YOUR NAME PLEASE:

NETID:

Computer Science 200b
Exam 2 - Practice
April 2016

Enter your netid at the bottom of each page.

Closed book and closed notes. No electronic devices. Show ALL work you want graded on the test itself. You may not hand in a Blue Book.

For problems that do not ask you to justify the answer, an answer alone is sufficient. However, if the answer is wrong and no derivation or supporting reasoning is given, there will be no partial credit.

GOOD LUCK!

Problem	Points	Actual
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
Total	60	

1.(a) (10 points)

Short answer.

What is a decorator? What purpose do decorators serve in Python?

> A decorator is a wrapper function that surrounds a function, optionally executing code before and after the function executes. It can be used for debugging, tracking, memoization, and other nifty purposes. It relies on the fact that functions can be passed as arguments in Python.

What is an exception? What purpose do exceptions serve in Python? Explain the related meaning of EAFP vs LBYL.

> An exception is a runtime error that interrupts normal control flow. They are not unconditionally fatal It is possible to catch an exception and deal with it programmatically. The idea is that certain errors are predictable, e.g., division by zero, array out of bounds, missing file or permission errors. Exceptions embody the EAFP philosophy (easier to ask forgiveness than permission) as opposed to the LBYL approach (look before you leap). See "**import this**" for more Python philosophy.

How does object oriented programming improve software reliability?

- Code reuse
- Encapsulation
- Structure
- Maintenance
- Consistency
- Polymorphism

See http://zoo.cs.yale.edu/classes/cs200/lectures/oop.html

2. (10 points)

Below we define several functions, which may or may not raise an exception when executed. For each function, indicate if an exception is raised always, sometimes, or never. For always and sometimes, indicate what exception is raised and for sometimes specify the conditions under which the exception occurs.

```
def f1():
    return 1/0
```

Always
ZeroDivisionError: division by zero

```
def f2():
    return a
```

Sometimes - if a is not defined
NameError: name 'a' is not defined

```
def f3():
    a = [1, 2]
    return a[2]
```

Always
IndexError: list index out of range

```
def f4():
    a = [1, 2, 3, 4]
    return a[2]
```

Never

3. (10 points) Define a python procedure x() that generates the following bytecode

```
>>> dis.dis(x)
  6           0 LOAD_CONST               1 (2)
              3 STORE_FAST               0 (a)
  7           6 LOAD_CONST               2 (3)
              9 STORE_FAST               1 (b)
  8          12 LOAD_FAST                0 (a)
             15 LOAD_FAST                1 (b)
             18 BINARY_ADD
             19 STORE_FAST               2 (c)
  9          22 LOAD_FAST                2 (c)
             25 LOAD_FAST                2 (c)
             28 BINARY_MULTIPLY
             29 STORE_FAST               3 (d)
 10          32 LOAD_FAST                3 (d)
             35 RETURN_VALUE
```

```
def x():
    a = 2
    b = 3
    c = a + b
    d = c * c
    return d
```

4. (10 points)

Provide the bytecode generated for the following Python function. Use dis.dis() format, but without the source code line numbers from the first column.

```
def y():
   a = 7
   if a % 2:
      return 1 + 3*a
   else:
      return a / 2
```

```
>>> y()
22
>>> dis.dis(y)
 20           0 LOAD_CONST               1 (7)
              3 STORE_FAST               0 (a)

 21           6 LOAD_FAST                0 (a)
              9 LOAD_CONST               2 (2)
             12 BINARY_MODULO
             13 POP_JUMP_IF_FALSE       28

 22          16 LOAD_CONST               3 (1)
             19 LOAD_CONST               4 (3)
             22 LOAD_FAST                0 (a)
             25 BINARY_MULTIPLY
             26 BINARY_ADD
             27 RETURN_VALUE

 24      >>  28 LOAD_FAST                0 (a)
             31 LOAD_CONST               2 (2)
             34 BINARY_TRUE_DIVIDE
             35 RETURN_VALUE
             36 LOAD_CONST               0 (None)
             39 RETURN_VALUE
```

5. (10 points)

Write the UNIX command(s) corresponding to **XXXX** in the transcript below.

```
-bash-4.2$ pwd
/home/accts/sbs5/cs201
-bash-4.2$ ls
bin  class  graded  handouts  hws  previous-years  README  SUBMIT
TESTS  www
-bash-4.2$ mkdir mt
-bash-4.2$ cd mt
-bash-4.2$ pwd
/home/accts/sbs5/cs201/mt
-bash-4.2$ ls .. > f1
-bash-4.2$ ls -l
total 4
-rw-rw-r-- 1 sbs5 cs201ta 72 Oct  6 16:12 f1
-bash-4.2$ cat f1
bin
class
graded
handouts
hws
mt
previous-years
README
SUBMIT
TESTS
www
-bash-4.2$ cp f1 f2
-bash-4.2$ ls -l
total 8
-rw-rw-r-- 1 sbs5 cs201ta 72 Oct  6 16:12 f1
-rw-rw-r-- 1 sbs5 cs201ta 72 Oct  6 16:13 f2
-bash-4.2$ diff f1 f2
-bash-4.2$ rm f1
-bash-4.2$ ls -l
total 4
-rw-rw-r-- 1 sbs5 cs201ta 72 Oct  6 16:13 f2
-bash-4.2$ wc f2
11 11 72 f2
```

6. (10 points)

Define the decorator xxx() that has the following behavior.

```
@xxx
def incr(a):
    return a + 1

@xxx
def add2(a, b):
    return a + b

>>> incr(10)
Calling: incr with args: (10,)
Exiting: incr with value: 11
11
>>> add2(2,3)
Calling: add2 with args: (2, 3)
Exiting: add2 with value: 5
5
```

```
def xxx(f):
    def g(*args):
        print ("Calling: {} with args: {}".format(f.__name__, args))
        val = f(*args)
     print ("Exiting: {} with value: {}".format(f.__name__, val))
     return val
    return g
```