Python for Data Science

Pre-school Lecture Series
(HEPAP-DAS, EHEP2024)

November 1, 2023 - Lecture 3

Subir Sarkar, SINP
subir.sarkar@cern.ch

List

List

- A list is a sequence of values that could be of any type
- The values in a list are called elements or items
- A list is mutable

List creation:

```
L = []                          # empty list
L = list()                      # empty list
L = list(sequence)
L = list(expression for variable in sequence)

# functional programming
L1 = map(function, L)
L2 = filter(function, L)

# (List) Comprehension
L = [expression, ...] # list display
L = [expression for variable in sequence]
```

List

```
>>> fruits = ['apple', 'pear', 'peach']
>>> type(fruits)
<class 'list'>
>>> len(fruits)
3
>>> fruits[-1]
'peach'
>>> dir(fruits)
[..., 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> fruits.append('fig')
>>> fruits[1:]
['pear', 'peach', 'fig']
>>> fruits.extend(['banana', 'apricot'])
>>> fruits[::2]
['apple', 'peach', 'banana']
```

List

```
>>> sorted(fruits)
['apple', 'apricot', 'banana', 'fig',
'peach', 'pear']
>>> fruits.pop()
'apricot'
>>> fruits
['apple', 'Pear', 'peach', 'fig', 'banana']
>>> fruits.insert(2,'apricot')
>>> fruits
['apple', 'pear', 'apricot', 'pear', 'fig',
'banana']
>>> fruits.remove('fig')
>>> fruits['apple', 'pear', 'apricot',
'peach', 'banana']
```

List

```
>>> fruits.append('apple')
>>> fruits.count('apple')
2
>>> fruits.sort() # sorted in-place
>>> fruits
['apple', 'apple', 'apricot', 'banana',
'fig', 'peach', 'pear']
>>> fruits.reverse()
>>> fruits
['pear', 'peach', 'fig', 'banana',
'apricot', 'apple', 'apple']
```

Copying List

```
import copy
w = ['Perl', 'Python', 'Ruby']
c1 = w[:]
c2 = list(w)
c3 = copy.copy(w)
c4 = copy.deepcopy(w)
c5 = [e for e in w]
c6 = []
for e in w:
    c6.append(e)
c7 = []
c7.extend(w)
```

Copying List

```
print("-> Before update\nc0: ", c0,
"\nc1: ", c1, "\nc2: ", c2, "\nc3: ", c3,
"\nc4: ", c4, "\nc5: ", c5, "\nc6: ", c6,
"\nc7: ", c7)

w.extend(['Julia','R','Raku'])

print("-> After update\nc0: ", c0,
"\nc1: ", c1, "\nc2: ", c2, "\nc3: ", c3,
"\nc4: ", c4, "\nc5: ", c5, "\nc6: ", c6,
"\nc7: ", c7)
```

Tuple

Tuple

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345

# tuples may be nested
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

# tuples are immutable
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Tuple

```
# tuples can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
>>> v[0][0] = 10

# tuple unpacking
>>> x,y,z = t

# single element tuple
>>> st = 'hello',  ← note trailing comma
>>> type(st)
<class 'tuple'>
```

Dictionary

Dictionary

- A dictionary is an unordered set of [key, value] pairs
- The keys must be unique within a dictionary

```
# 1
>>> play_makers = {'Zinedine': 'Zidane', 'Andrea':
'Pirlo', 'Andres': 'Iniesta', 'Luis': 'Figo'}
>>> play_makers
{'Zinedine': 'Zidane', 'Francesco': 'Totti',
'Andres': 'Iniesta', 'Luis': 'Figo'}

# 2
>>> play_makers = {} # empty dictionary
>>> play_makers['Zinedine'] = 'Zidane'
>>> play_makers['Andrea'] = 'Pirlo'
>>> play_makers['Andres'] = 'Iniesta'
>>> play_makers['Luis'] = 'Figo'
```

Dictionary

```
# 3: from 2 lists
>>> given = ['Zinedine','Andrea','Andres',
'Luis']
>>> family =
['Zidane','Pirlo','Iniesta','Figo']
>>> play_makers.clear()
>>> for (i, n) in enumerate(given):
...     play_makers[n] = family[i]
...
# 3a: Idiomatic
>>> dict(zip(given, family))
```

Dictionary

```
# 4
>>> play_makers = dict(Zinedine='Zidane',
Andrea='Pirlo', Andres='Iniesta',
Luis='Figo')

# 5: from a single list
>>> l = ['Zinedine','Zidane','Andrea',
'Pirlo','Andres', 'Iniesta', 'Luis', 'Figo']

>>> l1 = [l[i:i+2] for i in range(0,len(l),2)]
[['Zinedine', 'Zidane'], ['Andrea', 'Pirlo'],
['Andres', 'Iniesta'], ['Luis', 'Figo']]

>>> play_makers = dict(l1)
```

Dictionary

```
>>> for name in play_makers:
...     print (name, play_makers[name])
...
Zinedine Zidane
Andrea Pirlo
Andres Iniesta
Luis Figo
>>> list(play_makers.keys())
['Zinedine', 'Andrea', 'Andres', 'Luis']
>>> list(play_makers.values())
['Zidane', 'Pirlo', 'Iniesta', 'Figo']
>>> del play_makers['Luis']
>>> play_makers.items()
[('Zinedine', 'Zidane'), ('Andrea', 'Pirlo'),
('Andres', 'Iniesta')]
```

Dictionary

```
>>> for key,value in play_makers.items():
...     print(key, value)
...
Zinedine Zidane
Andrea Pirlo
Andres Iniesta
>>> play_makers.get('Zinedine')
'Zidane'
>>> 'Luis' in play_makers
False
>>> play_makers.pop('Andres')
'Iniesta'
>>> play_makers
{'Zinedine': 'Zidane', 'Andrea': 'Pirlo'}
```

Set

Set

- A set is an unordered collection with unique elements

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket) # create a set
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits
True
>>> 'mango' in fruits
False
```

Set

```
# set operations
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b              # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b              # letters in both a and b
set(['a', 'c'])
>>> a ^ b          # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Functions, Functional Programming Support

lambda, map, filter, reduce, list comprehension

Function

- In Python, everything is an object
  - functions are first-class objects too and can be
    - assigned to variables, or stored in data structures
    - passed as argument to another function
    - used in return value of a function

Function

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print (a, end=" ")
...         a, b = b, a+b
...
>>> fib
<function fib at 0x7f4f0958e848>
>>> type(fib)
<class 'function'>

# Now call the function we just defined
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Function arguments

```
# parameter(s) with default value
>>> def power(x, y=2):
...     r = 1
...     for i in range(y):
...         r *= x
...     return r
...
>>> power(3)
9
>>> power(3,3)
27
>>> power(3,5)
243
```

Function arguments

```
# All positional arguments to Python
# functions can also be passed by
# keyword
>>> def display(creator, year, lang):
...     print("Creator: ", creator)
...     print("year: ", year)
...     print("Language: ", lang)
...

>>> display(year=1987, creator='L Wall',
lang='Perl')

>>> display(creator="G van Rossum", year=91,
lang='Python')
```

Function arguments

```
# arbitrary number of arguments (tuple)
def sum(*args):
    '''Function returns the sum
    of all values'''
    r = 0
    for i in args:
        r += i
    return r

print sum.__doc__
print sum(1, 2, 3)
print sum(1, 2, 3, 4, 5)
```

Function arguments

```
# expects a dictionary

def display(**details):
   for i in details:
     print("%s: %s"%(i, details[i]))

display(creator='Larry Wall',
year=1987,lang='Perl')
```

Functions are objects

```
def function1():
  print('You chose one.')

def function2():
  print('You chose two.')

def function3():
  print('You chose three.')
# dictionary of functions
switch = {
    'one': function1, # just refer the
    'two': function2, # function
  'three': function3
}
```

Functions are objects

```
choice = input('Enter one, two, or three :')

# call the appropriate function
try:
  result = switch[choice]
except KeyError:
  print('Failed to understand your choice.')
else:
  result() # callable
```

http://www.voidspace.org.uk/python/articles/OOP.shtml#so-what-is-an-object

lambda

- Python *lambda* supports creation of anonymous functions at runtime

```
# standard function call
def f(x):
   return x*2
print f(3) -> 6

# unnamed function assigned to a variable
g = lambda x: x*2
print(g(3)) -> 6

# all on-the-fly
print((lambda x: x*2)(3))
```

lambda

```
# two variables
>>> print((lambda x, y: x*y)(3, 4))

# return a function
>>> def increment(n):
...     return lambda x: x + n
...
>>> increment(2)
>>> increment(2)(20)
```

Operation on sequence

```
>>> lang = ['Tcl', 'Perl', 'Python',
'Ruby', 'R', 'Julia']

# store string length in another list
>>> lst = []

>>> for e in lang:
...     lst.append(len(e))
...

>>> lst
[3, 4, 6, 4, 1, 5]
```

map (function , sequence , [sequence...]) → list

```
>>> list(map(len, lang)) # built-in
function
[3, 4, 6, 4, 1, 5]

>>> def to_upper(s): # user function
...     return s.upper()
...
>>> list(map(to_upper, lang))
['TCL', 'PERL', 'PYTHON', 'RUBY', 'R',
'JULIA']
```

map(function , sequence , [sequence...]) → list

```
# combine lambda() & map()
>>> list(map(lambda s: s.upper(), lang))
['TCL', 'PERL', 'PYTHON', 'RUBY', 'R',
'JULIA']

# operation on > 1 lists
>>> a = [ 1,  2,  3,  4]
>>> b = [11, 12, 13, 14]

>>> list(map(lambda x,y: x+y, a,b))
[12, 14, 16, 18]


>>> list(filter(lambda s:
s.startswith('P'), lang))
['Perl', 'Python']
```

filter(function or None, sequence) -> list, tuple, or string

```
>>> foo = list(range(20))
>>> filter(lambda x: x % 3 == 0, foo)
[0, 3, 6, 9, 12, 15, 18]

>>> map(lambda x: 2*x, \
    filter(lambda x: x%3 == 0, foo))
[0, 6, 12, 18, 24, 30, 36]
```

reduce(function, sequence[, initial]) -> value

```
>>> foo = list(range(20))
>>> reduce(lambda x, y: x + y, foo)
190
>>> reduce(lambda x,y: x+y,
    map(lambda x:2*x,
    filter(lambda x: x%3 == 0, foo)))
126

>>> def comp(a, b):
...   return a if (a > b) else b
...
>>> comp(2,4) # 4
>>> li = [1001, 9, 301, 450, 25]
>>> reduce(comp, li) # 1001
>>> reduce(lambda a,b: a if (a > b) else b, li)
1001
```

The operator module

- These functions are often useful in functional-style programming because they save you from writing trivial functions that perform a single operation

- `add(), sub(), mul(), div(), floordiv(), abs(), ...`
- `not_(), truth()`
- `and_(), or_(), invert()`
- `eq(), ne(), lt(), le(), gt(), and ge()`
- `is_(), is_not()`

```
>>> map(lambda x,y: x+y, foo, bar) and
>>> import operator
>>> map(operator.add, foo, bar)
```

are equivalent

List Comprehension

```
>>> lang = ['Tcl', 'Perl', \
'Python', 'Ruby', 'R','Julia']

>>> [len(l) for l in lang]
[3, 4, 6, 4, 1, 5]

>>> [l.upper() for l in lang]
['TCL', 'PERL', 'PYTHON', 'RUBY',
'R', 'JULIA']
```

List Comprehension

```
import pprint
sentence = '''
Truth can be stated in a thousand different
ways, yet each one can be true.
'''
words = sentence.split()

print ">>> Use map ..."
newlist = map(lambda w: [w,w.upper(),len(w)], words)
pprint.pprint(newlist, depth=2)

print ">>> Use List Comprehension ..."
newlist_2 = [[w,w.upper(),len(w)] for w in words]
pprint.pprint(newlist_2, depth=2)
```

List Comprehension

```
# nested LC
noprimes = [j for i in range(2, 8)
            for j in range(i*2, 50, i)]
print(noprimes)

primes = [x for x in range(2, 50)
          if x not in noprimes]
print(primes)
```

Dictionary Comprehension

```
server = {'OS': 'CentOS 7',
          'IP': '10.10.0.244',
    'hostname': 'ccs2',
    'location': 'Room 237',
       'model': '4451',
         'OEM': 'Fujitsu'
}
print(server)
server_lo = {key.lower(): value for
key, value in server.items()}
print(server_lo)
```

Set Comprehension

```
>>> lst = [1,2,3,4,4,5,6,6,6,7,7,8,8]
>>> sa = {v for v in lst if v % 2 == 0}
>>> sa
{2, 4, 6, 8}

>>> a = {x for x in 'abracadabra' if x
not in 'abc'}
>>> a
{'r', 'd'}
```