List Comprehensions

Dt 7/4/2020
Tuesday

#list, set, dictionary

```
my_list = []
for char in 'hello':
    my_list.append(char)
print(my_list)
```

my list = [char for char in iterable]

```
my_list = [char for char in 'hello']
print(my list)
my_list2 = [num for num in rang(0,100)]
print(my_list2)
```

Instead of ~~Expression~~ now this is a variable.
~~xxxx~~ expression ~~xx~~ how we want to act upon each Character

Eg: my_list = [num * 3 for num in rang (0, 100)

num * 3 → Expression

```
my_list = [num **2 for num in range(0,100)
                    if num%2 ==0]
```

list = [<Expression> for <variable>]

↳ we can also convert them into a set form

```
my_list ={char for char in 'hello'}
```

Enclosing them via {} braces

SET COMPREHENSIONS

DICTIONARY

HELSINKI

```
my_dict ={ key : value }
```

↳ This key & value can be acted upon.

Eg value = value **2,

points to parsed

```
dicta={key:value**2 for key,value in my_dict}
```

```
dict:{key: value **2 for key,value in mydict.items()}
print(dict).
```

short hand type of writing the code

without key

num = {num: num ** 2 for num in [1,2,3]}

print(numnum)

finding the Duplicates

some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicates = list(set([x for x in some_list if some_list.count(x) > 1]))

Here the whole duplicate elements are Counted & removed,

PYTHON DECORATORS

@<Decorator Name>

@ classMethod

@ Static Method

```
def hello():
    print('hello')

greet = hello
print(greet())
print(greet)
```

del hello

↓

deleting the function

All it does it delete the functio in Memory

// functions in Python act like variables they are first class citizens

Nesting functions Calling func inside a function

```
def hello (func):
    func()

def greet():
    print ('still here')

a = hello (greet)
print (a)
```

@ decorator name → underneath the hood they use the functions

Higher Order function — A function which Accepts a function as a parameter.

```
def greet (func):
    func()
```

or

```
def greet2 ():
    def func():
        return 5

    return func
```

A function which returns a function

Decorators → Supercharges the functions

```
def hello():
    print('hello')
def my_decorator(func):
    def wrap_func():
        print('* * * *')
        func()
    return wrap_func
```

// We can use it above.

```
@my_decorator
def hello():
    print('hello')
hello()
```

o/p

```
* * * *
hello
* * * *
```

The whole thing itself passes as an argument the main function

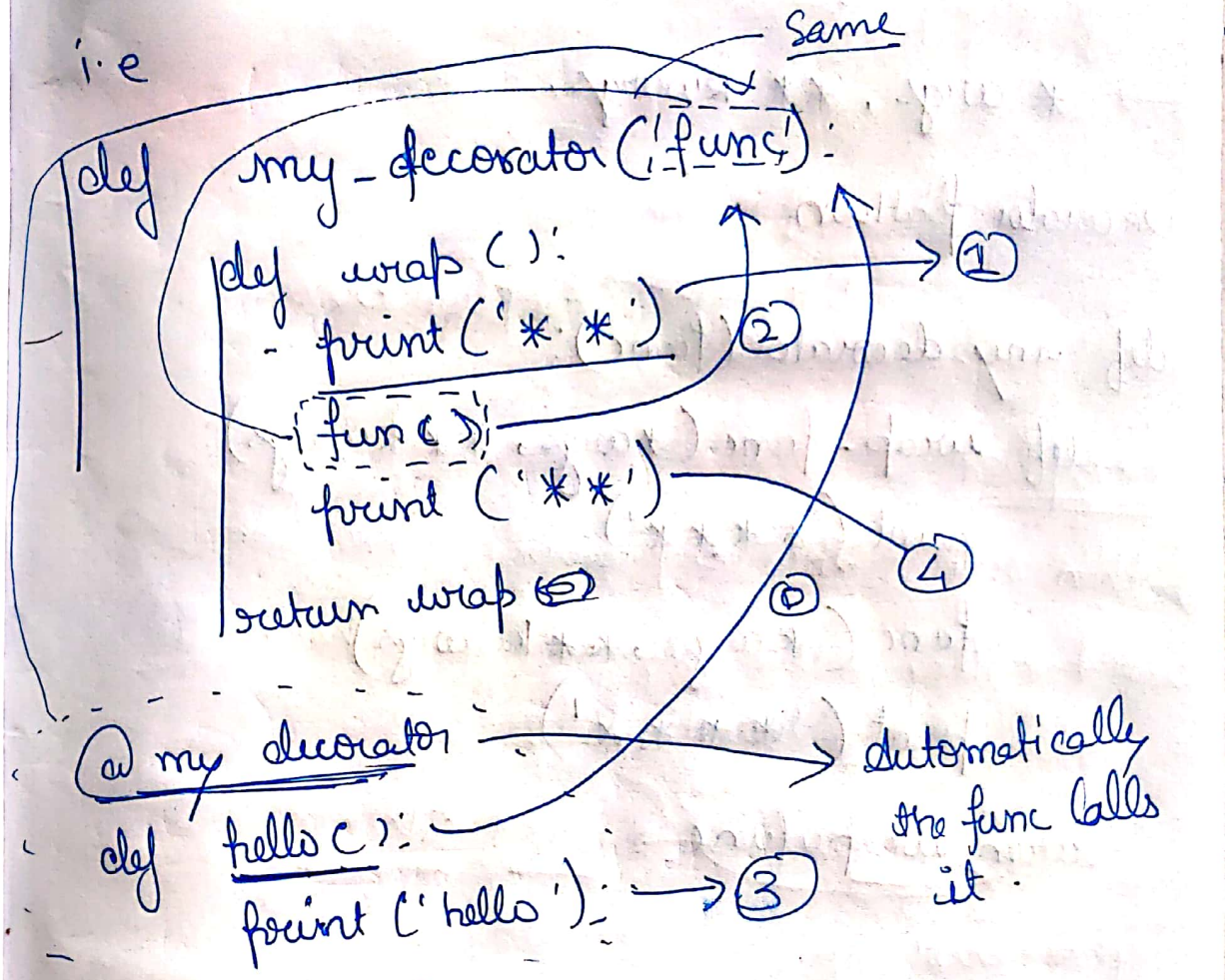
Eg:
We can directly send a function inside a function by just Mentioning it as @(name of funct.

Eg

```
def hello(func):
    print("* * * *")
    func()
@hello
```

↳ The Argument which we are passing will have the same name.

i.e



hello() → Function Calling

Calling the function via Decorator

```
hello2 = my_decorator(hello)
hello2()
```

// So, here my_decorator function just returns the whole thing that is 2 print statements & One function

We can also send arguments to the functions

*args, **kwargs,

Decorator pattern

```
def my_decorator(func):
    def wrap_func(*args, **kwargs):
        print('****')
        func(*args, **kwargs)
        print('*****')
    return wrap_func

@my_decor
def hello(greeti, emoj=':)'):
    print(greeti, emoji)

hello('hii')
```

Why do we need Decorators ?

from time import time

time → Module

time → function

Authentication

Privileges

```
from time import time

def performance(fn):
    def wrapper(*args, **kwages):
        t1 = time()
        result = fn(*ag)
        t2 = tin
        pri(t2-t1)
        return resul
    return result

@perfe
def dim():
    for in rang(10000):
        i * 5
```

time() → Calling the function

↳ meta programming as a part of the programmer tries to modify another part of the program at Compile time.

Error handling in Python

Indentation Error.

↔

printing the expression wrong.

Eg print ('1')) → Exception error

Exception

Python raises this expressions while Compiling.

Error handling allows us let the python script running even though it has errors

1 + 'True' → int + string

Syntax error
Refrence error } docs.python
Tab error
name error, Type error,

```
def hoo()
    li = {'a':1}
    d1['b'] → key error
hoo()
```

5/0 = Zero division Error.

Built-in Exceptions

int (input ("Enter age"))

↳ Asking in int type

if age > 18

↳ Perform Action Else do other.

try:
    age = int (input ('age'))
    print (age)

except:
    print ("please enter number")

similar to macros in C

looping → while True: → unless its True

Try:
except:
else:
break.
pass:
Continue:

do while

except ValueError:
    →
except ZeroDivisionError
    →
else
    P → Thale
    Break.

def Sum (num1, num2)
    return num1 + num2 → Concatination

print (Sum ('1' + '2')) → 12

except :) → has predefined value.

```
except TypeError as err:
    print (' ' + err)
raise ValueError ('hey cut it')
```

assert

finally

raise Exception ('hey cut it')

GENERATORS

→ It allow us to Generate a sequence of values over time

Eg : range(100)

→ It can pause & resume functions

```
def make_list(num):
    result = []
    for i in range(50)
        result.append(i*2)
    return result
my_list = make_list(100)
print(my_list)
```

range is a generator which is not held in memory

↳ It doesn't create on its own

Instead we can use generator

__iter__ iterable object to continuity

maintainance loops etc

(generator is a subset of iterable)

```
def generate_function(num):        ← Param
    for i in range(num):
        yield i*2
```

↳ It pauses the function & come back to it.

```
for item in generate_function(100):
    print(item)
```

stopIteration Error

```
g = generator_fun(100)
print(g).        next(g)
next(g)          next(g)
```

```
for i in range(num):
    yield i*2
print(next(g))
```

```
def special_for(iterable)
    iterator = iter(iterable)
    while True
        Try: first()
             next(iterator)
        except StopIteration
            break

special_for((1,2,3))
```

Ranges

```
class My_Range():
    def __init__(self, first, last)
        self.first = first
        self.last = last

    def __iter__(self):
        return self.

    def __next__(self):
        if MyGen.curr < self.last:
            num = MyGen.curr
            += 1
            return num

gen = MyGen(0,100)
for ir in gen:
    print(i)
```