Higher order functions

It is a **nested functions**

```
def maxpower(n, exp):
        def count(m, cnt):
                if m >= exp:
                        return cnt
                return count(m*n, 1+cnt)
        return count(1, 0)

print(maxpower(2, 1000))
print(maxpower(3, 1000))
```

The variables `n and exp` live in the scope of **count** but **not a local variable of count**

```
def partial(op):
        def action(a, b):
                print(op(a,b))
        return action

f1 = partial(lambda x, y : x * 2 + y * 2)

f1(5,10)
f2 = partial(f1)
f2(5,10)
```

The variable `op` live in the scope of **action** but **not local in action**?

Why? One Liners

You can use lambda for HOF, and use **map** and **filter**.

map()

Executes a specified function **for each item in an iterable**. The item is sent to the function as a parameter.

`map(function, iterables)`

Parameter	Description
function	The function to execute for each item
iterable	A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

```
def myfunc(n):
    return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))
print(x)
print(list(x))
```

filter()

Returns an **iterator where the items are filtered through a function** to test if the item is accepted or not.

`filter(function, iterable)`

Parameter	Description
function	A Function to be run for each item in the iterable
iterable	The iterable to be filtered

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
    if x < 18:
        return False
    else:
        return True

adults = filter(myFunc, ages)

print(adults)
print(list(adults))
```

reduce()

Applies a **rolling computation to sequential** pairs of values in an iterable.

> `reduce()` was a *built-in* function for python, but was later removed in 2016 but was moved to functools

`reduce(function, iterable[, initializer])`

Parameter	Description
function	A function that takes two arguments and returns a value

Parameter	Description
iterable	The iterable to be reduced
initializer	(Optional) The initial value for the reduction

```
from functools import reduce

sum_all = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
combine_all = reduce(lambda x, y: x + y, ['a', 'b', 'c'])

print(sum_all)
print(combine_all)
```

We can write our own reduce:

```
def reduce(function, iterable):
        if not iterable:
                return iterable
        first = iterable[0]
        for i in iterable[1:]:
                first = function(first, i)
        return first

sum_all = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
combine_all = reduce(lambda x, y: x + y, ['a', 'b', 'c'])

print(sum_all)
print(combine_all)
```

any()

Returns **True if at least one element of an iterable is true**. If the iterable is empty, it returns False.

`any(iterable)`

Parameter	Description
iterable	The iterable to be checked for truthiness

```
L = [1, 2, 3, 4]
result1 = any(x > 3 for x in L)
result2 = any(x > 9 for x in L)

def isPrime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

result3 = any(isPrime(x) for x in [4, 6, 8, 99])
result4 = any(isPrime(x) for x in [4, 6, 8, 97, 99])

print(result1)
print(result2)
print(result3)
print(result4)
```

all()

Returns **True if all elements of an iterable are true** (or if the iterable is empty). If any element is false, it returns False.

`all(iterable)`

Parameter	Description
iterable	The iterable to be checked for truthiness

```
L = [1, 2, 3, 4]
result5 = all(x > 3 for x in L)
result6 = all(x > 0 for x in L)

def isPrime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

result7 = all(isPrime(x) for x in [4, 6, 8, 9, 99])
result8 = all(isPrime(x) for x in [3, 5, 7, 11, 97])

print(result5)
print(result6)
print(result7)
print(result8)
```

Used as a one-liner

```python
L = [9, 2, 1, 3, 4, 5, 6]

bigger_than_2 = map(lambda x: x > 2, L)
print(list(bigger_than_2))

remove_less_than_2 = list(filter(lambda x: x > 2, L))
print(remove_less_than_2)

odd_even = map(lambda x: 'o' if x % 2 else 'e', L)
print(list(odd_even))

odd_even_filter = list(filter(lambda x: 'o' if x % 2 else 'e', L))
print(odd_even_filter)

make_odd_string = map(str, list(filter(lambda x: x % 2, L)))
print(list(make_odd_string))

squared = str(list(filter(lambda x: x > 30, map(lambda x: x*x, L))))
print(squared)

squared_all = map(lambda x: x*x, L)
print(list(squared_all))
```

More on one-liners

```python
print(list(str(123456)))
def digitsum(n):
        return sum(map(lambda x: int(x), list(str(n))))

print(digitsum(123345))

print("########")

def sds(n):
        return sum(map(lambda x: int(x) ** 2, list(str(n))))

print(sds(22222))
```

Fked up sht (Taylor Swift's Backshots)

We can try to implement HOF for calculating the Taylor's Series using map():

> In mathematics, the Taylor series or Taylor expansion of a function is an infinite sum of terms that are expressed in terms of the function's derivatives at a single point.

Taylor Series General Formula:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$n!$ - factorial of n
a - real or complex number
$f^{(n)}(a)$ - nth derivative of f evaluated at the point a

cos(x)

We can Try to calculate estimate the value cos(x) using the Taylor's Series, where

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}x^{2n} \quad = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots \quad \text{for all x}$$

1. The function is (x is the angle)
   `cf = lambda n: (x ** (2 * n) * ((-1) ** n) / factorial(2 * n))`
2. Just map the function cf to `[0, 1, 2, 3, 4, 5, …]`
3. The target should be
   `cf(0) + cf(1) + cf(2) + cf(3) + cf(4) + …`

```python
from math import factorial, cos

def TaylorSeriesCosine(x):
    def cf(n):
        return (x ** (2 * n) * ((-1) ** n) / factorial(2 * n))

    return sum(map(cf, range(0, 10)))

angle = 3.141592654
print(TaylorSeriesCosine(angle / 3))
print(cos(angle / 3))
```

sin(x)

Similar to cos(x), we can Try to calculate estimate the value sin(x) using the Taylor's Series, where

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{2n+1} \quad = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \quad \text{for all x}$$

1. The function is (x is the angle)
   `sf = lambda n: ((-1) ** n) * (x ** (2 * n + 1)) / factorial(2 * n + 1)`
2. Just map the function sf to `[0, 1, 2, 3, 4, 5, …]`
3. The target should be
   `sf(0) + sf(1) + sf(2) + sf(3) + sf(4) + …`

```python
from math import factorial, sin

def TaylorSeriesSine(x, terms=10):
    def sf(n):
        return ((-1) ** n) * (x ** (2 * n + 1)) / factorial(2 * n + 1)

    return sum(map(sf, range(terms)))

angle = 3.141592654
print(TaylorSeriesSine(angle / 3))
print(sin(angle / 3))
```

Alternative

```python
from functools import reduce
from math import factorial, sin

def mapfn(n, x):
    return (-1) ** n / factorial(2 * n + 1) * x ** (2 * n + 1)

def filterfn(x):
    return True

def reducefn(x1, x2):
    return x1 + x2

def sine(x, n):
    return reduce(reducefn, map(lambda i: mapfn(i, x), filter(filterfn, range(n + 1))))

angle = 3.141592654
print(sine(angle / 3, 10))
print(sin(angle / 3))
```

`mapfn(n, x):`

- Represents a term in the Taylor series for sine.
- Calculates the value of the term based on the current iteration `n` and the input angle `x`.

`filterfn(x):`

- A filter function that always returns `True`. In this example, it doesn't filter any values.

`reducefn(x1, x2):`

- A function for reducing two values. It simply adds the two values together.

`sine(x, n):`

- Uses the `reduce` function to sum up the terms of the Taylor series.
- Applies the `mapfn` to each term, filters the terms using `filterfn`, and then reduces them using `reducefn`.
- The result is an approximation of the sine function for the given angle `x` using `n` terms.