

Initiation à la programmation avec Python (v3)

Cours n°4

Copyright (C) 2015 - 2019

Jean-Vincent Loddo

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

Sommaire du cours n°4

- Notion n°5 : Structures de données : les **n-uplets**
- Notion n°5 : Structures de données : les **dictionnaires**
- Notion n°9 : Les modules ou bibliothèques de sous-programmes

Notion n°5 : La structure de données des **n-uplets** de valeurs (1)

- Les **n-uplets** (type **tuple**) sont des séquences finies de valeurs, comme les listes !
 - Il s'agit d'une **structure énumérable** similaire aux listes : il peut y avoir un 1^{er} élément (indice 0), un 2^{ème} (indice 1), un 3^{ème} (indice 2), ainsi de suite
 - En Python, on les exprime avec des **parenthèses** (au lieu des crochets pour les listes) et on les sépare avec des virgules (comme pour les listes). Exemples :

```
(1, "chien", 1.16, True) # ceci est un n-uplet (tuple)
[1, "chien", 1.16, True] # ceci est une liste (list)
```

- Exemples

```
T = (1, "chien", 1.16, True) # T contient un 4-uplet (tuple)
S = T[0] + T[2]             # S contient 2.16
C = T[1] * 2                # C contient "chienchien"
B = not(T[3])               # B contient False
```

3

Notion n°5 : La structure de données des **n-uplets** de valeurs (2)

- En python, les opérations sur les listes que nous connaissons ont le même sens sur les n-uplets :

```
T = (1, "chien", 1.16, True)
U = T + T # U contient (1, "chien", 1.16, True, 1, "chien", 1.16, True)
V = T * 2 # V contient la même chose que U
```

- Note sur la syntaxe des **1-uplets** : il faut rajouter une virgule pour éviter que les parenthèses prennent leur sens habituel :

```
"Célibataire" # c'est une chaîne (string)
("Célibataire") # c'est toujours une chaîne
("Célibataire",) # c'est un 1-uplet (contenant une chaîne)
```

```
("François", "Hollande", 61, "Avocat") + ("Célibataire",)
```

- Nous n'avons pas ce problème avec les listes : les crochets évitent toute ambiguïté, même pour les listes singletons

4

Notion n°5 : La structure de données des n-uplets de valeurs (3)

- La fonction **tuple()** permet de créer des tuples à partir de structures de données énumérables comme les listes (**list**) ou les chaînes de caractères (**string**) :
- Exemples :
 - `tuple("Hello")`
`('H', 'e', 'l', 'l', 'o')`
 - `tuple([1,3,5])`
`(1, 3, 5)`
 - `tuple(range(0,10))`
`(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)`
- La fonction **list()** est l'équivalent pour les listes : elle permet de créer des listes à partir de tout type de structure énumérable

5

Notion n°5 : La structure de données des n-uplets de valeurs (4)

- Les **n-uplets** (type **tuple**) sont des séquences finies de valeurs, comme les listes (type **list**) !
- On peut faire les mêmes opérations qu'avec les listes
- **Quelle est alors la différence** avec les listes ?
- Réponse **n°1**, de façon générale, au-delà du cas Python :
 - Une **liste** a vocation à contenir des valeurs du même type (séquences **homogènes**)
`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
 - Un **n-uplet** a vocation à contenir des valeurs de types différents (séquences **hétérogènes**)
`("Français", "Hollande", 61, "Avocat", True)`
 - Mais en Python il n'est pas interdit d'avoir des **listes hétérogènes** (ni des **n-uplets homogènes**)

6

Notion n°5 : La structure de données des n-uplets de valeurs (5)

- Les **n-uplets** (type **tuple**) sont des séquences finies de valeurs, comme les listes (type **list**) !
 - On peut faire les mêmes opérations qu'avec les listes
 - **Quelle est alors la différence** avec les listes ?
 - Réponse **n°2**, dans le cas particulier de **Python** :
 - Les composantes (cases) d'une **liste** sont **modifiables** (peuvent être affectées)
 - Les composantes (cases) d'un **n-uplet** ne sont **pas modifiables** (ne peuvent pas être affectés)
`L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
`T = ("Français", "Hollande", 61, "Avocat", True)`
`L[3] = 2015` # c'est possible (L est une liste)
`T[3] = "Président"` # non autorisé (T est un n-uplet), erreur d'exécution !
- ```
TypeError: 'tuple' object does not support item assignment
```

7

## Notion n°5 : La structure de données des dictionnaires (1)

- Un **dictionnaire** est un ensemble modifiables de couples clef-valeur (*associations* ou *liens*) à clef unique
  - En Python, on les exprime avec des **accolades** contenant les couples clef-valeur séparés par des **virgules**. Les couples clef-valeur s'expriment par deux expressions séparés par **deux points** (:)  
`{"Français", 61, "Avocat"}` # c'est une liste  
`("Français", 61, "Avocat")` # c'est un n-uplet  
`{"prenom":"Français", "age":61, "job":"Avocat"}` # c'est un dictionnaire
  - L'**accès** aux éléments n'est pas très différent par rapport aux listes et n-uplets : au lieu d'utiliser des index (entiers), on utilise la clef
  - Les clefs sont type quelconque **non modifiable**, par exemple de chaînes de caractères (**string**) ou des n-uplets (**tuple**)
- Exemples  
`D = {"prenom":"Français", "age":61, "job":"Avocat"}`  
`P = D["prenom"]` # P contient "Français"  
`D["job"] = "Président"` # mise à jour possible  
`J = D["job"]` # J contient "Président"

8

## Notion n°5 : La structure de données des dictionnaires (2)

- Un **dictionnaire** est un ensemble modifiables de couples clef-valeur (*associations* ou *liens*) à clef unique
  - La **clef** n'est pas forcément une chaîne de caractère (**string**), mais c'est ce que nous utiliserons le plus souvent
  - Les **clefs** présentes ne doivent pas forcément être **homogènes** (même type), mais c'est le plus naturel

### Exemples

```
STOCK = {"poires":15, "pommes":23, "bananes":12}
STOCK["bananes"] = STOCK["bananes"] + 18 # mise à jour (30 bananes)
STOCK["citrons"] = 20 # nouveau couple (clef,valeur)
del STOCK["poires"] # ce n'est plus la saison
"citrons" in STOCK # c'est True
"poires" in STOCK # c'est False

for i in STOCK: print "Le stock de",i,"est",STOCK[i]
```

- Affiche :

```
Le stock de bananes est 30
Le stock de pommes est 23
Le stock de citrons est 20
```

9

## Notion n°5 : La structure de données des dictionnaires (3)

- Quelques outils (fonctions) sur les dictionnaires :

- L'application **len(D)** rend le nombre de couples du dictionnaire D

```
len(STOCK) # rend 3
```

- L'application (infixe) **K in D** rend un booléen indiquant la présence de la clef K dans le dictionnaire D

```
"citrons" in STOCK # rend True
```

- L'application **list(D)** donne la liste des clefs du dictionnaire D

```
list(STOCK) # rend ["poires", "pommes", "citrons"]
```

- L'application **del(D[K])** élimine le lien ayant pour clef (unique) K du dictionnaire D

```
del(STOCK["citrons"]) # avec ça list(STOCK) == ["poires", "pommes"]
```

10

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (1)

### 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*)
- Le mot **bibliothèque** (library, package) est généralement utilisé pour indiquer un ensemble de composants (modules, classes) dans une même thématique (entrées-sorties, mathématiques, graphismes, ..)

11

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (2)

### 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

### Exemples

- Le module **turtle** en **Python** s'occupe de fournir tous les outils pour travailler avec la tortue graphique
- La classe **FileInputStream** en **Java** s'occupe de fournir des outils pour lire le contenu des fichiers
- La classe **Exception** en **PHP** s'occupe de fournir des outils pour contrôler l'exécution dans des situations erronées ou exceptionnelles

12

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (3)

- Pourquoi créer ses propres modules
  - Pour les mêmes raisons que les modules prédéfinis (**structurer**, **réutiliser**)
- Comment créer ses propres modules en Python
  - En réunissant le code dans un fichier, nommé par exemple **foo.py** (**Attention** à ne pas utiliser un nom de module déjà existant, par exemple **turtle.py**, car cela rendrait un des deux inaccessible)
  - Puis en déclarant vouloir l'utiliser dans un autre fichier source (**.py**), par le mot clef **import**, et en faisant appel aux fonctions à l'intérieur par le nom du module, un **point** et le nom de la fonction :

```
import foo
foo.trace_carre(100,"red") # trace_carre() est définie dans foo.py
```

Syntaxe d'accès à un élément du module :  
**MODULE.NOM**

13

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (4)

- Comment créer ses propres modules en Python
  - En réunissant le code dans un fichier, nommé par exemple **foo.py**
  - Puis en déclarant vouloir l'utiliser dans un autre fichier, par le mot clef **import**, et en faisant appel aux fonctions à l'intérieur par le nom du module, un **point** et le nom de la fonction
  - On peut aussi importer **plusieurs** modules à la fois :

```
import foo, bar, baz
foo.trace_carre(100,"red") # trace_carre() est définie dans foo.py
bar.cache_nombre(1000) # cache_nombre() est définie dans bar.py
baz.tire_lire("vide") # tire_lire() est définie dans baz.py
```

14

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (5)

- Comment créer ses propres modules en Python
  - En réunissant le code dans un fichier, nommé par exemple **foo.py**
  - Puis en déclarant vouloir l'utiliser dans un autre fichier, par le mot clef **import**, et en faisant appel aux fonctions à l'intérieur par le nom du module, un **point** et le nom de la fonction
  - On peut aussi **éviter la notation pointée** (avec le nom du module en préfixe) :

```
from foo import *
trace_carre(100,"red") # trace_carre() est définie dans foo.py
```

- On peut aussi importer seulement certaines fonctions, **pas toutes** :

```
from foo import trace_carre, trace_rectangle
trace_carre(100,"red") # trace_carre() est définie dans foo.py
trace_rectangle(200,150,"blue") # trace_rectangle() est définie dans foo.py
```

15

## Notion n°9 : Les modules ou bibliothèques de sous-programmes (6)

- Comment créer ses propres modules en Python
  - En réunissant le code dans un fichier, nommé par exemple **foo.py**
  - Puis en déclarant vouloir l'utiliser dans un autre fichier, par le mot clef **import**, et en faisant appel aux fonctions à l'intérieur par le nom du module, un **point** et le nom de la fonction
- Où est-ce que Python va chercher, sous Unix, les fichiers sources au moment des importations ?
  - (1) Le répertoire courant (**PWD**)
  - (2) Tour à tour les répertoires de la variables d'environnement **PYTHONPATH** (où les chemins sont séparés par des ":" comme pour **PATH**)
  - (3) Répertoire par défaut de l'installation Python (normalement **/usr/local/lib/python/**)

16