

Structure de données

Bernard CHAMBON

CC-IN2P3, Lyon - France

12, 13, 14 novembre 2018

- Liste, tuple
- Conversion liste | tuple ↔ chaîne
- Dictionnaire
- List comprehensions (dénomination anglaise)
- Les générateurs
- Les ensembles
- Structures de données du module `collections` : `deque`, `Queue`, `OrderedDict`
- Fonctionnalités avancées : `map(...)`, `filter(...)`, `reduce(...)`, `lambda`

■ 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 x in my_list:`

■ Premier exemple : création par `[]` , accès aux éléments par `[]` , itération

```
def playing_with_list():
    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")
```

Synthèses de quelques opérations possibles sur les listes

■ Fonctions

`len(l)` nombre d'élément de la liste

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

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

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

`del l[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. `if not my_list:` renvoie vrai si la liste est vide

opérateur `+` pour concaténer deux listes

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

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

`s.clear()` pour vider une liste (dispo avec Python 3)

`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():
    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)

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

    # pop will get (and remove) the last one (remember it's a LIFO => will be 'EndOfList')
    print(alist.pop())

    # clearing the list
    alist[:] = [] # to clear the list (reference of list will not change - check with id(list))

    # alist.clear() # same as previous with Python 3

    # providing a new list (id as changed, old list reference will be garbage collected)
    alist = []

    # Using 'not' to check empty list
    if not alist:
        print("The list is empty")
```

Les listes sont des structures LIFO (voir la classe `deque` 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

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

Il est suffisant de savoir que les tuples existent, les reconnaître 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]
my_zip = zip(l1, l2)
for x in my_zip: # will display (1, 1) (2, 4) (3, 9)
    print(x)
```

■ Conversion liste | tuple ↔ chaîne : méthodes `join` et `split`

- `str.join(iterable)`

génère une chaîne résultant de la concaténation des éléments de l'iterable passés en paramètre (ex une liste) et du contenu de la chaîne (qui peut être vu comme un séparateur)

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

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

    # will gives ['username=mylogon', 'password=mysecret', 'db=mydb']
    connection_chain_list = []
    for couple in myzip:
        x = "".join(couple)
        connection_chain_list.append(x)

    # will give "username=mylogon;password=mysecret;db=mydb"
    connection_chain = ";".join(connection_chain_list)
    return connection_chain
```

- `str.split(sep=None, maxsplit=-1)`

Génère une liste résultant du découpage de la chaîne 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))
```

■ 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, ce sont les méthodes à utiliser et d'ailleurs `iteritems()`, `iterkeys()`, `itervalues()` n'existent plus
 - . `has_key()` n'existe plus, 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 des éléments ne reflète pas l'ordre d'insertion

■ Syntaxe par l'exemple

■ Les dictionnaires par l'exemple

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

    # just to show how to address os.environ
    print("$USER env. var. is set to {}".format(os.environ.get("USER")))
```

- A propos des types `dict_keys` ou `dict_values` retournés par `items()`, `keys()`, `values()`, en Python 3
Les structures retournées sont itérables et non utilisables directement par la méthode `print()`

```
def about_dict_with_python3():
    # About .values() in Python 3

    my_dict = { "id1": [i for i in range (0, 5)] }

    # will be : [[0, 1, 2, 3, 4]]          with Python 2
    # will be : dict_values([[0, 1, 2, 3, 4]]) with Python 3
    print(my_dict.values())

    # With Python 3, .values() return a dict_values which can't be displayed via print but can be iterated
    # so let's iterate over dict_values to build a new list (list_of_values in this code)
    list_of_values = []
    for elt in my_dict.values():
        if (type(elt) is list):
            for sub_elt in elt:
                list_of_values.append(sub_elt)
    print(list_of_values)

    # Another solution
    list_of_values = []
    for elt in my_dict.values():
        if (type(elt) is list):
            list_of_values.extend(elt)

    print(list_of_values)
```

Exécution (avec Python 3)

```
dict_values([[0, 1, 2, 3, 4]])
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

■ List comprehensions

- Mécanisme puissant pour générer des collection de façon simple et selon cette syntaxe :
[expression-manipulation for expression in sequence if test] pour générer une liste
{expression-manipulation for expression in sequence if test} pour générer un dictionnaire

- 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)
```

- Deuxième exemple

```
# using function isdigit() in test
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']
```

- Troisième exemple

```
# using functions in test and in expression-manipulation
import math

def f(i):
    return(math.sin(i))

def g(i):
    return((i%2) ==0)

my_list = [ f(i) for i in range(0, 6) if g(i) ]
print(my_list) # will gives [0.0, 0.909, -0.756]
```

■ Exemples avancés

- Liste de listes, liste de dictionnaires, dictionnaires

```
# A list of lists
# will gives [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16]]
l1 = [[i, i**2] for i in range(0, 5)]
print(l1)

# A list of dict
# using format to restrict number of digits
# will gives [{0: '+0.00'}, {1: '+0.84'}, {2: '+0.91'}, {3: '+0.14'}, {4: '-0.76'}]
l1 = [{i : "+{:1.2f}".format(math.sin(i)) } for i in range(0, 5)]
print(l1)

# A dictionary
# will gives {0: '+0.00', 1: '+0.84', 2: '+0.91', 3: '+0.14', 4: '-0.76'}
l1 = { i : "{:1.2f}".format(math.sin(i)) for i in range(0, 5)}
print(l1)
```

- Avec plusieurs itérateurs

```
# A list of lists
# will gives : [[0, 1], [0, 1, 2], [0, 1, 2, 3]]
l1 = [[x1 for x1 in range(0, x2)] for x2 in [2, 3, 4]]
print(l1)

# a list of all tuples possible between [1,2,3] and [4, 5, 6]
# will gives : [(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
l2 = [(x1, x2) for x1 in [1, 2, 3] for x2 in [4, 5, 6]]
print(l2)

# a list of all tuples possible between [1,2,3] and [1,2,3] but without (1,1), (2, 2) etc
ref = [i for i in range(1, 4)] # will be [1,2,3]
# will gives : [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
l3 = [(x1, x2) for x1 in ref for x2 in ref if (x1 != x2)]
print(l3)
```

Exercice 2 - étape 1
prévisionnel de 1 h

■ Les générateurs

- fonction qui va renvoyer une liste d'objets (donc iterable) mais en créant les objets l'un après l'autre i.e. sans les stocker
- Ceci se fait par l'utilisation de la fonction `yield()` qui renvoie 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(...)` ou le symbole `|`
- `.intersection(...)` ou le symbole `&`
- `.difference(...)` ou le symbole `-`
- `.symmetric_difference(...)` ou le symbole `^`
(éléments contenus dans `s1` et pas dans `s2` plus les éléments contenus dans `s2` et pas dans `s1`)
- `.issubset(...)` OU `.issuperset(...)`

■ Syntaxe par l'exemple Code

```
def playing_with_sets():
    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)))
```

Execution

```
union           {'orange', 'apple', 'strawberry', 'green', 'red'}
intersection    {'orange'}
difference       {'strawberry', 'apple'}
symmetric_difference {'apple', 'strawberry', 'green', 'red'}
```

■ Rappel

- deque : double-ended queue
- ajout/retrait d'objet aux deux extrémités de la liste, de façon efficace
les listes vu précédemment (`[]` ou `list()`) ne fonctionne qu'en mode "Last-In, First-Out"
- même méthode que list avec en plus `appendleft()`, `popleft()`, `extendleft()`
- Dans le module `collections`

```
from collections import deque # available for Python 2 et 3
```
- 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"

■ Exemple

Code

```
from collections import deque

# appendleft + pop => mode First In First Out
my_dq = deque()
for i in range(0, 5):
    my_dq.appendleft(i)

print("Queue content is : ")
for i in my_dq:
    print(i, end=" ")
print()

print("Using pop ...")
while my_dq:
    print(my_dq.pop(), end=" ")
```

Exécution

```
Queue content is :
4 3 2 1 0
Using pop ...
0 1 2 3 4
```

■ Rappel

- Cette collection est `thread safe`, donc à utiliser pour échanger des données entre des threads d'exécution
- Ajout : `put(item[, block[, timeout]])`, Retrait : `get([block[, timeout]])`
Bloquant : `block True + timeout None`
Non bloquant : `block False + timeout None`
Bloquant + exception : `block True + timeout en secondes`
- On peut spécifier une taille `max` à la création, l'insertion sera bloquée (si `block True`) lorsque la taille `max` est atteinte
- Existe en mode FIFO `Queue(maxsize=0)`, en mode LIFO `LifoQueue(maxsize=0)`, et aussi avec tri des objects selon critère de comparaison `PriorityQueue(maxsize=0)`
- La classe `Queue` est contenu dans le module `Queue` en Python 2, module renommé `queue` en python 3

■ Exemple

Code

```
#from Queue import Queue # For Python 2.7
from queue import Queue # For Python 3

my_queue = Queue()
for i in range(0, 5):
    my_queue.put(i)

while not my_queue.empty():
    print(my_queue.get(), end=" ")
```

Exécution

```
0 1 2 3 4
```

■ Rappel

- Dictionnaire avec maintien de l'ordre d'insertion (pas le cas avec {} ou dict())
- Nouvelle méthode `popitem(last=True)` qui retourne (et supprime) une paire (clé, valeur) possiblement en mode LIFO | FIFO selon `last = True | False`
- Dans le module `collections`

```
from collections import OrderedDict # available for Python 2 et 3
```

■ Exemple

Code

```
from collections import OrderedDict

my_dict = OrderedDict()
my_dict["m1"] = "Welcome"
my_dict["m2"] = "to"
my_dict["m3"] = "our"
my_dict["m4"] = "lab"

# will be m4 m1 m3 m2 is using dict() with Python 2
# will be m1 m2 m3 m4 when using OrderedDict()
for key in my_dict.keys():
    print(key, end=" ")
print()
print(my_dict.popitem(True)) # will be('m4', 'lab') since mode LIFO
print(my_dict.popitem(False)) # will be('m1', 'Welcome') since mode FIFO
print(len(my_dict)) # will be 2 (4 - 2 pops)
```

Exécution

```
m1 m2 m3 m4
('m4', 'lab')
('m1', 'Welcome')
2
```

- Autres classes à découvrir <https://docs.python.org/3.6/library/collections.html>

- `map(function, iterable[, ...])`
 - Pour appliquer une fonction à chaque objet d'une entité itérable (ex une liste)
 - Syntaxe `map(function, iterable, ...)`
'...' exprime la possibilité de passer une liste contenant les paramètres pour la fonction
 - `map` est une built-in function qui renvoie une liste en Python 2, un iterable avec Python 3
 - Exemples

Code

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

# will be [0, 1, 4, 9, 16]
squares = map(square, range(0, 5))
for x in squares:
    print(x, end=" ")
print()

##
def my_pow(value1, value2):
    return (value1 ** value2)

# will be [0**1, 1**2, 2**3, 3**4, 4**5]
pows_list = map(my_pow, range(0, 5), [1, 2, 3, 4, 5])
for pow in pows_list:
    print(pow, end=" ")
print()
```

Exécution

```
# square function usage
0 1 4 9 16

# my_pow function usage
0 1 8 81 1024
```

■ `filter(function, iterable)`

- Pour filtrer les objets d'une entité itérable (ex une liste)
- Syntaxe : `filter(function, iterable)`
- `filter` est une built-in function qui renvoie une liste en Python 2, un iterable avec Python 3
- La fonction passée en paramètre de `filter` doit renvoyer un booléen
- Exemple : Liste des mots qui commencent par une lettre majuscule

Code

```
def starts_with_capitalized_char(value):
    return((value[0]).isupper())

msg = "Python is an INTERESTING language"
msg_as_list = msg.split();
words = filter(starts_with_capitalized_char, msg_as_list)

# will be ['Python', 'INTERESTING']
for word in words:
    print(word, end=" ")
```

Exécution

Python INTERESTING

■ `reduce(function, iterable[, initializer=None])`

- 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])`
- `reduce` est une built-in fonction en Python 2, mais a été déplacée dans le module `functools` en Python 3 : `from functools import reduce`
- Exemple : Calcul de la somme des N premiers nombres

Code

```
from functools import reduce
```

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

```
my_sum = reduce(my_add, range(1, 6))  
print(my_sum)
```

```
# other example with init value  
# it will be used once for the first iteration  
# result will be 100 + sum of 5 first numbers => 115  
my_sum = reduce(my_add, range(1, 6), 100)  
print(my_sum)
```

Exécution

15

115

■ `lambda`

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

Classiquement, avec définition d'une fonction

```
def my_sqr(x):  
    return (x**2)  
  
print(my_sqr(123))
```

Avec l'utilisation d'expression `lambda`

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

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

```
# lambda allows function definition and usage at the same time  
  
msg = "Python is an INTERESTING language"  
msg_as_list = msg.split();  
words = filter(lambda c: c[0].isupper(), msg_as_list)  
  
# will be ['Python', 'INTERESTING']  
for word in words:  
    print(word, end=" ")
```

- Contrainte majeure
Sur seule ligne et une seule instruction dans la fonction !

Exercice 2 - étapes 2 et 3
prévisionnel de 1h30