Python Programming for Linguists

Week 9

Shu-Kai Hsieh

LOPE lab at Graduate Institute of Linguistics,
National Taiwan University

- **[Lecture]**: Data Structure (2): List, Dictionary and Tuple
- **[Praxis]**: In-class exercises with NLTK
- **[Final Projects and Homework]**: see course webpage notes
- **[Reading Assignment]**: HTTLCS chapter 10-12; NLTK chapter 4

Review of Data Structure

- Expressions create and process objects (Something a variable can refer to. Also known as data structures).
- Data structures are basically just that - they are structures which can hold some data together. In other words, they are used to store a collection of related data.
- There are three main built-in data structures in Python - **list, tuple** and **dictionary**. We will see how to use each of them and how they make life easier.

Overview: List

- A list is an ordered collection of arbitrary objects. (similar to `array` in Java or C).
- delimited by square brackets [ ] and the individual list elements are separated by commas.
- A list can contain any type of objects, like numbers, strings, and other lists, and is mutable (can change the elements, unlike strings.)

Example

```
>> scores = [83, 97, 90, 86]
>> ablout_compounds = ["chitchat", "shillshally", "snipsnap", "c
>> stuff = ['the', 14, 'an', 37]
>> sentence = [ ['the', 'dog'], 'saw', ['the', 'cat']]
>> combined = scores + linguists
```

Indexing and slicing lists

- Indexing and slicing work like strings.
- Indexing returns the **object element**, and slicing returns a **list**.
- We can also use indexing and slicing to modify the list contents.

Example

```
>> ablout_compounds[1]
'shillshally'
>> ablout_compounds[3:4] = ['pingpong','zigzag']
```

Indexing and slicing lists

If there is no number on the right side of the colon the range goes to the end of the list:

Example

```
>> ablout_compounds[1:]
```

If there is no number on the left side of the colon the range starts at the beginning of the list:

Example

```
>> ablout_compounds[:2]
```

- Checking if something is a member of a list:

Example

```
>> 'pingpong' in ablout_compounds
True
>> 'pingping' in ablout_compounds
False
```

- The length of a list

Example

```
>> len(ablout_compounds)
```

- Iterating with for loop

Example

```
>> for compound in ablout_compounds:
       print(compound)
```

Other things you can do with lists

- Methods are a special type of function that is 'attached' to some object. That is, a class of Python objects may have a set of functions that are associated with it.
- Here's a weird analogy. Suppose I have two dogs, Kirby and Watson, that attended obedience school. They are so well trained that they only respond to commands issued to them and not the other dog. I issue these commands by first saying the dog's name and then the command:

Kirby, sit.

Watson, here.

Watson, down.

Kirby, here.

Other things you can do with lists

To make these look like Python method syntax, which uses a dot-notation, the commands would be

Example

```
Kirby.sit()
Watson.here()
Watson.down()
Kirby.here()
```

Other things you can do with lists

- So, looking at the first command `Kirby.sit()` I first reference the Python object - Kirby, in this case. Then I issue the command sit() and assume that the Python object (Kirby) has some internal set of instructions that tell it how to execute the command. It could be that Kirby and Watson execute the commands differently.
- For example, Watson might interpret 'here' as meaning to come to me, circle in back of me, come up on my left side, and sit. Kirby might interpret 'here' as coming to me and stand right in front of me.
- In a similar way, Python does have some useful methods associated with lists. NB: In some editors, when you type in an object name followed by a period, it will display all the methods applicable to that object.

List methods

Lists also have some useful methods, e.g.,

- `append()` : adds **an element** to the list.
- `extend()` : adds **multiple elements** to the list.
- `sort()` : orders of a list in place.
- `pop()` : remove the last element from the list and return it.

Example

```
linguists = ['Emily', 'Yvonne', 'Taco', 'Amber', 'Simon', 'Sally
linguists.append('Owen')
linguists.extend('John','Mary')
linguists.sort()
linguists.pop()
```

List operations

List operation	Explanation	Example
`sort`	Sorts a list in place	`x.sort()`
`+`	Adds two lists together	`x1 + x2`
`*`	Replicates a list	`x = ['y'] * 3`
`min`	Returns the smallest element in a list	`min(x)`
`max`	Returns the largest element in a list	`max(x)`
`index`	Returns the position of a value in a list	`x.index['y']`
`count`	Counts the number of times a value occurs in a list	`x.count('y')`
`in`	Returns whether an item is in a list	`'y' in x`

List operations

List operation	Explanation	Example
`[]`	Creates an empty list	`x = []`
`len`	Returns the length of a list	`len(x)`
`append`	Adds a single element to the end of a list	`x.append('y')`
`insert`	Inserts a new element at a given position in the list	`x.insert(0, 'y')`
`del`	Removes a list element or slice	`del(x[0])`
`remove`	Searches for and removes a given value from a list	`x.remove('y')`
`reverse`	Reverses a list in place	`x.reverse()`

Exercise

Find the differences: **sort()** and **sorted()**

Exercise

- `sort()` is a built-in method for lists themselves; while `sorted()` is the built-in function for other iterables in Python (e.g., string, dictionary key, tuple).

Example

```
> sorted({3: 'D', 1: 'A', 2: 'Z', 9: 'E', 15: 'A'})
[1, 2, 3, 9, 15]
```

- `sort()` re-orders the original list; while `sorted()` returns a new ordered list.

Example

```
> x = [3,2,2,5]
> x.sort()
> x
[2, 2, 3, 5]
```

Tuple

- The main difference between list and tuple is that you can change a list, but you can't change a tuple. (i.e, *immutable*)
- A tuple is created with parentheses rather than square brackets.
- Similar to other operations on sequence types, include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in methods for finding the length of a sequence, and for finding its largest and smallest elements, etc.

Example

```
>> tu = ('an', 'introduction', 'to', 'python', 'programming')
>> tu[-1]
>> tu[0:3] # slicing
>> len(tu)
>> max(tu) # get 'to', why?
>> tu[1]='brief'  # assigment to a tuple raises an error!!
```

Tuple: why bothers?

- Tuples take up slightly less memory and are faster to access. Probably not as flexible but are more efficient than lists.
- Usually used to serve as *dictionary keys*.

Dictionary (a.k.a. Associative Arrays)

A dictionary is:

- addressed by key, not by offset.
- unordered collections of arbitrary objects.
- mutable (can change the elements, like list)
- think of it as a set of **key:value pairs** (also called an **item**)
- use a *key* to access its *value*.

Example

```
score = {'Mike':95, 'Owen':94, 'Taco':96}
j_score = score['Taco']
n = len(score)
```

Dictionary

- Like lists, dictionaries can store objects of any type. To add items to the dictionary, use square brackets.
- Note: keys are unique. Therefore, if a key-value pair is added to the dictionary for a key that already exists, the new value will replace the old one.
- use `keys()` and `values()` to retrieve all the keys and values in a dictionary.

Example

```
> d = {}      # or: d = dict()
> d['shukai'] = 'friend'
> d['shukai']
> d['shukai'] = 'family'
> d['shukai']
> d.keys()
> d.values()
> d.items()
# items method returns all key-value pairs as a sequence of tuple, use list() function when necessary. list(d.items()
```

Dictionary: A more real example

Let's use the dictionary data structure to store part of a real dictionary. The values in a dictionary can be anything, like lists, so that we can create more realistic dictionaries that have multiple translation for each word defined.

Example

```
# Creating a dictionary with list as the type of values
> d = {'geklets': ['talking', 'gossiping'], 'geknabbel':['nibbling', 'gnawing'],
 'gekletter':['clatter']}

> print "gekletter can be understood as", d['geklets']

# Get keys and retrieve items by keys
> words = d.keys()
> words.sort()
```

Dictionary methods

Table 5-5. Python's dictionary methods: A summary of commonly used methods and idioms involving dictionaries

Example	Description
`d = {}`	Create an empty dictionary and assign it to d
`d[key] = value`	Assign a value to a given dictionary key
`d.keys()`	The list of keys of the dictionary
`list(d)`	The list of keys of the dictionary
`sorted(d)`	The keys of the dictionary, sorted
`key in d`	Test whether a particular key is in the dictionary
`for key in d`	Iterate over the keys of the dictionary
`d.values()`	The list of values in the dictionary
`dict([(k1,v1), (k2,v2), ...])`	Create a dictionary from a list of key-value pairs
`d1.update(d2)`	Add all items from d2 to d1
`defaultdict(int)`	A dictionary whose default value is zero

Exercise: A simple word counting script, AGAIN

How many times each word occurs in a sentence? (Using `strings` as keys)

Example

```
> test_string = "To be or not to be"
> counts = {}
> for word in test_string.split():
    counts[word] = counts.get(word, 0) + 1
> for word in counts:
    print "The word", word, "occurs", counts[word], \
    "times in the string"
```

What can be used as a key in dictionary?

- `keys` are immutable (i.e., can not be modified).
- This means that lists can't be used as dictionary keys!
- **Tuple**, the *immutable list* is the solution here.