Higher-Order Functions

A higher order function (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

Functions as Arguments

One way a higher order function can manipulate other functions is by taking functions as input (an argument). Consider this higher order function called `negate`.

`negate` takes in a function `f` and a number `x`. It doesn’t care what exactly `f` does, as long as `f` is a function, takes in a number and returns a number. Its job is simple: call `f` on `x` and return the negation of that value.

Questions

1.1 Here are some possible functions that can be passed through as `f`.

```python
def square(n):
    return n * n

def double(n):
    return 2 * n
```

What will the following Python statements display?

```python
>>> negate(square, 5)
-25

>>> negate(double, -19)
38

>>> negate(double, negate(square, -4))
32
```

Functions as Return Values

Often, we will need to write a function that returns another function. One way to do this is to define a function inside of a function:

```python
def outer(x):
    def inner(y):
        ...
        return inner
```
The return value of outer is the function inner. This is a case of a function returning a function. In this example, inner is defined inside of outer. Although this is a common pattern, we can also define inner outside of outer and still use the same return statement. However, note that in this second example (unlike the first example), inner doesn’t have access to variables defined within the outer function, like x.

Questions

1.1 Use this definition of outer to fill in what Python would display when the following lines are evaluated.

```python
def outer(n):
    def inner(m):
        return n - m
    return inner

>>> outer(61)

<function outer.inner ...>

>>> f = outer(10)
>>> f(4)

6

>>> outer(5)(4)

1
```

1.2 Write a function and_add that takes a one-argument function f and a number n as arguments. It should return a function that takes one argument, and does the same thing as the function f, except also adds n to the result.

```python
def and_add(f, n):
    """Return a new function. This new function takes an argument x and returns f(x) + n."

    >>> def square(x):
    ...        return x * x
    ...    >>> new_square = and_add(square, 3)
    ...    >>> new_square(4)  # 4 * 4 + 3
    19
    """

    def g(x):
        return f(x) + n
    return g
```
Environment Diagrams

1.1 Draw the environment diagram for the following code:

```python
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
five = add_three(2)
```
1.2 Draw the environment diagram that results from running the following code:

```python
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y())(f)
```

![Environment Diagram](image.png)
The following question is extremely difficult. Something like this would not appear on the exam. Nonetheless, it’s a fun problem to try.

Draw the environment diagram for the following code: (Note that using the + operator with two strings results in the second string being appended to the first. For example "C" + "S" concatenates the two strings into one string "CS")

```python
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```