise le mot clé `pass` (équivalent à NOP ou NO-OP)

Exemple : une boucle infinie

```
condition = True
while condition:
    pass
```

> (à ne pas exécuter sous peine de bloquer le noyau de ce notebook)

Tests conditionnels

Les instructions `if/elif/else` permettent d'exécuter des blocs d'instructions en fonction de conditions :

```
if test1:
    <bloc d instructions 1>
elif test2:
    <bloc d instructions 2>
else:
    <bloc d instructions 3>
```

`elif` est la contraction de "`else if`"

```
if True:
    print("c'est vrai!")
```

```
if False:
    print("je suis caché!")
else:
    print("mais moi je suis en pleine lumière...")
```

```
# Pour cet exemple, on itère sur les éléments d'un tuple (cf. boucle for plus loin)
for position in 2, 9, 3, 1, 8:
    if position == 1:
        print(position, "Or")
    elif position == 2:
        print(position, "Argent")
    elif position == 3:
        print(position, "Bronze")
    else:
        print(position, "Vestiaires")
```

```
taille = float(input("Votre taille ? "))
if taille >= 1.80:
    print('grand')
else:
    print('petit')
```

> **Exercices** :
>
> 1. Editez la cellule pour y mettre votre taille et exécutez-la pour savoir si vous êtes grand ou petit.
> 2. Gérez le cas des gens de taille moyenne.
> 3. Utilisez la fonction input() pour demander sa taille à l'utilisateur.

```
# Solution
taille = float(input('Quelle est votre taille ? '))

if taille >= 2.0:
    print('grand')
elif taille >= 1.80:
    print('moyen')
else:
    print('petit')
```

Boucles

Les boucles sont les structures de contrôle permettant de répéter l'exécution d'un bloc de code plusieurs fois.

Boucle `while`

La plus simple est la boucle de type `while` :

```
while <condition>:
    <bloc 1>
<bloc 2>
```

Tant que `<condition>` est True, le `<bloc 1>` est exécuté, quand la condition passe à False, l'exécution continue au `<bloc 2>`.

```
compteur = 3
while compteur > 0:
    print('le compteur vaut :', compteur)
    compteur -= 1
print('le compteur a été décrémenté 3 fois et vaut maintenant', compteur)
```

> **Exercice** :
> C'est la soirée du réveillon. Ecrivez une boucle `while` qui décompte les secondes à partir de `5` pour souhaiter la bonne année.
> Allez voir par ici pour passer le temps...

```
import time
# Votre code ici


# Solution
import time

secs = 5

while secs > 0:
    print(secs)
    secs -= 1
    time.sleep(1)

print('Happy new year !')
```

Boucle `for`

Une boucle plus complexe : `for/in`

```
for <variable> in <iterable>:
    <bloc 1>
<bloc 2>
```

A chaque tour de boucle, la variable `<variable>` va référencer un des éléménts de l'`<iterable>`. La boucle s'arrête quand tous les éléménts de l'itérable ont été traités. Il est fortement déconseillé de modifier l'itérable en question dans le `<bloc 1>`.

Si nous voulons itérer sur une liste mais avons besoin des indices liés a chaque élément, nous utilisons une combinaison de `range()` et `len()`.

```
nombres = [2, 4, 8, 6, 8, 1, 0, 1j]
print(list(range(len(nombres))))
for idx in range(len(nombres)):
    nombres[idx] **= 2
# Les carrés
print(nombres)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[4, 16, 64, 36, 64, 1, 0, (-1+0j)]
```

Il existe une forme raccourcie pour faire ce genre de choses, la fonction interne `enumerate()`

```
nombres = [2, 4, 8, 6, 8, 1, 0]
for (idx, item) in enumerate(nombres):
    nombres[idx] = bool(item % 2)
# Les impairs
print(nombres)
```

Instruction `break`

Il est possible d'arrêter prématurément une boucle grâce a l'instruction `break`.

L'instruction `break` est utilisable indifférement dans les boucles `for` ou `while`.

```
compteur = 3
while True:  # Notre boucle infinie
    compteur -= 1
    print('Dans la boucle infinie! compteur =', compteur)
    if compteur <= 0:
        break  # On sort de la boucle while immédiatement
    print('on contine, compteur =', compteur)
print("c'était pas vraiment une boucle infinie...")
```

En cas d'imbrication de plusieurs boucles, l'instruction `break` sort de la plus imbriquée (la plus proche).

```
for i in (1, 2, 3):
    for j in (1, 2, 3, 4):
        if i == 2:
            break
        print("i, j = {}, {}".format(i, j))
```

Instruction `continue`

Si, dans une boucle, on veut passer immédiatement à l'itération suivante, on utilise l'instruction `continue`.

```
compteur = 9
while compteur > 0:
    compteur -= 1
    if compteur % 2:
        compteur /= 2
        print('impair, on divise :', compteur)
        continue # retourne immédiatement au début de la boucle
    print("pair, RAS")
print("c'est fini...")
```

Fonctions

Les fonctions permettent de réutiliser des blocs de code plusieurs endroits différents sans avoir à copier ce bloc.

En python, il n'y a pas de notion de sous-routine. Les procédures sont gérées par les objets de type fonctions, avec ou sans valeur de retour.

```
def <nom fonction>(arg1, arg2, ...):
    <bloc d instructions>
    return <valeur>  # Instruction optionnelle
```

On distingue :

- les fonctions avec `return` des fonctions sans `return`
- les fonctions sans arguments (pour lesquelles `()` est vide) des fonctions avec arguments `(arg1, arg2, ...)`

`<nom fonction>(arg1, arg2, ...)` est appelé **signature** de la fonction.

Fonctions sans arguments

Fonction sans `return`

Pour définir une fonction :

```
def func():  # Definition de la fonction
    print('Je suis une fonction sans valeur de retour')
```

Pour utiliser une fonction que l'on a défini :

```
func()  # 1er Appel de la fonction
func()  # 2eme appel
func()  # 3eme appel, etc...
print(func())  # On l'appelle et on affiche sa valeur de retour
```

Fonction avec `return`

```
def func():  # Definition de la fonction
    return "I'm happy"  # La fonction retourne une chaine de caractère

print("1er appel:")
func()  # 1er Appel de la fonction : la valeur retournée n'est pas utilisée
print("2eme appel:")
ret_val = func()  # Le retour du 2eme appel est stocké
print("La fonction func() nous a renvoyé la valeur:", ret_val)
```

Fonctions avec arguments

Pour définir une fonction qui prend des arguments, on nomme ces arguments entre les parenthèses de la ligne `def`. Ces paramètres seront définis comme des variables à l'intérieur de la fonction et recevrons les valeurs passées lors des appels de celle-ci.

```
def somme(x, y):
    return x + y
```

Pour utiliser cette fonction avec diverses valeurs, il suffit de l'appeler plusieurs fois :

```
print(somme(1, 2))
print(somme(4, 7))
print(somme(2 + 2, 7))
print(somme(somme(2, 2), 7))
```

> **Exercice** :
>
> - Définissez une fonction nommée `chevalier`, qui prend un paramètre `n`, et qui répète `n` fois (affiche avec `print`) la chaîne de caractères `test!`
> - appelez cette fonction pour vérifier que `chevalier(3)` dit bien `test!` trois fois comme il convient !

```
def chevalier(n):
    # Votre code ici
    pass


# Vérifions que tout fonctionne bien:
chevalier(1)
chevalier(3)
chevalier(6)
```

```
# Solution
def chevalier(n):
    print('test!' * n)

chevalier(3)
```

```
test!test!test!
```

Utilisation de valeurs par défaut

```
def somme(x, y=1):
    return x + y

print(somme(1, 2))
```

Si sa valeur n'est pas spécifiée lors de l'appel, le paramètre `y` prend la valeur par défaut (ici : `1`)

```
print(somme(4))
```

Note : Les arguments ayant une valeur par défaut doivent être placés **en dernier**.

Utilisation des arguments par leur nom

Si les arguments sont explicitiment nommés lors de l'appel, leur ordre peut être changé :

```
print(somme(y=7, x=4))
```

Espace de nommage et portée des variables

1er exemple

On veut illustrer le mécanisme de l'espace de nommage des variables :

```
def func1():
    a = 1
    print("Dans func1(), a =", a)
```

```
def func2():
    print("Dans func2(), a =", a)

a = 2
func1()
func2()
print("Dans l'espace englobant, a =", a)
```

Cet exemple montre deux comportements :

1. Une variable définie localement à l'intérieur d'une fonction cache une variable du même nom définie dans l'espace englobant (cas de `func1()`).
2. Quand une variable n'est pas définie localement à l'intérieur d'une fonction, Python va chercher sa valeur dans l'espace englobant (cas de `func2()`).

2ème exemple

On veut illustrer le mécanisme de portée des variables au sein des fonctions :

```
def func():
    a = 1
    b=2
    print('Dans  func() : a =', a)

a = 2
print("Avant func() : a =", a)
func()
print("Après func() : a =", a)
```

```
Avant func() : a = 2
Dans  func() : a = 1
Après func() : a = 2
```

Les variables définies localement à l'intérieur d'une fonction sont détruites à la sortie de cette fonction. Ici, la variable `b` n'existe pas hors de la fonction `func()`, donc Python renvoie une erreur si on essaye d'utiliser `b` depuis l'espace englobant :

```
# Cette cellule génère une erreur
print(b)
```

Exercices sur les fonctions

Exercice 1

Ecrire une fonction `stat()` qui prend en argument une séquence d'entiers et retourne un tuple contenant :

- la somme
- le minimum
- le maximum

des éléments de la liste

```
def stat(a_list):
    # votre fonction
    pass

stat([1, 4, 6, 9])
```

```
# Solution
def stat(a_list):
    """return sum, min and max of a list as a tuple"""
    return sum(a_list), min(a_list), max(a_list)

print(stat([1, 4, 6, 9]))
```

Exceptions

Pour signaler des conditions particulières (erreurs, évenements exceptionnels), Python utilise un mécanisme de levée d'exceptions

```
# Cette cellule génère une erreur
raise Exception
```

Ces exceptions peuvent embarquer des données permettant d'identifier l'évenement producteur.

```
# Cette cellule génère une erreur
raise Exception('Y a une erreur')
```

La levée d'une exception interrompt le cours normal de l'exécution du code et "remonte" jusqu'à l'endroit le plus proche gérant cette exception.

Pour intercepter les exceptions, on écrit :

```
 try:
     <bloc de code 1>
 except Exception:
     <bloc de code 2>
```

```
try:
    print('ici ca fonctionne')
    # ici on détecte une condition exceptionnelle, on signale une exception
    raise Exception('y a un bug')
    print("on n'arrive jamais ici")
except Exception as e:
    # L'excécution continue ici
    print("ici on peut essayer de corriger le problème lié à l'exception : Exception({})".format(e))
print("et après, ça continue ici")
```

Exemple illustrant le mécanisme de remontée des exceptions d'un bloc à l'autre :

```
def a():
    raise Exception('coucou de a()')

def b():
    print('b() commence')
    a()
    print('b() finit')

try:
    b()
except Exception as e:
    print("l'exception vous envoie le message :", e)
```

Exercice

Ecrivez une fonction `openfile()` :

- qui demande à l'utilisateur un fichier a ouvrir
- qui gère correctement les fichiers inexistants.
- qui affiche la première ligne du fichier ouvert
- qui retourne une valeur booléenne indiquant que le fichier a été ouvert ou non.

> **Attention :** Python attend un nom de fichier complet. Sous Windows, les extensions de fichier sont cachées par défaut...

```
# Votre code ici
def openfile():
    pass

print(openfile())
```

```
# Solution
def openfile():
    filename = input('Donnez un chemin de fichier : ')
    try:
        f = open(filename, 'r')
        print(f.readline())
```

```
        f.close()
        return True
    except FileNotFoundError as err:
        # Ignore les erreurs lorsqu'on essaie d'ouvrir le fichier
        pass
    return False

print(openfile())
```

```
Donnez un chemin de fichier : bb
False
```

Pour plus d'informations sur les exceptions, lire ce tutoriel.

Les gestionnaires de contexte

Pour faciliter la gestion des obligations liées à la libération de ressources, la fermeture de fichiers, etc... Python propose des gestionnaires de contexte introduits par le mot-clé `with`.

```
from pathlib import Path  # Pour la compatibilité avec Windows
with open(Path('exos/interessant.txt'), 'r') as fichier_ouvert:
    # Dans ce bloc de code le fichier est ouvert en lecture, on peut l'utiliser normalement
    print(fichier_ouvert.read())
# Ici, on est sorti du bloc et du contexte, le fichier à été fermé automatiquement
```

```
# Cette cellule génère une erreur
print(fichier_ouvert.read())
```

> **Exercice** : Reprenez le code de l'exercice précédent, et utilisez `with` pour ne pas avoir à utiliser la méthode `close()`.

```
# Votre code ici
```

```
# Solution
def openfile():
    filename = input('Donnez un chemin de fichier : ')
    try:
        with open(filename, 'r') as f:
            print(f.readline())
            return True
    except FileNotFoundError as err:
        # Ignore errors while trying to open file
        pass
    return False

print(openfile())
```

Il est possible de créer de nouveaux gestionnaires de contexte, pour que vos objets puissent être utilisés avec `with` et que les ressources associées soient correctement libérées.

Pour plus d'informations sur la création de gestionnaires de contexte, voir la documentation.

Les compréhensions de listes

Python a introduit une facilité d'écriture pour les listes qui permet de rendre le code plus lisible car plus concis.

```
# On construit une liste ne contenant que les éléments pairs de la liste Liste1
Liste1 = list(range(10))
print(Liste1)
ListePaire = []
for i in Liste1:
    if (i % 2) == 0:
        ListePaire.append(i)
print(ListePaire)
```

```
# Ici, on fait la même chose en compréhension de liste
ListePaire = [i for i in Liste1 if (i % 2) == 0]
print(ListePaire)
```

Les expressions génératrices

C'est une forme d'écriture, très proche des compréhensions de listes, mais qui ne crée pas de nouvel objet liste immédiatement. Les items sont produits à la demande.

```
tuplePairs = (i for i in Liste1 if (i % 2) == 0)
print(tuplePairs)
print(list(tuplePairs))
```

Plus d'informations dans ce tutoriel.

Modules

- Python fournit un système de modularisation du code qui permet d'organiser un projet contenant de grandes quantités de code et de réutiliser et de partager ce code entre plusieurs applications.
- L'instruction `import` permet d'accéder à du code situé dans d'autres fichiers. Cela inclut les nombreux modules de la bibliothèque standard, tout comme vos propres fichiers contenant du code.
- Les objets (variables, fonctions, classes, etc.) du module sont accessibles de la manière suivante :

```
module.variable
module.fonction(parametre1, parametre2, ...)
```

Ou plus généralement :

```
module.attribut
```

```
# Pour utiliser les fonctions mathématiques du module 'math'
import math

pi = math.pi
print('{:.2f}'.format(pi))
print('{:.2f}'.format(math.sin(pi)))
```

Créer ses propres modules

Il suffit de placer votre code dans un fichier avec l'extension `.py` et vous pourrez l'importer comme module dans le reste de votre code :

Nom de fichier dans le sytème	Nom d'objet dans le code python
`mon_module.py`	`mon_module`

Cas 1 : le fichier module est directement importable

- **Cas 1.a :** le module est déjà installé dans l'environnement d'exécution
  - il fait partie de la bibliothèque standard
  - il a été installé (avec `conda`, avec `pip`, etc.)
- **Cas 1.b :** le fichier module se trouve dans le même répertoire que le script qui l'importe

On peut importer un module sous un autre nom (pour le raccourcir, en général) :

```
import mon_module as mm
mm.ma_fonction()
```

On peut importer un objet particulier d'un module :

```
from mon_module import ma_fonction
ma_fonction()
```

Ou encore

```
from mon_module import ma_fonction as my_func
my_func()
```

Cas 2 : le fichier module se trouve ailleurs

On peut distinguer deux cas d'usage :

- **Cas 2.a :** on veut aller chercher du code dans un autre projet python qui n'est pas installé dans l'environnement courant
- **Cas 2.b :** on travaille sur un gros projet structuré en modules stockés dans une arborescence de sous-répertoires.

Quelques modules de la stdlib

La bibliothèque standard de Python est incluse dans toute distribution de Python. Elle contient en particulier une panoplie de modules à la disposition du développeur.

`string`

- `find()`
- `count()`
- `split()`
- `join()`
- `strip()`
- `upper()`
- `replace()`

`math`

- `log()`
- `sqrt()`
- `cos()`
- `pi`
- `e`

`os`

- `listdir()`
- `getcwd()`
- `getenv()`
- `chdir()`
- `environ()`
- `os.path : exists(), getsize(), isdir(), join()`

`sys`

- `argv`
- `exit()`
- `path`

Mais bien plus sur la doc officielle de la stdlib !

Bonnes pratiques

Commentez votre code

- pour le rendre plus lisible
- pour préciser l'utilité des fonctions, méthodes, classes, modules, etc...
- pour expliquer les parties complexes

Documentez avec Docstring

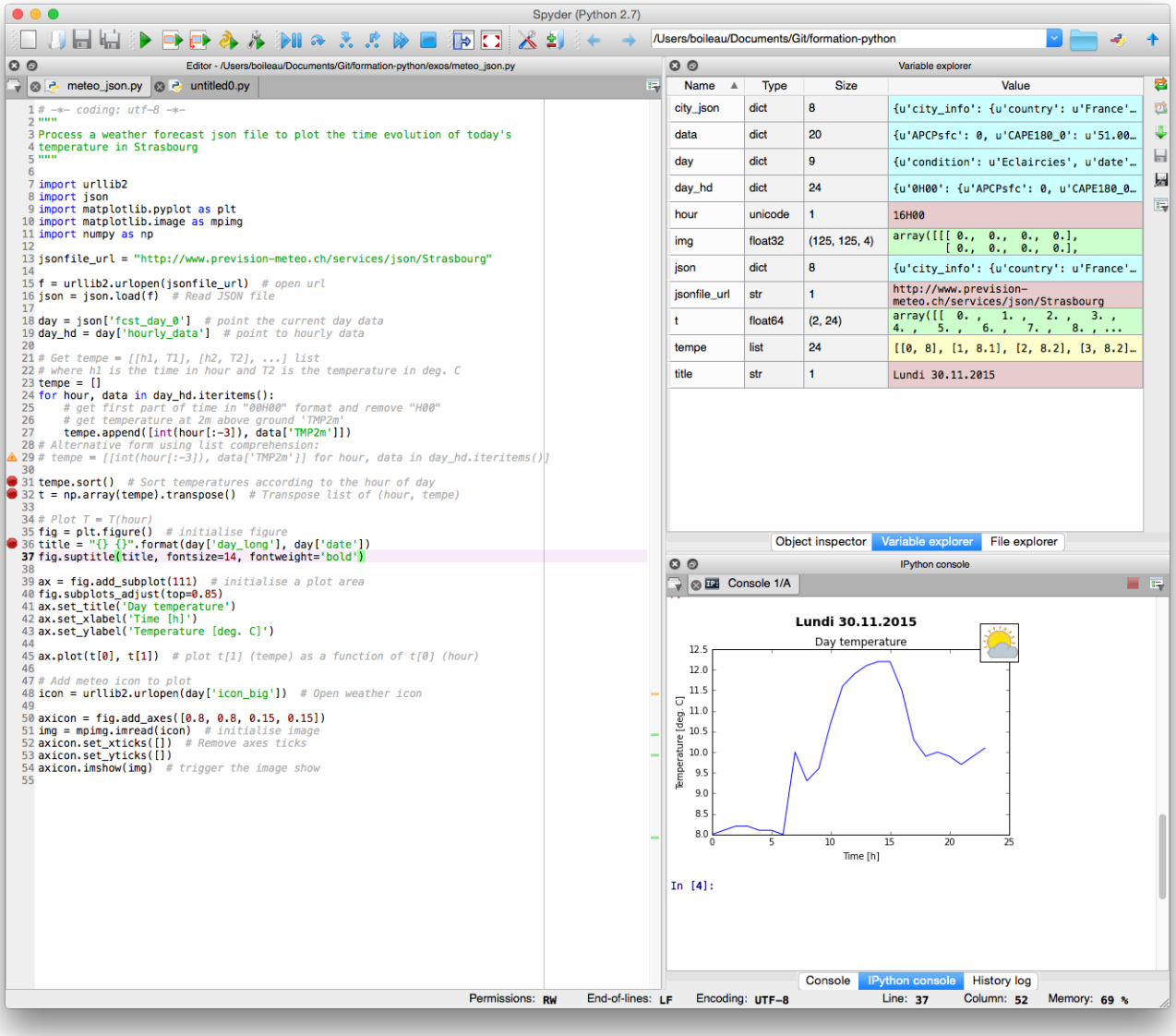
- Juste après la signature de la fonction, on utilise une chaîne de caractère appelée *docstring* délimitée par `""" """`
- la *docstring* permet de documenter la fonction au plus près de sa définition
- cette *docstring* est affichée par `help(fonction)`
- la *docstring* est utilisée par les outils de documentation automatique comme Sphinx.

```
def nrien(n):
    """Retourne n fois 'rien'"""
    return 'rien' * n

help(nrien)
```

```
Help on function nrien in module __main__:

    nrien(n)
        Retourne n fois 'rien'
```

Utiliser les environnements de développement intégrés :

- pydev
- eclipse
- spyder
- pycharm



Tests en Python

En programmation, on utilise des tests de non régression pour vérifier que les nouveaux développements et les corrections de bugs n'entraînent pas de pertes de fonctionalités et de nouveaux bugs. On distingue généralement :

- les tests unitaires (comportement de fonctions prises séparément)
- les tests d'intégration (interaction entre plusieurs parties de programme)
- les tests du système complet

Python dispose de nombreux modules dédiés aux deux premières catégories. Quelques exemples :

- `unittest` : fait partie de la bibliothèque standard
- `doctest` : le test est basé sur des sorties de sessions interactives stockées généralement dans la *docstring* de la fonction testée (alourdit le code...)
- `nose` : une extension de `unittest`
- `pytest` : syntaxe simple et nombreuses fonctionnalités

Une synthèse des outils existants sur cette page.

Exercices complémentaires

Chaines de caractères

Ecrivez les fonctions suivantes, sans utiliser upper() / lower():

- `majuscule('azERtyUI') -> 'AZERTYUI'`
- `minuscule('azERtyUI') -> 'azertyui'`
- `inverse_casse('azERtyUI') -> 'AZerTYui'`
- `nom_propre('azERtyUI') -> 'Azertyui'`

> **Indice:** cet exercice s'apparente à de la "traduction"...

```
def majuscule(chaine):
    # Votre code ici
    pass

def minuscules(chaine):
    # Votre code ici
    pass

def inverse_casse(chaine):
    # Votre code ici
    pass

def nom_propre(chaine):
    # Votre code ici
    pass

assert majuscule('azERtyUI') == 'AZERTYUI'
assert minuscule('azERtyUI') == 'azertyui'
assert inverse_casse('azERtyUI') == 'AZerTYui'
assert nom_propre('azERtyUI') == 'Azertyui'


# Solution

lower = 'abcdefghijklmnopqrstuvwxyz'
upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

toupper = dict(zip(lower, upper))
tolower = dict(zip(upper, lower))

inverse = dict(toupper)
inverse.update(tolower)

def to_case(chaine, dico):
    ret = []
    for c in chaine:
        d = dico.get(c, None)
        ret.append(d or c)
    return ''.join(ret)

def majuscules(chaine):
    return to_case(chaine, toupper)

def minuscules(chaine):
    return to_case(chaine, tolower)

def inverse_casse(chaine):
    return to_case(chaine, inverse)

def nom_propre(chaine):
    return majuscules(chaine[0]) + minuscules(chaine[1:])

print(majuscules('aTtEnTioN'))
print(minuscules('aTtEnTioN'))

print(inverse_casse('aTtEnTioN'))
print(nom_propre('aTtEnTioN'))
```

Récursion

Les fonctions dites "récursives" sont des fonctions qui font appel à elles-même, en résolvant une partie plus petite du problème à chaque appel, jusqu'à avoir un cas trivial à résoudre.

Par exemple pour calculer : "la somme de tout les nombres de 0 jusqu'à x", on peut utiliser une fonction récursive:

La somme de tous les nombres de 0 à 10 est égale à 10 plus la somme de tous les nombres de 0 à 9, etc...

```
def sum_to(x):
    if x == 0:
        return 0
    return x + sum_to(x - 1)


print(sum_to(9))
```

```
45
```

La fonction mathématique factorielle est similaire, mais calcule : "le produit de tout les nombres de 1 jusqu'à x".

```
def fact(x):
    if x == 1:
        return 1
    return x * fact(x - 1)


print(fact(5), fact(9))
```

La fonction mathématique qui calcule [la suite des nombres de Fibonacci](), peut être décrite de la façon suivante :

- `fibo(0) = 0`
- `fibo(1) = 1`

Et pour toutes les autres valeurs :

- `fibo(x) = fibo(x - 1) + fibo(x - 2)`

> **Exercice** : écrivez une fonction récursive `fibo(x)` qui renvoie le x-ième nombre de la suite de Fibonnaci.

```
def fibo(x):
    # Votre code ici
    pass


print(fibo(9))
```

```
# Solution
def fibo(x):
    if x < 2:
        return 1
    return fibo(x - 1) + fibo(x - 2)

print(fibo(8))
```

```
34
```