Excercise_3_solutions

May 30, 2019

1 Lecture 3

Calculate a value of a polynomial for a vector of x coordinates using single line list comprehension expression and a generator.

You must apply the enumerate() and sum() functions

1st problem generator

```
In [1]: c = [1,2,3]
        x = [0, 2, 4, 8]
        #squared = [i ** 2 for i in c]
        #squared = list(map(lambda c: c**2, c))

        import matplotlib.pyplot as plt
        c = [1,2,3]
        x = [0, 2, 4, 8]

        def polynomial():
            for i in range(len(x)):
                ans = 0
                for n,a in enumerate(c):
                    ans = sum((a*x[i]**n, ans))
                yield ans

        p = (list(polynomial()))

        ans = 0

        #lambda_function = list(map(lambda ans, c, x: sum((a*x[i]**n, ans))  for i in range(le
        print(p)
```

```
[1, 17, 57, 209]
```

1st problem list comprehension

```
In [2]: c = [1,2,3]
        x = [0, 2, 4, 8]
        p = [ sum([b*(i**a) for a,b in list(enumerate(c))]) for i in x]
        print(p)
```

```
[1, 17, 57, 209]
```

2nd problem
Generate a smooth polynomial plot from Task 1 using dense vector of x coordinates

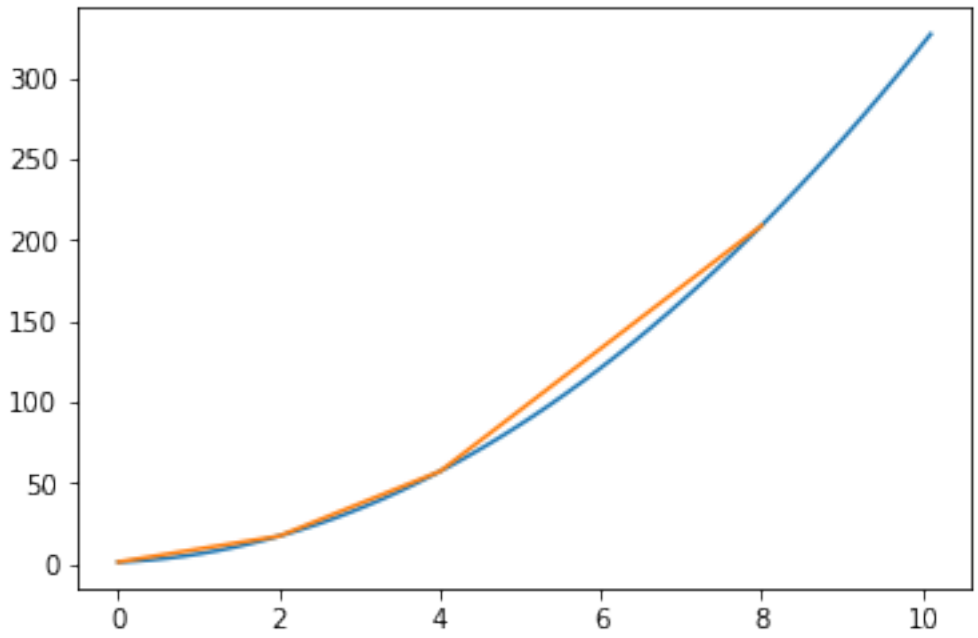
In [3]:
```
# n = 100
# xd = [8*x/n for x in range(n)]

# import matplotlib.pyplot as plt
# x = [8*i/100 for i in range(0,100)]
# y = [x**2 for x in x]
# plt.plot(x,y)

import matplotlib as plt
import numpy as np

c = [1,2,3]
x  = np.linspace(0.,10.1000)
x_old = [0,2,4,8]

plt.pyplot.plot(x,[ sum([b*(i**a) for a,b in list(enumerate(c))]) for i in x])
plt.pyplot.plot(x_old,[ sum([b*(i**a) for a,b in list(enumerate(c))]) for i in x_old])
plt.pyplot.show()
```



3rd

Using a conditiional list comprehension, convert all occurrences of the letter 'e' in the following string to uppercase: thepurposeoflife

You can use ".join(alist) to join a list of characters into a single string.

```
In [4]: upperstr = 'thepurposeoflife'

        upperstr = [x.upper() if x =="e" else x for x in list(upperstr)]
        upperstr = ''.join(upperstr)
        print(upperstr)
```

```
thEpurposEoflifE
```

2 4th

Use the following tuple of tuples records = (('Sam', 19, 'CS'), ('Nicole', 21, 'Biochemistry'), ('Paul', 20, 'Fine Arts'), ('Ashley', 18, 'History')) iterate over the records, unpack them, and print the results This might be useful syntax for printing: print('%s and %d and %s', % ('a string', 10, 'another string')) Make a function as well to complete the following: def showrecords(records): "'Unpack records stored in a tuple of tuples and print each one in a nice format"' ... showrecords(___)

```
In [5]: records = (('Sam', 19, 'CS'),
        ('Nicole', 21, 'Biochemistry'),
        ('Paul', 20, 'Fine Arts'),
        ('Ashley', 18, 'History'))

        records = (('Sam', 19, 'CS'),
        ('Nicole', 21, 'Biochemistry'),
        ('Paul', 20, 'Fine Arts'),
        ('Ashley', 18, 'History'))

        def showrecords(records):
          for record in records:
            name,age,faculty = record
            print(f'{name}\t{age}\t{faculty}')

        showrecords(records)
```

```
Sam         19          CS
Nicole        21          Biochemistry
Paul         20          Fine Arts
Ashley        18          History
```

5th

Make a nested function and a python closure to make functions to get multiple multiplication functions using closures. That is using closures, one could make functions to create multiply_with_5() or multiply_with_4() functions using closures.

```
In [6]: def multiplier_of(n):
            def multiplier(number):
                return number*n
            return multiplier

        multiplywith5 = multiplier_of(5)
        print(multiplywith5(9))

        multiply_with_45 = multiplier_of(multiplywith5(9))
        print(multiply_with_45(2))
```

```
45
90
```

6th

Make a decorator factory which returns a decorator that decorates functions with one argument. The factory should take one argument, a type, and then returns a decorator that makes function should check if the input is the correct type. If it is wrong, it should print("Bad Type") (In reality, it should raise an error, but error raising isn't in this tutorial). Look at the tutorial code and expected output to see what it is if you are confused (I know I would be.) Using isinstance(object, type_of_object) or type(object) might help.

```
In [7]: def type_check(correct_type):
            def check(old_function):
                def wrapped_func(arg):
                  if(isinstance(arg,correct_type)):
                    return old_function(arg)
                  else:
                    return print('Bad type')
                return wrapped_func
            return check

        @type_check(int)
        def times2(num):
            return num*2

        print(times2(2))
        times2('Not A Number')

        @type_check(str)
        def first_letter(word):
            return word[0]

        print(first_letter('Hello World'))
        first_letter(['Not', 'A', 'String'])
```

```
4
Bad type
```

```
H
Bad type
```

7th

Decorators don't have to wrap the function they're decorating. They can also simply register that a function exists and return it unwrapped. This can be used, for instance, to create a light-weight plug-in architecture:

```
In [8]: import random
        PLUGINS = dict()

        def register(func):
            PLUGINS[func.__name__] = func
            return func

        @register
        def say_hello(name):
            return f"Hello {name}"

        @register
        def be_awesome(name):
            return f"Yo {name}, together we are the awesomest!"

        def randomly_greet(name):
            greeter, greeter_func = random.choice(list(PLUGINS.items()))
            print(f"Using {greeter!r}")
            return greeter_func(name)

        randomly_greet('John')
```

```
Using 'be_awesome'


Out[8]: 'Yo John, together we are the awesomest!'
```