

Initiation à la programmation avec Python (v3)

Cours n°6

Copyright (C) 2015-2019

Jean-Vincent Lodo

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

Sommaire du cours n°6

- Retour sur l'appel de fonction
 - L'instruction **global** en Python
- Notion n°13 : **Programmation orientée objet** (POO) - Initiation en Python

Retour sur l'appel de fonction (1) : L'instruction **global**

- Souvenirs du cinquième cours :

- Syntaxe de **définition** d'une fonction :
`def NOM (ARG1, ... , ARGn): ACTIONS`

- Syntaxe d'**appel** : `NOM (EXPR1, ... , EXPRn)`

- 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**)

- L'instruction **global** va changer ce comportement : les variables déclarées globales seront **cherchées** et **affectées** dans l'environnement global

```
def nb_octets(X): global Y; Z=X*Y*Y; return Z
```

ENVIRONNEMENT GLOBAL		ENVIRONNEMENT LOCAL	
nom	adresse	nom	adresse
TAILLE	0x1313d40	Z	0x2255e90
Y	0x1753db8	X	0x1313d40
X	0x1953ab2		

Retour sur l'appel de fonction (2) : L'instruction **global**

- 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)

ENVIRONNEMENT GLOBAL		ENVIRONNEMENT LOCAL	
nom	adresse	nom	adresse
TAILLE	0x1313d40	Z	0x2255e90
Y	0x1753db8	X	0x1313d40
Z	0x24900ae		
X	0x1953ab2		

- 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**)

- L'instruction **global** va changer ce comportement : les variables déclarées telles seront **recherchées** et **affectées** dans l'environnement global

```
def nb_octets(X): global Y; Z=X*Y*Y; return Z
```

```
Z = 1024./1000. # facteur utilisé par la suite (après les 4 prochaines lignes)
X = "Taille du disque annoncée par le constructeur en Mo ? "
TAILLE = int(input(X))
Y=1000; print("S'il s'agit de méga, cela fait ", nb_octets(TAILLE), "octets")
Y=1024; print("S'il s'agit de mébi, cela fait ", nb_octets(TAILLE), "octets")
```

variables globales

Retour sur l'appel de fonction (3) : L'instruction **global**

```
def nb_octets(X): global Y; Z=X*Y*Y; return Z
```

```
Z = 1024./1000. # facteur utilisé par la suite (après les 4 prochaines lignes)
X = "Taille du disque annoncée par le constructeur en Mo ? "
TAILLE = int(input(X))
Y=1000; print("S'il s'agit de méga, cela fait ", nb_octets(TAILLE), "octets")
Y=1024; print("S'il s'agit de mébi, cela fait ", nb_octets(TAILLE), "octets")
```

Affiche :

```
Taille du disque annoncée par le constructeur en Mo ? 500
S'il s'agit de méga, cela fait 500000000 octets
S'il s'agit de mébi, cela fait 524288000 octets
```

ENVIRONNEMENT GLOBAL		ENVIRONNEMENT LOCAL (1 ^{er} appel)	
nom	adresse	nom	adresse
TAILLE	0x1313d40	Z	0x18010ce
Y	0x1753db8	X	0x1313d40
Z	0x24900ae		
X	0x1953ab2		

ENVIRONNEMENT LOCAL (2 ^{ème} appel)	
nom	adresse
Z	0x2255e90
X	0x1313d40

MEMOIRE

0x1313d40	500
	:
0x1753db8	1000
	:
0x18010ce	500000000
	:
0x1953ab2	"Taille du disque annoncée.. en Mo ?"
	:
0x2255e90	524288000
0x24900ae	1.024

5

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (1)

- Rappels sur la notion de **module** (cours n°4)
Motivation :

- Quand le code devient **complexe**, il est nécessaire de le structurer en plusieurs composants distincts (**modules**, **bibliothèques**, **classes**, ...) dont le **but** aura été clairement identifié
- Le composant fournira un **ensemble** d'outils **cohérents** en rapport avec son **but** (d'où le qualificatif de **conteneur**)
- Ce sera donc un conteneur d'outils **réutilisables** dans d'autres projets
- En **Python**, il existe à la fois la notion de **module** classique (conteneur de fonctions), et celle de la *programmation orientée objet* (POO), c'est-à-dire les notions de **classe** et **objet** (conteneur de valeurs, qu'on appelle *champs* ou *propriétés*, et de fonctions, qu'on appelle *méthodes*)

6

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (2)

- La notion de **module** correspond à l'idée de **réunir** un ensemble d'outils (fonctions) cohérents par rapport à un certain objectif

- Un module est une sorte de **boîte à outils** spécialisée pour certaines tâches



- Par exemple, le module **turtle** de **Python** regroupe un ensemble de fonctions permettant de dessiner sur un écran graphique
 - Le **but** est de dessiner sur un écran graphique
 - On programme et on regroupe alors des fonctions pour ce faire dans un module et on lui donne un nom (**turtle**)
 - Ensuite, pour dessiner sur un écran graphique, il nous suffit d'utiliser (**importer**) le module et faire appel aux outils dans la boîte

7

Adresse de l'image : <https://openclipart.org/detail/2921/toolkit>

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (3)

- On peut considérer la notion d'**objet** comme une évolution de celle de **module**
- Pas juste **conteneur d'outils** (de manipulation de l'information), mais aussi **conteneur d'information** (que les outils manipulent)



- À bien regarder, le module **turtle** de **Python** pourrait déjà mériter le titre d'**objet** :
 - Il contient l'**information** sur la **position** actuelle de la tortue (couple d'entiers)
 - Il contient l'**information** sur la **couleur** actuelle du crayon (triplet d'entiers)
 - Il contient l'**information** sur l'état **appuyé/relevé** du crayon (booléen), etc
 - Les **outils** qu'il contient sont le plus souvent des **procédures** permettant de **modifier l'état** de la tortue (c'est-à-dire l'information stockée dans l'objet)

8

Adresse de l'image : <https://openclipart.org/detail/202408/raseone-file-cabinet>

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (4)

- L'**objet** comme évolution de la notion de **module** :
 - pas juste **conteneur d'outils**, mais aussi **conteneur d'information**

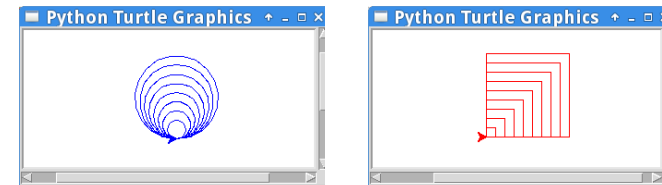


- Puisque l'information à l'intérieur de l'objet (**état interne**) peut **changer** (à la différence des outils), il se peut qu'on ait besoin de plusieurs objets (**instances**) du même type
- C'est plutôt naturel : les **mêmes outils**, certes, mais avec des **états internes différents**
- Par exemple, on pourrait avoir besoin de deux **instances** du module **turtle** de **Python**, une pour dessiner des carrés, une pour dessiner des cercles

9

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (5)

- Exemple des deux **instances** du module **turtle** de **Python**, une pour dessiner des cercles (bleus), une pour dessiner des carrés (rouges)

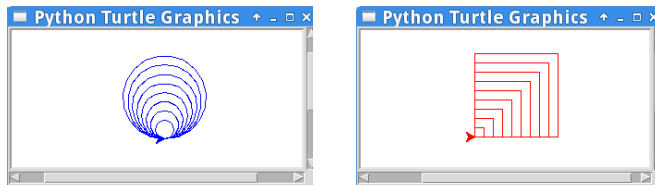


```
import imp # Module standard permettant des importations plus sophistiquées
t1 = imp.load_module('t1', *imp.find_module('turtle')) # t1 instance du module turtle.py
t2 = imp.load_module('t2', *imp.find_module('turtle')) # t2 instance du module turtle.py
t1.color("blue") # couleur crayon dans t1 : bleu
t2.color("red") # couleur crayon dans t2 : rouge
t2.forward(10) # position de la tortue dans t2 : (10,0)
for i in range(10,50,5): t1.circle(i) # trace cercles bleus dans fenêtre de t1
def trace_carre(t, taille): # Le 1er paramètre est un module (objet)
    for i in range(4): t.forward(taille); t.left(90)
for i in range(10,100,10): trace_carre(t2,i) # trace carré rouges dans fenêtre de t2
```

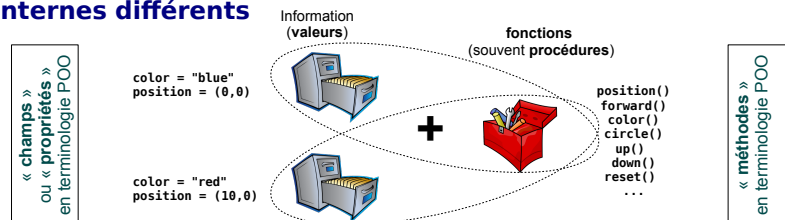
10

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (6)

- Exemple des deux **instances** du module **turtle** de **Python**, une pour dessiner des cercles (bleus), une pour dessiner des carrés (rouges)



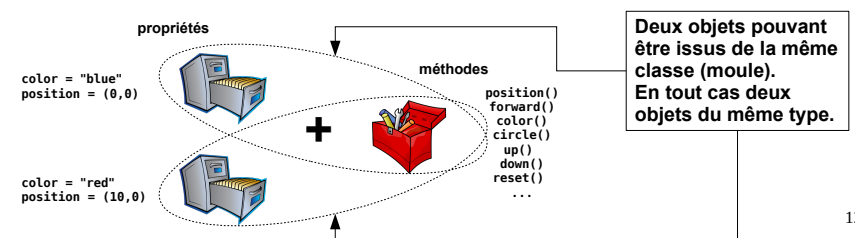
- Les **mêmes outils**, certes, mais qui s'appliquent à des **états internes différents**



11

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (7)

- Dans le paradigme objet, les **outils** sont appelés « **méthodes** », les **informations** sont appelées « **champs** » ou « **propriétés** »
- L'ensemble des **valeurs** des champs constitue l'**état interne** de l'objet
- La présence d'un état interne **variable** donne intérêt à avoir une **multiplicité d'instances** (exemple des deux tortues)
- Les langages de POO offrent typiquement la possibilité d'écrire un « **moule** » pour fabriquer des **objet similaires** (même méthodes, différents états internes) : c'est ce qu'on appelle le plus souvent « **classe** »



12

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (8)

- Définition d'un **moule à objets**, c'est-à-dire une **classe**, en Python :
`class NOM[(NOM1, ..., NOMk)]:`
COMMANDES
- Ressemble à un module (fichier contenant des commandes), sauf que :
 - toutes les **variables** définies sont des **propriétés**
 - toutes les **fonctions** définies sont des **méthodes**
 - **ATTENTION** : le 1^{er} argument des méthodes est **un nom pour l'objet lui-même** (typiquement **self, this, me,...**), ce qui permet d'accéder à ses propriétés
- Exemple :

```
class point:
    x = 0
    y = 0
    def move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy
```

début de la définition de classe
coordonnée horizontale (propriété)
coordonnée verticale (propriété)
déplacement (méthode)
mise à jour de la propriété x
mise à jour de la propriété y

13

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (9)

- Définition d'un **moule à objet**, c'est-à-dire une **classe**, en Python :
`class NOM[(NOM1, ..., NOMk)]:`
COMMANDES
- Application du moule (création d'une instance ou objet de la classe) :
 - on utilise le **nom** de la classe **comme si c'était une fonction**
 - le **résultat** renvoyé est l'**objet** nouvellement créé
- Exemple :

```
p = point()
p.x
p.y
p.move(10,0)
p.move(5,5)
p.x
p.y
```

création d'un point
l'interpréteur répond 0
l'interpréteur répond 0
déplace horizontalement de 10
déplace de 5 dans les deux directions
l'interpréteur répond 15
l'interpréteur répond 5

14

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (10)

- La notion de **constructeur** :
 - une méthode particulière, nommée `__init__()`, si présente, est appelée automatiquement à la création de l'objet
- Application du moule (création d'une instance ou objet de la classe) :
 - on utilise le **nom** de la classe **comme si c'était une fonction** et on fournit éventuellement ce que `__init__` demande comme arguments
 - typiquement `__init__` utilisera des paramètres optionnels
- Exemple :

```
class point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy
```

début de la définition de classe
constructeur (méthode)
coordonnée horizontale (propriété)
coordonnée verticale (propriété)
déplacement (méthode)
mise à jour de la propriété x
mise à jour de la propriété y

15

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (11)

- Définition d'un **moule à objet**, c'est-à-dire une **classe**, en Python :
`class NOM[(NOM1, ..., NOMk)]:`
COMMANDES
- Notion d'**héritage** : les noms optionnels délimités par des parenthèses et situés entre le **NOM** et les deux points, sont **les noms des classes** dont la classe définie **hérite (propriétés et méthodes)**
 - si $k=1$ il s'agit d'**héritage simple** (ordinaire), si $k>1$ il s'agit d'**héritage multiple** (rare et difficile à utiliser)
- Exemple

```
class colored_point(point):
    c = "black"
    def set_color(self, v):
        self.c = v
```

hérite de la classe point
couleur de type chaîne (propriété)
changement de couleur (méthode)
mise à jour de la propriété c

- Remarque : on aurait eu envie de l'héritage dans l'exemple des deux tortues
 - au lieu de définir la fonction `trace_carre()`, on aurait défini une classe **héritant de tortue** et rajoutant une méthode **square** !

16

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (12)

Plusieurs notions fondamentales apparaissent avec la POO :

- L'**encapsulation**
 - un objet peut cacher complètement les données qu'il contient (propriétés) et **exposer** (à l'utilisation) seulement ses **méthodes** (encore l'exemple de **turtle**)
- Le **sous-typage** et le **polymorphisme** des objets
 - si un objet x est plus **riche** en méthodes et propriétés qu'un autre objet y, alors x pourra être utilisé partout où y pourrait l'être (le type de x est **sous-type** de celui de y)
 - On appelle cela du « **polymorphisme** » : un objet pourra toujours « **se comporter** » comme tout autre objet **moins riche** (dont il est sous-type).
 - Par exemple, les objet de type `colored_point` sont plus « **riches** » que les (**sous-type** des) objets de type `point`, donc :
 - les objet de type `colored_point` pourront être utilisés, si besoin, comme des simples `point` (il suffit en effet d'oublier la couleur !)
 - Autre exemple : la tortue enrichie avec `square` pourra être utilisée, si besoin, comme une tortue ordinaire (on oubliera `square`)

17

Notion n°13 : Introduction en Python à la Programmation orientée objet (POO) (13)

Plusieurs notions fondamentales apparaissent avec la POO :

- Le **redéfinition** de méthodes (et la **liaison tardive**)
 - les objets (ou classes) qui héritent d'autres objets (ou classes), peuvent **redéfinir** certaines méthodes **héritées**.
 - dans ce cas, s'ils sont utilisés comme des objets **moins riches** (**polymorphisme**), les **nouvelles versions** de leurs méthodes seront de toute façon utilisées, et non pas celles héritées des ancêtres (« **liaison tardive** »)
 - Exemple : les objet de type `colored_point` pourront être utilisés, si besoin, comme des simples `point` (en oubliant la couleur). Or, si on redéfinissait la méthode `move()` héritée de `point`, ce serait de toute façon la **nouvelle version** qui serait utilisée, même en les utilisant comme de simples `point`.

18