Tips for Common Problems

PS 452: Text as Data

Fall 2014

Department of Political Science, Stanford University

Created by **Frances Zlotnick**

Please contact me at zlotnick@stanford.edu with questions or comments.

This document provides some tips based on common python problems observed in the problem sets. It covers the following topics:

Iterators

One of the great benefits of using python is the functionality provided by its built-in iterators. Iterating over collections is very simple in python, but sufficiently different from `R` that it can cause confusion for new python users.

Python collections can be easily iterated over using the simple `for` syntax. `R` users often iterate over vector or dataframe indexes by writing `for` loops that iterate through the range of numbers corresponding to the dimensions of a collection. Python allows you to iterate directly over the objects without indexing the locations, which is particularly valuable when dealing with unordered (and thus un-indexible) collections. All collections have built-in iterators that behave somewhat differently. The `list` iterator is most straight-forward and iterates over each element in the list in order.

In [1]:
```
random_cities = ["San Francisco", "Stockholm", "Cleveland", "Washington, D.C.", "Warsaw"]

for i in random_cities:
    print i
```

```
San Francisco
Stockholm
Cleveland
Washington, D.C.
Warsaw
```

Iterating over `tuples` is very similar.

In [2]:
```
tup = ("This", "is", "my", "silly", "tuple.")
for i in tup:
    print i
```

```
This
is
my
silly
tuple.
```

By default, the `dict` iterator iterates over the keys in a dictionary only. The order of the elements reurned is arbitrary and does not refect the actual ordering of the data. You should not rely on this ordering for other operations.

In [3]:
```
capitals = {
    "United States": "Washington, D.C.",
    "Sweden": "Stockholm",
    "California": "Sacramento",
    "Poland": "Warsaw",
    "Ohio": "Columbus"
}

for i in capitals:
    print i
```

```
United States
Ohio
Sweden
California
Poland
```

You can also easily iterate over the values, or both keys and values, with slight modifications.

In [4]:
```
for i in capitals.values():
    print i
```

```
Washington, D.C.
Columbus
Stockholm
Sacramento
Warsaw
```

In [5]:
```
for k, v in capitals.items():
    print k, v
```

```
United States Washington, D.C.
Ohio Columbus
Sweden Stockholm
California Sacramento
Poland Warsaw
```

You can even iterate over all of the characters in a `string` using the same syntax!

In [6]:
```
string = "my string"

for i in string:
    print i.upper()
```

```
M
Y

S
T
R
I
N
G
```

If you do need to iterate over a range of numbers, remember that python is zero indexed and that the stop value in `range` is non-inclusive. To successively acces every element in a collection, you should iterate over all of the numbers from 0 to the length of the collection. Note that if you use this method you are no longer iterating over the elements directly, so you need to explicilty index the collection in order to process an element on each loop.

In [7]:
```
test = ["This", "is", "a", "complete", "sentence."]

for i in range(0, len(test)):
    print test[i]
```

```
This
is
a
complete
sentence.
```

If you subtract one from the stop value, you will miss the last element.

In [8]:
```
for i in range(0, len(test)-1):
    print test[i]
```

```
This
is
a
complete
```

Casting to and concatenating strings

When casting objects into `string` type, be certain to cast the elements themselves and not the collection holding them. Casting a collection will result in unexpected additions to your text.

In [9]:
```
nums = range(1,11)
wrong = str(nums)
wrong
```

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

In [10]:
```
wrong[0:3]
```

Out[10]:
```
'[1,'
```

Casting the entire collection to a `string` adds all of the structural information from the `list` data structure into the text. The square brackets denoting the `list` type, and the commas separating elements have all been added into the data. You would then need to do a bunch of extra processing to strip out these additions, which would be difficult if you wanted to maintain the original punctuation marks in the data.

This problem often occurs when you get data in the form of an unfamiliar data structure and try to turn it into something familiar by casting it to a string.

If you want to cast elements of a collection to another data type, you must cast each element individually, not the collection as a whole.

In [11]:
```
right = [str(x) for x in nums]
right
```

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

In [12]:
```
right[0:3]
```

Out[12]:
```
['1', '2', '3']
```

If you want to concatenate a collection of strings, use the `join` function rather than casting the collection into a `string`. This function has unusual syntax: you first specify the separator you want between the elements, then call `join`, providing the collection of strings as an argument. Usually you want to separate the items with a space.

In [13]:
```
" ".join(test)
```

Out[13]:
```
'This is a complete sentence.'
```

You can specify any string object as a separator, however.

In [14]:
```
wrong.join(test)
```

Out[14]:
```
'This[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]is[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]a[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]complete[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]sentence.'
```

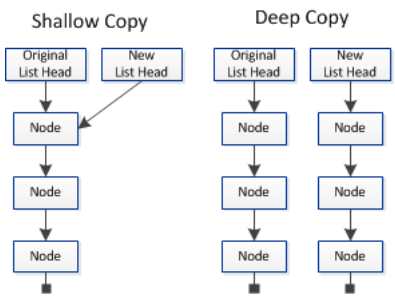
Copying mutable objects

Often you may want to copy an object so that you can manipulate it without destroying the original data. This is tricky when dealing with mutable objects because python, unlike `R`, makes a *shallow* copy of mutable objects.

There are two ways to copy data. A deep copy produces a full and complete duplication of the object, while a shallow copy simply creates a *pointer* that points to the memory location of the object to be copied. This is more efficient than copying all of the values and taking up memory with duplicate information. However, it means that **there is only one set of data accessed by both references**.

In [15]:
```
from IPython.display import Image
Image("http://i.stack.imgur.com/hOPkR.png")
```

Out[15]:



The diagram above illustrates the difference between a shallow copy and a deep copy of a linked list. The big implication of shallow copying is that *if you modify the object by calling either one of the references, the other reference will reflect those changes.*

In [16]:
```
word_list = ["this", "is", "a", "list", "of", "words"]
copy = word_list
copy
```

Out[16]:
```
['this', 'is', 'a', 'list', 'of', 'words']
```

In [17]:
```
word_list.append("dude")
copy
```

Out[17]:
```
['this', 'is', 'a', 'list', 'of', 'words', 'dude']
```

In [18]:
```
copy.remove("this")
word_list
```

Out[18]:
```
['is', 'a', 'list', 'of', 'words', 'dude']
```

In [19]:
```
print capitals
```

```
{'United States': 'Washington, D.C.', 'Ohio': 'Columbus', 'Sweden': 'Stockholm', 'California': 'Sacramento', 'Poland': 'Warsaw'}
```

This behavior applies to all mutable objects, so it occurs with `dictionaries` as well.

In [20]:
```
cap_copy = capitals
cap_copy["Uzbekistan"] = "Tashkent"
```

In [21]:
```
capitals
```

Out[21]:
```
{'California': 'Sacramento',
 'Ohio': 'Columbus',
 'Poland': 'Warsaw',
 'Sweden': 'Stockholm',
 'United States': 'Washington, D.C.',
 'Uzbekistan': 'Tashkent'}
```

This behavior does not apply to immutable objects like `ints` or `strings`.

In [22]:
```
a = 5
b = a
b
```

Out[22]: 5

In [23]:
```
b = 3
a
```

Out[23]: 5

In [24]:
```
s = "a string"
s_copy = s
s_copy = s + ", man."
s_copy
```

Out[24]: 'a string, man.'

In [25]:
```
s
```

Out[25]: 'a string'

If you want to make a deep copy, use the copy module.

In [26]:
```
from copy import deepcopy
list_1 = ["this", "is", "a", "test"]
list_2 = deepcopy(list_1)
list_2.append("TEST!")
list_2
```

Out[26]: ['this', 'is', 'a', 'test', 'TEST!']

In [27]:
```
list_1
```

Out[27]: ['this', 'is', 'a', 'test']

Inserting vs. Appending to lists

Whenever possible, you should append to `lists` rather than inserting elements. Inserting an element requires that every element after the insertion point be moved in memory to accomodate the new item. Inserting at index 0 is O(n), since it requires moving all n elements in your `list`. Appending is in general O(1), because it does not require moving any existing elements, simply adding one after the last item.^ This is the same as the difference between searching over unhashed and hashed objects!

^This is not always true; if your list grows to the point where the allocated memory is no longer sufficient to hold it, python will copy it into a new location with more available memory, which diminishes the efficiency. This occurs whether the length grows through appending or inserting.

In [28]:
```
number_list = range(1, 10000)
```

In [29]:
```
%%timeit
number_list.insert(0, 500)
```

```
100000 loops, best of 3: 160 µs per loop
```

In [30]:
```
%%timeit
number_list.append(500)
```

```
10000000 loops, best of 3: 167 ns per loop
```

There are 1,000 nanoseconds in each microsecond, so appending is around 1000 times faster than inserting, in this example.

The lesson here is that you should add elements to lists in the order you want them to occur, whenever possible. Sometimes you will need to `insert` items, but if it isn't necessary, you should avoid it in favor of `append`.