Team #:_____ Name:_____
Absent:

1. A *function* is a procedural abstract (a named body of code to perform some action and return a resulting value). The syntax of a function definition is:

def functionName([parameter [, parameter]*]):
<functionBody>

where a parameter can be either:

- formal parameter name
- a keyword argument of the form: id = value which assigns the formal parameter id a specified default value. Note: keyword arguments can only appear as the last parameters in a parameter list
- a pseudo-argument of the form *args that captures all of the remaining non-keyword arguments in a tuple.
- a pseudo-argment keyword argument of the form **kwargs which captures all of the remaining keyword arguments into a dictionary

For example, the function definition "def foo(x, y, *args, **kwargs):" called with "foo(1, 2, 3, 4, a=5, b=6)" will result with *formal parameter* x containing 1, y containing 2, args containing (3, 4), and kwargs containing {'a':5, 'b':6}. Python uses pass-by-value parameter passing, which copies the value of the *actual parameters* to the *formal parameters*. Since variables associated with built-in collections and objects contain references, *actual parameters* to these only copy their reference values to corresponding formal parameters.

Predict the output of the following Python code segment.

```
def foo(a, b):
      a = 5
      b[0] = 99

myInt = 4
myList = [ 1, 2, 3, 4 ]
foo(myInt, myList)
print 'myInt =', myInt
print 'myList =', myList
```

2. A *module* in Python is a file containing a collection of function and variable definitions. Python has a vast collection of standard library modules that are installed with the Python interpreter. All but the __builtins__ module must be imported before being used. For example, the *sys* module contains system specific parameters and functions. We can find out the current system path by:

```
>>> import sys
>>> sys.path
['C:\\Python25\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python25.zip', ... ]
```

A *namespace* is a mapping from names to objects. In Python, namespaces are disjoint, i.e., identical names in different namespaces typically refer to different objects. The following table shows different namespaces in Python and their lifetime:

Namespace	When created	Lifetime
__built-in__ (built-in functions like abs(), etc.)	Interpreter start up	whole execution
a module's global namespace	module definition read in	whole execution
__main()__ namespace (top-level interpreter)	interpreter start up	whole execution
local namespace for function	function called	until function returns

Team #:_____ Name:_____
Absent:

The *scope* refers to the textual region in a program where a namespace is directly accessible (without the "dotted" qualifiers). At least 3 nested scopes are directly accessible:

- inner-most scope (local scope) is searched first
- namespace of any enclosing functions are searched from the nearest enclosing function outward
- "*middle scope*" - current module's global namespace
- built-in namespace is searched last

If a name is declared "global," then all references to it go to the middle scope.

```
# scopeTest.py tests the scope of names

a = 1
b = 2
c = 99

def f():
    def g():
        print 'IN g(): a=', a, 'b=', b, 'c=', c

    a = 3
    def h():
        global a
        print 'IN h(): a=', a, 'b=', b, 'c=', c
        a = 15
        print 'IN h(): a=', a, 'b=', b, 'c=', c
    b = 4
    print 'IN f(): a=', a, 'b=', b, 'c=', c
    g()
    print 'IN f(): a=', a, 'b=', b, 'c=', c
    h()
    print 'IN f(): a=', a, 'b=', b, 'c=', c

f()
print 'At top-level:  a=', a, 'b=', b, 'c=', c
```

a. Predict the output of the above Python code.

The *dir*([object]) function returns an alphabetized list of names comprising (some of) the attributes of the given object, and of attributes reachable from it. If no object is specified, then the names in the current scope are returned.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning', 'EOFError', 'Ellipsis', ... , 'zip']
```

Team #:_____ Name:_____
Absent:

```
# scopeTestList.py tests the scope of names

a = [1]
b = [2]
c = [99]

def f(x):
    def g():
        print 'IN g(): x=', x, 'a=', a, 'b=', b, 'c=', c

    x.append(10)
    b = 4
    print 'IN f(): x=', x, 'a=', a, 'b=', b, 'c=', c
    g()
    x = [1, 2, 3]
    print 'IN f(): x=', x, 'a=', a, 'b=', b, 'c=', c


f(a)

print 'At top-level:  a=', a, 'b=', b, 'c=', c
```

b. Predict the output of the above Python code.

3. Classes in Python have the following characteristics:

- all class members (instance variables and methods) are public, but name mangling can provide some private protection
- all data types are objects, so they can be used as inherited base classes
- most built-in operators can be redefined for a class
- objects are passed by their references
- all classes have a set of standard methods provided, but may not work properly (__.str__, __.doc__, etc.)
- The general format of class definition is:

class MyClass [(superClass1 [, superClass2]*)]:
    "Document comment which becomes the __.doc__ attribute for the class"
    def __init__(*self*, [param [, param]*):
        "Document comment for constructor method with *self* referencing to object itself"
        <__init__body>

    # defs of other class methods and assignments to class attributes

end class MyClass

Consider the following Fraction class that stores the numerator and the denominator as exact values. A Fraction objects can be created by:

```
>>> myFraction = Fraction(3,4)
>>> anotherFraction = Fraction(11, 8)
>>> fraction3 = myFraction + anotherFraction
```

Team #:_____ Name:_____
Absent:

```
def gcd(m, n):
    "Calculates the greatest-common-divisor using Euclid's Algorithm"
    # Assumes the m and n are greater than zero!!!
    while m%n != 0:
        oldm = m
        oldn = n
        m = oldn
        n = oldm % oldn

    return n
# end def gcd

class Fraction:
    "Simple Fraction class to allow rational number"
    def __init__(self, top, bottom):
        "Constructor for the Fraction class that takes a \
        numerator and denominator as parameters."
        self.num = top
        self.den = bottom
    # end def __init__

    def __str__(self):
        "Overrides the standard method, converts a Fraction to a string"
        return str(self.num)+"/"+str(self.den)
    # end def __str__

    def show(self):
        "Displays a Fractional by printing it"
        print self.num,"/",self.den
    # end def show

    def __add__(self, rhs_fraction):
        "Overrides the '+' operator for Fractions"
        newnum = self.num * rhs_fraction.den + self.den*rhs_fraction.num
        newden = self.den * rhs_fraction.den
        common = gcd(newnum, newden)
        return Fraction(newnum/common, newden/common)
    # end def __add__

    def __cmp__(self, rhs_fraction):
        "Overrides the '__cmp__' operator for Fractions, to allow for \
        to allow for a deep comparison of two Fractions"
        num1 = self.num*rhs_fraction.den
        num2 = rhs_fraction.num*self.den

        if num1 < num2:
            return -1
        elif num1 == num2:
            return 0
        else:
            return 1
    # end def __cmp__
# end class Fraction
```

a) Modify the Fraction class to include a method to perform multiplication, called __mul__.

b) How would you call __mul__ to assign `productFraction` the value of `myFraction` and `anotherFraction`?

Team #:_____ Name:_____
Absent:

4. Consider the following modules:

```
# File myModuleA:  This is an example module.

# Module variables -- executed first time the module is imported
modVar1 = 1
modVar2 = 2

def fn1(x):
    locA = x
    modVar1 = x
    print dir()
    print "locA =", locA, "modVar1 =", modVar1


def fn2(y):
    locA = y
    print "locA =", locA, "modVar1 =", modVar1
```

```
# File myModuleB:  This is another example module.

# Module variables -- executed first time the module is imported
modVar1 = 1
modVar2 = 2

def fn1(x):
    locA = x
    modVar1 = x
    print dir()
    print "locA =", locA, "modVar1 =", modVar1


def fn3(y):
    locA = y
    print "locA =", locA, "modVar1 =", modVar1
```

Predict the output of the following commands:

```
print dir()
import myModuleA
myModuleA.fn1(5)
print dir()
print dir(myModuleA)
import myModuleB
print dir(myModuleB)
from myModuleB import fn3
print dir()
```