Initiation à la programmation avec Python (v3)

Cours n°2

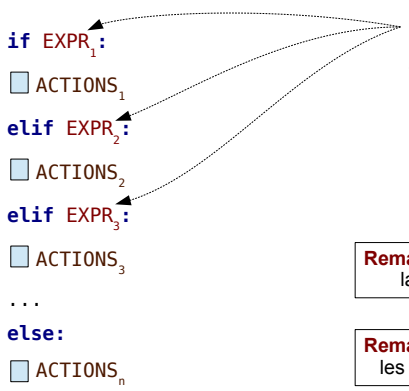
Jean-Vincent Loddo

Sommaire du cours n°2

- Syntaxe générale de la conditionnelle (**if-elif-else**)
- Notion n°5 : structures de données
- Notion n°6 : itérations ou boucles sans condition de sortie (**for**)
- Notion n°7 : sous-programmes (**fonctions**)

Syntaxe générale de la conditionnelle (prise de décision)

- Syntaxe générale :



Expressions booléennes **d'aiguillage** :

True => aller à la partie « alors » suivante

False => aller à la partie « sinon-si » (elif) ou « sinon » (else) suivante

Remarque n.1 : toutes les parties « elif » et la partie « else » **sont optionnelles**

Remarque n.2 : au plus **une** des ACTIONS après les parties « then » ou « else » **sera exécutée**

Syntaxe générale de la conditionnelle Exemple (1) :

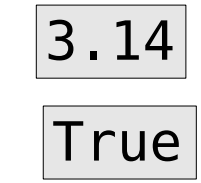
```
X = input("Quel âge avez vous ?")
AGE = int(X)

if (AGE<=2):
  print("Vous êtes un bébé")
elif (AGE<=11):
  print("Vous êtes un enfant")
elif (AGE<18):
   print("Vous êtes un adolescent")
else:
   print("Vous êtes un adulte")
```

Syntaxe générale de la conditionnelle Exemple (2) :

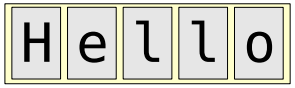
```
X = input("Quel âge avez vous ?")
AGE = int(X)
if (AGE<=2):                                  EXPR1
  print("Vous êtes un bébé")                  ACTIONS1
elif (AGE<=11):                               EXPR2
  print("Vous êtes un enfant")                ACTIONS2
elif (AGE<18):                                EXPR3
   print("Vous êtes un adolescent")           ACTIONS3
else:
   print("Vous êtes un adulte")               ACTIONS4
```


Notion n°5 : structures de données (1)

- La plupart des types de base (**int**, **float**, **bool**) ne sont pas structurés
  - les valeurs sont **simples**, c'est-à-dire :
  - ils n'ont **pas de structure interne**, c'est-à-dire :
  - ils ne contiennent **pas de sous-informations**, c'est-à-dire :
  - ils ne sont pas **décomposables** en plusieurs parties

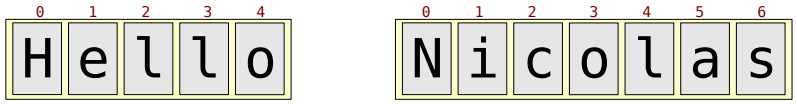


- En revanche, le type de base **str** est **structuré**
  - une chaîne de caractère est composée d'une séquence finie de caractères
  - les « parties » de la chaîne `"Hello"` sont les caractères `'H' 'e' 'l' 'l' 'o'`
  - il y a structure, chaque caractère étant un composant de la chaîne



Notion n°5 : structures de données (2)

- Le type de base **str** est **structuré**
  - une chaîne de caractère est composée d'une séquence finie de caractères
  - les parties de la chaîne `"Hello"` sont les caractères `'H' 'e' 'l' 'l' 'o'`
  - il y a structure, chaque caractère étant un composant de la chaîne



- on peut donc extraire un composant (un caractère) :

```
NOM = "Nicolas"
INITIALE = NOM[0]
print("La lettre initiale de",NOM,"est",INITIALE)
```

  - Ce programme affiche :

    La lettre initiale de Nicolas est N

Notion n°5 : structures de données (3) Les **listes** de valeurs

- Les **listes** (type **list**) sont des séquences finies de valeurs
  - La **structure** est similaire aux chaînes de caractères : il peut y avoir un 1er élément, un 2ème, un 3ème, ainsi de suite
  - En Python, on les exprime avec des crochets et on les sépare avec des virgules. Exemples :

```
[1, 3, 5, 7, 9]          ["chien", "chat", "loup", "tigre"]
[1.16, 3.14, 5.721]      [ True,    True,   False,  False ]
```

- on peut voir les listes (**list**) comme une généralisation des chaînes de caractères (**str**)
  - Au lieu de contenir seulement des caractères, une liste peut contenir **tout type** de valeurs (même des listes bien sûr !)
  - Une liste a vocation à contenir des valeurs du même type (listes **homogènes**), mais en Python il n'est pas interdit d'avoir des listes **hétérogènes**, comme par exemple : `[1, "chien", 1.16, True ]`

Notion n°5 : structures de données (4)
Les **listes** de valeurs

- Exemples :

```
ANIMAUX    = ["chien", "chat", "loup", "tigre"]
DOMESTIQUE = [ True,    True,   False,  False ]
```

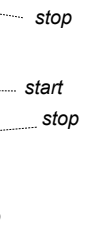

```
print("Le", ANIMAUX[0], "est le meilleur ami de l'homme")
print("Le", ANIMAUX[2], "est le meilleur ami du Petit Chaperon rouge")

i = int(input("Entrez un index entre 0 et 3 : "))
if DOMESTIQUE[i]:
  print("Le", ANIMAUX[i], "est un animal domestique")
else:
  print("Le", ANIMAUX[i], "est un animal sauvage")
```


Notion n°5 : structures de données (5)
Les **listes** de valeurs

- La fonction **range** permet de générer des **listes d'entiers**
- Très utiles pour construire des traitements en boucle (**for**)
- Depuis la version 3, **range** rend un *itérateur*, c'est-à-dire une séquence où les éléments sont générés quand nécessaire. Toutefois, on peut toujours transformer la séquence rendue en liste ordinaire par la fonction **list**. Exemples :
  - `list(range(10))`
    `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - `list(range(4,10))`
    `[4, 5, 6, 7, 8, 9]`
  - `list(range(4,10,2))`
    `[4, 6, 8]`
- Usage :
  - **range**(*stop*)
  - **range**(*start*, *stop* [, *step*])

Notion n°5 : structures de données (5)
Les **listes** de valeurs

- La fonction **range** permet de générer des **listes d'entiers**
- Très utiles pour construire des traitements en boucle (**for**)
- Depuis la version 3, les listes sont « paresseuses » (éléments générés quand nécessaire), mais on peut les transformer en listes ordinaires par la fonction **list**. Exemples :
  - `list(range(10))`
    `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - `list(range(4,10))`
    `[4, 5, 6, 7, 8, 9]`
  - `list(range(4,10,2))`
    `[4, 6, 8]`



- Usage :
  - **range**(*stop*)
  - **range**(*start*, *stop* [, *step*])

Remarque importante : l'entier *stop* n'est pas inclus dans la liste générée

Notion n°5 : structures de données (6)
Les **listes** de valeurs

- En Python les listes sont **modifiables**, ce qui signifie qu'une fois construites :
  - on peut les **rallonger**, c'est-à-dire **rajouter** des éléments
  - on peut les **raccourcir**, c'est-à-dire **éliminer** des éléments
  - on peut aussi modifier un composant quelconque, par l'**affectation** :

```
LISTE = ["chien", "chat", "loup", "tigre"]
LISTE[2] = "renard"
print(LISTE)
```

Ce programme affiche :

```
["chien", "chat", "renard", "tigre"]
```

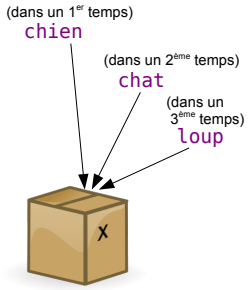
Notion n°6 : **itérations**
ou boucles sans condition de sortie (**for**) (1)

- On doit répéter la **même action** plusieurs fois
  - Pas tout à fait la même action, mais à quelque chose près...
  - Exemple : vous voulez afficher des noms de fichiers JPEG correspondants à plusieurs animaux :
    `ANIMAUX/chien.jpg  ANIMAUX/chat.jpg  ANIMAUX/loup.jpg`
  - Vous pouvez faire (méthode brute, problématique si la liste s'allonge) :

```
print("ANIMAUX/chien.jpg")
print("ANIMAUX/chat.jpg")
print("ANIMAUX/loup.jpg")
```

  - Vous pouvez faire (méthode rusée) :

```
for X in ["chien", "chat", "loup"]:
  print("ANIMAUX/%s.jpg" % X)
```

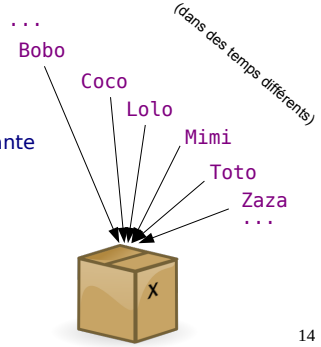


Action (`print`) répétée 3 fois (pour chacune des valeurs prises tour à tour par X)

Notion n°6 : **itérations**
ou boucles sans condition de sortie (**for**) (2)

- Exemple de l'**appel en classe** : pour chaque case dans la liste d'émargement, le prof super cool répète mécaniquement les instructions suivantes :
  - Lire le nom écrit dans la case actuelle
  - Prononcer le nom
  - Écouter l'éventuelle réponse
  - Cocher la case « **Présent** » si réponse, sinon cocher la case « **Absent** » correspondante

Actions répétées pour toutes les lignes de la feuille d'émargement



Notion n°6 : **itérations**
ou boucles sans condition de sortie (**for**) (3)

- Exemple de l'**appel en classe** en Python maintenant :

```
ETUDIANTS = ["Bobo", "Coco", "Lolo", "Mimi", "Toto", "Zaza"]
PRESENT   = [False,  False,  False,  False,  False,  False ]
for LIGNE in range(6):
  X = ETUDIANTS[LIGNE]              # lire le nom
  print(X)                          # prononcer le nom
  REP = input("Présent ?")          # écouter la réponse
  if (REP == "oui"):
    PRESENT[LIGNE] = True           # cocher présent
  else:
    PRESENT[LIGNE] = False          # cocher absent
```

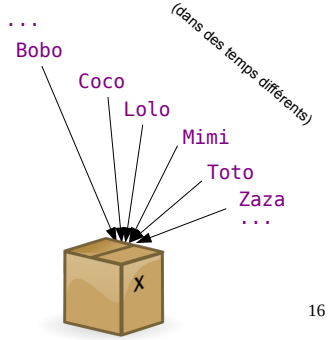
Actions répétées pour les six lignes (de 0 à 5) de la feuille d'émargement

Notion n°6 : **itérations**
ou boucles sans condition de sortie (**for**) (4)

- Exemple de l'**appel en classe** en Python maintenant :

```
ETUDIANTS = ["Bobo", "Coco", "Lolo", "Mimi", "Toto", "Zaza"]
PRESENT   = [False,  False,  False,  False,  False,  False ]
for LIGNE in range(6):
  X = ETUDIANTS[LIGNE]
  print(X)
  REP = input("Présent ?")
  if (REP == "oui"):
    PRESENT[LIGNE] = True
  else:
    PRESENT[LIGNE] = False
```

Actions répétées pour les six lignes (de 0 à 5) de la feuille d'émargement



Notion n°6 : **itérations**
ou boucles sans condition de sortie (**for**) (5)

- Syntaxe :

```
for VARIABLE in EXPR:
    ACTIONS
```

Expression de valeur structurée énumérable (**str**, **list**, ...)

- D'autres langages appellent cette construction « **foreach** »

- Exemple avec des chaînes de caractères (au lieu d'une liste) :

```
for CARACTERE in "0123456789ABCDEF":
  print(CARACTERE)
```

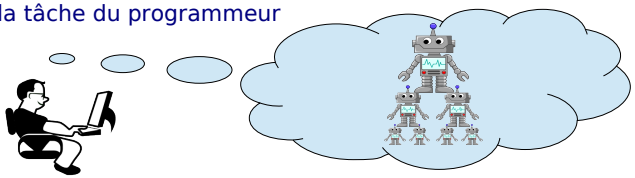
Action répétée pour les 16 caractères de la chaîne (caractères hexadécimaux)

Notion n°7 : paramétrisation
(des programmes ou sous-programmes) (1)

- Les programmes interactifs sont intéressants pour l'utilisateur :

```
#!/bin/bash
JPG=$(zenity --file-selection --title="Choisissez un fichier jpeg")
convertisseur.py ${JPG} -resize 50% ${JPG%.jpg}.png
```

- mais... il n'y a que lui qui peut **en faire appel**, puisque il n'y a que lui pour répondre interactivement aux questions posées (clavier, souris,...)
- C'est bien aussi qu'un robot (programme) puisse faire appel à... un autre robot (sous-programme) déjà construit
  - C'est un principe de sous-traitance des services à rendre, qui facilite la tâche du programmeur



Notion n°7 : paramétrisation
(des programmes ou sous-programmes) (2)

- Pour que les (sous-)programmes puissent se faire appel, on utilise les **paramètres d'appel** :
  - Tous les (sous-)programmes ont une sorte de « **casier** » où on peut déposer des informations (valeurs) utiles au service qu'ils doivent rendre
  - De cette façon, ils ne doivent pas poser de question (interagir) : les informations sont **dès le départ** dans le casier des paramètres
  - N'importe qui (utilisateur ou autre programme) peut faire appel à leur service en remplissant le casier avec les bonnes informations ; c'est en réalité un principe que vous connaissez très bien, puisque vous l'utilisez :

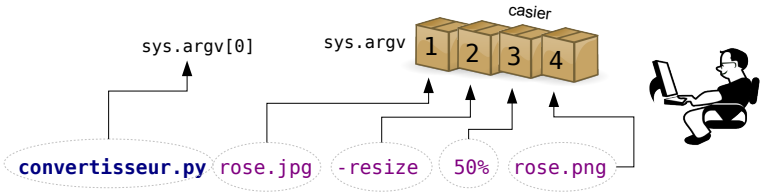
    ```
    convertisseur.py rose.jpg -resize 50% rose.png
    ```

    Vous faites appel au programme **convertisseur.py** en remplissant son casier :

    - Dans la case n°**1** du casier vous mettez la valeur "rose.jpg"
    - Dans la case n°**2** du casier vous mettez la valeur "-resize"
    - Dans la case n°**3** du casier vous mettez la valeur "50%"
    - Dans la case n°**4** du casier vous mettez la valeur "rose.png"

Notion n°7 : paramétrisation
(des programmes ou sous-programmes) (3)

- Comment le développeur Python peut manipuler les valeurs déposées dans le casier du programme par l'appelant (utilisateur ou autre programme) ?
  - Le « **casier** » **est simplement une liste**, dont le nom est `sys.argv` (liste `argv` du module `sys`, module qu'il faudra importer)



  - Le développeur peut donc accéder à ces informations (qu'il ne connaît pas mais ce n'est pas un problème) par `sys.argv[1]` `sys.argv[2]` `sys.argv[3]` ...

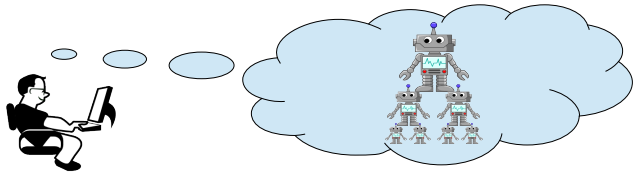
Notion n°7 : paramétrisation (des programmes ou sous-programmes) (4)

Exemple du robot pour footballeurs professionnels (v0)

```
#!/usr/bin/python3
# coding: utf-8
# Version 0 : basée sur l'âge, non interactif
# Usage   : pronostic_ballon_d_or.py PRENOM NOM AGE
# Exemple : pronostic_ballon_d_or.py Lionel Messi 32

PRENOM = sys.argv[1]              # au lieu de : input("Votre prénom ?")
NOM = sys.argv[2]                 # au lieu de : input("Votre nom ?")
AGE = int(sys.argv[3])            # au lieu de : int(input("Votre âge ?"))
if (AGE > 35):
  print("Aucune chance de gagner le Ballon d'Or !")
else:
  print("Vous avez de fortes chances de gagner le Ballon d'Or ! ")
```


Notion n°7 : paramétrisation (des programmes ou sous-programmes) (5)

- Grâce aux paramètres un programme peut faire appel à d'autres programmes (déjà construits) pour rendre son service, dans une logique de sous-traitance

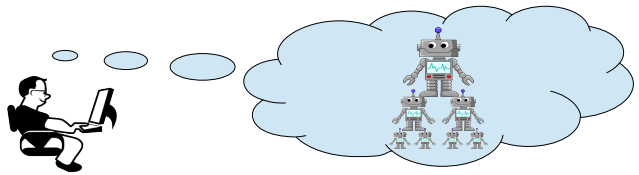


Terminologie :

- les paramètres traités par le développeur (sys.argv) sont appelés **paramètres** (ou **arguments**) **formels**
- Les valeurs utilisées à l'appel (rose.jpg, -resize, ...) sont appelés **paramètres** (ou **arguments**) **actuels**

Notion n°7 : Sous-programmes internes (fonctions) - Introduction (1)

- Grâce aux paramètres un programme peut faire appel à d'autres programmes (déjà construits) pour rendre son service, dans une logique de sous-traitance



- Or, il est **possible** mais n'est **pas nécessaire** que ces sous-traitants soient des programmes (fichiers exécutables) **indépendants**
- Il est bien **plus courant** de définir des sous-programmes **à l'intérieur** du programme lui même (dans le même fichier exécutable)

Notion n°7 : Sous-programmes internes (fonctions) - Introduction (2)

- Les fonctions sont des sous-programmes définissables **à l'intérieur** du programme lui même.
- Syntaxe Python des **définitions** de fonction (*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'appelant) la valeur représentée par EXPR.
Exemples de définitions :

```
def double(X):
  return(X*2)
```

```
def dix_fois(Y):
  return(Y*10)
```

- Syntaxe d'**appel** (*application*) : à la place de toute EXPR on pourra écrire :

```
NOM (EXPR1, … ,EXPRn)
```

  - Exemples :

```
print("Le double de %d est %d" % (7, double(7)))
C = dix_fois(A) + double(B)
```

Adresse des images utilisées

- Boite fermée https://openclipart.org/detail/15872/closed-box-by-mcol
- Robot sympa https://openclipart.org/detail/170101/cartoon-robot-by-sirrob01
- Développeur https://openclipart.org/detail/37129/personnage_ordinateur-by-antoine
- Utilisateur https://openclipart.org/detail/37135/personnage_ordinateur-by-antoine-37135