Approche technique de l'espace numérique

Pablo Rauzy
pr@up8.edu
pablo.rauzy.name/teaching/aten

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS
Institut Français de Géopolitique / M2 Cyberstratégie & Datascience

Séance 3

Premiers pas avec Python
Introduction à la programmation réseau

Premiers pas avec Python

Python

- Python a été créé en 1991 par Guido van Rossum.
- C'est un langage très répandu pour lequel il existe énormément de *bibliothèques*.
- Python est *dynamiquement typé*.
- Python est plutôt prévu pour la programmation *impérative*.
- Python offre un support de la programmation *orientée objet* (à base de classe).

La syntaxe de Python

- La *syntaxe* de Python se veut très lisible.
- L'indentation et les retours à la ligne sont significatifs :
  - une instruction se termine avec un retour à la ligne,
  - les blocs sont marqués par l'indentation.
- Les commentaires sont tout ce qui suit le symbole # sur une ligne jusqu'à la fin de celle-ci.

Variables, expressions, affectations

- Les variable n'ont pas de types et il n'y a pas besoin de les déclarer.
- La convention en Python est de nommer les variables en minuscules en séparant les mots par des underscores.
- L'opérateur d'affection est `=`.
- Exemples :
  - `distance = speed * duration`
  - `length_in_cm = 2.54 * length_in_inch`

! Attention, ce `=` est différent du $=$ des mathématiques !

True, False, None

- Les valeurs booléennes en Python sont `True` et `False`.
- Il existe aussi une valeur `None` qui veut dire “rien”.
- Les opérateurs de comparaison booléenne renvoient `True` ou `False`.
- L’égalité se teste avec $=$, l’inégalité avec $\neq$.
- Pour tester si une valeur `expr` est `None` on utilise `expr is None`.

Nombres

- Les nombres peuvent être entier ($\mathbb{Z}$) ou flottant ($\sim \mathbb{R}$).
- Exemples :
  - `13.51`
  - `42`

Chaînes de caractères

- Les chaînes de caractères sont notées entre simple ou double quote (`'` ou `"`).
- Selon lequel on utilise il faut l'échapper avec un `\`.
- Exemples :
  - `"Je m'appelle Python"`
  - `"dites \"AAAAAAH\"."`
  - `'Je m\'appelle Python'`
  - `'dites "AAAAAAH".'`
- On peut utiliser certains opérateurs sur les chaînes :
  - `"cou" * 2 # vaut "coucou"`
  - `"géo" + "politique" # vaut "géopolitique"`

Tuples

- Python permet de manipuler des paires, des triplets, des quadruplets, ...
- La syntaxe est de séparer les expressions par des virgules.
- Exemples :
  - `nom, age = "Sam", 24`
  - `a, b = b, a`

! Attention `1,23` est la paire composée de `1` et `23`, pas le nombre `1.23`.

Listes

- Une *liste* (ou *tableau*/*vecteur*, c'est confondu en Python) se notent entre crochets et leurs éléments séparés par des virgules.
- On accède à un élément d'une liste en donnant son indice entre crochets.
- Exemples :
  - `one_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
  - `one_to_ten[3] # 4`

Dictionnaires

- Un *dictionnaire* (ou *tableau associatif*) en Python permet de stocker des associations clef-valeur.
- On note les dictionnaires entre accolades, leurs entrées séparées par des virgules, et les clefs séparées des valeurs par des `:`.
- On accède à une valeur avec sa clef entre crochets.
- Exemple :

```
1  mdlp = {
2    'niveau': 'L1',
3    'semestre': 1,
4    'enseignant·es': {
5      'A': 'P Rauzy',
6      'B': 'JJ Bourdin',
7      'C': 'A Pappa',
8      'D': 'S Chalençon'
9    }
10 }
11
12 mdlp['niveau'] # 'L1'
13 mdlp['enseignant·es']['A'] # 'P Rauzy'
```

- La syntaxe des conditions est la suivante :
  - ```
    if condition:
      then-block
    elif other-condition:
      otherwise-block
    else:
      else-block
    ```
- Il peut y avoir zéro ou plusieurs bloc **`elif`** après un bloc **`if`**.
- Il peut y avoir zéro ou un bloc **`else`** à la fin.
- Les branches sont introduites par un symbole `:` puis délimitées par l'indentation.
- La convention en Python est d'utiliser 4 espaces comme indentation
  (j'en utilise parfois 2 par habitude, mais c'est mieux de suivre les conventions !).
    ```

- Il existe deux types de boucles :
  - “tant que”, qui répète un bloc d’instructions tant qu’une condition est vraie,
  - “pour … dans”, qui répète un bloc d’instructions pour chaque valeur dans un conteneur.
- Leur syntaxe sont :
  - ```
    while bool-expr:
      do-block
    ```
  - ```
    for v in iterable:
      do-block
    ```
- Même syntaxe que pour les branches conditionnelles.

Compréhension de liste

- Il y a une syntaxe spéciale pour faire des opérations sur les éléments d'une liste.
- Pour créer une nouvelle liste à partir des éléments d'une liste existante :
  - `[expr(x) for x in lst]`
- On peut au passage filtrer certains éléments de la liste :
  - `[expr(x) for x in lst if pred(x)]`
- Exemples :
  - `lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - `[n * 2 for n in lst]`
  - `[n * n for n in lst if (n % 2) == 0]`

Fonctions

- Les fonctions en Python sont définies avec le mot-clef **`def`**.
- La convention en Python est de nommer les fonctions en minuscules en séparant les mots par des underscores.
- On utilise **`return`** `expr` pour renvoyer une valeur et quitter la fonction.
  - Utilisé sans rien derrière c'est équivalent à **`return`** `None`.
- Exemples :

```
def fact (n):
  result = 1
  while n > 1:
    result = n * result
    n = n - 1
  return result
```

```
def get_distance (speed, duration):
  return speed * duration
```

- Appels de fonctions :
  - `fact(10)`
  - `distance = get_distance(50, 0.25)`

Portées des variables

- On appelle la *portée* d'une variable la partie du code dans laquelle on peut y accéder.
- Les variables qui sont initialisées en dehors d'une fonction sont *globales* (accessibles partout).
- Les variables qui sont initialisées dans une fonction sont *locales*, elles n'existent que dans cette fonction.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  a = 5 # global
2
3  def fct (arg):
4    b = "foo"
5    r = b * (a + arg) # ici a est accessible
6    a = 10
7    return r
8
9  # b et r ne sont plus accessibles ici
10 # ici a vaut
```

Portées des variables

- On appelle la *portée* d'une variable la partie du code dans laquelle on peut y accéder.
- Les variables qui sont initialisées en dehors d'une fonction sont *globales* (accessibles partout).
- Les variables qui sont initialisées dans une fonction sont *locales*, elles n'existent que dans cette fonction.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  a = 5 # global
2
3  def fct (arg):
4    b = "foo"
5    r = b * (a + arg) # ici a est accessible
6    a = 10
7    return r
8
9  # b et r ne sont plus accessibles ici
10 # ici a vaut 5
```

Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a)
9  print(b)
10 f(a, b)
11 print(a)
12 print(b)
```

Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b)
10 f(a, b)
11 print(a)
12 print(b)
```

Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a)
12 print(b)
```

Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a) # [0, 1, 0, 1]
12 print(b)
```

Passage des arguments

- En Python les arguments sont passés aux fonctions *par référence*.
- Cela veut dire que si on fait une modification *en place* de ces variables, alors elles seront affectées en dehors de la fonction aussi.
- L'affectation n'est pas une modification "en place", elle crée une variable locale.
- Exemple :

```
1  def f (lst1, lst2):
2    lst1 = [1, 2, 3] # affectation d'une nouvelle liste
3    lst2.append(13) # ajout d'élément en place
4
5  a = [0, 1, 0, 1]
6  b = [10, 11, 12]
7
8  print(a) # [0, 1, 0, 1]
9  print(b) # [10, 11, 12]
10 f(a, b)
11 print(a) # [0, 1, 0, 1]
12 print(b) # [10, 11, 12, 13]
```

Arguments par défaut

- Il est possible de spécifier une valeur par défaut pour certains arguments (qui doivent tous être après ceux qui n'ont pas de valeur par défaut).
- Exemple :

```
1 def get_distance (speed, duration = 1/6):
2   # par défaut, on calcul la distance parcouru en 10 minutes}
3   return speed * duration
4
5 get_distance(50, 0.5) # une demi-heure en agglomération
6 get_distance(130)     # 10 minutes sur l'autoroute
```

1. Écrire un programme qui compte de 0 à 100, mais remplace les nombres divisibles par 3 par “Fizz”, ceux divisibles par 5 par “Buzz”, et ceux qui le sont par 3 et par 5 par “Fizz Buzz”.
2. Écrire une fonction qui prend une liste de contacts et une lettre et renvoie la liste des emails de celleux dont le nom commence par cette lettre (on imagine que chaque contact est enregistré comme dictionnaire avec comme informations son prénom, son nom, et son adresse email).

- Informations (peut-être) utiles :
  - La fonction `print(a)` affiche la valeur de son argument `a`.
  - `range(min,max)` retourne une liste dont les éléments sont les nombres de `min` à `max` (non-inclus).
  - Les caractères d’une chaîne de caractères sont accessibles comme les éléments d’une liste.
  - La fonction `str.lower(s)` renvoie la chaîne `s` mais avec toutes les lettres en minuscules.

Démonstration : petit jeu

- Regardons ensemble la construction étape par étape d'un petit jeu développé avec la bibliothèque `pygame`.

Introduction à la programmation réseau

- On appelle “programmation réseau” le code qui sert à communiquer entre machines/logiciels, du côté application.
- On utilise pour cela des abstractions qui permettent de ne se soucier que du minimum : les *sockets*.

Les sockets

- L'idée est que lire et écrire sur une connexion réseau devrait se faire de la même manière que lire et écrire dans un fichier.
- Une *socket* est un fichier *virtuel* qui sert de point d'accès au réseau.
- Grâce aux sockets, un *client* peut se connecter à un *serveur*.

- Des deux côtés, il faut commencer par ouvrir une socket.
- Pour cela, il faut lui spécifier un *domaine*, un *type*, et un *protocole*.
- Le domaine correspond au protocole de la couche réseau (IPv4 ou IPv6 par exemple).
- Le type va généralement correspondre au protocole de la couche transport (TCP ou UDP).
- Le protocole, dans le cas où le type n'est pas TCP ou UDP mais "RAW", indique le protocole qui va être utilisé au dessus de la couche réseau.

- Des deux côtés, il faut commencer par ouvrir une socket.
- Pour cela, il faut lui spécifier un *domaine*, un *type*, et un *protocole*.
- Le domaine correspond au protocole de la couche réseau (IPv4 ou IPv6 par exemple).
- Le type va généralement correspondre au protocole de la couche transport (TCP ou UDP).
- Le protocole, dans le cas où le type n'est pas TCP ou UDP mais "RAW", indique le protocole qui va être utilisé au dessus de la couche réseau.
- Pour la suite on va se mettre dans le cas le plus courant : une socket TCP.

- Une fois la socket ouverte, côté serveur, il faut lui dire, dans l'ordre :
  - de s'attacher à une adresse, càd un nom d'hôte et un numéro de port ;
  - de se mettre en mode passif pour écouter ;
  - d'attendre une connexion d'un client.
- Cela correspond aux appels à `bind`, `listen`, et `accept`.

- Du côté du client, il suffit de demander à se connecter à une adresse (toujours un nom d'hôte et un numéro de port).
- Cela correspond à l'appel `connect`.

- Une fois connectés, un client et un serveur peuvent échanger dans les deux sens.
- Il suffit de lire ou écrire dans la socket, comme si il s'agissait d'un fichier local.

- En Python, la bibliothèque standard du langage contient un module `socket` :
  - `import socket`
- Pour créer une socket TCP :
  - `sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)`
- Du côté serveur :
  - ```
    addr = "localhost", 2345
    sock.bind(addr)
    sock.listen(8)
    client, _ = sock.accept()
    ```
- Du côté client :
  - ```
    addr = "localhost", 2345
    sock.connect(addr)
    ```
- Ensuite on peut utiliser les méthodes `send` et `recv`.

- Regardons ensemble des exemples simples d'un client et d'un serveur.