Les données de types construits

Introduction

Dans le chapitre "Premiers pas avec Python", nous avons utilisé des données de type `int`, `float`, et `bool` ce sont des données de `type simple`.

Le type `str` est particulier car chaque caractère de la chaine a un indice (voir ci-dessous).

Les données de `type simple` sont suffisantes pour traiter un petit nombre de données, mais si le nombre de données devient plus important (ou s'il est intéressant d'en regrouper certaines comme par exemple les coordonnées d'un point), nous avons besoin de définir des variables dont les valeurs sont des ensembles de valeurs.

Nous utiliserons alors des données de `types construits` comme les `séquences`.

Une `séquence` est un conteneur (un objet ayant vocation à en contenir d'autres) ordonné d'éléments indicés par des entiers, Python dispose de trois types prédéfinis de séquences :

- Les chaînes de caractères
- Les listes
- Les tuples

Sur ces objets que sont ces `séquences`, nous pourrons utiliser des méthodes permettant d'agir sur ces objets.

À la différence des `séquences` qui sont indexées par des nombres, les `dictionnaires` sont indexés par des `clés` (qui peuvent être des chaînes de caractères, des nombres, des tuples...), on parle alors de `collections`.

Les chaînes de caractères

Nous avons déjà abordé la notion de chaîne de caractères puisqu'il s'agit de variables de type `str` (`string`) :

```
>>> nom = 'Dupont'        # entre apostrophes
>>> nom
'Dupont'
>>> type(nom)
<class 'str'>
>>> prenom = "Pierre"     # on peut aussi utiliser les guillemets

>>> prenom
'Pierre'
>>> print(nom, prenom)    # ne pas oublier la virgule
Dupont Pierre
```

On peut réaliser une concaténation de chaînes de caractères à l'aide de l'opérateur + :

```
>>> chaine = nom + prenom   # concaténation de deux chaînes decaractères
>>> chaine
'DupontPierre'

>>> chaine = prenom + nom   # concaténation de deux chaînes de caractères
>>> chaine
PierreDupont

>>> chaine = prenom + ' ' + nom
>>> chaine
'Pierre Dupont'

>>> chaine = chaine + ' 18 ans'
>>> chaine
'Pierre Dupont 18 ans'
```

La fonction `len()` retourne la longueur (length) de la chaîne de caractères :

```
>>> len(chaine)
20
```

On peut facilement atteindre un élément ou une partie de la chaîne de caractères :

```
>>> chaine[0]            # premier caractère (indice 0)
P
>>> chaine[1]            # deuxième caractère (indice 1)
i
>>> chaine[1:4]          # slicing
ier
>>> chaine[2:]           # slicing
erre Dupont 18 ans
>>> chaine[-1]           # dernier caractère (indice -1)
s
>>> chaine[-6:]          # slicing
18 ans
```

Mais on ne peut pas modifier un élément (ou plusieurs) d'une chaîne de caractères :

```
>>> chaine[7]
'D'
>>> chaine[7] = 'd'
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Remarque : Quand une chaîne de caractères contient une apostrophe, on peut utiliser le caractère d'échappement `\` ou les guillemets `"` pour éviter son interprétation.

```
>>> chaine = 'Aujourd'hui'
SyntaxError: invalid syntax

>>> chaine  = 'Aujourd\'hui'    # séquence d'échappement \'
>>> chaine
"Aujourd'hui"

>>> chaine  = "Aujourd'hui"
>>> chaine
"Aujourd'hui"
```

On retiendra donc que les chaînes de caractères (`type str`) sont des séquences où chaque élément de la chaîne est accessible par un indice (qui débute à 0) mais les éléments ne sont pas modifiables directement (elles sont `immutables`).

Les listes

Les `listes` vont nous permettre de stocker plusieurs valeurs (chaîne de caractères, nombre, ...) dans une structure unique.

Ce sont donc des `séquences`, comme les chaînes de caractères, mais au lieu de contenir des caractères, elles peuvent contenir n'importe quel objet.

Attention : dans la plupart des autres langages, on parlera plutôt de `tableau` que de `liste` mais les concepteurs de Python ont choisi d'utiliser le terme de `liste`. Ne soyez pas étonné d'entendre parler de l'un ou l'autre.

Créer une liste

Dans l'exemple ci-dessous, la variable `villes` contient plusieurs villes de France :

```
villes = ["Bordeaux", "Paris", "Marseille", "Lyon", "Lille"]
```

Remarque : les listes sont délimitées par deux crochets en python entre lesquels on trouve les éléments (les items) séparés par une virgule.

Dans l'exemple ci-dessous, la variables `ages` contient plusieurs valeurs numériques :

```
ages = [10, 11, 12, 13, 14, 15, 16, 17, 18]
```

Les items d'une liste peuvent être de type varié :

```
jours = ["lundi", "mardi", "mercredi", 8, 10, 12]
```

Ces trois variables sont bien de type `list` :

```
>>> type(villes)
<class 'list'>
>>> type(ages)
<class 'list'>
>>> type(jours)
<class 'list'>
```

Afficher un item d'une liste

Pour afficher un item d'une liste, il suffit d'utiliser son indice de position (index), comme pour les chaînes de caractères, le 1er élément de la liste a l'indice 0.

```
>>> villes = ["Bordeaux", "Paris", "Marseille", "Lyon", "Lille"]
>>> villes[1]
'Paris'
```

Remarque : oublier que l'indice de position du 1er élément d'une liste est 0 et pas 1 est une erreur « classique ».

Utiliser une boucle while pour parcourir une liste

Parcourir une liste, peut se faire à l'aide d'une boucle while :

```
semaine = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
i = 0
while i < 5:
    print(semaine[i])
    i = i + 1
print("C'est terminé")

# On obtient
lundi
mardi
mercredi
jeudi
vendredi
C'est terminé
```

Utiliser une boucle for pour parcourir une liste

Il est plus facile de parcourir les éléments d'une liste avec une boucle `for` qu'avec une boucle `while` :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
for index in range(5):
    print(semaine[index])
print("C'est terminé")

# on obtient
lundi
mardi
mercredi
jeudi
vendredi
C'est terminé
```

Mais attention à ne pas demander un item de la liste inexistant sous peine d'obtenir un message d'erreur du type `list index out of range` :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
for index in range(6):
    print(semaine[index])
print("C'est terminé")

# on obtient
lundi
mardi
mercredi
jeudi
vendredi
Traceback (most recent call last):
  File "C:\Users\Frédéric\Desktop\test.py", line 3, in <module>
    print(semaine[index])
IndexError: list index out of range
```

Pour éviter cela, on peut itérer en tenant compte de la longueur de la liste :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
for index in range(len(semaine)):
    print(semaine[index])
print("C'est terminé")

# on obtient
lundi
mardi
mercredi
jeudi
vendredi
C'est terminé
```

Mais la manière la plus élégante est d'utiliser l'instruction `in` :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
for jour in semaine:
    print(jour)
print("C'est terminé")

# On obtient
lundi
mardi
mercredi
jeudi
vendredi
C'est terminé
```

Remarque : La variable `jour` prendra successivement toutes les valeurs contenues dans la liste `semaine`. Par contre, on ne récupère pas le numéro de l'index.

Modification d'une liste

Contrairement aux `chaînes de caractères`, il est possible de modifier une `liste` après sa création en ajoutant des items, en les modifiant, voire en les supprimant : les listes sont `mutables`.

Ajouter un item

Il est possible d'ajouter un item à une liste (en fin de liste plus précisément) grâce à la méthode `append`.

Pour ajouter l'item `samedi` à la fin de la liste `semaine`, il faut écrire :

```
>>> semaine.append('samedi')
>>> semaine
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
```

Modifier un item

Il est possible de modifier un item d'une liste en le remplaçant directement à l'aide de son index :

```
>>> semaine[0] = 'monday'
>>> semaine
['monday', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

Supprimer un item

Pour supprimer un item, il y a deux manières possibles :

```
# avec la commande del (et le numéro de l'item):
>>> del semaine[5]               # effacera le 'samedi'
# avec la méthode remove (et la valeur de l'item):
>>> semaine.remove('dimanche')    # effacera le 'dimanche'
```

Quelques méthodes supplémentaires

```
# inverser les items d'une liste
>>> semaine
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
>>> semaine.reverse()
>>> semaine
['dimanche', 'samedi', 'vendredi', 'jeudi', 'mercredi', 'mardi', 'lundi']

# compter les occurences d'une valeur
>>> semaine = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi',
'dimanche', 'lundi', 'lundi']
>>> semaine.count('lundi')
3

# trouver l'index d'un item
>>> semaine = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi',
'lundi', 'dimanche']

>>> semaine.index('lundi')   # attention s'il existe plusieurs items

0                            # identiques c'est l'index le plus petit
                             # qui sera renvoyé
```

Créer une liste contenant n éléments identiques

Pour créer une liste contenant n items identiques, il suffit de mettre l'élément entre crochets et de le multiplier par n :

```
>>> liste = [5] * 10   # création d'une liste contenant 10 fois l'item 5
>>> liste
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]

>>> liste = ['merci'] * 5
>>> liste
['merci', 'merci', 'merci', 'merci', 'merci']
```

Si on désire créer une liste d'items, on peut aussi créer une liste vide que l'on complète ensuite avec une boucle :

```
resultat = []
for i in range(10):
    resultat.append(i*2)

# on obtient :
>>> resultat
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Cela fonctionne très bien mais en Python, il existe une écriture plus compacte qui utilise la création de liste par `compréhension` :

```
resultat = [i*2 for i in range(10)]

# on obtient :
>>> resultat
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

On peut aussi ajouter des conditions dans la création de listes par `compréhension` :

```
# le code suivant :
resultat = []

for i in range(10):
    if(i % 2 == 0):
        resultat.append(i)

# s'écrire plutôt :
resultat = [i for i in range(10) if i % 2 == 0]
```

Les tableaux de nombres (ou matrice)

Dans certains cas, il est utile de manipuler des données sous forme de matrice (un tableau de nombres) :

$$\begin{pmatrix} 2 & 3 & 5 \\ 8 & 4 & 1 \\ 7 & 6 & 9 \end{pmatrix}$$

Si on souhaite représenter des matrices à partir de listes, l'astuce à utiliser sera de construire des listes de listes.

Si l'on veut représenter la matrice ci-dessus, on écrira :

```
>>> matrice = [[2, 3, 5], [8, 4, 1], [7, 6, 9]]
```

Ainsi, pour accéder à l'élément d'indice (i, j) de cette matrice, on écrira `matrice[i][j]` (on demande le j-ème élément de la i-ème ligne de la matrice) :

```
>>> matrice[1][0]
8
>>> matrice[0][2]
5
>>> matrice[2][2]
9
```

On peut modifier un élément de la matrice :

```
>>> matrice[2][2] = 25
>>> matrice
[[2, 3, 5], [8, 4, 1], [7, 6, 25]]
```

Astuce : pour créer une matrice remplie de 0, on peut utiliser le code suivant :

```
>>> matrice = [[0 for j in range(3)] for i in range(5)]
>>> matrice
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Attention aux copies de listes

La copie de liste en Python est un exercice délicat.

En effet, si l'on crée une première liste `semaine` que l'on désire copier en une seconde liste appelée `week`, il est tentant d'écrire :

```
>>> semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
>>> week = semaine
```

le contenu de `semaine` ne sera pas recopié dans `week`, les deux objets ne feront qu'un. Cela signifie notamment que si l'on modifie l'une des deux listes, la modification sera répercutée sur l'autre liste :

```
>>> semaine[4] = 'friday'
>>> semaine
['lundi', 'mardi', 'mercredi', 'jeudi', 'friday']
>>> week
['lundi', 'mardi', 'mercredi', 'jeudi', 'friday']
```

Remarque : ne pas hésiter à utiliser python tutor pour s'en rendre compte.

Ainsi, si l'on veut copier le contenu d'une liste dans une autre, on devra utiliser une astuce syntaxique. Plusieurs approches sont possibles.

La première utilise une liste vide que l'on complète :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
week = []
for index in range(len(semaine)):
    week.append(semaine[index])

>>> semaine
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> week
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> week[4] = 'friday'
>>> semaine
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> week
['lundi', 'mardi', 'mercredi', 'jeudi', 'friday']
```

La deuxième qui consiste à créer la copie par compréhension :

```
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
week = [jour for jour in semaine]
```

La troisième en utilisant la méthode `copy()` :

```
import copy
semaine = ['lundi','mardi','mercredi','jeudi','vendredi']
week = copy.copy(semaine)
```

Attention : si la liste est imbriquée (liste de liste), il faudra utiliser `deepcopy()` sinon les items de deuxième niveau seront partagés (à voir avec python tutor).

Les tuples

Comme déjà dit ci-dessus, un `tuple` est une `séquence`.

L'utilisation du terme anglais `tuple`, abréviation de quin-tuple/sex-tuple/..., est courante dans des ouvrages de programmation. En français, on parle plutôt de `n-uplet` : doublet, triplet...

Pour créer un `tuple`, on place les items séparés par des virgules entre parenthèses :

```
>>> mon_tuple = (5, 8, 6, 9)
>>> mon_tuple
(5, 8, 6, 9)

>>> type(mon_tuple)
<class 'tuple'>
```

Dans le code ci-dessus, la variable `mon_tuple` référence un `tuple` qui est constitué des entiers 5, 8, 6 et 9.

On constate que la variable `mon_tuple` est bien de type `tuple`.

Remarque : on peut créer un `tuple` sans mettre de parenthèses mais il est conseillé de les ajouter afin d'améliorer la lisibilité :

```
>>> mon_tuple = 5, 8, 6, 9
>>> mon_tuple
(5, 8, 6, 9)

>>> type(mon_tuple)
<class 'tuple'>
```

Un tuple ne contient pas forcément que des nombres entiers, il peut aussi contenir des nombres décimaux, des chaînes de caractères, des booléens... voir même un mélange de ces différents types de variables.

```
>>> tuple = (1, 6, 9, "Bonjour", False)
>>> tuple
(1, 6, 9, 'Bonjour', False)
```

Et pourquoi pas des tuples de tuples...

```
>>> t1 = 1, 6, 8
>>> t2 = 5, 8, 10
>>> tuple = (t1, t2)
>>> tuple
((1, 6, 8), (5, 8, 10))
```

Comme indiqué dans la définition d'une `séquence`, chaque élément du `tuple` possède un indice (ou index).

On peut ainsi facilement à l'élément de son choix dans un `tuple`, le principe est exactement le même que pour les `listes` :

```
mon_tuple = (5, 8, 6, 9)
>>> mon_tuple[1]
8
>>> mon_tuple[-1]
9
>>> mon_tuple[1:]
(8, 6, 9)
```

On peut tester l'appartenance à un tuple avec in

```
>>> mon_tuple = (5, 8, 6, 9)
>>> 5 in mon_tuple
True
>>> 12 in mon_tuple
False
```

Un `tuple` l'affectation multiple de valeurs :

```
>>> a, b, c, d = (5, 8, 6, 9)
>>> a
5
>>> b
8
>>> c
6
>>> d
9
```

Grâce au tuple, une fonction peut renvoyer plusieurs valeurs :

```
def add(a, b):
    c = a + b
    return (a, b, c)

>>> mon_tuple = add(5, 8)
(5, 8, 13)
```

Un tuple est immutable

Il n'est pas possible de modifier un tuple après sa création (on parle d'objet `immutable`), si vous essayez de modifier un `tuple` existant, l'interpréteur Python vous renverra une erreur.

```
>>> mon_tuple = (5, 8, 6, 9)
 >>> mon_tuple[2]
 6
 >>> mon_tuple[2] = 4
 Traceback (most recent call last):
 File "<pyshell>", line 1, in <module>
 TypeError: 'tuple' object does not support item assignment
```

Les dictionnaires

Les dictionnaires ressemblent aux `listes` mais chaque élément d'un dictionnaire est composé de 2 parties, on parle de pairs "clé/valeur".

Voici un exemple de dictionnaire :

```
monDico = {"nom": "Durand", "prenom": "Christophe", "date de naissance":
"29/02/1981"}
```

Comme vous pouvez le constater, nous utilisons des accolades {} pour définir le début et la fin du `dictionnaire` (alors que nous utilisons des crochets [] pour les listes).

Dans le dictionnaire ci-dessus, "nom", "prenom" et "date de naissance" sont des clés et "Durand", "Christophe" et "29/02/1981" sont des valeurs.

La clé "nom" est associée à la valeur "Durand", la clé "prenom" est associée à la valeur "Christophe" et la clé "date de naissance" est associée à la valeur "29/02/1981".

Les clés sont des chaînes de caractères ou des nombres. Les valeurs peuvent être des chaînes de caractères, des nombres entiers, des booléens (des objets non mutables), mais ne peuvent pas être des listes (objets mutables).

Pour créer un `dictionnaire`, il est aussi possible de procéder comme suit :

```
monDico = {}
monDico["nom"] = "Durand"
monDico["prenom"] = "Christophe"
monDico["date de naissance"] = "29/02/1981"
```

Pour afficher le contenu d'un dictionnaire, ou une valeur à partir de sa clé :

```
>>> monDico
{'nom': 'Durand', 'prenom': 'Christophe', 'date de naissance': '29/02/1981'}
>>> monDico['prenom']
'Christophe'
>>> monDico['nom']
'Durand'
>>> monDico['date de naissance']
'29/02/1981'
```

Remarque : si on recherche dans un dictionnaire avec une clé inexistante, on obtient un message d'erreur (KeyError) :

```
>>> monDico['name']
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
KeyError: 'name'
```

Il est donc préférable d'utiliser la méthode `get` qui ne renverra pas de message d'erreur (sauf si le spécifiez) :

```
>>> monDico.get('nom')
Durand
>>> monDico.get('name')
>>> monDico.get('name',"clé inexistante attention")
'clé inexistante attention'
```

Il est facile d'ajouter un élément à un dictionnaire :

```
>>> monDico['lieu naissance'] = 'Bonneville'
>>> monDico
{'nom': 'Durand', 'prenom': 'Christophe', 'date de naissance': '29/02/1981', 'lieu
naissance': 'Bonneville'}
```

On peut facilement vérifier l'existence d'une clé dans un dictionnaire avec l'instruction `in` :

```
>>> 'nom' in monDico
True
>>> 'name' in monDico
False
```

L'instruction `del` permet du supprimer une paire "clé / valeur" du dictionnaire :

```
>>> del monDico['date de naissance']
>>> monDico
{'nom': 'Durand', 'prenom': 'Christophe', 'lieu naissance': 'Bonneville'}
```

Il est possible de modifier une valeur :

```
>>> monDico['lieu naissance'] = 'Paris'
>>> monDico
{'nom': 'Durand', 'prenom': 'Christophe', 'lieu naissance': 'Paris'}
```

Il est possible de parcourir un dictionnaire à l'aide d'une boucle for. Ce parcours peut se faire selon les clés ou les valeurs.

Commençons par parcourir les clés à l'aide de la méthode `keys` (nous verrons la notion de méthode un peu plus tard) :

```
mesFruits = {"poire":3, "pomme":4, "orange":2, "pamplemousse":1}
for fruit in mesFruits.keys():
    print (fruit)

# on obtient :
poire
pomme
orange
pamplemousse
```

La méthode `values` permet de parcourir le dictionnaire selon les valeurs :

```
mesFruits = {"poire":3, "pomme":4, "orange":2, "pamplemousse":1}
for quantite in mesFruits.values():
    print (quantite)

# on obtient :
3
4
2
1
```

On peut obtenir facilement la liste de clés et des valeurs d'un dictionnaire :

```
>>> monDico.values()
dict_values(['Durand', 'Christophe', '29/02/1981'])
>>> monDico.keys()
dict_keys(['nom', 'prenom', 'date de naissance'])
```

Enfin, il est possible de parcourir un dictionnaire à la fois sur les clés et les valeurs en utilisant la méthode `item` :

```
mesFruits={"poire":3, "pomme":4, "orange":2, "pamplemousse":1}
print ("Stock de fruits :")
for fruit,quantite in mesFruits.items():
    print (fruit + " : " + str(quantite))

# on obtient :
Stock de fruits :
poire : 3
pomme : 4
orange : 2
pamplemousse : 1
```