CS 463 - Homework 1, Python Code

deadline:
5:00pm, Monday January 31st.

Notes

- New to python? Here is a Python Quick Primer that should help you get up to speed.
- **Python Version**: we are using Python 3. Oddly enough, it is **not** backwards compatible with Python 2.7! You can use the `python3` command directly on `zeus` if you don't have Python installed on your own machine, though that is also relatively straightforward. (Windows users may need to append to their `PATH`, and the command might just be `py` depending on how you install it).
- This file tester1h.py lets you test your code. put it in the same directory, and from that directory run this command:

```
python3 tester1h.py yourfile.py
```

- In this assignment, you will write out some basic functions into a .py file named `netID_h1.py` (use your own id, e.g. mine would be `msnyde14_h1.py`).
- To turn in your work, submit the single file to BlackBoard under the appropriate submission.
- This is supposed to be the easiest language of the semester in which we'll implement these functions - figure out your imperative style algorithm now, because we're going to be implementing most or all of these functions in multiple languages. Calling some built-in function right now won't help at all when we get to some more bizzare language in the future! In general, as long as you don't import anything, and don't phone-in the whole solution, we are not going to obsess over what modules and built-in definitions you use. When in doubt, just ask on piazza with enough time to get a response.
- this tester allows you to run all tests, or only tests for one function at a time.
  - running all tests:

```
python3 tester1h.py yourcode.py
```

  - running only tests for some functions: just name them after your source file.

```
python3 tester1h.py yourcode.py   prime_factors coprime trib
```

- **What Can I Use??** The point of these assignments is to experience implementing an algorithm in various languages.
  - **you must not import anything** unless expressly allowed in the homework specs
  - if you find that there's a built in function that does exactly the same thing (perhaps by a different name), you can't use it. (For instance, though python has a `reversed` function, you must not call it or other functionality that does the entire task for you). If it feels like you're dodging the assignment's task, that's where you won't get points. When in doubt you can ask in a private piazza post, but of course leave enough time for us to respond. More to the point, use these assignments as chances to explore the languages themselves, don't view them as roadblocks to reaching the end of the semester. These homeworks will be good practice for the quizzes and tests.

Functions

`prime_factors(n)`

given a positive integer `n`, return a list of its prime factors, in increasing order. Include the correct number of occurrences, e.g. `prime_factors(24)` is `[2,2,2,3]`.

```
>>> prime_factors(5)
[5]
>>> prime_factors(50)
[2, 5, 5]
>>> prime_factors(66)
[2, 3, 11]
>>> prime_factors(463)
[463]
>>> prime_factors(512)
[2, 2, 2, 2, 2, 2, 2, 2, 2]
```

`coprime(a,b)`

Given two positive integers `a` and `b`, return whether they are co-prime or not. Coprime numbers don't share any divisors other than 1.

```
>>> coprime(60,340)
False
>>> coprime(7,11)
True
>>> coprime(10,21)
True
>>> coprime(50,100)
False
>>> coprime(367,463)
True
>>> coprime(330,463)
True
>>> coprime(222,262)
False
```

`trib(n)`

Given a non-negative integer n, calculate the nth tribonnaci number (where values are the sums of the previous three items in the sequence, or 1 when there aren't enough values ahead of it). For our purposes, the sequence begins as `1,1,1,...`. You need to implement a fast version; if your code takes some exponential amount of time to complete, you will not get credit for this function.

```
>>> trib(0)
1
>>> trib(1)
1
>>> trib(2)
1
>>> trib(3)
3
>>> trib(4)
5
```

```
>>> trib(5)
9
>>> trib(6)
17
>>> list(map(trib, range(20)))
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 4063, 7473, 13745, 25281, 46499]
>>>
```

`max_two(xs)`

Given a list of integers `xs`, find the two largest values and return them in a list (largest first). Largest duplicates should be preserved(included) in the output. Do not modify the original list.

```
>>> max_two([1,2,3,4,5,1,2])
[5, 4]
>>> max_two([3,6,1,2,6,4])
[6, 6]
>>> max_two([-3,-4,-5,-2,-10])
[-2, -3]
```

`reversed(xs)`

Given a list of values, create the list whose values are in the opposite order. Do not modify the original list.

```
>>> reversed([1,2,3,4,5])
[5, 4, 3, 2, 1]
>>> reversed([True, "poly", [1,2,3],{"five":5, "three":3},0])
[0, {"five":5, "three":3}, [1, 2, 3], "poly", True]
>>> reversed([4])
[4]
```

`clockwise(grid)`

Given a list of lists, when it is rectangular (all rows have same length), create and return the list of lists that contains the same values, but rotated clockwise. If this 2D list is not rectangular, `raise` a `ValueError` with the message "`not rectangular`". Do not modify the original 2D list.

```
>>> clockwise([[1,2,3],[4,5,6]])
[[4, 1], [5, 2], [6, 3]]
>>> clockwise([[1,2,3,4,5]])
[[1], [2], [3], [4], [5]]
>>> clockwise([[1], [2], [3], [4], [5]])
[[5, 4, 3, 2, 1]]
>>> clockwise([[1,2,3],[1,2,3,4,5,6]])
...Traceback: ValueError: "not rectangular"
```

`any(xs)`

Given a list of bool values, return `True` if any of them is `True`, and return `False` if every single one of them is `False`. Do not modify the original list.

```
>>> any([False, False, False])
False
```

```
>>> any([True, False, True, True])
True
>>> any([])
False
```

`select(p, xs)`

Given a predicate function `p` (which accepts a single argument and returns a boolean), as well as a list of values `xs`, create a list of all items from `xs` that pass predicate function `p`. Preserve ordering in your output, and do not modify the original list.

- note: the tester defines even() and odd() functions, but they are not built-in functions.

```
>>> def even(n): return n%2==0
...
>>> select(even, [1,2,3,4,5])
[2, 4]
>>> select(lambda x: x%2==1, [1,2,3,4,5])  # fancy way to say "odd"
[1,3,5]
>>> def window(x): return 5 < x < 10
...
>>> select(window, [1,2,3,4,5,6,7,8,9,10,11,12])
[6, 7, 8, 9]
```

`zip_with(f, xs, ys)`

Given a two-argument function and two lists of arguments to supply, create a list of the results of applying the function to each same-indexed pair of arguments from the two lists. (Zip up the two lists with the function). The answer is as long as the shortest of the two input lists. Do not modify the argument lists.

```
>>> def add(x,y): return x+y  # just as an example value for the f parameter
...
>>> zip_with(add, [1,2,3], [10,20,30])
[11, 22, 33]
>>> zip_with(add, [1,2,3], [10,20,30,40,50,60])
[11, 22, 33]
>>> zip_with(add, [1,2,3], [])
[]
>>> zip_with(add, [], [1,2,3])
[]
```

`augdentity(r,c)`

Given positive ints `r` and `c` indicating number of rows and columns, create a 2D list that represents the "augmented identity matrix" with that dimension: It's the `k x k` identity matrix (where `k = min(r,c)`), and augmented rightwards or downwards as needed with zeroes in order to be of size `r x c`. Stated another way, it's an `r x c` matrix filled with zeroes that has ones along its main diagonal.

```
>>> augdentity(3,3)
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> augdentity(3,5)
[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0]]
>>> augdentity(5,3)
[[1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0], [0, 0, 0]]
```

```
>>> augdentity(2,2)
[[1, 0], [0, 1]]
```