

Initiation à la programmation avec Python (v3)

Cours n°3

Copyright (C) 2015 - 2019

Jean-Vincent Loddo

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

Sommaire du cours n°3

- Retour sur les sous-programmes internes (**fonctions**)
- Méthodologie de programmation des fonctions (généralisation progressive)
- Notion n°8 : boucles avec condition de sortie (**while**)
- Simuler un **for-range** avec un **while**

Retour sur les sous-programmes (**fonctions**) (1)

Sens du mot *abstraction*

- Syntaxe de **définition** (*construction* ou *abstraction*) :

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

où dans **ACTIONS** pourra se trouver une occurrence ou plus de l'instruction **return(EXPR)**, pour renvoyer (à l'appellant) la valeur représentée par **EXPR**

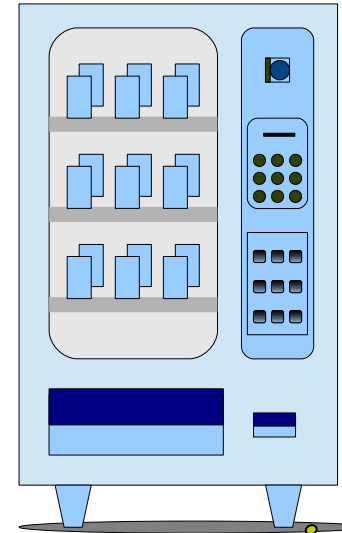
- Exemple :

```
def repond_present(X):  
    print(X) # prononcer le nom  
    REP = input("Présent ?") # écouter la réponse  
    if (REP == "oui"):  
        return True # rendre vrai (présent)  
    else:  
        return False # rendre faux (absent)
```

En construction

Remarquer la présence de plusieurs return

Action définie quelle que soit la valeur de **X**, c'est-à-dire **en faisant abstraction** de la valeur du **paramètre** ou des paramètres.
Il faudra toutefois que cette valeur soit donné (à l'utilisation, c'est-à-dire à l'**appel**)



Retour sur les sous-programmes (**fonctions**) (2)

Variables locales

- Syntaxe de **définition** (*construction* ou *abstraction*) :

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

Paramètre (*formel*)

Variable **locale**

En construction

- Exemple :

```
def repond_present(X):  
    print(X) # prononcer le nom  
    REP = input("Présent ?") # écouter la réponse  
    if (REP == "oui"):  
        return True # rendre vrai (présent)  
    else:  
        return False # rendre faux (absent)
```

Les variables **locales** sont des variables utilisées par la fonction en interne, pour rendre le service souhaité.
Une fois la mission accomplie (**return**), les variables locales disparaissent sans laisser de traces dans le reste du programme.

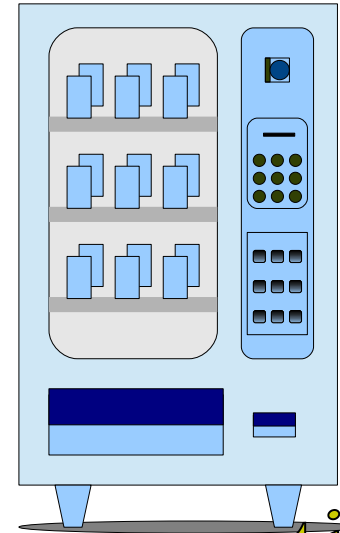
Retour sur les sous-programmes (**fonctions**) (3)

Procédures

- Syntaxe de **définition** (*construction* ou *abstraction*) :

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

où dans **ACTIONS** pourra se trouver une occurrence ou plus de l'instruction **return**(**EXPR**) pour renvoyer (à l'appellant) la valeur représentée par **EXPR**

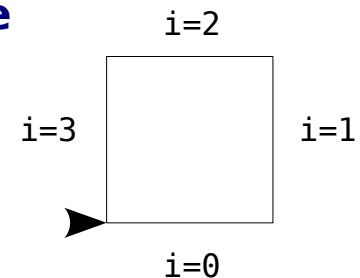


En construction

- Une fonction qui ne renvoie aucune valeur, s'appelle **procédure** (pas de **return**, ou **return** sans expression)
 - Exemple de procédure pour la tortue : (on révisé au passage les boucles **for**)

```
def trace_carre(L):  
    for i in range(4):  
        forward(L)  
        left(90)
```

```
# L est la longueur  
# répéter 4 fois (i locale)  
# avancer de X pixels  
# tourner à gauche
```



Retour sur les sous-programmes (**fonctions**) (4)

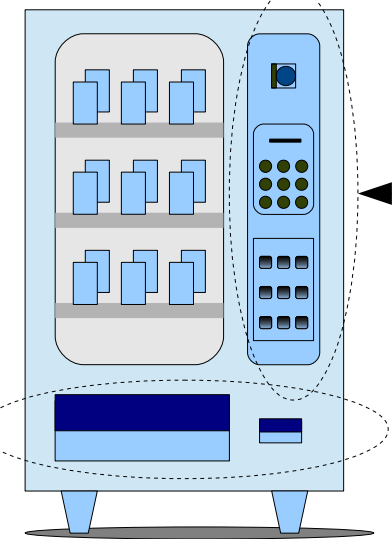
Donner pour recevoir

- Syntaxe d'**appel** (*utilisation* ou *application*) :
à la place de toute **EXPR** ou **ACTION** (si procédure)
on pourra écrire :

NOM (**EXPR**₁, ... , **EXPR**_n)

- Faire **appel** signifie **utiliser**
- Pour utiliser **il faut fournir** les **arguments** (actuels),
on aura en contrepartie un **résultat**
- Comme pour le distributeur de cannettes :
il faudra fournir de l'argent et un choix de boisson (**arguments**)
pour avoir en contrepartie une cannette (**résultat**)

Utilisable



- Dans le cas d'une expression, le résultat remplacera l'appel
Exemple :

```
print("Le double de %d est %d" % (7, double(7))) # 14 remplacera double(7)
```

- Comme si on écrivait :

```
boire (distributeur(1.50, 43)) # choix n.43, prix 1.50€
```

Retour sur les sous-programmes (**fonctions**) (5)

Re-écriture de l'appel

- Syntaxe d'**appel** (*utilisation* ou *application*) :
à la place de toute **EXPR** ou **ACTION** (si procédure)
on pourra écrire :

NOM (**EXPR**₁, ... , **EXPR**_n)

- Faire **appel** signifie **utiliser**
- Pour utiliser **il faut fournir** les **arguments** (actuels),
on aura en contrepartie un **résultat**
- Dans le cas d'une expression, le résultat **remplacera** l'appel
- Exemple :

```
ETUDIANTS = ["Bobo", "Coco", "Lolo", "Mimi", "Toto", "Zaza"]  
PRESENT = [False, False, False, False, False, False]
```

```
for i in range(6):
```

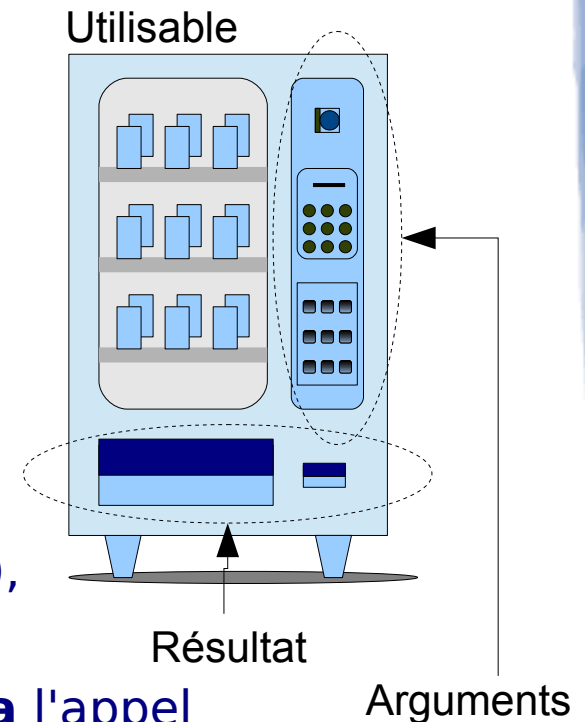
```
    X = ETUDIANTS[i]
```

```
    PRESENT[i] = repond_present(X)
```

```
# lire le nom
```

```
# cocher présent ou absent
```

sera remplacé par **True** ou **False**



Retour sur les sous-programmes (fonctions) (6)

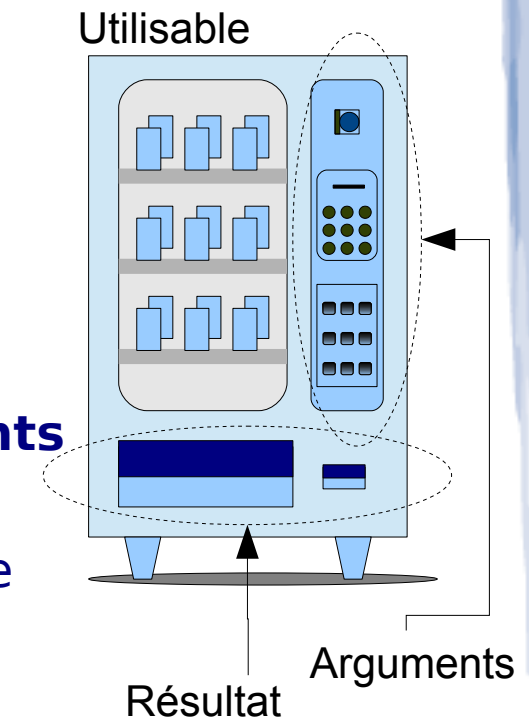
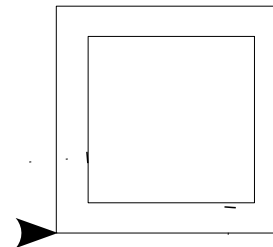
Exemple des procédures

- Syntaxe d'**appel** (*utilisation* ou *application*) :
à la place de toute **EXPR** ou **ACTION** (si procédure)
on pourra écrire :

NOM (**EXPR**₁, ... , **EXPR**_n)

- Faire **appel** signifie **utiliser**, il faut fournir les **arguments** (*actuels*) et on aura en contrepartie le **résultat**
- Dans le cas d'une action (procédure), l'action se termine quand la procédure revient
- Exemple :

```
trace_carre(100) .....  
up()  
backward(10); right(90); forward(10); left(90)  
down()  
trace_carre(120) .....
```



```
# relever le crayon  
# déplacer de (-10,-10)  
# abaisser le crayon
```

Remarque : avec la tortue la plupart des outils sont des procédures, grâce auxquelles nous en construisons de nouvelles

Retour sur les sous-programmes (**fonctions**) (7)

Fonctions qui utilisent des fonctions

- Une fois définie, une fonction devient utilisable **n'importe où** dans le programme
- On peut alors **l'utiliser pour en définir une autre**
- Exemple :

```
def trace_deux_carres():
```

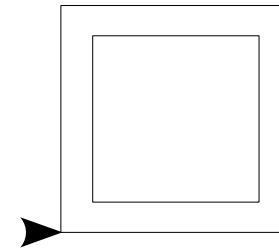
```
    trace_carre(100)
```

```
    up()
```

```
    backward(10); right(90); forward(10); left(90)
```

```
    down()
```

```
    trace_carre(120)
```



```
# relever le crayon
```

```
# déplacer de (-10,-10)
```

```
# abaisser le crayon
```

- Ici la fonction `trace_carre` est utilisée pour définir `trace_deux_carres`

Remarque : une fonction peut avoir **aucun argument**. C'est comme dire qu'elle n'a besoin de rien pour fonctionner, pour rendre son service. Il faut juste lui donner le feu vert, c'est-à-dire l'appeler. Dans cet exemple, l'appel se fera ainsi : `trace_deux_carres()`

Retour sur les sous-programmes (**fonctions**) (8)

Fonctions qui utilisent des fonctions

Cas spécial des fonctions **récurives**

- Une fois définie, une fonction devient utilisable
- En réalité, la **fonction est utilisable dans sa propre définition**, si cela est utile...
- Ces fonctions qui font appel à elles mêmes sont dites « **récurives** »
- Exemple :

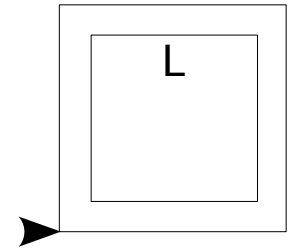
```
def factorielle(n):  
    if (n == 0):  
        return 1  
    else:  
        k = factorielle(n-1)  
        return (n * k)
```

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 7 * (6!) \\ \begin{array}{ccc} \uparrow & & \uparrow \\ n & & k \end{array}$$

- Ici la fonction `factorielle` est utilisée pour définir `factorielle`, c'est-à-dire pour définir elle même

Méthodologie de programmation des **fonctions** (1)

- Lorsque on programme une fonction, il est bien plus simple d'**introduire les arguments petit à petit**, pas tout de suite et tous ensemble
- Autrement dit, il est nettement plus facile de faire semblant de travailler avec des valeurs données, puis de **généraliser (abstraire) progressivement**
- Exemple :

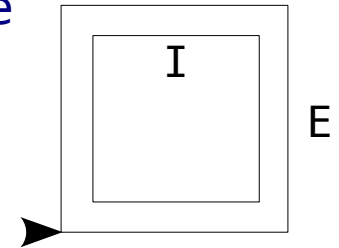


```
- def trace_deux_carres(L):  
    trace_carre(L) # taille spécifiée dans L  
    up() # dessine carré interne  
    backward(10); right(90); forward(10); left(90) # relever le crayon  
    down() # déplacer de (-10,-10)  
    trace_carre(L+20) # abaisser le crayon  
 # dessine carré externe
```

- Ici la taille du carré interne devient un paramètre (L), et l'outil devient plus générique, c'est-à-dire **d'application plus large**

Méthodologie de programmation des **fonctions** (2)

- Lorsque on programme une fonction, il est bien plus simple d'**introduire les arguments petit à petit**, pas tout de suite et tous ensemble
- Autrement dit, il est nettement plus facile de faire semblant de travailler avec des valeurs particulières, puis de **généraliser (abstraire) progressivement**
- Exemple :



```
- def trace_deux_carres(I,E):  
    trace_carre(I) # taille interne I, externe E  
    up() # dessine carré interne  
    # relever le crayon  
    D = (E-I)/2 # D est la distance  
    backward(D); right(90); forward(D); left(90) # déplacer de (-D,-D)  
    down() # abaisser le crayon  
    trace_carre(E) # dessine carré externe
```

- Après avoir généralisé (fait abstraction de) la taille du carré interne (I), nous généralisons (faisons abstraction de) la taille du carré externe (E)
- L'outil est encore plus générique, c'est-à-dire **d'application encore plus large** 12

Notion n°8 : boucles avec condition de sortie (**while**) (1)

- **while** : on répète des actions plusieurs fois, **tant qu'une condition est satisfaite**
 - on sort dès que la condition n'est plus **vraie** (n'est plus évaluée à True)
- Exemple :
 - ```
REP = "oui"
while (REP == "oui"):
 print("Bonjour, je suis un vieux programme rabâcheur")
 REP = input("Voulez-vous continuer ? ")
```

Exécution du programme :

```
Bonjour, je suis un vieux programme rabâcheur
Voulez-vous continuer ? oui
Bonjour, je suis un vieux programme rabâcheur
Voulez-vous continuer ? oui
Bonjour, je suis un vieux programme rabâcheur
Voulez-vous continuer ? oui
```

...

# Notion n°8 : boucles

## avec condition de sortie (**while**) (2)

- **while** : on répète des actions plusieurs fois, **tant qu'une condition est satisfaisante**.

- Syntaxe

```
while EXPR :
 ACTIONS
```

Expression booléenne **d'aiguillage** (prise de décision : **rester ou sortir de la boucle**), évaluée au départ, puis ré-évaluée à la fin de chaque cycle

**Actions répétées** (éventuellement zéro fois si l'expression est tout de suite fausse)

- Exemple

```
REP = "oui"
while (REP == "oui"):
 print("Bonjour, je suis un vieux programmeur rabâcheur")
 REP = input("Voulez-vous continuer ? ")
```

- Les actions ont vocation à **changer** la valeur de l'expression booléenne d'aiguillage, ce qui à un certain moment provoquera la **sortie** de la boucle (sinon boucle infinie)
- Dans l'exemple, l'expression (**REP == "oui"**) porte sur la variable **REP** (réponse) qui peut changer à cause de l'action `input`

# Notion n°8 : boucles avec condition de sortie (**while**) (3)

- Exemple : calculer la première puissance de 2 qui dépasse 1000

```
X = 1
while (X <= 1000):
 X = X * 2
```

1024=2<sup>10</sup> of course



- Comme fonction (généralisation progressive) :

```
def premiere_puissance_de_2_superieure_a(LIMITE):
 X = 1
 while (X <= LIMITE): # début de la boucle
 X = X * 2 # intérieur de la boucle
 return X # extérieur de la boucle, intérieur de la fonction
```

- Encore plus général :

```
- def premiere_puissance_de_superieure_a(BASE, LIMITE):
 X = 1
 while (X <= LIMITE): # début de la boucle
 X = X * BASE # intérieur de la boucle
 return X # extérieur de la boucle, intérieur de la fonction
```

# Simuler un **for-range** avec un **while**

## Nécessité du **while**

- On peut voir la construction **for** couplé avec un appel à la fonction **range()** comme une facilité d'écriture. Une boucle **while** pourrait faire la même chose :

```
for X in range(START,STOP,STEP):
 ACTIONS
```

- est parfaitement équivalent à :

```
X = START
while (X < STOP):
 ACTIONS
 X = X + STEP
```

- Quand est-ce que **while** est strictement **nécessaire** par rapport à **for** ?
  - quand il **n'est pas possible** pour le programmeur **d'annoncer** à l'utilisateur, juste avant de rentrer dans la boucle, **combien de cycles seront exécutés**