L2-Python-Data-Structures

October 16, 2015

1 Principal built-in types in Python

- numerics: int, float, long, complex
- sequences: str, unicode, **list**, **tuple**, bytearray, buffer, xrange
- mappings: **dict**
- files:
- classes:
- instances:
- exceptions:

2 Lists

The list data type has some more methods. Here are all of the methods of list objects in **Python 3**:

list.append(x)

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

```
In [1]: a = [1, 2, 4, 5,8, 100 ,1005]

In [2]: a.append(2)
        a

Out[2]: [1, 2, 4, 5, 8, 100, 1005, 2]
```

list.extend(L)

Extend the list by appending all the items in the given list. Equivalent to a[len(a):] = L.

```
In [3]: b = ["Feng", "Li", "Love"]
        a.extend(b)
        a

Out[3]: [1, 2, 4, 5, 8, 100, 1005, 2, 'Feng', 'Li', 'Love']
```

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).

```
In [4]: a.insert(1,"Hello")
        a

Out[4]: [1, 'Hello', 2, 4, 5, 8, 100, 1005, 2, 'Feng', 'Li', 'Love']
```

list.remove(x)
Remove the first item from the list whose value is x. It is an error 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[:].

```
In [5]: a.pop(2)
        a

Out[5]: [1, 'Hello', 4, 5, 8, 100, 1005, 2, 'Feng', 'Li', 'Love']

In [6]: a.pop()
        a

Out[6]: [1, 'Hello', 4, 5, 8, 100, 1005, 2, 'Feng', 'Li']

In [7]: a.remove("Hello")
        a

Out[7]: [1, 4, 5, 8, 100, 1005, 2, 'Feng', 'Li']
```

list.index(x)
Return the index in the list of the first item whose value is x. It is an error if there is no such item.
list.count(x)
Return the number of times x appears in the list.
list.sort()
Sort the items of the list in place.

```
In [8]: a.append("Love")

In [9]: a.index("Love")

Out[9]: 9

In [10]: a.count("Love")

Out[10]: 1

In [11]: b = [1,7,2,14]
         b.sort()
         b

Out[11]: [1, 2, 7, 14]
```

list.reverse()
Reverse the elements of the list in place.
list.copy()
Return a shallow copy of the list. Equivalent to a[:].

```
In [12]: a.reverse()
         a

Out[12]: ['Love', 'Li', 'Feng', 2, 1005, 100, 8, 5, 4, 1]
```

```
In [13]: b = a.copy()
         b

Out[13]: ['Love', 'Li', 'Feng', 2, 1005, 100, 8, 5, 4, 1]
```

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list

```
In [14]: a

Out[14]: ['Love', 'Li', 'Feng', 2, 1005, 100, 8, 5, 4, 1]

In [15]: del a[2]
         a

Out[15]: ['Love', 'Li', 2, 1005, 100, 8, 5, 4, 1]
```

`del` can also be used to delete entire variables

```
In [16]: del a
         # a will not be there
```

2.1 Variables and pass-by-reference

When assigning a variable (or name) in Python, you are creating a **reference** to the object on the right hand side of the equals sign.

```
In [17]: a = [1,2,3,4]
         b = a
         a.append(100)
         print(a)
         print(b)

[1, 2, 3, 4, 100]
[1, 2, 3, 4, 100]
```

Understanding the semantics of references in Python and when, how, and why data is copied is especially critical when working with larger data sets in Python. If you really need to make hard copy, use method like `list.copy()`

```
In [18]: c1 = a.copy()
         c2 = a[:]
         a.append(23232)
         print(a)
         print(c1)
         print(c2)

[1, 2, 3, 4, 100, 23232]
[1, 2, 3, 4, 100]
[1, 2, 3, 4, 100]
```

2.2 Using Lists as Stacks

The list methods make it very easy to use a list as a **stack**, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index

```
In [19]: stack = [3, 4, 5]

In [20]: stack.append(2)
         stack

Out[20]: [3, 4, 5, 2]

In [21]: stack.pop()
         stack

Out[21]: [3, 4, 5]
```

2.3 Using Lists as Queues

It is also possible to use a list as a **queue**, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose.

To implement a queue, use *collections.deque* which was designed to have fast appends and pops from both ends. For example:

```
In [22]: from collections import deque
         queue = deque(["Eric", "John", "Michael"])
         queue.append("Terry")           # Terry arrives
         queue.append("Graham")          # Graham arrives
         queue

Out[22]: deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])

In [23]: queue.popleft()                 # The first to arrive now leaves
         queue.popleft()                 # The second to arrive now leaves
         queue

Out[23]: deque(['Michael', 'Terry', 'Graham'])
```

2.4 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
In [24]: squares = [] # the usual way
         for x in range(10):
             squares.append(x**2)
         squares

Out[24]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can calculate the list of squares without any side effects using the following. This is very similar to R programming language's apply type functions.

```
In [25]: squares = [x**2 for x in range(10)]
         squares
```

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

which eventually did this

```
In [26]: squares = list(map(lambda x: x**2, range(10)))
         squares

Out[26]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

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.

```
In [27]: [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

Out[27]: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

which is equivalent to:

```
In [28]: combs = []
         for x in [1,2,3]:
             for y in [3,1,4]:
                 if x != y:
                     combs.append((x, y))

         combs

Out[28]: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

2.5 Nested List Comprehensions

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

```
In [29]: matrix = [[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]]
         matrix

Out[29]: [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

The following list comprehension will transpose rows and columns:

```
In [30]: [[row[i] for row in matrix] for i in range(4)]

Out[30]: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

the magic behind this was

```
In [31]: transposed = []
         for i in range(4):
             transposed.append([row[i] for row in matrix])

         transposed

Out[31]: [[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 would do a great job for this use case:

```
In [32]: list(zip(*matrix))

Out[32]: [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

3 Tuples

A tuple consists of a number of values separated by commas

```
In [33]: t = 12345, 54321, 'hello!'
         t[0]

Out[33]: 12345

In [34]: t

Out[34]: (12345, 54321, 'hello!')
```

Tuples may be nested and tuples are immutable, but they can contain mutable objects.

```
In [35]: u = t, (1, 2, 3, 4, 5)
         u

Out[35]: ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

In [36]: v = ([1, 2, 3], [3, 2, 1])
         v

Out[36]: ([1, 2, 3], [3, 2, 1])
```

Note On output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

4 Dictionaries

Unlike sequences, which are indexed by a range of numbers, **dictionaries** are indexed by *keys*. Dictionaries are sometimes found in other languages as `associative memories'' or`associative arrays".

The keys can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys.

It is best to think of a dictionary as an unordered set of `key:value` pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of `key:value` pairs within the braces adds initial `key:value` pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key.

```
In [37]: tel = {'jack': 4098, 'sape': 4139}
         tel['guido'] = 4127
         print(tel)

{'jack': 4098, 'sape': 4139, 'guido': 4127}
```

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

```
In [38]: dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

Out[38]: {'guido': 4127, 'jack': 4098, 'sape': 4139}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
In [39]: dict(sape=4139, guido=4127, jack=4098)

Out[39]: {'guido': 4127, 'jack': 4098, 'sape': 4139}

In [40]: list(tel.keys())

Out[40]: ['jack', 'sape', 'guido']

In [41]: sorted(tel.keys())

Out[41]: ['guido', 'jack', 'sape']

In [42]: 'guido' in tel

Out[42]: True

In [43]: 'jack' not in tel

Out[43]: False
```

It is also possible to delete a `key:value` pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

```
In [44]: del tel['sape']
```

`dict` comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
In [45]: {x: x**2 for x in (2, 4, 6)}

Out[45]: {2: 4, 4: 16, 6: 36}
```

5 Sets

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces {} or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not {}; the latter creates an empty dictionary.

```
In [46]: basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
         print(basket)                      # show that duplicates have been removed

{'apple', 'pear', 'orange', 'banana'}

In [47]: 'orange' in basket

Out[47]: True

In [48]: a = set('abracadabra')
         b = set('alacazam')
         print(a)
         print(b)

{'c', 'b', 'a', 'd', 'r'}
{'c', 'm', 'l', 'a', 'z'}

In [49]: a - b

Out[49]: {'b', 'd', 'r'}
```

```
In [50]: a | b

Out[50]: {'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}

In [51]: a & b

Out[51]: {'a', 'c'}

In [52]: a ^ b # XOR: exclusive OR

Out[52]: {'b', 'd', 'l', 'm', 'r', 'z'}

In [53]: a = {x for x in 'abracadabra' if x not in 'abc'} # set comprehensions

In [54]: a

Out[54]: {'d', 'r'}
```

6 Looping

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
In [55]: knights = {'gallahad': 'the pure', 'robin': 'the brave'}
         for k, v in knights.items():
             print(k, v)

robin the brave
gallahad the pure
```

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

```
In [56]: 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.

```
In [57]: 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.

```
In [58]: 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.

```
In [59]: basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
         for f in sorted(set(basket)):
             print(f)
```

```
apple
banana
orange
pear
```

To change a sequence you are iterating over while inside the loop (for example to duplicate certain items), it is recommended that you first make a copy. Looping over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
In [60]: words = ['cat', 'window', 'defenestrate']
         for w in words[:]:  # Loop over a slice copy of the entire list.
             if len(w) > 6:
                 words.insert(0, w)

         words
```

```
Out[60]: ['defenestrate', 'cat', 'window', 'defenestrate']
```

```
In [ ]: words2 = ['cat', 'window', 'defenestrate']
        for w in words2:  # No copies.
            if len(w) > 6:
                words2.insert(0, w)

        words2
```

7 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.

```
Keyword (Scalar) | Function      | Bitwise  | True if . . .
-----------------------------------------------------------------
and              | logical_and() |    &     | Both True
or               | logical_or()  |    |     | Either or Both True
not              | logical_not() |    ~     | Not True
                 | logical_xor() |    ^     | One True and One False
```

7.1 Logical Operators

The core logical operators are

- **Greater than**: >, greater()
- **Greater than or equal to**: >=, greater_equal()
- **Less than**: <, less()
- **Less than or equal to**: <= less_equal()
- **Equal to**: == equal()
- **Not equal to**: != not_equal

All comparisons are done element-by-element and return either True or False. Note that in Python, unlike C, assignment cannot occur inside expressions. it avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

7.2 Multiple tests

The functions **all** and **any** take logical input and are self-descriptive. all returns True if all logical elements in an array are 1. If all is called without any additional arguments on an array, it returns True if all elements of the array are logical true and 0 otherwise. any returns logical(True) if any element of an array is True .

7.3 is*

A number of special purpose logical tests are provided to determine if an array has special characteristics. Some operate element-by-element and produce an array of the same dimension as the input while other produce only scalars. These functions all begin with is .

```
isnan
isinf
isfinite
isposfin , isnegfin
isreal
iscomplex
isreal
is_string_like
is_numlike
isscalar
isvector
```