Discussion 12: Procedures as Data

Lambda Functions

1. Write a lambda function called `f` that takes in a number and outputs that number squared.

`f` = lambda n: n ** 2

2. Now, use a list comprehension and your lambda function `f` to return a list the squares of all numbers between 1-5, inclusive.

[f(n) for n in range(1, 6)]

Functions as Data

What would the Python interpreter display for the following lines of code? If you believe a line errors, just write "Error." **Each subproblem is independent and does not depend on the other subproblems.**

```
>>> lst = [1, 2, [3, 4]]
>>> lst[2].pop()
>>> lst
```

```
[1, 2, [3]]
```

```
>>> [x * 2 for x in range(4) if x % 2 == 1]
```

```
[2, 6]
```

```
>>> "".join([word[0] for word in "Univ of Calif at Berkeley".split()
... if not(len(word) == 2)])
```

```
'UCB'
```

```
>>> "".join([word[0] for word in "Univ of Calif at Berkeley"
... if not(len(word) == 2)])
```

```
'Univ of Calif at Berkeley'
```

```
>>> f1 = lambda x: x + x
>>> f2 = lambda x: x > 9
>>> [f(10) for f in [f1, f2]]
```

```
[20, True]
```

```
>>> f = lambda x: lambda: x + x
>>> f(2)
```

```
<function lambda ... >
```

```
>>> y = 3
>>> f = lambda x: lambda: x + y
>>> f(2)()
5
>>> g = lambda y: x + y
>>> g(2)
Error (x is not defined)
```

2. Now, continue the exercise, instead **assuming that each subproblem is a continuation of the previous subproblems.**

```
>>> def make_adder(x):
...     def inner(y):
...         return x + y
...     return inner
>>> make_adder(5)
<function make_adder ... >
>>> make_adder(5)(6)
11
>>> functions = [lambda x: x, lambda x: x * x, lambda x: x * 3]
>>> functions[2](3)
9
>>> def returnMax():
...     return max
... returnMax()
<built-in function max>
>>> returnMax()(2, 3)
3
>>> max = min
>>> max(5, 4)
4
>>> returnMax()
<built-in function min>
returnMax()(2, 3)
2
```

3. Write a function called `functionList` that takes in a list of functions, `functions`, and a number, `n`, and returns a list of the results of calling each function on `n`.

```
>>> functionList([lambda x: x + x, lambda x: x * x], 4)
[8, 16]
```

```
def functionList(functions, n):
    lst = []
    for function in functions:
        lst.append(function(n))
    return lst
```

3. Write a recursive function called `recursiveSum` that takes in a function `func` and a number `n`, and returns the summed results of `func` applied from 1 to `n`.

```
>>> recursiveSum(lambda x: x * x, 3)
14 # 3*3 + 2*2 + 1*1
```

```
def recursiveSum(func, n):
	if n == 0:
		return 0
	else:
		return func(n) + recursiveSum(func, n - 1)
```

Tree Recursion in Python

1. The Fibonacci sequence is a sequence of numbers where each number is the sum of the previous two. Here is the start of the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, …

In the space below, write the function `fib(n)` that returns the nth Fibonacci number in the sequence, assuming the first one is n = 0.

```
def fib(n):
	if n == 0:
		return 0
	elif n == 1:
		return 1
	else:
		return fib(n - 1) + fib(n - 2)
```

What is the runtime of this function? exponential

2. We find ourselves at the bottom of a staircase with `num_steps` steps. We can either climb the stairs one at a time or two at a time (or a mix of the two). Fill in the function below to return the number of ways you can climb the staircase.

```
def climb_staircase(num_steps):
      if num_steps == 0:
            return 1
      elif num_steps < 0:
            return 0
      else:
            return climb_staircase(num_steps - 1) + climb_staircase(num_steps - 2)
```

3. Now, when we are climbing the staircase, we can take any from 1 to `max_steps` number of steps at a time (not just 1 or 2). Fill in the blanks below to write rewrite `climb_staircase` to return the number of ways you can now climb the staircase.

```
def climb_staircase(num_steps):
      if num_steps == 0:
            return 1
      elif num_steps < 0:
            return 0
      Else:
            return sum([climb_staircase(num_steps - i, max_steps) for i in
            range(1, max_steps + 1)])
```