This is a Python programming assignment. You will edit your ongoing copy of `qc.py` and submit it for grading. The grader will `import qc` and then run their own testing code against it. Probably the grader will also inspect your code.

A. Implement the function `function` based on this specification. To understand what is meant by `deutschTest`, look ahead to problem C.

```
def function(n, m, f):
    '''Assumes that n = m = 1. The argument f is a Python function that takes
    as input an n-bit string alpha and returns as output an m-bit string
    f(alpha). See deutschTest for examples of f. This function returns the
    (n + m)-qbit gate F that corresponds to f.'''
```

B. Implement Deutsch's algorithm in the function `deutsch` based on this specification.

```
def deutsch(f):
    '''Given a two-qbit gate representing a function f : {0, 1} -> {0, 1},
    outputs ket1 if f is constant and ket0 if f is not constant.'''
```

C. Paste the following function into your `qc.py`. It tests all four cases of Deutsch's algorithm while also testing the `function` function. Use it to test your code, but do not submit the test results for grading. (Yes, this part of the assignment is literally copying and pasting. If you find that this function doesn't work, then your problem is in `function` or `deutsch`.)

```
def deutschTest():
    print("One should see ket0s.")
    def f(x):
        return (1 - x[0],)
    print(deutsch(function(1, 1, f)))
    def f(x):
        return x
    print(deutsch(function(1, 1, f)))
    print("One should see ket1s.")
    def f(x):
        return (0,)
    print(deutsch(function(1, 1, f)))
    def f(x):
        return (1,)
    print(deutsch(function(1, 1, f)))
```

D. You already have a function `application`. Improve it to match this specification. It is possible that your one- or two-qbit implementation is already sufficient.

```
def application(gate, state):
    '''Assumes n >= 1. Applies the n-qbit gate to the n-qbit state, returning
    an n-qbit state.'''
```

E. You already have a function `tensor`. Improve it, if necessary, to match this specification. By the way, `numpy.kron` is nearly identical in its functionality. You may not use `numpy.kron` to implement this function, but it wouldn't be a bad idea to test your implementation against `numpy.kron`. My implementation is about five or seven lines of code organized into two or three cases.

```
def tensor(a, b):
    '''Assumes that a and b are both gates or a and b are both states. Let a be
    n-qbit and b be m-qbit, where n, m >= 1. Returns the tensor product of a
    and b, which is (n + m)-qbit.'''
```

F. You already have a function `first`. Improve it to match this specification. You might need to do some algebra on paper, to figure out the details of the calculation. Do not submit your paper work for grading.

```
def first(state):
    '''Assumes n >= 1. Given an n-qbit state, measures the first qbit. Returns
    a pair (a tuple or list of two elements) consisting of a classical one-
    qbit state (either ket0 or ket1) and an (n - 1)-qbit state.'''
```

G. Paste the following function into your `qc.py`. Use it to test your implementation of `first`, but do not submit the test results for grading.

```
def firstTest345(n, m):
    '''Assumes n >= 1. Uses one more qbit than that, so that the total number
    of qbits is n + 1. The parameter m is how many tests to run. Should return
    a number close to 0.64 --- at least for large m.'''
    psi0 = 3 / 5
    beta = uniform(n)
    psi1 = 4 / 5
    gamma = uniform(n)
```

```
    chi = psi0 * tensor(ket0, beta) + psi1 * tensor(ket1, gamma)
    def f():
        if (first(chi)[0] == ket0).all():
            return 0
        else:
            return 1
    acc = 0
    for i in range(m):
        acc += f()
    return acc / m
```

H. You already have a function `last`. Improve it to match this specification.

```
def last(state):
    '''Assumes n >= 1. Given an n-qbit state, measures the last qbit. Returns
    a pair (a tuple or list of two elements) consisting of an (n - 1)-qbit
    state and a classical one-qbit state (either ket0 or ket1).'''
```

I. Write a function `lastTest345`, based on `firstTest345`, to test your `last` function. Your implementation of `lastTest345` should be in your `qc.py` for grading, but the test results should not be.