Formation Python : Structures de données

par Bernard CHAMBON

CC-IN2P3, Lyon - France

26 et 27 juin 2017

Structures de données

- Liste, tuple
- Conversion liste | tuple $\leftrightarrow$ chaine
- Dictionnaire
- List comprehensions (dénomination anglaise)
- Les générateurs
- Les ensembles
- Structures de données du module `collections` : `dequeue`, `Queue`, `OrderedDict`
- Fonctionnalités avancées sur les listes : `map(...)`, `filter(...)`, `reduce(...)`

- Liste
  Création par `[]` ou en instanciant la classe `list()`
  Accès aux éléments par `[i]`, possiblement par tranche `[i:j]` voire en spécifiant le step `[i:j:step]`
  mais une liste est un object itérable i.e. accessible dans une séquence for `for x in my_list:`
- Premier exemple : création par `[]` , accès aux éléments par `[]` , itération

```
def playing_with_list(self):
  alist = ["Welcome", "to",  "the", "CC-IN2P3"]  # using [] to make a list

  # type, len
  print(type(alist), len(alist))  # will be <type 'list'>    4
  # access to element
  print(alist[3])  # will display CC-IN2P3 (index starts at 0)
  alist[3] = "CNRS"  # a list may be modified
  print(alist[3])  # will display CNRS
  print(alist[2:3+1])  # will display the sublist ['the', 'CNRS'],

  # iterate over a list
  for element in alist:
    print(element, end=" ")  # will display Welcome to the CNRS
  print()

  # checking if a value is in list using 'in'
  if ("CNRS" in alist):
    print("CNRS is in the list")
  else:
    print("CNRS keyword NOT found in the list")

  return
```

Synthèses de quelques opérations possibles sur les listes

- Fonctions

  `len(s)` nombre d'élément de la liste s

  `min(s)` | `max(s)` plus petite | grande valeur de la liste s

  `sum(s)` somme des valeurs de la liste s

  `del s[i]` :pour supprimer un objet à la position i

  `del s[i : j]` pour supprimer des objets entre les positions i et j

  `del alist[:]` ou `s[:]=[]` pour supprimer tous les objets de la liste

  opérateur `in` pour tester l'existence d'un objet dans un liste

  opérateur `not` pour tester si une liste est vide. Ex `if not my_list:`

  opérateur `+` pour concaténer deux listes

  `reversed(s)` renvoie un iterateur inversé

  Pour plus de possibilités voir les Built-in Functions

- Méthodes de la classe `list`

  `s.append(value)`, `s.extend(another_list)`, `s.insert(index, value)`

  `s.pop()`, `s.remove(value)`

  `s.index(x)` index de la premiere occurrence de x dans s. ValueError si non trouvé, utiliser `try - except`

  `s.count(x)` nombre d'occurrence de `x` dans `s`

  `s.reverse()` inverse la liste courante

  Pour plus de possibilités, voir les Méthodes de la classe List

- Deuxième exemple utilisant la classe `list()`

```
def playing_with_list_again(self):
  alist = list()  # alternative to alist = []

  # append, pop, insert, remove, clear the list
  for i in range(6, 0, -1):  # adding 6, 5, ..., 1
    alist.append(i)

  # to get a reverse iterator on the list
  riterator = reversed(alist)
  for riter in riterator :
    print("{} ".format(riter), end="") # will be 1 2 3 4 5 6

  # sub-list using slicing
  even_list =  alist[0:len(alist):2] # will be 6 4 2

  # other trick using slicing, to get a reversed list !
  rlist = alist[::-1] # will be 1 2 3 4 5 6

  #  pop will get (and remove) the last one (remember it's a LIFO)
  print("last element is {}, new length is {}".format(alist.pop(), len(alist)))

  alist.append("EndOfList")  # a list can store heterogeneous object's type

  # clearing list, check that after clear it's the same list (via id(list))
  alist[:]=[]  # also possible via del alist[:]
  # providing a new list (id as changed)
  alist = []

  return
```

Les listes sont des structures LIFO (voir la classe `dequeue` si besoin FIFO)

- Les tuples
  Séquences itérables comme les listes, mais NON modifiables
  expression par `()`
  accès aux éléments `[]` ou par iteration (comme pour les listes)
- Syntaxe par l'exemple

```
def playing_with_tuple():
  atuple =("Welcome", "to the", "CC-IN2P3") # using () to a make tuple

  # type, len
  print type(atuple), len(atuple) # will be <type 'tuple'>    3
  # access to element
  print atuple[2]                 # will display CC-IN2P3
  # atuple[2]="CNRS"              # error, a tuple CAN'T be modified
  # atuple.append(" bye")         # error, remember this is a list's method !

  # iterate over a tuple
  for element in atuple:
    print(element)              # will display Welcome to the CC-IN2P3
  return
```

Il est suffisant de savoir que les tuples existent, les reconnaitre via la notation `()` et savoir qu'on peut itérer
Exemple la fonction `zip` qui permet 'd'associer' des listes (ou des tuples)
et qui renvoi une liste de tuples avec Python 2, un itérable de tuples avec Python 3

```
l1=[1, 2, 3]
l2=[1, 4, 9]
print(zip(l1,l2))  # will give  [(1, 1),(2, 4),(3, 9)]
print(zip(l1,l2)[1][1]) # will give 4
```

- Conversion liste | tuple ↔ chaine : méthodes `join` et `split`
  - `join`
    génère une chaine résultant de la concaténation des éléments de la liste, passés en paramètre et du contenu de la chaine (à comprendre comme le séparateur)

```
alist = ["username", "password", "db"]
atuple =("mylogon", "mysecret", "mydb")

# join to concatenate with separator and get a string, possible on list or tuple
print(";".join(alist)) # will be "username;password;db"
print(" and ".join(atuple)) # will be "mylogon and mysecret and mydb"

# objective is to get  "username=mylogon;password=mysecret;db=mydb"
myzip = zip(alist, atuple) # [('username', 'mylogon'),('password', 'mysecret'),('db', 'mydb')]

connection_chain_list = []
for couple in myzip:
  x = "=".join(couple)
  connection_chain_list.append(x)
connection_chain = ";".join(connection_chain_list)
print(connection_chain) # will be "username=mylogon;password=mysecret;db=mydb"
```

  - `split`
    Génère une liste résultant du découpage de la chaine selon le séparateur passé en paramètre et le nombre de tronçons voulu

```
astr = "Welcome to training sessions on Python language"

# ['Welcome', 'to', 'training', 'sessions', 'on', 'Python', 'language']
print(astr.split())

# '[Welcome to training ', ' on Python language']
print(astr.split("sessions"))

# ['Welcome', 'to training sessions on Python language']
print(astr.split(" ", 1)) #  astr.split(maxsplit=1) possible with Python 3
```

- Les dictionnaires
  - Containers pour stocker des object sous la forme clé - valeur
  - Création par `{}` ou `dict()` puis spécification de couple clé `:` valeur
  - Exemple : `person = {"firstname" : "Jean", "lastname" : "Dupond", "age" : 30}`
    clé et valeur peuvent être n'importe quel objet (ex : une instance de classe)
  - Méthodes : `items()` , `keys()`, `values()` renvoient des listes (sur lesquelles on peut itérer) et représentent une copie des données
  - Il existe aussi `iteritems()`, `iterkeys()`, `itervalues()` renvoie un itérateur (ne travaille pas sur la copie) => `RuntimeError` si dictionnaire modifiée en même temps
  - En Python 3 :
    `items()`, `keys()`, `values()` ne travaillent plus sur des copies.
    `iteritems()`, `iterkeys()`, `itervalues()` n'existent plus de même que `has_key()` , il faut utiliser l'opérateur `in`
  - l'existence d'une clé se fait par l'opérateur `in`
  - `get(key[, default])` renvoie la valeur associée à un clé (default si non trouvé)
  - `clear()` pour vider le dictionnaire
  - `copy()` pour disposer d'une copie
  - Un dictionnaire notable est celui des variables d'environnement renvoyé par `os.environ`
  - L'ordre d'insertion n'est pas garantit
- Syntaxe par l'exemple

- Les dictionnaires par l'exemple

```
def playing_with_dict(self):
  adict = {"m1": "Welcome", "m2": "to the", "m3": "CC-IN2P3"}

  # type, len
  print(type(adict), len(adict))  # will be <type 'dict'>   3

  # iteration : methodes items(), keys(), values()
  for (key, value) in adict.items():
    print("key = %s value = %s" %(key, value))

  # will be ['m1', 'm3', 'm2'] (note : order is NOT insert order; return a list)
  print(adict.keys())
  # will be ['Welcome', 'CC-IN2P3', 'to the']
  print(adict.values())

  # checking existence
  if ("m0" not in adict):  # adict.has_key("m0") only with Python 2
    adict["m0"] = "I wish you"
  print(adict.get("m0", "Unknown"))

  # cleaning one entry, all entries
  del adict["m0"]  #
  adict.clear() #
  print("dict is empty " if not(adict) else "dict is not empty")

  print("$USER env. var. is set to {}".format(os.environ.get("USER")))
  return
```

- List comprehensions
  - Mécanisme puissant pour générer des listes de façon simple
  - Syntaxe : `[expression-manipulation for expression in sequence if test]`
  - Premier exemple

```
alist = [ i for i in range(0, 5) ]                # gives [0, 1, 2, 3, 4]
print(alist)

elist = [ i for i in range(0, 5) if ((i%2) == 0) ] # gives [0, 2, 4]
print(elist)
```

  - Autre exemple

```
sentence = "8 à 12 slides for a talk of 20 mns"
sentence_as_list = sentence.split(" ")

numbers = [n for n in sentence_as_list if n.isdigit()]

print(numbers)  # gives ['8', '12', '20']
```

- Exemple plus complet :

```
def playing_with_list_comprehension(self):
  # a list of values
  alist = [i for i in range(0, 5)]  # gives [0, 1, 2, 3, 4]
  print(alist)

  alist = [i for i in range(0, 5)  if ((i % 2) == 0)]  # gives [0, 2, 4]
  print(alist)

  # list of squared of odd numbers  [25, 9, 1, 1, 9, 25]
  slist = [i**2 for i in range(-5, 5+1) if ((i % 2) == 1)]
  print(slist)

  # a list of lists
  # will gives [["index-0", "value-0"], ["index-1", "value-1"], ["index-2", "value-4"]]
  # values = [["index-%d" % i, "value-%d" %(i * i)] for i in range(0, 3)]
  values = [["index-{}".format(i), "value-{}".format(i**2)] for i in range(0, 3)]
  print(values)

  # a list of dicts
  # will gives [{"id0": "value-0"}, {"id1": "value-1"}, {"id2": "value-4"}]
  values = [{"id{}".format(i): "value-{}".format(i**2)} for i in range(0, 3)]  # gives [0, 2, 4]
  print(values)

  # iterate over list, then dict
  for adict in values:
    if (type(adict) is dict):
      for key, value in adict.items():
        print("{}:{}, ".format(key, value), end="")  # will gives id0:value-0, id1:value-1, id2:value-4,

  return
```

- Les générateurs
  - fonction qui va renvoyer une liste d'objets (donc iterable) mais en créant les objects l'un après l'autre i.e. sans les stocker (une fois qu'il a été généré et envoyé l'objet est perdu)
  - Ceci se fait part l'utilisation de la fonction `yield()` qui renvoi une valeur et permet de continuer l'itération en cours
  - Exemple
    Création d'un générateur de nombres au carré, itération sur la liste des nombres générés

```
# This is a generator (because yield usage), I can iterate over result
# print just to show that values are generated one after the other
def my_square_generator():
   for i in range(1, 6):
     print("just before yield, i= {} ".format(i), end="")
     yield(i**2)

# let's iterate over my generator
for value in  my_square_generator():
  print(" just after yield, value = {} ".format(value))
```

Résultat d'exécution

```
just before yield, i= 1  just after yield, value = 1
just before yield, i= 2  just after yield, value = 4
just before yield, i= 3  just after yield, value = 9
just before yield, i= 4  just after yield, value = 16
just before yield, i= 5  just after yield, value = 25
```

- Les ensembles
  Python supporte la notion d'ensemble (au sens mathématique) et les opérations associées (union, intersection, etc.)
  Les éléments sont unique
  Construction par le constructeur `set()` , à partir d'une liste `set(alist)` ou d'un tuple, opération inverse par `list(aset)`
  Possibilité de créer un ensemble non modifiable via la fonction `frozenset()`
- Méthodes disponibles :
  - `.union(...)`
  - `.intersection(...)`
  - `.difference(...)`
  - `.symmetric_difference(...)`
    (éléments contenus dans s1 et pas dans s2 + les éléments contenus dans s2 et pas dans s1)
  - `.issubset(...)` **ou** `.issuperset(...)`
  - `.copy()`

- Syntaxe par l'exemple
  Code

```
def playing_with_sets(self):
  fruits = set(["apple", "strawberry", "orange"])

  colors = set(["orange", "red", "green"])
  colors.add("red") # will do nothing, a set contains unique element

  print("{:22} {}".format("union", fruits.union(colors)))
  print("{:22} {}".format("intersection", fruits.intersection(colors)))
  print("{:22} {}".format("difference", fruits.difference(colors)))
  print("{:22} {}".format("symmetric_difference", fruits.symmetric_difference(colors)))

  # a set from a list
  aset = set(i for i in range(6))
  # display a string, using join on list(list made using List comprehensions, iterating on set , ouf !)
  print("string from list from set from list : {} ".format(", ".join(str(e) for e in aset))) # will print 0, 1, 2, 3, 4, 5
```

Execution

```
union                  {'orange', 'apple', 'strawberry', 'green', 'red'}
intersection           {'orange'}
difference             {'strawberry', 'apple'}
symmetric_difference   {'apple', 'strawberry', 'green', 'red'}
string from list from set from list : 0, 1, 2, 3, 4, 5
```

Exercise 2 - étape 1
prévisionnel de 20 mn

- Classes `dequeue` et `Queue` du module `collections`
  - `deque` (= double-ended queue)
    ajout/retrait d'objet aux deux extrémités de la liste, de façon efficace
    la liste `[]` ou `list()` vu précédemment, n'est efficace que en mode "last-in, first-out"
    même méthode que list + `appendleft()`, `popleft()`, `extendleft()`
    possibilité de spécifier une taille maximale à la création, décalage (gauche ou droite) si ajout dans une queue pleine
    Attention, `deque` n'est pas thread safe pour toutes les opérations, mais "`appends` and `pops` from either side are thread-safe"
  - `Queue` module à utiliser en env. multi-threads
    Classes `Queue` (mode FIFO) ou `LifoQueue` ou `PriorityQueue`

    ```
    put(item[, block[, timeout]])
    get([block[, timeout]])
    ```

    On peut spécifier une taille max à la création, l'insertion sera bloquée lorsque la taille max est atteinte
    Attention : La classe `Queue` est contenu dans le module `Queue`, module renommé `queue` en python 3 !

- Classes `OrderedDict` du module `collections`
  - Comme les dict déjà vu mais avec maintient de l'ordre d'insertion
  - nouvelle méthode `popitem(last=True)` qui retourne (et supprime) une paire (clé, valeur), possible en mode LIFO | FIFO selon `last = True | False`
  - Exemple

```
from collections import OrderedDict
...
    odict =  OrderedDict()
    odict["m1"] = "Welcome"
    odict["m2"] = "to"
    odict["m3"] = "CC-IN2P3"

    for key in odict.keys():
      print(key),                  # will be m1 m2 m3 (order is insert order, not the case with built-in dict())

    print(odict.popitem(True) )    # will be('m3', 'CC-IN2P3')    mode LIFO
    print(odict.popitem(False))    # will be('m1', 'Welcome')     mode FIFO
    print(len(odict))              # will be 1 (3 - 2 pops)
```

- Le module `collections` est riche d'autres classes à découvrir
  https://docs.python.org/3.6/library/collections.html

- `map(...)`
  - Pour appliquer une fonction à chaque object d'une entité itérable (ex une liste)
  - `map` et `filter` renvoie une liste en Python 2, un iterable avec Python 3
  - syntaxe `map(function, iterable)`
  - Exemples

```
def square(value):
  return(value * value)

# build a list of square value from 0 to 4
squares = map(square, range(0, 5))  # will be [0, 1, 4, 9, 16]
print(squares)  # With Python 3 map return an iterator, please use print(list(squares))

# built a list of lengths, length of string from a list
lengths = map(len, ["name-%d" % i for i in range(9, 12)])  # will be [6, 7, 7]
print(lengths) # With Python 3 map return an iterator, please use print(list(lengths))
```

- `filter(...)`
  - Pour filtrer les objets d'une liste
    Non prise en compte des objets qui ne passent pas le filtre
  - syntaxe : `filter(function, iterable)`
  - Exemple : Liste des mots qui commencent par une lettre majuscule

```
def starts_with_capitalized_char(self, value):
  # value supposed to be a non empty string(str type)
  return((value[0]).isupper())

words = filter(self.starts_with_capitalized_char, "Python is an INTERESTING language".split())
print("Capitalized words list is {}".format(list(words))) #  Capitalized words list is ['Python', 'INTERESTING']
```

- `reduce(...)`
  - Pour appliquer une fonction à 2 arguments, et réduire le résultat à une valeur retournée par la fonction.
  - cette valeur retournée sera le 1er argument, le 2eme étant pris dans la séquence lors de l'itération suivante, etc.
  - syntaxe : `reduce(function, iterable[, initializer])` retourne une liste
  - Exemple : Calcul de la somme des N premiers nombres

```
def my_sum(v1, v2):
  return(v1+v2)

def sum_of_first_numbers():  # better way is N*(N+1)/2
  #  list of numbers from [1 to 10]
  # my_sum will apply to first and second, then on previous result(pr) and third, then pr and forth, etc.
  x = reduce(my_sum, range(1, 10))
  print(x)
  return

sum_of_first_numbers() # method invocation
45
```

  - Attention
    `reduce` est une built-in function en Python 2, mais a été déplacée dans le module `functools` en Python 3

```
from functools import reduce
```

- `lambda`
  - Pour définir une fonction anonyme
  - Syntaxe : `lambda` argument_list : expression
  - Exemple

Classiquement, avec définition d'une fonction

```
def mysqr(x):
  return (x**2)

print(mysqr(123))
```

Avec l'utilisation d'expression lambda

```
other_sqr = lambda x: x**2
print(other_sqr(123))
```

  - Autre exemple, déjà vu avec la fonction `filter`, mais ré-ecrit avec le mécanisme `lambda`

```
# lambda keyword allow definition only when usage is needed
words = filter(lambda c: c[0].isupper(), "Python is an INTERESTING language".split())
print(list(words))
```

```
['Python', 'INTERESTING']
```

Exercise 2 - étapes 2 et 3
prévisionnel de 50 mn