Handout 10: Modification, Dicts, Functions

Modifying objects

1. Every object has an **ID** (memory location)

   a. The `id` function:

```
>>> x = ['hi', [10, 20], 'bye']
>>> id(x)
4299886104
```

   b. The `is` operator tests whether the IDs are the same

```
>>> id(x[1])
4299859512
>>> y = [10, 20]
>>> id(y)
4300096144
>>> x[1] == y
True
>>> x[1] is y
False
```

   c. What is stored in the variable is actually the ID: it is a **pointer** to the object

```
>>> nums = x[1]
>>> id(nums)
4299859512
>>> nums is x[1]
True
```

2. It is possible to change a list after it has been created

   a. Example:

```
>>> nums[1] = 'new'
>>> nums
[10, 'new']
```

   b. This is dangerous. We just changed `x`! (But not `y`.)

```
>>> x
['hi', [10, 'new'], 'bye']
>>> y
[10, 20]
```

   c. Assignment to a list element is **destructive**.

3. Some other destructive methods: adding new elements

a. `append` to a list:

```
>>> nums.append(30)
>>> nums
[10, 'new', 30]
>>> x
['hi', [10, 'new', 30], 'bye']
```

b. Contrast with `+`, which is not destructive:

```
>>> nums + [40]
[10, 'new', 30, 40]
>>> nums
[10, 'new', 30]
```

c. `add` to a set:

```
>>> s = set(['a', 'b'])
>>> s
set(['a', 'b'])
>>> s.add('z')
>>> s
set(['a', 'b', 'z'])
```

d. Note this difference:

```
>>> nums.append(30)
>>> nums
[10, 'new', 30, 30]
>>> s.add('z')
>>> s
set(['a', 'b', 'z'])
```

4. Sorted versus sort

```
>>> w1 = ['horse', 'dog', 'cow']
>>> w2 = w1
>>> sorted(w1)
['cow', 'dog', 'horse']
>>> w2
['horse', 'dog', 'cow']
>>> w1.sort()
>>> w2
['cow', 'dog', 'horse']
```

5. *Exercise.* Which of these steps are destructive?

```
>>> x = [4,6,8]
>>> x[1] = 10
>>> x = x + [50, 60]
```

```
>>> x.append(70)
>>> print(x[1])
>>> y = x[1:3]
>>> d = {'hi':1, 'there':2}
>>> d['hi']
>>> d['foo'] = 3
```

6. Strings and tuples are **immutable**. Attempting to modify them causes an error.

Dicts

7. A dict
   a. Maps from **keys** to **values**

```
>>> d = {'apple':2, 'pear':5, 'banana':3}
```

   b. A FreqDist is a specialization of dict
   c. Look up:

```
>>> d['apple']
2
```

   d. Attempting to access a nonexistent element gives an error
   e. Can use "`in`" to test first

```
>>> if 'horse' in d:
...     print(d['horse'])
... else:
...     print('Not found')
...
Not found
```

   f. Can add, change, delete entries:

```
>>> d['apple'] = 42
>>> d['horse'] = 'foo'
>>> del d['horse']
```

8. Creation
   a. From a set of pairs:

```
>>> pairs = [('apple', 2), ('pear', 5), ('banana', 3)]
>>> d2 = dict(pairs)
>>> d2['pear']
5
```

   b. List comprehension:

```
>>> d3 = dict( (w,len(w)) for w in ['apple', 'pear', 'banana'] )
>>> d3['banana']
6
```

9. Iteration

a. Over the dict: iterates over keys **in random order**:

```
>>> list(d)
['pear', 'apple', 'banana']
```

b. Iterating over the values:

```
>>> list(d.values())
[5, 2, 3]
```

c. Iterating over key-value pairs:

```
>>> list(d.items())
[('pear', 5), ('apple', 2), ('banana', 3)]
```

Functions

10. Local and global variables

a. Which are which? `s`, `c`, `nouns`

```
>>> nouns = ['cat', 'dog', 'horse']
>>> def nnouns (s):
...     return sum(1 for c in s if c in nouns)
...
>>> nnouns('cat and dog'.split())
2
```

b. The rule: if you assign to it, it is local

```
>>> def nnouns (s):
...     return sum(1 for c in s if c in nouns)
...     vowels += 'rabbit'
...
>>> nnouns('dog chase rabbit'.split())
Traceback (most recent call last):
    ...
NameError: free variable 'vowels' referenced before assignment in enclosing sc
```

c. Explain this behavior:

```
>>> def addnoun1 (x):
...     nouns += x
...
>>> addnoun1('horse')
Traceback (most recent call last):
    ...
UnboundLocalError: local variable 'nouns' referenced before assignment
>>> def addnoun2 (x):
...     nouns.append(x)
...
```

```
>>> addnoun2('rabbit')
>>> nouns
['cat', 'dog', 'horse', 'rabbit']
```

11. Instead of global variables, use objects

a. Example:

```
class NounCounter (object):

    def __init__ (self):
        self.nouns = ['cat', 'dog', 'horse']

    def count (self, s):
        return sum(1 for c in s if c in self.nouns)
```

b. Usage:

```
>>> c = NounCounter()
>>> c.count('cat and dog'.split())
2
```

c. Making it callable:

```
class NounCounter (object):
    ...
    def __call__ (self, s):
        return sum(1 for c in s if c in self.nouns)
```

d. Then:

```
>>> nnouns = NounCounter()
>>> nnouns('cat and dog'.split())
2
```

12. A function should either return a value or have a side effect, but not both.

```
def count_vowels (s):
    return sum(1 for c in s if c in 'aeiou')

def upcase_vowels (lst):
    for i in range(len(lst)):
        if lst[i] in 'aeiou':
            lst[i] = lst[i].upper()
```

13. Optional and keyword arguments

a. Example:

```
def count_vowels (s, i=0, j=None, include_y=False):
    if include_y: vowels = 'aeiouy'
    else: vowels = 'aeiou'
    return sum(1 for c in s[i:j] if c in vowels)
```

b. Usage:

```
>>> count_vowels('history')
2
>>> count_vowels('history', 3)
1
>>> count_vowels('history', 2, 4)
0
>>> count_vowels('history', include_y=True)
3
```