PyLearn-PyTutorial-5.DataStructures

April 21, 2020

#

Learning Python

1 The Python Tutorial -> Data Structures¶

Link: https://docs.python.org/3/tutorial/datastructures.html#data-structures

1.1 More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

list.**append**(x) Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

list.**extend**(iterable) Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

list.**insert**(i, x) Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

list.**remove**(x) Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.

list.**pop**([i]) Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.**clear**() Remove all items from the list. Equivalent to `del a[:]`.

list.**index**(x[, start[, end]]) Return zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item.

The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

list.**count**(x) Return the number of times x appears in the list.

list.**sort**(key=None, reverse=False) Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.**reverse**() Reverse the elements of the list in place.

list.**copy**() Return a shallow copy of the list. Equivalent to `a[:]`.

```
[10]: fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
      print( fruits.count('apple') )

      print( fruits.count('tangerine') )

      print( fruits.index('banana') )

      print( fruits.index('banana', 4) )  # Find next banana starting a position 4

      fruits.reverse()
      print( fruits )

      fruits.append('grape')
      print( fruits )

      fruits.sort()
      print( fruits )

      print( fruits.pop() )
```

```
2
0
3
6
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
pear
```

You might have noticed that methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. This is a design principle for all mutable data structures in Python.

1.1.1 Using Lists as Stacks

```
[12]: stack = [3, 4, 5]
      stack.append(6)
      stack.append(7)
      print(stack)

      stack.pop()

      print(stack)
      print( stack.pop() )

      print( stack.pop() )

      print(stack)
```

```
[3, 4, 5, 6, 7]
[3, 4, 5, 6]
6
5
[3, 4]
```

1.1.2 Using Lists as Queues

lists are not efficient for this purpose.
While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow
To implement a queue, use collections.deque (https://docs.python.org/3/library/collections.html#collections.deque
which was designed to have fast appends and pops from both ends. Example:

[15]:
```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")
queue.append("Graham")

print( queue.popleft() )

print( queue.popleft() )

print( queue )
```

```
Eric
John
deque(['Michael', 'Terry', 'Graham'])
```

1.1.3 List Comprehensions

[60]:
```
squares = []
for x in range(10):
    squares.append(x**2)

print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Side effect : This creates (or overwrites) a variable named x that still exists after the loop completes.

[66]:
```
squares = list(map(lambda x: x**2, range(10)))
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

No side effect like example above

[22]:
```
squares = [x**2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

more concise and readable.

A list comprehension consists of
- brackets containing an expression followed by a `for` clause, - then zero or more `for` or `if` clauses.

The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it.

For example, this list comprehension combines the elements of two lists or two tuples if they are not equal:

[29]:
```
list = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
print(list)

list = [(x, y) for x in (1,2,3) for y in (3,1,4) if x != y]
print(list)
```

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

note: can be list or tuple

it's equivalent to:

[43]:
```
list = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            list.append((x, y))

print(list)
```

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

More examples:

[33]:
```
vec = [-4, -2, 0, 2, 4]

# create a new list with the values doubled
print( [x*2 for x in vec] )
```

```
# filter the list to exclude negative numbers
print( [x for x in vec if x >= 0] )

# apply a function to all the elements
print( [abs(x) for x in vec] )
```

```
[-8, -4, 0, 4, 8]
[0, 2, 4]
[4, 2, 0, 2, 4]
```

[34]:
```
# call a method on each element
freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
print( [weapon.strip() for weapon in freshfruit] )
```

```
['banana', 'loganberry', 'passion fruit']
```

[69]:
```
# create a list of 2-tuples like (number, square)
print( [(x, x**2) for x in range(6)] )
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

[36]:
```
# ERROR: the tuple must be parenthesized, otherwise an error is raised
print( [x, x**2 for x in range(6)] )
```

```
      File "<ipython-input-36-84a7ab7ffe80>", line 2
    print( [x, x**2 for x in range(6)] )
                    ^
SyntaxError: invalid syntax
```

[81]:
```
# flatten a list using a listcomp with two 'for'
vec = [[1,2,3], [4,5,6], [7,8,9]]
print( [num for elem in vec for num in elem] )
```

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

`[num  for num in elem for elem in vec]` would be wrong, we need to read from **left to right**!

[55]:
```
# list comprehensions can contain complex expressions and nested functions:
from math import pi
print( [str(round(pi, i)) for i in range(1, 6)] )
```

```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

1.1.4 Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

[82]:
```
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]

matrix_transposed = [[row[i] for row in matrix] for i in range(4)]
print(matrix_transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the for that follows it, so this example is equivalent to:

[87]:
```
matrix_transposed = []
for i in range(4):
    matrix_transposed.append([row[i] for row in matrix])
print(matrix_transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

[85]:
```
matrix_transposed = []
for i in range(4):
    # the following 3 lines implement the nested listcomp
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    matrix_transposed.append(transposed_row)
print(matrix_transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The zip() function (https://docs.python.org/3/library/functions.html#zip) would do a great job for this use case:

[114]:
```
print(*matrix)
print(zip(*matrix))
print(list(zip(*matrix)))

matrix_transposed = list(zip(*matrix))
print(matrix_transposed)
```

```
[1, 2, 3, 4] [5, 6, 7, 8] [9, 10, 11, 12]
<zip object at 0x7f0be420e440>
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See Unpacking Argument Lists (https://docs.python.org/3/tutorial/controlflow.html#tut-unpacking-arguments) for details on the asterisk in this line.

1.2 The del statement

[119]:
```python
a = [-1, 1, 66.25, 333, 333, 1234.5]
print(a)

del a[0]
print(a)

del a[2:4]
print(a)

del a[:]
print(a)

del a
# print(a)  # This will produce ERROR as 'a' no longer exists
```

```
[-1, 1, 66.25, 333, 333, 1234.5]
[1, 66.25, 333, 333, 1234.5]
[1, 66.25, 1234.5]
[]
```

1.3 Tuples and Sequences

[129]:
```python
t = 12345, 54321, 'hello!'
print(t[0])
print(t)

# Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
print(u)
```

```
12345
(12345, 54321, 'hello!')
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples are immutable:

[135]:
```
# ERROR: Tuples are immutable:**
t = 1, 2, 3

t[0] = 4
```

```
       ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-135-f8903324229a> in <module>
         2 t = 1, 2, 3
         3
   ----> 4 t[0] = 4


       TypeError: 'tuple' object does not support item assignment
```

[133]:
```
# but they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
print(v)
```

```
([1, 2, 3], [3, 2, 1])
```

- on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses,

- Though tuples may seem similar to lists, they are often used in different situations and for different purposes.
  - **Tuples** are **immutable**, and usually contain a **heterogeneous** sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples).
  - **Lists** are **mutable**, and their elements are usually **homogeneous** and are accessed by iterating over the list.

- Empty tuples are constructed by an empty pair of parentheses;
- a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective.

[136]:
```
empty = ()
print(empty)
print(len(empty))

singleton = 'hello',    # <-- note trailing comma
print(singleton)
print(len(singleton))
```

```
()
0
('hello',)
1
```

[152]:
```
# tuple packing
myTuple = 123, 'hello', 456
print(myTuple)
myList = [123, 'hello', 456]
print(myList)

# sequence (i.e. tuple) unpacking
a1, b1, c1  = myTuple
print(a1, b1, c1)

# sequence (i.e. list) unpacking
a2, b2, c2  = myList
print(a2, b2, c2)

# multiple assignments is really just a combination of tuple packing and␣
 ↪sequence unpacking
a3, b3, c3 = 6, 7, 8
print(a3, b3, c3)
```

```
(123, 'hello', 456)
[123, 'hello', 456]
123 hello 456
123 hello 456
6 7 8
```

1.4 Sets

A set is an **unordered collection** with **no duplicate** elements.

[153]:
```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

print(basket)    # show that duplicates have been removed

print('orange' in basket)         # fast membership testing
print('crabgrass' in basket)      # fast membership testing

# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
print(a)                # unique letters in a
print(b)                # unique letters in a
print(a - b)            # letters in a but not in b
print(a | b)            # letters in a or b or both
print(a & b)            # letters in both a and b
print(a ^ b)            # letters in a or b but not both (exclusive or)
```

```
{'orange', 'banana', 'pear', 'apple'}
True
False
{'d', 'b', 'c', 'r', 'a'}
{'m', 'c', 'z', 'l', 'a'}
{'r', 'b', 'd'}
{'m', 'd', 'c', 'a', 'z', 'l', 'r', 'b'}
{'c', 'a'}
{'r', 'm', 'd', 'b', 'z', 'l'}
```

[181]:
```
# Create set using curly brackets
basket1 = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket1)

# Create set from a tuple
basket2 = set(('apple', 'orange', 'apple', 'pear', 'orange', 'banana'))
print(basket2)

# Create set from a list
basket3 = set(['apple', 'orange', 'apple', 'pear', 'orange', 'banana'])
print(basket3)

# Create set with ONE element using curly brackets
basket4 = {'coconut'}
print(basket4)

# Using curly brackets with ZERO element, it is an empty dictionary, NOT set
this_is_dict = {}
print(this_is_dict)

# Use set() with ZERO element to creaty an empty set
this_is_set = set()   # or set(()) or set([])
print(this_is_set)
```

```
{'orange', 'banana', 'pear', 'apple'}
{'orange', 'banana', 'pear', 'apple'}
{'orange', 'banana', 'pear', 'apple'}
{'coconut'}
{}
set()
```

Set comprehensions

```
[155]: a = {x for x in 'abracadabra' if x not in 'abc'}
       print (a)
```

```
{'r', 'd'}
```

1.5 Dictionaries

Dictionaries are **indexed by keys**, which can be **any immutable type** such as: - strings - numbers - tuples that contain only strings, numbers, or tuples

Can NOT be keys:
- tuple that contains mutable object either directly or indirectly - lists, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend()

Best to think of a dictionary as **a set of key: value pairs**, with the requirement that the keys are unique (within one dictionary).

```
[184]: tel = {'jack': 4098, 'sape': 4139}  # adds initial key:value pairs to the␣
        ↪dictionary

       tel['guido'] = 4127                 # adds extra entry
       print(tel)
       print(tel['jack'])

       del tel['sape']
       tel['irv'] = 4127
       print(tel)

       print('guido' in tel)
       print('jack' not in tel)

       # returns a list of all the KEYS used in the dictionary,
       print(list(tel))
       print(sorted(tel))
```

```
{'jack': 4098, 'sape': 4139, 'guido': 4127}
4098
{'jack': 4098, 'guido': 4127, 'irv': 4127}
True
False
['jack', 'guido', 'irv']
['guido', 'irv', 'jack']
```

```
[182]: # Create an initial dictionary as an empty dictionary
       this_is_dict = {}
       print(this_is_dict)
```

```
{}
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

[258]:
```
dict1 =     {'one': 1, 'two': 2, 'three': 3}
dict2 = dict( one = 1, two = 2, three = 3   )  # keys are simple strings,␣
↪easier to specify using keyword arguments
dict3 = dict( zip(['one', 'two', 'three'], [1, 2, 3]) )
dict4 = dict( [('two', 2), ('one', 1), ('three', 3)]  )
dict5 = dict( {'three': 3, 'one': 1, 'two': 2}        )
print(dict1)
print(dict2)
print(dict3)
print(dict4)
print(dict5)
print( dict1 == dict2 == dict3 == dict4 == dict5 )
```

```
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3}
{'two': 2, 'one': 1, 'three': 3}
{'three': 3, 'one': 1, 'two': 2}
True
```

[196]:
```
dict6 = {x: x**2 for x in (2, 4, 6)}
print(dict6)
```

```
{2: 4, 4: 16, 6: 36}
```

1.6 Looping Techniques

The key and corresponding value can be retrieved at the same time using the `items()` method

[203]:
```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

```
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

[241]:
```
for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)
```

```
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

[242]:
```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

[243]:
```
for i in reversed(range(1, 10, 2)):
    print(i)
```

```
9
7
5
3
1
```

To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

[244]:
```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
    print(f)
```

```
apple
banana
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

[248]:
```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
    if not math.isnan(value):
        filtered_data.append(value)

print(filtered_data)
```

```
[56.2, 51.7, 55.3, 52.5, 47.8]
```

1.7 More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence.

The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists.

All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, $a < b == c$ tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable.

[253]:
```
string1, string2, string3 = '', 'apple', 'orange'
non_null = string1 or string2 or string3
print(non_null)
```

```
apple
```

1.8 Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type.

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods

```
[257]: print( (1, 2, 3)              < (1, 2, 4)              )
       print( [1, 2, 3]              < [1, 2, 4]              )
       print( 'ABC' < 'C' < 'Pascal' < 'Python'               )
       print( (1, 2, 3, 4)           < (1, 2, 4)              )
       print( (1, 2)                 < (1, 2, -1)             )
       print( (1, 2, 3)             == (1.0, 2.0, 3.0)        )
       print( (1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4) )
```

```
True
True
True
True
True
True
True
```

2 END OF The Python Tutorial -> Data Structures