Lab 4: Functional Programming

Assignment

In this lab, you are a tester trying to verify that an Auction code adheres to the specification. Each time you submit your tests to Github, a Github Action will run your tests against **8 different Auction** classes. Only one of them is correct; the others all have at least one defect.

1. Create a unit test file named **`test_auction.py`** to test that the Auction class code obeys the rules of the auction. You should verify the auction *behavior,* not just *auction methods*. Details of how each method behaves are in the method docstring comments.
2. Test the *specification*, not the actual implementation. And **do not** use attributes of Auction in your tests (attributes are part of implementation). Tests should use only the auction methods.
3. Use descriptive test method names such as test_bid_when_auction_is_stopped.
   Not a descriptive name: test_borderline
4. Please do **not** write tests for obvious type errors such as auction.bid(None, "1.5") where amount is not a number and name not a string.

Part 1 - Complete Functions in `funclib.py` [Push work by 13:00 today]

Implement the functions in funclib.py. The identity function is a normal function; the others are all higher-order functions that return a new function.

identity(*args)	The identify function. This is not a higher order function. The only tricky part is to return the args correctly even if there is only 1 value or no values. f = identity means f(x) -> x, f(x,y) -> (x,y)
apply(*f*, iterable)	Apply a function *f* of one variable to each element in a list or *Iterable* object, and return the results as a list.
compose(f, g)	Returns the composite of f and g, such that compose(*f, g*)(*args*) = *f*(*g*(*args*)). Don't assume that *g* has only one parameter (should work for functions that have multiple parameters).
inverse(f)	Returns the reciprocal of a function as a new function, i.e. 1/*f*
product(f, g)	Returns the product of two functions of one parameter. h = product(f, g) means h(x) = f(x) * g(x).
polynomial(c1, c2, ...)	Returns a polynomial of one variable (*x*) with coefficients *c1, c2, ...* where *c1* is the leading coefficient. It should return a function even if there is zero or one coefficient. See docstring for details. $polynomial(a, b, c)(x) \rightarrow (ax + b)x + c$

An object or collection is ***iterable*** if you can create an *iterator* to sequentially access each element in the object or collection.. In actual use, *iterable* and *iterator* are used as the target of a "for" statement:

```
    for x in iterable:
        print(x)
```
or
```
    [print(x) for x in iterable]
```

Collections like list, set, and dict are all "Iterable"; range(start, end) is also Iterable. This is why they can all be used as targets of "for", even though they have different methods for getting their individual elements.

Part 2 - Use funclib to construct novel functions

Using only the functions in funclib.py and Python builtin functions (including the math module) define the following functions. Write your code in the file `appfun.py`
You should need only **one** Python statement to define each function.

1. Define *f* such that $f(x) = x^4 + 2x^3 + 6x^2 - 10$
2. Define *g* such that $g(x) = (\frac{1}{x^2+1})^2 + 2(\frac{1}{x^2+1}) - 1$
3. Define *h* such that $h(x) = x\sqrt{|x| + 1}$

Part 3 - Doctest & Unit tests

1. Create a file named `test_funclib.py`. Write at le`assertAlmostEqual(expected, actual, places=`ast one **unit test** for each function defined in Part 1. In your tests, if you need to compare inexact values (e.g. sqrt(2)) use `n)`
2. Write doctests for the 3 functions in Part 2 in the function named `doc` in `appfun.py`. You should test 3 values of x for each function, using values that you can *manually* verify are correct. x = 0 is an obvious choice. (It would be stupid to simply evaluate f, g, h at some values and copy the results into the doctest. If the function is wrong, the doctest would be wrong, too.)

Examples

```
>>> p = polynomial(1, 5, 4)
>>> p(0)
4
>>> p(10)

>>> from math import sqrt
>>> compose(sqrt, max)(10, 25, 18, -1)

>>> ip = inverse(p)
>>> ip(0)
0.25
>>> ip(1)
0.1
>>> prod = product(max, min)
>>> prod(3, 2, 10, 7, 5)      # 10*2
20
>>> square = polynomial(2, 0, 0)
>>> sum_of = compose(sum, apply)
>>> sum_of(square, [1,2,3,4])        # this may be hard to get to work
```

```
sum(apply(square, [1,2,3,4]) ->
sum([square(1),square(2),square(3),square(4)])
```