Initiation à la programmation avec Python (v3)

Cours n°5

Copyright (C) 2015-2019

Jean-Vincent Loddo

Licence Creative Commons Paternité
Partage à l'Identique 3.0 non transposé.

Sommaire du cours n°5

- Retour sur l'appel de fonction :
  - Fonctions avec paramètres **nommés** et **optionnels** en Python
- Notion n°10 : Les **exceptions** en programmation
- Notion n°11 : La vérité sur le stockage des variables : **Environnement**, **Mémoire**, **Adresses**, **Pointeurs**
- Notion n°12 : La vérité sur l'appel de fonction : Environnement **global** et **local**

Retour sur l'appel de fonction (1) : **Paramètres nommés**

- Nous savons qu'une fonction s'utilise (s'appelle) en écrivant son nom suivi du n-uplet de ses arguments ; l'association entre paramètres **formels** et **actuels** se fait donc « **par position** »

```
def foo(a,b,c,d): return((a + b*c) * 100 / d)
...
x = foo(1,2,3,4)     # Par position : a=1 b=2 c=3 d=4  donc  x=175
y = foo(1,2,3,1)     # Par position : a=1 b=2 c=3 d=1  donc  y=700
```

- En Python, comme dans d'autres langages, il est possible de rendre explicite l'affectation **formel** = **actuel**, mais il faut alors écrire :
  - **en premier** les actuels **par position**
  - **en dernier** toutes les affectations explicites **formel = actuel**, **dans n'importe quel ordre**
- Cette modalité d'appel se dit « **par argument nommé** » Exemple :

```
x = foo(1,2,3,d=4)              # Par position : a=1 b=2 c=3  Nommés : d=4      donc  x=175
y = foo(1,2,c=3,d=1)            # Par position : a=1 b=2      Nommés : c=3 d=1  donc  y=700
y = foo(c=3,d=1,a=1,b=2)        # Nommés       : a=1 b=2 c=3 d=1                donc  y=700
```


Retour sur l'appel de fonction (2) : **Paramètres (nommés) optionnels**

- En Python, comme dans d'autres langages, il est aussi possible de définir une **valeur par défaut** pour certains paramètres
- De cette façon, on pourra éviter de les fournir au moment de l'appel, autrement dit, ces paramètres seront **optionnels** (fournis de façon optionnelle)
- Dans le n-uplet des formels de la définition il faudra écrire les affectations **formel** = **valeur-par-défaut** dans le bon ordre
  - **en premier** les formels **non optionnels**
  - **en dernier** les formels **optionnels**

```
def foo(a,b,c,d=1): return((a + b*c) * 100 / d)       # d est optionnel (par défaut égal à 1)
...
x = foo(1,2,3,4)              # Par position : a=1 b=2 c=3 d=4  donc  x=175
y = foo(1,2,3,1)              # Par position : a=1 b=2 c=3 d=1  donc  y=700
y = foo(1,2,3)                # Par position : a=1 b=2 c=3  Par défaut : d=1  donc  y=700

y = foo(c=3,d=1,a=1,b=2)      # Nommés       : a=1 b=2 c=3 d=1               donc  y=700
y = foo(c=3,a=1,b=2)          # Nommés       : a=1 b=2 c=3  Par défaut : d=1  donc  y=700
```

Notion n°10, les « **exceptions** » (1)

- Il est possible de **récupérer le contrôle** de l'exécution lorsqu'une situation d'erreur ou exceptionnelle survient
  - **Division par zéro**, **conversion de type** impossible, accès à un **indice inexistant** (listes, n-uplets) ou à une **clef inexistante** (dictionnaires), tentative d'**affectation** d'une structure **non mutable** (tuple, chaîne), ouverture d'un **fichier non existant** ou sans droits suffisants,...
  - Si tout se passe bien, les actions « de rattrapage » sont ignorées
  - Si quelque chose tourne mal (une « exception est soulevée »), alors les actions de rattrapage (ou de secours) sont exécutées
- **Syntaxe simplifiée :**

```
try:
    ACTIONS1
except:
    ACTIONS2
```

Actions **ordinaires**, pouvant « soulever » une excéption

Actions **extraordinaires** (de rattrapage), exécutées seulement si une excéption est soulevée dans les $ACTIONS_1$

Notion n°10, les « **exceptions** » (2)

- Il est possible de **récupérer le contrôle** de l'exécution lorsqu'une situation d'erreur ou exceptionnelle survient
- Exemple n°1 (`ValueError`) :

```
try:
  s = input("Quel entier svp ? ")        # s est une chaîne de caractères
  i = int(s)                             # conversion string → int
except:                                  # exception « ValueError »
  print("Désolé mais %s n'est pas un entier !" % s)
  i = 42                                 # cette valeur conviendra !
```

- Exemple n°2 (`IndexError`) :

```
try:
  i = int(input("Quel indice d'étudiant ? "))    # danger conversion !
  e = ETUDIANTS[i]                               # danger accès !
except:                                          # ValueError ou IndexError
  e = ETUDIANTS[0]                               # le premier conviendra !
```


Notion n°10, les « **exceptions** » (3)

- Il est possible de **récupérer le contrôle** de l'exécution lorsqu'une situation d'erreur ou exceptionnelle survient
- Exemple n°3 (différencier les exceptions) :

```
def choix_etudiant(ETUDIANTS):
  try:
    i = int(input("Quel indice d'étudiant ? "))     # danger conversion !
    e = ETUDIANTS[i]                                # danger accès !
    return(e)
  except ValueError:
    print("L'indice doit être un nombre entier !")
    return (choix_etudiant(ETUDIANTS))              # appel récursif
  except IndexError:
    n = len(ETUDIANTS)
    print("L'indice doit être un nombre entre 0 et %d !" % (n-1))
    return (choix_etudiant(ETUDIANTS))              # appel récursif
  except:
    return (ETUDIANTS[0])    # le premier conviendra pour tous les autres cas!
```


Notion n°10, les « **exceptions** » (4)

- En Python, comme dans d'autres langages, les exceptions correspondent à des **valeurs** qui précisent la nature de l'erreur

`Value_error, Index_error, EOFError, ZeroDivisionError, IOError,...`

- **Syntaxe générale :**

```
try:
    ACTIONS0
[except EXPR1: ACTIONS1]
...
[except EXPRn: ACTIONSn]
[except: ACTIONSn+1]
```

Actions **ordinaires**, pouvant « soulever » une excéption

Actions **extraordinaires** (de rattrapage) : les $ACTIONS_i$ sont exécutées seulement si une excéption est soulevée dans $ACTIONS_0$ et si l'expression $EXPR_i$ correspond à l'exception soulevée

Remarque : pour **regrouper** le traitement de plusieurs exceptions à la fois, les expressions $EXPR_i$ après le mot clef **except** pourront aussi être des **n-uplets** d'exceptions, par exemple `(Value_error, Index_error)`

Notion n°10, les « **exceptions** » (5)

- En Python, comme dans d'autres langages, les exceptions correspondent à des **valeurs** qui précisent la nature de l'erreur

  `Value_error, Index_error, EOFError, ZeroDivisionError, IOError,...`

- Il existe aussi une instruction, qui s'appelle **raise**, qui permet de soulever manuellement une exception

- **Syntaxe**

  `raise EXPR`

  Expression d'exception

- **Exemple** (regrouper plusieurs types d'erreur) **:**

```
def choix_etudiant(ETUDIANTS):
  try:
    i = int(input("Quel indice d'étudiant ? "))  # danger conversion !
    e = ETUDIANTS[i]                             # danger accès !
    return e
  except:
    raise Value_error          # une seule exception, peut-être rattrapée ailleurs...
```


Notion n°11 : La vérité sur le stockage des variables **Environnement**, **Mémoire**, **Adresses**, Pointeurs (1)

- Souvenirs du premier cours :
  - Pour traiter l'information que le programmeur connaît mais surtout celle qu'il **ne connaît pas**, les langages de programmation proposent les « variables »
  - Les variables sont des boîtes qui ont un **nom** et un **contenu**

```
TOTO = "salut le monde"
Y = 16
X = 3.14159
```

  - Le contenu est une **valeur**, c'est-à-dire une information traitée par le langage de programmation
- Ce n'est pas faux, juste un peu simpliste...
  - La réalité est un peu plus complexe : il existe une **mémoire** qui permet de stocker une **valeur** à une certaine **adresse**, et un catalogue des noms (**environnement**) qui donne les adresses des valeurs associées

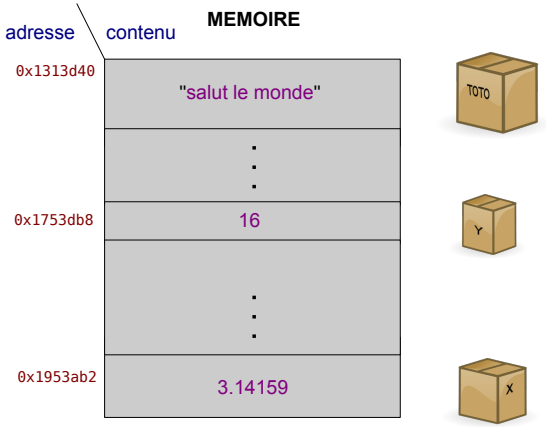
Notion n°11 : La vérité sur le stockage des variables **Environnement**, **Mémoire**, **Adresses**, Pointeurs (2)

- Il existe une **mémoire** qui permet de stocker une **valeur** à une certaine **adresse**, et un catalogue des noms (**environnement**) qui donne les adresses des valeurs associées

CATALOGUE des noms (**ENVIRONNEMENT**)

nom	adresse
TOTO	0x1313d40
Y	0x1753db8
X	0x1953ab2

Remarque :
Les « boîtes » ou « box » (emplacements) en mémoire peuvent être de tailles différentes (en nombre d'octets)



Notion n°11 : La vérité sur le stockage des variables Environnement, Mémoire, Adresses, **Pointeurs** (3)

- Pour comprendre ce que sont les **pointeurs**, supposons de faire l'affectation suivante :

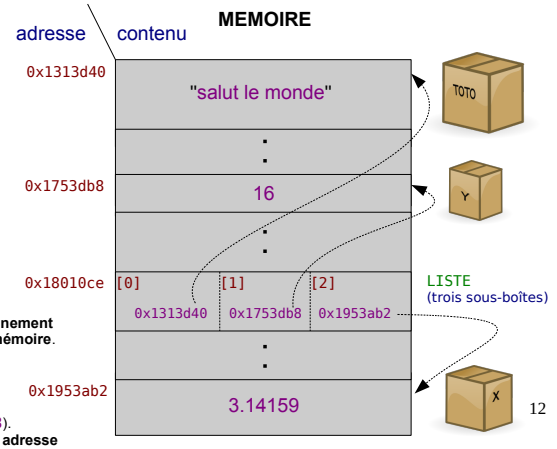
```
LISTE = [TOTO, Y, X]
```

CATALOGUE des noms (**ENVIRONNEMENT**)

nom	adresse
TOTO	0x1313d40
Y	0x1753db8
X	0x1953ab2
LISTE	0x18010ce

Remarque :
Il y a des **noms**,
tels que TOTO, Y, X, LISTE
et des **sous-noms**,
tels que LISTE[0], LISTE[1], LISTE[2]
Et il y a donc :
des **adresses** (de noms) **stockées dans l'environnement**
des **adresses** (de sous-noms) **stockées dans la mémoire**.

Un **pointeur** est une adresse
(par exemple `0x18010ce[1]`)
où est stockée comme valeur
une autre adresse (par exemple `0x1753db8`).
C'est une adresse qui indique (pointe) une autre adresse



Notion n°11 : La vérité sur le stockage des variables Environnement, Mémoire, Adresses, Pointeurs (4)

- Par extension du sens, un **nom** qui est associé à un pointeur **est** lui même **un pointeur** (tout comme on dit qu'un nom associé à un entier est un entier, etc)
- Les **noms** sont directement associés à des adresses dans l'environnement.
  Exemple : `Y → 0x1753db8`
- Les **sous-noms** sont indirectement associés à des adresses, en passant par l'environnement et la mémoire. Exemple : `LISTE[1] → 0x18010ce[1] → 0x1753db8`
- On appellera **référence**, tout ce qui est associé à une adresse de façon directe (nom) ou indirecte (sous-nom)
- En Python, l'unique **affectation** qui existe est une **création ou mise-à-jour de référence** :

  `REFERENCE = EXPR`

  - Le **sens** (sémantique) de l'affectation se comprend en deux étapes :

  (1) L'expression EXPR est **évalué** et donne une valeur v qui se trouvait déjà en mémoire à l'adresse a, ou qui a été fraîchement calculée et placée à l'adresse a. De toute façon, l'évaluation de EXPR **donne une adresse** a **où se trouve le résultat** v

  (2) La **référence** est associée maintenant à l'adresse a (si c'est un **nom**, en ajoutant ou modifiant un lien dans l'**environnement** ; si c'est un **sous-nom**, en ajoutant ou modifiant un lien dans la **mémoire**)

Notion n°11 : La vérité sur le stockage des variables Environnement, Mémoire, Adresses, Pointeurs (5)

- En Python, l'unique **affectation** qui existe est une **création ou mise-à-jour de référence** :

  `REFERENCE = EXPR`

  (1) L'évaluation de EXPR **donne une adresse** a **où se trouve le résultat** v

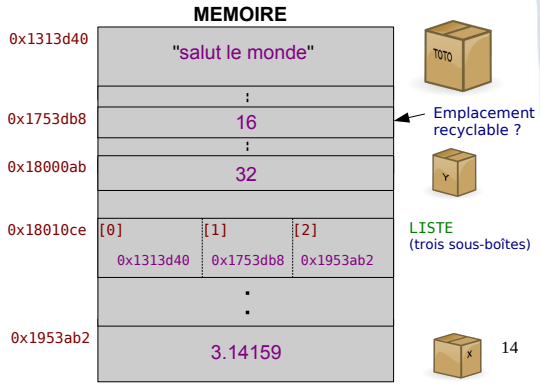
  (2) La **référence** est associée maintenant à l'adresse a

- **Exemple** (la référence est un **nom**)

```
Y = Y * 2          # Y est un nom
```

CATALOGUE des noms (**ENVIRONNEMENT**)

nom	adresse
TOTO	0x1313d40
~~Y~~	~~0x1753db8~~
Y	0x18000ab
X	0x1953ab2
LISTE	0x18010ce



Notion n°11 : La vérité sur le stockage des variables Environnement, Mémoire, Adresses, Pointeurs (6)

- En Python, l'unique **affectation** qui existe est une **création ou mise-à-jour de référence** :

  `REFERENCE = EXPR`

  (1) L'évaluation de EXPR **donne une adresse** a **où se trouve le résultat** v

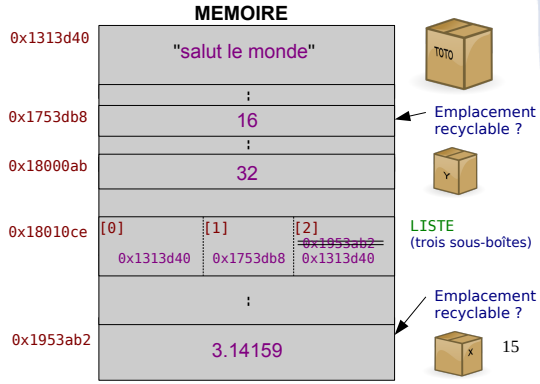
  (2) La **référence** est associée maintenant à l'adresse a

- **Exemple** (la réf. est un **sous-nom**)

```
Y = Y * 2
LISTE[2] = TOTO          # sous-nom
```

CATALOGUE des noms (**ENVIRONNEMENT**)

nom	adresse
TOTO	0x1313d40
~~Y~~	~~0x1753db8~~
Y	0x18000ab
X	0x1953ab2
LISTE	0x18010ce



Notion n°12 : La vérité sur l'appel de fonction : Environnement **global** et **local** (1)

- Souvenirs du deuxième cours :
  - Syntaxe de **définition** (*abstraction*) d'une fonction :

    `def NOM (ARG1, … ,ARGn):`
    `  ACTIONS`

  - Syntaxe d'**appel** (*application*) :

    `NOM (EXPR1, … ,EXPRn)`

ENVIRONNEMENT GLOBAL

nom	adresse
TOTO	0x1313d40
Y	0x1753db8
X	0x1953ab2

ENVIRONNEMENT LOCAL

nom	adresse
Z	0x2255e90
X	0x2258faa

- Au moment de l'appel, un **environnement local** est créé pour être utilisé **pendant l'exécution de la fonction** (il sera **éliminé** à la fin)
  - Il est **prioritaire** par rapport à l'environnement global : en **écriture**, les affectations concernent l'environnement local ; en **lecture**, on cherche avant dans le local, puis dans le global (ce qui résout le problème des **homonymes**)
  - **En premier lieu, pour commencer l'appel, sont exécutés les affectations :**

    `ARG1=EXPR1; … ; ARGn= EXPRn;`

Notion n°12 : La vérité sur l'appel de fonction : Environnement **global** et **local** (2)

- Que se passe t'il lorsqu'on fait appel à une fonction qui fait des affectations (dans ACTIONS) ? Est-ce que l'**effet** est **visible** à l'extérieur de la fonction ?

```
def foo(Z): Z=[]                    # remet la liste à vide
def bar(X): X[0]=0                  # (X homonyme) remet à zero le premier élément de la liste
TOTO = "salut le monde"
Y = 16
X = 3.14159
LISTE = [TOTO, Y, X]
foo(LISTE)
print(LISTE)                        # affiche ['salut le monde', 16, 3.14159]
bar(LISTE)
print(LISTE)                        # affiche [0, 16, 3.14159]
```

ENVIRONNEMENT GLOBAL

nom	adresse
TOTO	0x1313d40
Y	0x1753db8
X	0x1953ab2
LISTE	0x18010ce

ENVIRONNEMENT LOCAL Pour foo(LISTE)

nom	adresse
Z	0x18010ce

Résultat de l'affectation préliminaire `Z = LISTE`

ENVIRONNEMENT LOCAL Pour bar(LISTE)

nom	adresse
X	0x18010ce

Résultat de l'affectation préliminaire `X = LISTE`

- Pourquoi 0 ?

Notion n°12 : La vérité sur l'appel de fonction : Environnement **global** et **local** (3)

- Que se passe t'il lorsqu'on fait appel à une fonction qui fait des affectations (dans ACTIONS) ? Est-ce que l'**effet** est **visible** à l'extérieur de la fonction ?

```
def foo(Z): Z=[]
def bar(X): X[0]=0
foo(LISTE)                          # LISTE ne change pas
bar(LISTE)                          # LISTE devient [0, 16, 3.14159]
```

ENVIRONNEMENT GLOBAL

nom	adresse
TOTO	0x1313d40
Y	0x1753db8
X	0x1953ab2
LISTE	0x18010ce

ENVIRONNEMENT LOCAL Pour foo(LISTE)

nom	adresse
Z	~~0x18010ce~~ 0x1907ace

ENVIRONNEMENT LOCAL Pour bar(LISTE)

nom	adresse
X	0x18010ce

MEMOIRE

0x1313d40	"salut le monde"
	:
0x1753db8	16
	:
0x1800feb	0
0x18010ce	[0] ~~0x1313d40~~ 0x1800feb | [1] 0x1753db8 | [2] 0x1953ab2
	:
0x1907ace	[ ]
0x1953ab2	3.14159