Lecture 8

Recursion, Higher Order Functions

Announcements

- Thursday’s lecture will be a review of all the material that we have covered so far
- There is going to be a “final” project that will be assigned soon which will be much longer and you will be given approximately a week and a half to complete it. This project will incorporate everything that we have learned so far into one comprehensive project.

ICS 32 Contents

ICS 32 is simply a review of 31 for the first half of the class. The only new topics taught in 32 are:

- Recursion
- Software libs
- Classes

We won’t be going over software libs due to the variation in picking which one to use and the time it takes to apply and get API keys, and I have no educators pass.

Recursion

Recursion is the idea of calling a function **within** itself on a smaller version of it's parameters until it reaches a **base case** in which it will return a final value, ending the recursion call.

If you can do it with a loop you can do it with recursion.

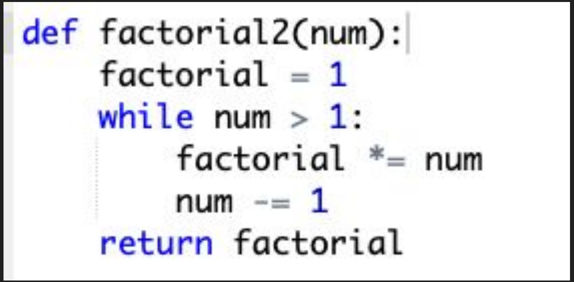
Most of the time recursion simplifies a problem.

We use recursion because it is extremely fast in comparison to iteration.

Recursion will most likely use more memory than iteration but when used correctly/optimally the difference can be negligible in most cases.

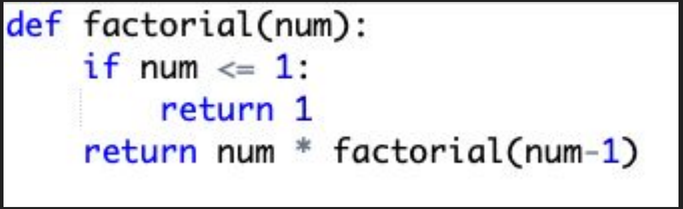
Recursion Example: Factorial

Iteration:



```
def factorial2(num):
    factorial = 1
    while num > 1:
        factorial *= num
        num -= 1
    return factorial
```

Recursion:



```
def factorial(num):
    if num <= 1:
        return 1
    return num * factorial(num-1)
```

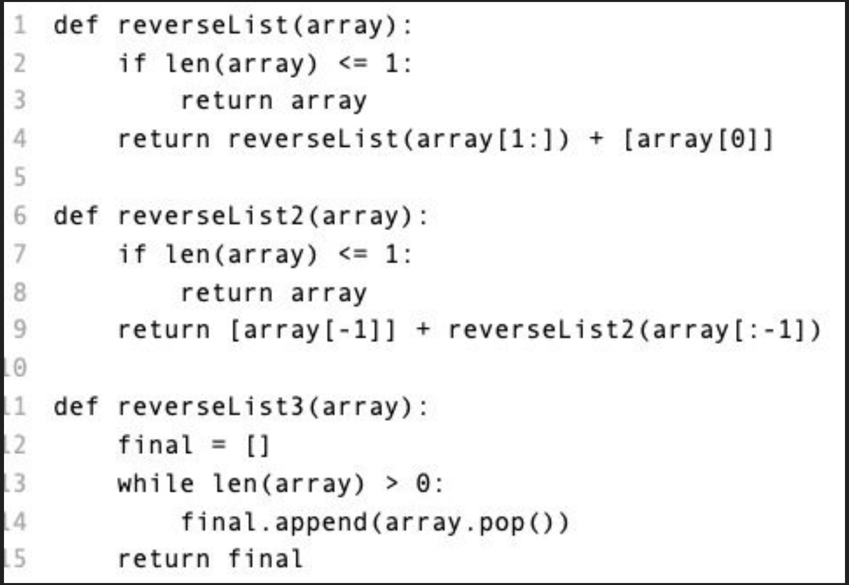
In the recursive version I have:

Base Case: If the number is 0 or 1 it will return 1 (assume non negative inputs)

Recursive Step: If the number is greater than 1, return the number times the returned value of the function call on number - 1.

Eventually after many function calls, number will reach 1 which is the base case and it will return 1.

Recursion Example: Reversing a List



```python
def reverseList(array):
    if len(array) <= 1:
        return array
    return reverseList(array[1:]) + [array[0]]

def reverseList2(array):
    if len(array) <= 1:
        return array
    return [array[-1]] + reverseList2(array[:-1])

def reverseList3(array):
    final = []
    while len(array) > 0:
        final.append(array.pop())
    return final
```

There are many ways of doing the same problem the first version I'm taking the 0th index and appending it to the end, the second version I'm taking the last index and appending it to the front.

The base case is if the length of the list is 1 or 0 then return it list (nothing to reverse) otherwise repeat the step above.

Recursion Note:

Although what we are doing right now is very basic, a recursion problem can get more complex. Just know that it is possible to have more than 1 base case / recursive call.

This can be done by having more conditionals and different kinds of return statements based on the condition

Try it yourself: Recursion

We want to find the length of a string using recursion.

Think about how we would approach this.

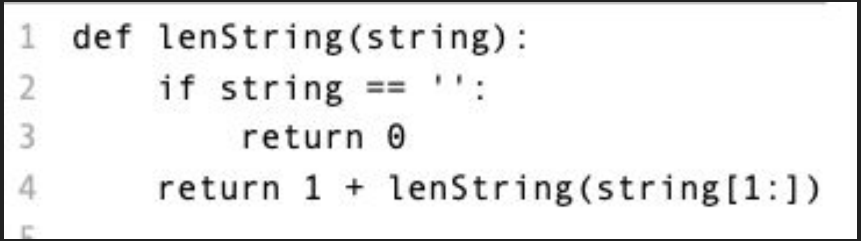
Solution:

Try it yourself: Recursion - Solution

We want to find the length of a string using recursion.

Think about how we would approach this.

Solution:



```python
def lenString(string):
    if string == '':
        return 0
    return 1 + lenString(string[1:])
```

Try it yourself: Recursion

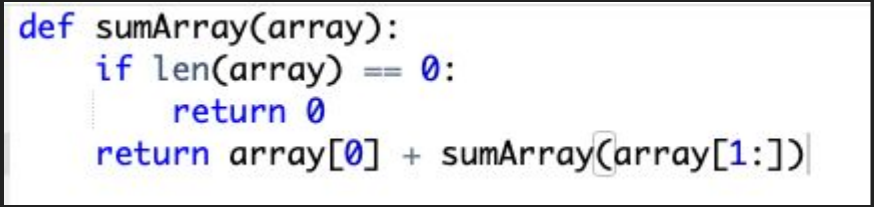
We want to use recursion to find the sum of all ints/floats in an array.

Solution:

Try it yourself: Recursion

We want to use recursion to find the sum of all ints/floats in an array.

Solution:

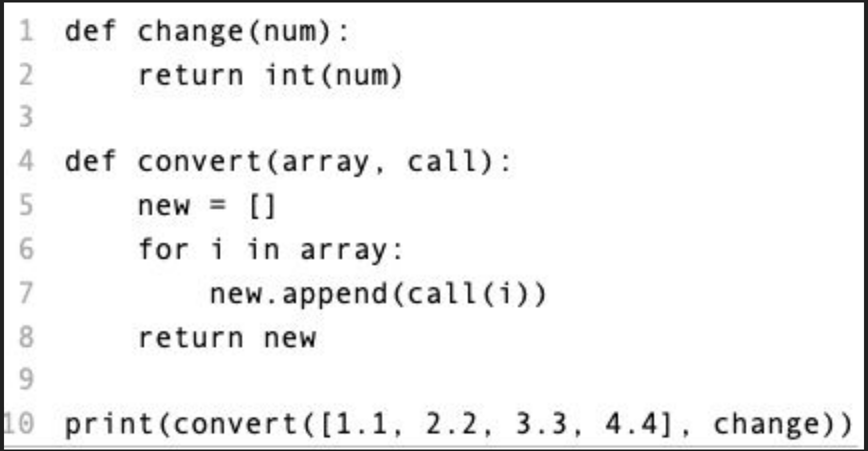


```
def sumArray(array):
    if len(array) == 0:
        return 0
    return array[0] + sumArray(array[1:])
```

Functions as Objects

We won't be getting too deep into this but it's a really powerful concept that has limitless potential.

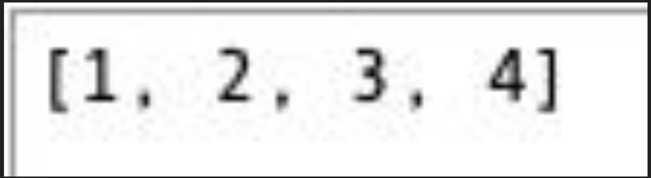
The general idea behind this is that functions are objects like anything else and because of this you can pass them around just like what we have done with other objects so far.



```python
def change(num):
    return int(num)

def convert(array, call):
    new = []
    for i in array:
        new.append(call(i))
    return new

print(convert([1.1, 2.2, 3.3, 4.4], change))
```

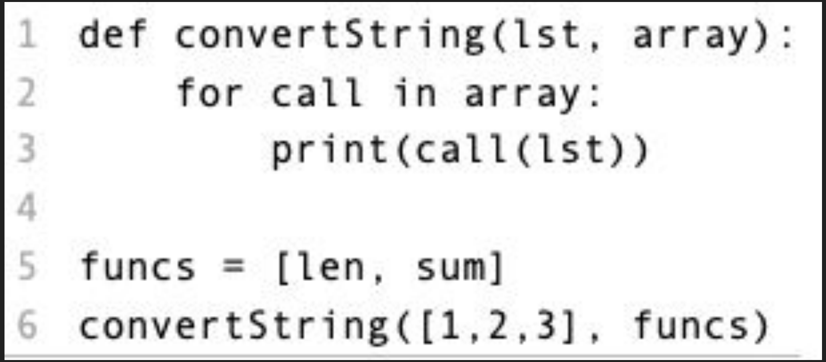


```
[1, 2, 3, 4]
```

You can store them in data structures

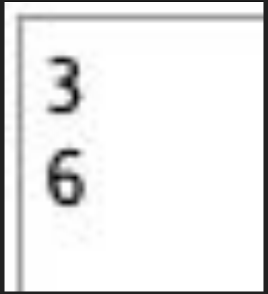
Since functions are objects you can store them in different kinds of data structures and call them from the data structure.

In this example we have



```
def convertString(lst, array):
    for call in array:
        print(call(lst))

funcs = [len, sum]
convertString([1,2,3], funcs)
```

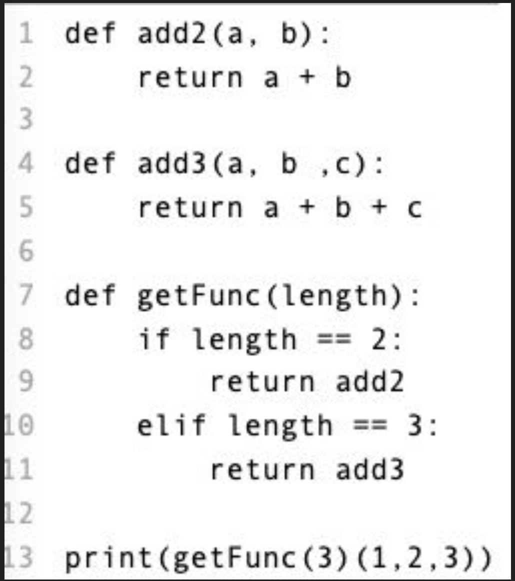


```
3
6
```

Function that returns functions

Functions can also be returned as an object so that it can be called on from the return value.

In this example we have 2 functions that add depending on how many parameters it has and another function that takes in an int as length and returns the appropriate function to be called.

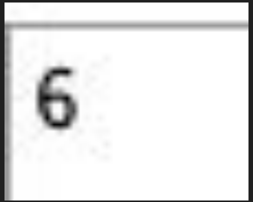


```python
def add2(a, b):
    return a + b

def add3(a, b ,c):
    return a + b + c

def getFunc(length):
    if length == 2:
        return add2
    elif length == 3:
        return add3

print(getFunc(3)(1,2,3))
```



```
6
```