30 Days Of Python: Day 14 - Higher Order Functions

Higher Order Functions

In Python functions are treated as first class citizens, allowing you to perform the following operations on functions:

- A function can take one or more functions as parameters
- A function can be returned as a result of another function
- A function can be modified
- A function can be assigned to a variable

In this section, we will cover:

1. Handling functions as parameters
2. Returning functions as return value from another functions
3. Using Python closures and decorators

Function as a Parameter

```
def sum_numbers(nums):  # normal function
    return sum(nums)    # a sad function abusing the built-in sum function :<

def higher_order_function(f, lst):  # function as a parameter
    summation = f(lst)
    return summation
result = higher_order_function(sum_numbers, [1, 2, 3, 4, 5])
print(result)       # 15
```

Function as a Return Value

```
def square(x):          # a square function
    return x ** 2

def cube(x):            # a cube function
    return x ** 3

def absolute(x):        # an absolute value function
    if x >= 0:
        return x
    else:
        return -(x)

def higher_order_function(type): # a higher order function returning a function
    if type == 'square':
        return square
    elif type == 'cube':
        return cube
    elif type == 'absolute':
        return absolute

result = higher_order_function('square')
print(result(3))       # 9
result = higher_order_function('cube')
print(result(3))       # 27
result = higher_order_function('absolute')
print(result(-3))      # 3
```

You can see from the above example that the higher order function is returning different functions depending on the passed parameter

Python Closures

Python allows a nested function to access the outer scope of the enclosing function. This is is known as a Closure. Let us have a look at how closures work in Python. In Python, closure is created by nesting a function inside another encapsulating function and then returning the inner function. See the example below.

Example:

```
def add_ten():
    ten = 10
    def add(num):
        return num + ten
```

```
    return add

closure_result = add_ten()
print(closure_result(5))  # 15
print(closure_result(10))  # 20
```

Python Decorators

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are usually called before the definition of a function you want to decorate.

Creating Decorators

To create a decorator function, we need an outer function with an inner wrapper function.

Example:

```
# Normal function
def greeting():
    return 'Welcome to Python'
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper
g = uppercase_decorator(greeting)
print(g())          # WELCOME TO PYTHON

## Let us implement the example above with a decorator

'''This decorator function is a higher order function
that takes a function as a parameter'''
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper
@uppercase_decorator
def greeting():
    return 'Welcome to Python'
print(greeting())   # WELCOME TO PYTHON
```

Applying Multiple Decorators to a Single Function

```
'''These decorator functions are higher order functions
that take functions as parameters'''

# First Decorator
def uppercase_decorator(function):
```

```
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper

# Second decorator
def split_string_decorator(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string

    return wrapper

@split_string_decorator
@uppercase_decorator     # order with decorators is important in this case -
.upper() function does not work with lists
def greeting():
    return 'Welcome to Python'
print(greeting())   # WELCOME TO PYTHON
```

Accepting Parameters in Decorator Functions

Most of the time we need our functions to take parameters, so we might need to define a decorator that accepts parameters.

```
def decorator_with_parameters(function):
    def wrapper_accepting_parameters(para1, para2, para3):
        function(para1, para2, para3)
        print("I live in {}".format(para3))
    return wrapper_accepting_parameters

@decorator_with_parameters
def print_full_name(first_name, last_name, country):
    print("I am {} {}. I love to teach.".format(
        first_name, last_name, country))

print_full_name("Suniksha", "Patel",'Finland')
```

Built-in Higher Order Functions

Some of the built-in higher order functions that we cover in this part are *map()*, *filter*, and *reduce*. Lambda function can be passed as a parameter and the best use case of lambda functions is in functions like map, filter and reduce.

Python - Map Function

The map() function is a built-in function that takes a function and iterable as parameters.

```
    # syntax
```

```
    map(function, iterable)
```

Example:1

```
numbers = [1, 2, 3, 4, 5] # iterable
def square(x):
    return x ** 2
numbers_squared = map(square, numbers)
print(list(numbers_squared))    # [1, 4, 9, 16, 25]
# Lets apply it with a lambda function
numbers_squared = map(lambda x : x ** 2, numbers)
print(list(numbers_squared))    # [1, 4, 9, 16, 25]
```

Example:2

```
numbers_str = ['1', '2', '3', '4', '5']  # iterable
numbers_int = map(int, numbers_str)
print(list(numbers_int))    # [1, 2, 3, 4, 5]
```

Example:3

```
names = ['Suniksha', 'Lidiya', 'Ermias', 'Abraham']  # iterable

def change_to_upper(name):
    return name.upper()

names_upper_cased = map(change_to_upper, names)
print(list(names_upper_cased))    # ['SUNIKSHA', 'LIDIYA', 'ERMIAS', 'ABRAHAM']

# Let us apply it with a lambda function
names_upper_cased = map(lambda name: name.upper(), names)
print(list(names_upper_cased))    # ['SUNIKSHA', 'LIDIYA', 'ERMIAS', 'ABRAHAM']
```

What actually map does is iterating over a list. For instance, it changes the names to upper case and returns a new list.

Python - Filter Function

The filter() function calls the specified function which returns boolean for each item of the specified iterable (list). It filters the items that satisfy the filtering criteria.

```
    # syntax
    filter(function, iterable)
```

Example:1

```
# Lets filter only even nubers
numbers = [1, 2, 3, 4, 5]  # iterable

def is_even(num):
    if num % 2 == 0:
        return True
    return False

even_numbers = filter(is_even, numbers)
print(list(even_numbers))       # [2, 4]
```

Example:2

```
numbers = [1, 2, 3, 4, 5]  # iterable

def is_odd(num):
    if num % 2 != 0:
        return True
    return False

odd_numbers = filter(is_odd, numbers)
print(list(odd_numbers))       # [1, 3, 5]
```

```
# Filter long name
names = ['Suniksha', 'Lidiya', 'Ermias', 'Abraham']  # iterable
def is_name_long(name):
    if len(name) > 7:
        return True
    return False

long_names = filter(is_name_long, names)
print(list(long_names))         # ['Suniksha']
```

Python - Reduce Function

The *reduce()* function is defined in the functools module and we should import it from this module. Like map and filter it takes two parameters, a function and an iterable. However, it does not return another iterable, instead it returns a single value. **Example:1**

```
numbers_str = ['1', '2', '3', '4', '5']  # iterable
def add_two_nums(x, y):
    return int(x) + int(y)

total = reduce(add_two_nums, numbers_str)
print(total)    # 15
```

💻 Exercises: Day 14

```
countries = ['Estonia', 'Finland', 'Sweden', 'Denmark', 'Norway', 'Iceland']
names = ['Suniksha', 'Lidiya', 'Ermias', 'Abraham']
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Exercises: Level 1

1. Explain the difference between map, filter, and reduce.
2. Explain the difference between higher order function, closure and decorator
3. Define a call function before map, filter or reduce, see examples.
4. Use for loop to print each country in the countries list.
5. Use for to print each name in the names list.
6. Use for to print each number in the numbers list.

Exercises: Level 2

1. Use map to create a new list by changing each country to uppercase in the countries list
2. Use map to create a new list by changing each number to its square in the numbers list
3. Use map to change each name to uppercase in the names list
4. Use filter to filter out countries containing 'land'.
5. Use filter to filter out countries having exactly six characters.
6. Use filter to filter out countries containing six letters and more in the country list.
7. Use filter to filter out countries starting with an 'E'
8. Chain two or more list iterators (eg. arr.map(callback).filter(callback).reduce(callback))
9. Declare a function called get_string_lists which takes a list as a parameter and then returns a list containing only string items.
10. Use reduce to sum all the numbers in the numbers list.
11. Use reduce to concatenate all the countries and to produce this sentence: Estonia, Finland, Sweden, Denmark, Norway, and Iceland are north European countries
12. Declare a function called categorize_countries that returns a list of countries with some common pattern (you can find the countries list in this repository as countries.js(eg 'land', 'ia', 'island', 'stan')).
13. Create a function returning a dictionary, where keys stand for starting letters of countries and values are the number of country names starting with that letter.
14. Declare a get_first_ten_countries function - it returns a list of first ten countries from the countries.js list in the data folder.
15. Declare a get_last_ten_countries function that returns the last ten countries in the countries list.

🎉 CONGRATULATIONS ! 🎉