Python Guide for CMSC 416

This guide goes over some Python knowledge that will help you complete assignments in this class.

Command line arguments

Data passed to Python from the command line is always available, but has to be imported.

```
# python my_file.py 3 4 hello goodbye
>>> from sys import argv
>>> argv
['my_file.py', '3', '4', 'hello', 'goodbye']
```

The arguments are given in a list of strings (even if a string only has numeric characters), and name of the file is always the first element.

If a string is all digits, you can convert it to an int like this: `int('42')` .

List slicing

Recall that the (mutable) ordered sequence type in Python is named "list". They are similar to the data type called "arrays" in C-style languages, though their length is not fixed. When one talks about "arrays" in Python, it's assumed that they're talking about the array data type defined in Numpy. Avoid using the two terms interchangeably.

Lists have a few properties that make them easy to work with: you can use negative indices to count from the end of the list, and you can use `start:stop:step` syntax to pick out specific slices. You don't have to include all three values, and when one is missing, `start` is assumed to be `0` , `stop` is assumed to be the current length of the list, and `step` is assumed to be `1` .

All of this works for strings as well.

```
>>> stuff = ['a', 'b', 'c', 'd']
>>> stuff[-1]
'd'
>>> stuff[1:]  # from the second element onward
['b', 'c', 'd']
>>> stuff[:-2]  # up to the next-to-last element
['a', 'b']
>>> stuff[1:-1]  # the middle, without the first and last element
['b', 'c']
>>> stuff[::2]  # every other element
```

```
['a', 'c']
>>> stuff[::-1]  # step through the list backwards (i.e. reverse it)
['d', 'c', 'b', 'a']
```

Hashing tuples

Recall that dictionaries are the hash table implementation in Python. You can use tuples as dictionary keys and look things up by n-tuples rather than n nested dictionaries.

```
>>> new_dict = {('a', 'b', 'c'): 2, ('b', 'c', 'a'): 5}
>>> new_dict['b', 'c', 'a']
5
>>> new_dict['a', 'b', 'c']
2
>>> new_dict['a', 'b', 'c'] += 1
>>> new_dict['a', 'b', 'c']
3
```

Counting with special dictionary subclasses

The standard library has several modules that will save you from needing to re-invent the wheel during this course, except for the wheel-reinvention that is the point of the assigments. This section details those that will help you with with counting.

`collections.defaultdict`

`defaultdict` will automatically supply a value if a key is looked up that isn't present. For example, it's often useful to have a `defaultdict` that automatically supplies `0`.

```
>>> from collections import defaultdict
>>> new_dict = defaultdict(int)  # supply whatever `int()` returns, which is `0`, when a key isn't pre
>>> new_dict
defaultdict(<class 'int'>, {})
>>> new_dict['a', 'b', 'c'] += 1
>>> new_dict
defaultdict(<class 'int'>, {('a', 'b', 'c'): 1})
```

Another common use case is to have `defaultdict(list)` so that you can append values to a list without having to worry if the list has been created for that key yet.

`collections.Counter`

`Counter` will solve a lot of the same problems as `defaultdict`, but has more features. One constructs a `Counter` by passing it an iterable of some kind (usually a list).

```
>>> from collections import Counter
>>> counts = Counter(['a', 'b', 'a', 'c', 'c', 'a'])
>>> counts
Counter({'a': 3, 'c': 2, 'b': 1})
>>> counts['a']
3
>>> counts['z']  # This hasn't been counted!
0
>>> counts + Counter(['x', 'y', 'x', 'z'])  # You can merge Counters with the + operator
Counter({'a': 3, 'c': 2, 'x': 2, 'b': 1, 'y': 1, 'z': 1})
```

Repeating something n times

If you want to repeat certain code a certain number of times, don't do this:

```
# bad
num_times = 5
while num_times > 0:
    do_stuff()
    num_times -= 1
```

Do this instead

```
# good
for _ in range(5):
    do_stuff()
```

A lone underscore is used as a variable name when a variable has to be created (you can't have `for in range(5)`) but you don't plan to use it. In situations where you can just not assign an expression to a variable, that is preferred.

Sets

A set is like a list, but with no duplicates, and where the elements don't have any particular order. In this course, it may sometimes be useful to have a set of strings. They look like lists but with curly braces instead of square brackets.

```
>>> string_set = {'virginia', 'commonwealth', 'university'}
>>> 'virginia' in string_set
True
>>> string_set.add('rams')
```

```
>>> string_set  # the order that the elements are displayed is arbitrary!
{'virginia', 'rams', 'university', 'commonwealth'}
>>> string_set[0]  # causes an error, as order does not apply here
>>> string_set.add('rams')  # has no effect, as the element is already present
```

Like lists, sets are mutable, and thus cannot be used as dictionary keys. The immutable analog is `frozenset`, which you can make by passing a set to `frozenset()`. Frozensets can be used as dictionary keys.

You cannot create an empty set with `{}`, as that creates an empty dict. The expressions `set()` and `frozenset()` create empty (frozen)sets.

List/dict comprehensions

Instead of this:

```
>>> squares = []
>>> for n in range(5):
...     squares.append(n ** 2)
>>> squares
[0, 1, 4, 9, 16]
```

Write this:

```
>>> squares = [n ** 2 for n in range(5)]
```

It also works for dicts.

```
>>> {word.lower(): len(word) for word in ('Virginia', 'Commonwealth', 'University')}
{'virginia': 8, 'commonwealth': 12, 'university': 10}
```

Iterating over dicts

If you need to iterate over a dict, don't look up the value every time from the key. There are three methods for making dict iteration easy to understand.

```
for key in some_dict.keys():
    ...

for value in some_dict.values():
    ...
```

```
for key, value in some_dict.items():
    ...
```