deque

According to the Python documentation, **deques** “are a generalization of stacks and queues”. They are pronounced “deck” which is short for "double-ended queue". They are a replacement container for the Python list. Deques are thread-safe and support memory efficient appends and pops from either side of the deque. A list is optimized for fast fixed-length operations. You can get all the gory details in the Python documentation. A deque accepts a **maxlen** argument which sets the bounds for the deque. Otherwise the deque will grow to an arbitrary size. When a bounded deque is full, any new items added will cause the same number of items to be popped off the other end.

As a general rule, if you need fast appends or fast pops, use a deque. If you need fast random access, use a list. Let’s take a few moments to look at how you might create and use a deque.

```
from collections import deque
import string
d = deque(string.ascii_lowercase)
for letter in d:
    print(letter)
```

Here we import the deque from our collections module and we also import the **string** module. To actually create an instance of a deque, we need to pass it an iterable. In this case, we passed it **string.ascii_lowercase**, which returns a list of all the lower case letters in the alphabet. Finally, we loop over our deque and print out each item. Now let’s look at at a few of the methods that deque possesses.

```
d.append('bork')
print (d)
#deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
#        'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'bork'])

d.appendleft('test')

print (d)
#deque(['test', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
#        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'bork'])

d.rotate(1)
print (d)
#deque(['bork', 'test', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
#        'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'])
```

Let's break this down a bit. First we append a string to the right end of the deque. Then we append another string to the left side of the deque… Lastly, we call **rotate** on our deque and pass it a one, which causes it to rotate one time to the right. In other words, it causes one item to rotate off the right end and onto the front. You can pass it a negative number to make the deque rotate to the left instead.

Let's finish out this section by looking at an example that's based on something from the Python documentation:

```
from collections import deque


def get_last(filename, n=5):
    """
    Returns the last n lines from the file
    """
    try:
        with open(filename) as f:
            return deque(f, n)
    except OSError:
        print("Error opening file: {}".format(filename))
        raise
```

This code works in much the same way as Linux's **tail** program does. Here we pass in a filename to our script along with the n number of lines we want returned. The deque is bounded to whatever number we pass in as n. This means that once the deque is full, when new lines are read in and added to the deque, older lines are popped off the other end and discarded. I also wrapped the file opening **with** statement in a simple exception handler because it's really easy to pass in a malformed path. This will catch files that don't exist for example.

Now we’re ready to move on and learn about the namedtuple.