Lab 4: Functional Programming

Part 0: Submission/Reference Materials

The amount of explanatory material in the labs will begin to decrease from this lab onwards. This is because a key skill in programming is reading (and writing!) documentation and finding examples. Finding and using appropriate/accurate reference materials is hard to teach directly but tends to be what separates bad programmers from good ones!

If you get stuck:

- Review the lecture material
- Check the docstrings of functions you are trying to use (e.g., to check out python's own explanation for how `map` works you can just type `?map` into your notebook or interactive prompt)
- Look at python's official documentation and guides (https://docs.python.org/3/howto/functional.html)
- Use online resources like stackoverflow, w3schools, realpython etc (but make sure you don't blindly copy code without working out HOW it works).
- Looking up materials is totally fine but remember if you copy code (or autocomplete it) directly from any source you MUST cite where you got it from in a comment next to the code.

Submit this assignment as a formatted notebook (as per lab 3's instructions) - make sure every function has a clear docstring that explains what it does, its arguments and what it returns. This will be part of the grading of each of your answers.

Part 1: Args, Kwags, and Returns

Q1 [4 points]. Write a function called "summary_statistics" that takes 0 or more numbers as input variables (as separate arguments not as a list) and returns a dictionary containing the sum, mean, median, minimum, and maximum values of these inputs. Make sure your functions generate the correct output if there are no input arguments.

Example of functionality:

```
stats = summary_statistics(1, 52, 2, 5, 9, 10, 88, 44)
print(stats)
{'sum': 211
```

```
 'mean': 26.375
 'median': 9.5
 'minimum': 1
 'maximum': 88}
```

Q2 [5 points]. Write a function **format_name** that takes in 4 optional keyword args (**first, last, title, suffix**) with appropriate default values and correctly formats the name (i.e., title if present followed by first name if provided followed by last name if provided followed by suffix if provided**.** Provide at least 1 example of you executing this function using a dictionary containing the parameters instead of specifying them using the `first=` syntax. Example of functionality**:**

```
print(format_name(first="John", last="Smith"))
John Smith
print(format_name(title="Dr.", last="Jones"))
Dr. Jones
print(format_name(first="Jane", suffix="PhD"))
Jane PhD
print(format_name())
'' # Empty string
```

Q3 [3 points]. Explain why `results` and `different_results` contain the same experimental data? Rewrite this code to fix this issue.

```
def create_result(experiment_name, measurement, result={}):
    """
    Records experimental measurements for different
    scientific experiments.  Optionally provide a pre-existing
    Results dictionary to append new results to that dictionary instead
    Of creating a new results dictionary.
    Args:
        experiment_name - string containing name of experiment
        measurement - numerical measurement from experiment
        result - optional kwarg to add new measurements to existing
                 experiment
    Returns:
        result - a dictionary containing experimental results
    """
    if experiment_name in result:
        result[experiment_name].append(measurement)
    else:
        result[experiment_name] = [measurement]
    return result

quantum_results = {"tunnelling": [25]}
results = create_result("photoelectric_effect", 1.5)
```

```
quantum_results=create_result("double_slit",0.7, result=quantum_results)
different_results = create_result("pizoelectric_effect", 1.6)
```

Part 2: Comprehensions

Q4 [3 points]. Rewrite the code below into a single list comprehension (this should only require 1 line of code; excluding the initial creation of the separate wavelengths and intensities lists):

```
wavelengths = [480, 520, 560, 600]
intensities = [100, 200, 150, 300]
# rewrite all following as a single list comprehension
normalised_spectra  = []
for wavelength, intensity in zip(wavelengths, intensities):
      if wavelength > 500:
            normalised_wavelength = wavelength / intensity
            normalised_spectra.append(normalised_wavelength)
```

`normalised_spectra` should contain `[2.6, 3.7333333333333334, 2.0]` for these inputs.

Q5 [3 points]. Complete this function using only a dictionary comprehension

```
def process_experiment_data(raw_readings):
   """
   Process experimental readings by dividing each raw_value by /100
   And removing values <= 0


   Args:
       raw_readings: Dict of sample_id: raw_value pairs

   Returns:
       Dict of processed readings
   """
   pass # replace this

data = {'A1': 150, 'A2': -5, 'A3': 200}
processed_data = process_experiment_data(data)
print(processed_data)
# {'A1': 1.5, 'A3': 2.0}
```

Part 3: Lambda & map

Lambda functions can use many parameters but only a single expression, their syntax works like this:

`lambda PARAMETERS: EXPRESSION`

Q6 [1 point]. Replace the **…** with a lambda function that will sort this list of tuples by the grade in each subject.

```
subject_marks = [('English', 88), ('Science', 90), ('Maths', 97),
('Social sciences', 82)]

print("Original list of tuples:")
print(subject_marks)
print(subject_marks.sort(key=...)
```

Q7 [2 points] Use `map` and a lambda function to create a new list that adds together two equal length lists of numbers (do not use a list comprehension!). For the inputs:

```
x_list = [1, 2, 3, 4, 5]
y_list = [4, 5, 6, 7, 8]
```

The output should be:

```
[5, 7, 9, 11, 13]
```

Part 4: Recursion

Write each of the following functions using recursion. Do not use `for` loops or `while` loops for these problems. Also do not use the `in` syntax, e.g. expressions like `8 in [7,9,10]`. You are however welcome to use `if/elif/else`.)

Remember that in recursive functions, counters don't work because the counter will be different in each recursive call! In general, any variable that you define inside a recursive function will be local only and not seen by any other function - not even other recursive calls to the same function! Therefore, in this assignment, do not create new variables inside your functions! That is, don't have things like counter = blah, blah, blah.

Q8 [5 points]. A dot product (also called a scalar product) is a way to multiply two lists of numbers (of equal length) together to get a single number as a result. It is a regularly used operation in linear algebra and geometry. To calculate it for the lists `a` and `b` you would use the following formula:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

If a and b are both empty then the dot product would be 0. Here's an example of how this calculation works in python:

```
a = [1, 2, 3]
b = [4, 5, 6]
dot_product = a[0] * b[0] + a[1] * b[1] + a[2] * b[2]
dot_product == 1 * 4 + 2 * 5 + 3 * 6
```

Without using a loop (i.e., iteration) write a recursive function `dot(a,b)` that calculates the dot product of 2 lists of numbers of equal length. Remember recursion is about breaking down a problem into a simpler version of itself. To do this you need to work out how to code the recursive case and the base case.

Q9 [6 points]. In the book *Liber Abaci*, Leonardo of Pisa considered pairs of breeding rabbits:

- At the start of month zero there is a single infant pair of rabbits.
- A pair of rabbits matures in 2 months.
- Rabbits do not breed in their first month of life.
- Each mature pair of rabbits produces an infant pair of rabbits on the last day of each month, starting with the second month.

So if we break this down:

- At the end of the first month, our first pair mate, but there is still only 1 pair.
- At the end of the second month they birth a new pair, so now there are 2 pairs of rabbits in the field.
- At the end of the third month, the original pair produces a second pair, making 3 pairs in all in the field.
- At the end of the fourth month, the original pair has produced yet another new pair, the pair two months ago produced their first pair also, making 5 total pairs.

So new pairs: 0, 1, 1, 2, 3, 5

Leonardo asks in this book: How many pairs will there be one year after this pair begins breeding.

If we use `F(j)` to denote the number of rabbit pairs in month (`j`) we can simplify Leonard's descriptions:

- `F(0) = 0`
- `F(1) = 1`
- When j ≥ 2 `F[j] = F[j - 1] + F[j - 2]`

Write a recursive function called `breeding_rabbits` that takes an argument `j` and returns the numbers of rabbits in month `j`.

Test this function with j=1, j=100, and j=1000

Part 5: Generators

Q10 [3 points]. Write a generator function that takes an input `n` and `yield`s the cubed values between 1 and n (including n).

For example:

```
n = 5
cubes = cube_generator(n)
for num in cubes:
    print(num)
1
8
27
64
125
```