

Python Course: Lecture 4

January 10, 2006

1 Sequences

1.1 Lists

- Computers are good at memorizing things, so we want to give them things to memorize. We've already seen variables that can store individual things (e.g. numbers, strings), but we'd also like to store *collections* of things. The simplest collection is a list.
- Basic list syntax: `[1, 2, 3]`.
- Lists are heterogeneous: `[1, "a", None, 15.3, "dog", [4,5]]`.
- Lists are other Python objects. They can be assigned to variables.
- Operators are defined for lists

```
>>> ['a', 'b'] + ['c', 'd']
['a', 'b', 'c', 'd']
>>> ['a'] * 3
['a', 'a', 'a']
```

- Lists embody two concepts: the idea of *collection* and the idea of *sequence*.
- Collections have a certain size: `len([1,2,3])`
- An item can either be in or not in a collection.

```
>>> 1 in [1,2,3]
True
False
>>> 'a' in [None, 'b']
```

- Sequence means lists have a first element, a next element, and so on (note: the sequence is zero-based). Refer to them by *index*.

```
>>> [1,2,3][0]
1
>>> ['a', 'b'][1]
'b'
```

- Sequences have a finite length: `len([1,2,3]) == 3`
- Note that the last element of the list has index `length - 1`. (There is a whole class of errors in computer programming called “off-by-one”. The way to work around this confusion in Python is to do simple examples in the interactive mode.)
- Slices
- Negative indices
- `append`, `extend`
- `del`, `insert`, `pop`, `find`
- lists are mutable

```
>>> a = ['a', 'b', 'c']
>>> b = a
>>> a
['a', 'b', 'c']
>>> b
['a', 'b', 'c']
>>> a[0] = 'z'
>>> a
['z', 'b', 'c']
>>> b
['z', 'b', 'c']
```

1.2 Tuple

- Tuples are immutable lists.
- Tuples look like lists except they use `()` instead of `[]`.
- They have all the same properties, except you can't make assignment to tuple elements.
- Single element tuples require a comma, e.g. `(1,)`.
- When you return multiple values from a function, they come back as tuples.

```

>>> def squareit(n):
...     return n,n**2
...
>>> squareit(4)
(4, 16)
>>> a,b = squareit(5)
>>> a
5
>>> b
25

```

- Use tuples to do string interpolation with multiple elements.

```

>>> "Name: %s, Age: %d" % ('John Doe', 40)
'Name: John Doe, Age: 40'

```

- Generally you use tuples when you know how many data elements you're dealing with; lists when it's more open-ended.

1.3 Sorting

- Lists can be sorted.

```

>>> l = [2, 7, 1, 5, 3]
>>> l.sort()
>>> l
[1, 2, 3, 5, 7]

```

- Note that the sorting happens *in place*—e.g. you do not say `sorted = l.sort()`. This is a purposeful design decision.
- Any object that defines comparison operators can be sorted, though not all objects do.

```

>>> l = ['z', 'm', 'a', 'k']
>>> l.sort()
>>> l
['a', 'k', 'm', 'z']
>>> l = [2+3j, 1+4j]
>>> l.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot compare complex numbers using <, <=, >, >=

```

- The general purpose comparison function is `cmp()`.

```

>>> cmp(1, 100) # first < second returns -1
-1
>>> cmp(100, 1) # first > second returns 1
1
>>> cmp(1, 1) # first == second returns 0
0

```

- You can customize sort order by specifying an optional sort function argument that implements its own `cmp()` functionality.
- For example, say we have a list of (first name, last name) tuples. We can sort them either way.

```

>>> people = [('Xavier', 'Abburri'), ('Amy', 'Zeller')]
>>> def byfirst(a,b):
...     return cmp(a[0], b[0])
...
>>> def bylast(a,b):
...     return cmp(a[1], b[1])
...
>>> people.sort(byfirst)
>>> people
[('Amy', 'Zeller'), ('Xavier', 'Abburri')]
>>> people.sort(bylast)
>>> people
[('Xavier', 'Abburri'), ('Amy', 'Zeller')]

```

How would you sort by last name in reverse order?

```

>>> def bylast(a,b):
...     return cmp(b[1], a[1])

```

- The above example also demonstrates that functions are objects in Python like anything else.

```

>>> byfirst
<function byfirst at 0x4b630>
>>> x = byfirst
>>> people.sort(x)
>>> people
[('Amy', 'Zeller'), ('Xavier', 'Abburri')]

```

1.4 An Object-Oriented Aside: Anonymous Functions

- Sometimes we have short functions like sort orders that only get called from one place in the code. Instead of writing out a full-fledged function with a name, we can write an *anonymous* function using the `lambda` keyword.
- Here's the same sort example.

```
>>> people.sort(lambda a,b: cmp(a[0], b[0])) # Sort by first name
```

- The combination of anonymous functions and functions-as-objects allows us to package up pieces of behavior inside Python objects that can be passed around a program. This is a strength of object-oriented programming.

```
>>> adder = (lambda a: a+1)
>>> adder
<function <lambda> at 0x4ff70>
>>> adder(4)
5
```

2 Sets

- Sets embody the idea of *collection* without the notion of *sequence*.
- You get at sets via a library.

```
>>> import sets
>>> s = sets.Set()
>>> s
```

- the `in` operation is defined, but indexing is not.
- These are just like sets in mathematics, so they have intersection and union defined.
- Can be initialized with a list
- Why are lists built into the syntax of Python directly but sets implemented as a library? It's easier to handle lists in lower-level computer languages, even though mathematically sets are a more fundamental concept. This is a wart from Python's evolution—earlier versions of the language did not have sets at all.
- The set object created above is the first example we've seen of an instance of a Python *class*. More on classes later.

3 More List Manipulation

- An equivalent command is `map`.

```
>>> map(lambda i:i**2, [1,2,3,4])
[1,4,9,16]
```

- `filter` returns all items in a list that evaluate to `True`.

```
>>> filter(lambda x: x>= 0, [-1, 5, -2, 10])
[5, 10]
>>> filter(None, [0, 5, "a", "", None, True, False])
[5, 'a', True]
```

- `sum` adds together all the items in a list.

```
>>> sum([1,2,3])
6
```