Homework 4: Higher-order Functions

Due: 09/26/2025 11:59 PM ET

This homework primarily covers defining and using higher-order functions.

Goals

In this homework you will:

1. Write higher order functions that accept one or more functions as an input.
2. Write higher order functions that return a function as output.
3. Develop a basic signal processing algorithm (convolution).

Background and Instructions

0) Download the homework

Download the files for HW 4 from Brightspace.

The folder should contain 3 files:

1. This README and its PDF.
2. `hw4.py`, a skeleton script for Problems 1 and 2.
3. `given_tests.py`, test functions to help you verify your implementation.

1) Problem 1: Higher-order functions

Problem 1 will get you familiar with several ways to build and use higher order functions. In this problem, we ask you to fill in the code for a number of missing functions in `hw4.py`:

1. `conditional_map`: This takes in a predicate function, two transformation functions (`if_func` and `else_func`), and a list `L` as its arguments. It returns a new list where each element is processed by `if_func` if the predicate returns `True` for that element, and by `else_func` if the predicate returns `False`. This is a fundamental pattern in functional programming for conditional data transformation.

2. `compose_map`: This takes in two functions, `func_1` and `func_2`, and a list `L` as its arguments and returns a list, which consists of applying `func_1` then `func_2` to each element of the input list `L`. Note that you must apply the functions in that order: call `func_1` first, then `func_2` on the output of `func_1` for each element in the input list `L`.

3. `compose`: This takes in two functions, `func_1` and `func_2`, and returns a new function `ret_fun`. The new function takes in a single argument `i` then calls `func_2` followed by `func_1` on that argument, returning the

result. Note again that the order matters. *Note: Expected return type for* `compose` *is a function and not a value.*

4. `repeater`: This takes in a list of functions `funlist` and a list of integers `num_repeats`, and returns a new function, `ret_fun`. The new function takes an input `x` and calls the first function in `funlist` repeated a number of times equal to the first number in the list `num_repeats`, and then calls the second function in `funlist` repeated a number of times equal to the second number in the list `num_repeats`, continuing this pattern until the end of `funlist` is reached. *Expected return type for* `repeater` *is a function and not a value.*

Testing

To test your implementation, run the provided test file:

```
python given_tests.py
```

If your code works correctly, you should get the following output:

```
[6, 12, 18, 24, 30, 30, -4, -7, -8]
[5, 11, 17, 23, 29, 29, -10, -19, -22]
[3, 9, 15, 21, 27, 27, -12, -21, -24]
9
[3, 9, 15, 21, 27, 27, -12, -21, -24]
11
[5, 11, 17, 23, 29, 29, -10, -19, -22]
repeat 0 times: 5
repeat 1 time: 14
repeat 2 times: 43
repeat 3 times: 132
```

2) Problem 2: Stencils

In Problem 2, we ask you to write two functions. The first is a generic *stencil* function, and the second is a function that generates *box filters* that, when combined, will let you perform a specific operation known as convolution (Explained later in the section).

Stencils

A stencil (in the most general sense) is a way of computing a value for a given data point by using the values of neighboring data points. Suppose we have a *stencil function* `f`. `f` is a function that accepts a list of some fixed length (for the purposes of explanation, let's say 3) and outputs a single value that is some combination of the values in that list. Here is one example:

```
def f(L) :
    return L[0] * L[1] + L[2]
```

Applying a stencil function means treating the stencil function like a "window" and "sliding" it across a list, computing the result of the stencil function at each point. So if we start with the following list:

```
[a, b, c, d, e, f, g]
```

Applying the stencil function produces the following output list:

```
[f([a, b, c]), f([b, c, d]), f([c, d, e]), f([d, e, f]), f([e, f, g])]
```

> Note two things: 1. The length of the output list is smaller than the length of the input list. If the input list is of length `k`, the output list will be of length `k - width + 1`, where `width` is the width of the stencil that `f` applies. 2. You can think of a stencil as a generalization of map that uses neighboring elements to compute the output instead of just the element itself. If `f` has a window of width 1, the output is basically like using map function.

We will ask you to write a function for applying stencils that takes in an input list, a stencil function, and a third parameter that tells you how wide the stencil function is, in the case shown above, the stencil function uses a width of 3 to slide over the list `[a, b, c, d, e, f, g]`.

Filter, Cross-correlation and Convolution

Given a filter of length `k` that specifies a list of `k` numbers (for a 1-D box filter), when applied to a input signal, computes its result by multiplying the first element of the filter by the first element of the input, the second element of the filter by the second element of the input, and so on, and then adding all the results together to output one single value.

In the signal processing literature, a filter is a signal that when applied to a target signal, removes some unwanted components or features from the target signal.

So if you have a filter (list), say `b`, of size 3 with elements `b[0], b[1], b[2]` and you slide it over the list `[a, b, c, d, e, f, g]`, the expected output would be:

```
[b[0] * a + b[1] * b + b[2] * c,
 b[0] * b + b[1] * c + b[2] * d,
 b[0] * c + b[1] * d + b[2] * e,
 b[0] * d + b[1] * e + b[2] * f,
 b[0] * e + b[1] * f + b[2] * g]
```

In one dimension (1-D), applying a filter in this way (shown above) computes the (discrete) cross-correlation of the signal in the input list with the filter. This is a measure of similarity between the signal and the filter.

Additionally, the convolution operation in 1-D is very similar to the cross-correlation operation but has a slight variation. In the convolution operation,

the filter is first flipped and then applied to the input signal. In other words, convolution is simply a cross-correlation on the input signal with a flipped version of the given filter.

So, if you have a filter with elements `b[0], b[1], b[2]` and you perform a convolution operation on the list `[a, b, c, d, e, f, g]`, you are basically sliding over the list with a flipped version of the filter, hence, the output you get would be:

```
[b[2] * a + b[1] * b + b[0] * c,
 b[2] * b + b[1] * c + b[0] * d,
 b[2] * c + b[1] * d + b[0] * e,
 b[2] * d + b[1] * e + b[0] * f,
 b[2] * e + b[1] * f + b[0] * g]
```

Convolution conveys how the shape of signal is modified by the filter.

> In two dimensions, filters can be used to perform image-processing tasks like blurring and edge detection. These types of filters are also one of the key steps in deep neural networks that operate on images (they form the core of the "convolution" layers).

In the second part of Problem 2, we ask you to write a function that, when given a list of filter elements, *creates a stencil function* from those elements.

Finally, coming to the point, we ask you to fill in the last two functions in `hw4.py`:

1. `stencil(data, f, width)`: This function takes in a list `data`, a function `f`, and an int `width`, and returns a list. This function returns an output list of length `k-width+1`, the result of applying the stencil function `f` to the input list `data`

2. `create_box(box)`: This function accepts a list `box`, and returns two outputs: a new function and a width (int). The width is the length of the box (the number of elements that the filter looks at). The new function is a stencil function that operates on items from an input list and applies a convolution operation to them as described above. The function should check, at the beginning, if the length in the input parameter is the same as the length of `box`. If not, the following error should be printed:

   ```
   Calling box filter with the wrong length list. Expected
   length of list should be n.
   ```

   where `n` is the length of `box`. For example if `n=7`, the printed warning should be:

   ```
   Calling box filter with the wrong length list. Expected
   length of list should be 7.
   ```

   and then after the print statement, return 0.

Moreover, the comments under `create_box()` in hw4.py provide further explanation on what `create_box(box)` is supposed to do.

`pass` statement in the function definitions

The `pass` statement is used as a placeholder for future code (as done by us). When the `pass` statement is executed, nothing happens, but you avoid getting an error when you have empty code. Empty code is not allowed in loops, function definitions, class definitions, or in if statements. You should delete/comment out the `pass` statement of a function once you write your code for that particular function.

Testing

The Problem 2 functions are also tested by running `python given_tests.py`. You should see additional output for Problem 2 after the Problem 1 results:

```
[2.0, 3.3333, 4.6667, 6.0, 6.6667, 2.3333, -3.0, -4.3333]
[220, 316, 309, 309, 294]
[5.0, 7.0, 8.5, 6.25, 2.75, -1.5]
[-1.8, -1.8, -1.4, 2.2, 7.1, 4.3]
```

Helpful documentation

Python slicing https://www.geeksforgeeks.org/python-list-slicing/

List comprehension https://www.w3schools.com/python/python_lists_comprehension.asp

What to Submit

Please submit ONLY `hw4.py` with all the appropriate functions filled in for Problem 1 and Problem 2.

Before submitting, make sure to test your code by running:

```
python given_tests.py
```

This will help you verify that your functions work correctly and that your file imports properly.

Submitting your code

Please submit the latest version of your code to Gradescope. Do not change the submission file name or the function names within the handout.