Assignment 2, Due 4/19

Do (at least?) five problems from the first section, and both problems from the second section.

Smaller programs

1. Something nice and easy to get us started: let's write a function `fibonacci_list` that takes an integer `n` as input, and returns a list containing the first `n` Fibonacci numbers.

2. Write a function called `flatten`, which takes a list and "flattens" it — that is, removes any internal lists, while keeping the elements in the same order. This is best explained by example:

```
>>> flatten([1,2,3])
[1, 2, 3]
>>> flatten(['house', [['boat'], 3]])
['house', 'boat', 3]
>>> flatten([[1,2], [[[3]], 4, [[5, 6]]]])
[1, 2, 3, 4, 5, 6]
```

3. Write a function `deep_substitute` that takes as input a list `ls` and two non-lists `old` and `new`, and walks the list `ls`, replacing each occurrence of `old` with `new`, including in any sublists occurring in `ls` (or sublists occurring in those sublists, etc). For instance:

```
>>> ls = [['foo', 3], 5, [[['bar']]]]
>>> deep_substitute(ls, 'foo', 2)
[[2, 3], 5, [[['bar']]]]
>>> deep_substitute(ls, 'spam', 'eggs')
[['foo', 3], 5, [[['bar']]]]
```

4. Write a function `get_name` that converts strings of the form

```
Lastname, Firstname Middlename(s) emailaddress
```

into valid email addresses, i.e. something of the form

```
"Firstname Middlename(s) Lastname" <emailaddress>
```

5. Write a function `fixed_point_free_permutations` that takes two inputs `n` and `k`, and returns a list of `k` lists, satisfying the following:

   - each list is a permutation of the numbers `0` through `n-1` (represented as just a list)
   - for each permutation, no element is sent to itself (so `ls[i] != i` for all `i` and for all permutations `ls` in the result)
   - no entry is mapped to the same thing by two different permutations in the result. (You'll probably find the python library module `random` useful for this problem.)

   Again, here's an example:

```
>>> fixed_point_free_permutations(5,2)
[[1,2,3,4,0], [2,3,4,0,1]]
```

whereas this would be invalid:

```
>>> fixed_point_free_fail(5,2)
[[1,2,3,4,0],[1,3,4,0,2]]
```

since 0 is mapped to 1 by both of the resulting permutations.

6. Write a function sigma which takes a positive integer n as input and returns the sum of the proper divisors of n (where “proper” just means ”don’t include n itself). So:

```
>>> sigma(6)
6
>>> sigma(28)
28
>>> sigma(496)
496
```

   You should be able to do this in one line with a list comprehension — don’t worry about making it fast, just do something naive and easy to implement.

7. Write a function prod which takes a list of integers as input and returns the product of them. You should be able to do this in one line with map/reduce/filter. Using this, implement the “double factorial” — that is, the function that inputs a positive integer n and returns the product of all the **odd** integers less than or equal to n.

Longer Problems

8. It’s often nice to print out a readable table of values of a function quickly. Write a function print_table that takes as input a function f and a list ls of inputs, and prints out a nicely formatted table of output values. Here’s an example:

```
>>> print_table(fact, range(3))
+---+--------+
| n | output |
+---+--------+
| 0 |      1 |
| 1 |      1 |
| 2 |      2 |
+---+--------+
>>> print_table(fact, [3, 5, 11])
+----+----------+
|  n |   output |
+----+----------+
|  3 |        6 |
|  5 |      120 |
| 11 | 39916800 |
+----+----------+
```

   In particular, here are the requirements:

   - the inputs and outputs should be right-justified, as should the labels n and output

- each side should be wide enough to accomodate all the entries
- there should be at least one space between any entry and the nearest `|`
- you should make some pretty ASCII art for the border — it doesn't have to match this exactly, but it should look nice. `:)`

9. Write a function called `copy_and_flatten_directory` that takes as input two directory names. It then copies all files in all subfolders of the first directory into the second subdirectory, ignoring the directory structure completely. (That is, it flattens it.) So, for instance, if the first folder had these files

   ```
   mydir/foo/bar/baz.txt
   mydir/foo/bar/pie.pdf
   mydir/foo/bar/stuff.txt
   mydir/myfile.py
   mydir/other_stuff.ss
   ```

   then, after this function is run, the second folder should contain

   ```
   baz.txt
   myfile.py
   other_stuff.ss
   pie.pdf
   stuff.txt
   ```