Advanced Python Topics

Overview

- Comprehensions.
- Iterators.
- Generators.
- *Lambda* Functions.
- Closures.
- Decorators.

PER SCHOLAS

Comprehensions

A **Comprehension** in *Python* is a syntactic construct for creating a new sequence. It provides a concise "*pythonic*" way to construct a list, a dictionary, a set, or a generator.

Comprehensions should be used for simple cases to make our code shorter and clearer. If using comprehensions makes the code harder to follow, avoid using them; stretching the code into different logical blocks, using descriptive names and adding comments will often make the code much clearer and readable.

The basic structure of a comprehension is as follows:

```
expression for variable in sequence
```

Comprehension Examples

```
my_list = [e**2 for e in range(1, 10, 2)]
# OR
my_list = []
for e in range(1, 10, 2):
    my_list.append(e**2)
```

- *if* statements:

```
my_list = [e**2 for e in range(10) if e%2 == 1]
# OR
my_list = []
for e in range(10):
    if e%2 == 1:
        my_list.append(e**2)
```

Comprehension Examples (continued)

- multiple conditional statements using *ternary operators*:

```
grades = [95, 55, 83, 75, 91]


grades_l = [('A' if g>=90 else ('B' if g>=75 else 'C')) for g in grades]
# OR
grades_l = []
for g in grades:
    if g >= 90:
        grades_l.append('A')
    elif g >= 75:

        grades_l.append('B')
    else:
        grades_l.append('C')
```

PER SCHOLAS

Comprehension Examples (continued)

- nested for loops:

```
pairs = [(i, j) for i in range(3) for j in range(2)]
# OR
pairs = []
for i in range(3): for j in
    range(2):
        pairs.append((i, j))
```

- sets and dictionaries:

```
letters = ['a', 'b', 'a', 'f']

d = {l: ord(l) for l in letters}                # dict

s = {ord(l) for l in letters if l < 'd'}
                                                 # set
```

PER SCHOLAS

Comprehension Exercises

1. Use *list comprehension* to create a list containing a solution for the famous *FizzBuzz* problem. For integers 1 to 100, inclusively, the value should be:
   - 'Fizz' if divisible by 3.
   - 'Buzz' if divisible by 5.
   - 'FizzBuzz' if divisible by both 3 and 5.
   - The integer itself if not divisible by both 3 and 5.
2. From the *ndarray* with the given values, use *dictionary comprehension* to create a dict with string key in the format "c- avg(c)" and list value representing the column values.

```
# Values for the ndarray
 [[ 1    2  3]
  [ 4    5  6]
  [ 7    8  9]
 [10    11 12]]
```

```
# Expected output
    {'1-5.5': [1,   4,  7,   10],
     '2-6.5': [2,   5,  8,   11],
     '3-7.5': [3,   6,  9,   12]}
```

Iterators

- A *Python* ***iterator*** is an object that can be iterated upon. An *iterator* returns data, one element at a time. They are implicitly implemented within *for* loops, comprehensions, etc.

- *Iterator* object must implement two special methods:
  - *iter* Gets called by the *iter* function.
  - *next* Gets called by the *next* function.

- A *python* object is called an *iterable* if an *iterator* can be returned from it by calling the *iter* function. The *next* function can be called to the *iterator* to get the next value in the object. Lists, tuples, strings, and dicts are all commonly used *iterables.* Current value cannot be accessed from the *iterator*.

Iterator Example

- When we exhaust all the elements of an iterator by manually calling the *next* function, there would be no more data to be returned. Calling the *next* function at that state will raise a *StopIteration* exception.

```
my_list = ['python', 'java', 'scala','javascript']

my_iter = iter(my_list)

print(next(my_iter))                        # 'python'
print(next(my_iter))                        # 'java'
print(my_iter.__next__())                   # 'scala'
print(my_iter.__next__())                   # 'javascript'
print(next(my_iter))                        # 'StopIteration' raised
```

PER SCHOLAS

Iterator within For Loop

```
my_list = ['python', 'java', 'scala', 'javascript']

for e in my_list:
    print(e)

# The actual implementation for the above for loop is the following:
my_iter = iter(my_list)

while True:

    try:
        print(next(my_iter))
    except StopIteration:

        break
```

PER SCHOLAS

Generators

- Building a *Python iterator* is generally NOT simple. We have to implement a class with *iter* and *next* methods, keep track of internal states, and raise *StopIteration* exceptions appropriately.

- A Python generator allows us to produce the same functionality through a much simpler way.

- A *generator* is a function that behaves like an *iterator*. It returns a *generator* object, which can be iterated over one value at a time. The returned *generator* object can only be iterated through once.

yield Statement

- A *generator* can be created by defining a normal function with a ***yield*** statement instead of a ***return*** statement. If a function contains at least one *yield* statement, it becomes a *generator* function.

- As we have seen, a *return* statement completely terminates a function. However, a *yield* statement pauses the function, saving all of its states, and later continues from there on the following calls.

- A function may contain multiple *yield* AND/OR *return* statements depending on the logic.

Generator Example

```python
def powers_of_two():
n = 1
     while True:
          n *= 2
          yield n
powers = powers_of_two()
for _ in range(20):
     print(next(powers))
```

```python
squares = (i**2 for i in range(20))


for s in squares:
     print(s)
```

```python
def even_n(max_n=1):
     n = 1
     while n <= max_n:
     yield n * 2
     n += 1


i = even_n(3)
print(next(i))
print(next(i))
print(next(i))
```

PER SCHOLAS

Generator Exercises

1. Create a generator, *primes_gen* that generates prime numbers starting from 2.

```
gen = primes_gen()
for _ in range(10): print(next(gen),
      end=' ')

# Expected output
# 2 3 5 7 11 13 17 19 23 29
```

2. Create a generator, *unique_letters* that generates unique letters from the input string. It should generate the letters in the same order as from the input string.

```
for letter in unique_letters('hello'):
      print(letter, end=' ')
# Expected output
# h e l o
```

lambda Functions

- In a *Python* program, a number or a string can be stored in a *variable* defined with the given value. A *function* can also be stored in a *variable*, as we have seen already:

```
def hello(name):
    print('hello', name)
# Here, a function is stored in the variable named 'hello'
```

- In cases where we need a number or a string for a single usage or as a parameter, we can use a number *literal* or a string *literal* without using a *variable*. What about the functions?
- We can define very simple *anonymous* functions on the fly using ***lambda*** keyword.

lambda Functions

- An *anonymous* function is a function that is defined without a name. While a normal function is defined with a ***def*** keyword, an *anonymous* function is defined with a ***lambda*** keyword. Therefore, an *anonymous* function is also called a ***lambda*** function.

- A *lambda* function has the following syntax:

```
lambda arguments : expression
```

- A *lambda* function can have any number of arguments but accept only ONE expression; hence, it should only be used for creating a very simple function.

PER SCHOLAS

lambda Function Examples

normal

```
def hello1():
    print('hello')


def hello2(name):
    print('hello', name)


def product(x, y):

    return x * y
```

anonymous

```
hello1 = lambda : print('hello')


hello2 = lambda name: print('hello', name)


product = lambda x, y: x *
y
```

lambda Function Examples

```
from functools import reduce


my_list = [-10, -20, -4, 5, 30]


list(map(lambda x: x**2, [1, 2, 3, 4]))
list(filter(lambda x: x>0, my_list))
sorted(my_list, key=lambda x: abs(x))
reduce(lambda a, b: a*b, my_list)
```

- ***map*** is a *Python* built-in function that takes in a function and a sequence as the arguments, calls the input function on each item of the sequence, and then returns an iterable *Map* object.
- ***filter*** is a *Python* built-in function that filters a sequence or any iterable object and returns an iterable *Filter* object.
- ***reduce*** is a function from ***functools*** module that applies a function of two arguments cumulatively to the items of a sequence or an iterable. It returns a single value.

lambda Exercises

- Consider the list:

```
prog_lang = [('Python', 3.8),
                ('Java', 13),
                ('JavaScript', 2019),
                ('Scala', 2.13)]
```

1. Sort the list by each language's version in ascending order.

```
[('Scala', 2.13), ('Python', 3.8), ('Java', 13), ('JavaScript', 2019)]
```

2. Sort the list by the length of the name of each language in descending order.

```
[('JavaScript', 2019), ('Python', 3.8), ('Scala', 2.13), ('Java', 13)]
```

3. Filter the list so that it only contains languages with 'a' in it.

```
[('Java', 13), ('JavaScript', 2019), ('Scala', 2.13)]
```

4. Filter the list so that it only contains languages whose version is in integer form.

```
[('Java', 13), ('JavaScript', 2019)]
```

lambda Exercises (continued)

```
prog_lang = [('Python', 3.8),
             ('Java', 13),
             ('JavaScript', 2019),
             ('Scala', 2.13)]
```

5. Transform the list so that it contains the tuples in the form, ("language in all lower case", length of the language string)

```
[('python', 6), ('java', 4), ('javascript', 10), ('scala', 5)]
```

6. Generate a tuple in the form, ("All languages separated by commas", "All versions separated by commas").

```
('Python,Java,JavaScript,Scala', '3.8,13,2019,2.13')
```

PER SCHOLAS

Closures

- A ***closure*** is a function object that remembers values in enclosing scopes even if they are not present in the memory. A *closure* allows the access to those "captured" variables through the *closure*'s copies of their values, even when the function is invoked outside of their scope.

- A *closure* is created with a ***nested*** function that accesses a ***nonlocal*** variable from the ***enclosing*** function. The *enclosing* function must return the *nested* function.

- This technique itself, by which some data gets attached to the code, is also called a ***closure*** in *Python*.

PER SCHOLAS

Closure Example

```
def outer(msg): lang =
     'Python' def
     inner():
          print(lang, msg)
     return inner

my_func = outer('is fun!!!')
my_func()  # output: 'Python is fun!!!'
```

- In the above example, the '*outer*' function was called with the string '*is fun!!!*' and the returned function was bound to the name *my_func.* On calling *my_func*(), the message and the value of *lang* were still remembered although we had already finished executing the '*outer*' function.
- When using a *closure*, the values in the enclosing scope are remembered even when the variables go out of scope or the function itself is removed from the current namespace.

PER SCHOLAS

Closures - Advantages

- *Closures* can provide simpler solution to the classes with a few methods.

```python
class Multiplier:
     def  init__(self, n): self.n = n
     def multiply(self, k):
          return k * self.n
multiplier3 = Multiplier(3)
print(multiplier3.multiply(5))
```

```python
# OR

def make_multiplier_of(n):
     def multiply(k):
          return k * n
     return multiply
multiplier3 = make_multiplier_of(3)
print(multiplier3(5))
```

- *Decorators* can be implemented using *closures*.
- *Closures* also provide some form of data hiding, giving us access to nonlocal variables.

PER SCHOLAS

Closure Exercise

- Using a *closure*, create a function, *multiples_of*(n), which we can use to create generators that generate multiples of n less than a given number.

```
m3 = multiples_of(3)

m3_under30 = m3(30)

m7_under30 = multiples_of(7)(30)

print(type(m3_under30))

# output: <class 'generator'>

print(*m3_under30)

# output: 3 6 9 12 15 18 21 24 27

print(*m7_under30)
# output: 7 14 21 28
```

PER SCHOLAS

Decorators

- A ***decorator*** is a function used to transform some function into another form. A *decorator* creates a kind of composite function. *Decorators* use the *closures* technique.
- A *decorator* is used as a *higher-order function* (a function that accepts a function as one of its arguments and/or return another function). It returns another function, applied as a function transformation, usually using the `@wrapper` syntax. Common examples for *decorators* are *classmethod()* and *staticmethod()*.

```
def my_decorator(func):
        ...
        return other_func

 def some_function():
        ...
some_function = my_decorator(some_function)
# some_function is decorated with modified behavior
```

PER SCHOLAS

Decorators Examples

```
from datetime import datetime


def my_dec(func): # some_function is passed
    def wrapper():     # nested function is created
        print(datetime.now())
        func()
    return wrapper     # the nested function is returned


def my_func():
    print('my_func is executed...')


my_func = my_dec(my_func)     # my_func is decorated by my_dec


my_func()
# output: 2018-11-10 10:50:20.334598
             my_func is executed...
```

Decorators Examples (continued)

```python
def run_n_times(n):
     def dec(func):
          def wrapper():
               for _ in range(n):
                    func()
          return wrapper
          return dec
def say_hello():
     print('hello')


say_hello = run_n_times(5)(say_hello)
# say_hello is decorated by run_n_times
say_hello()
```

```
# output:
hello
hello
hello
hello
hello
```

Decorators Examples (continued)

```
import time

def execution_timer(func): def
     wrapper(*args):
          start = time.perf_counter()
          result = func(*args)

          end = time.perf_counter()
          print(f"Execution time of '{func.__name__}'"
                f" with {len(args)} args: {end-start:.8f} secs") return
          result

     return wrapper


@execution_timer
def average(*args): sum_total =
     sum(args)

     n = len(args)
     return round(sum_total/n, 2)
```

PER SCHOLAS

Decorators Examples (continued)

```
print(average(4.3, 2.22, 5.55, 9, 100))
print(average(*range(5423569)))


# output:
Execution time of 'average' with 5 args: 0.00002150 secs
24.21
Execution time of 'average' with 5423569 args: 0.28669390 secs
2711784.0
```

Decorator Exercises

- Create following decorators:

1. *make_upper* – make every letter of a string returned from the decorated function uppercase.

```
@make_upper
def hello_world():
    return 'hello young, good day!!'

  print(hello_world())     # output: HELLO YOUNG, GOOD DAY!!
```

2. *print_func_name* – print the name of the decorated function before executing the function.

```
  @print_func_name
  def my_func():
     print('Python is fun!!')
my_func()   # output: my_func is running...
                      Python is fun
```

PER SCHOLAS

Decorator Exercises (continued)

3. *give_name(name)* – concatenate the given name at the end of a string returned from the decorated function.

```
@give_name('Theresa')

def greeting():

    return 'Hello'
print(greeting()) # output: Hello Theresa
```

4. *print_input_type* – print a data type of the input argument before executing the decorated function.

```
@print_input_type
def square(n):
    return n ** 2

print(square(3.5))        # output: The input data type is <class 'float'>
                                    12.25
```

Decorator Exercises (continued)

5. *check_return_type(return_type)* – check if the return type of the decorated function is *return_type* and print the result before executing the function.

```
@check_return_type(str)
def square(n):
    return n ** 2
print(square(6))    # output: =========Error!!
                              =========The return type is NOT <class 'str'>
                              36

@check_return_type(float)
def square(n):
    return n ** 2
print(square(2.9))
                    # output: The return type is <class 'float'>
                              8.41
```

PER SCHOLAS

Decorator Exercises (continued)

6. *execute_log* – write a function execution log on the log file.

```
@execute_log
def multiply(*nums):

     mult = 1

    for n in nums:
        mult *= n

    return mult
@execute_log

def hello_world():

    return 'hello world!!'


print(multiply(6, 2, 3))          # 36
print(hello_world())              # hello world!! #
print(multiply(2.2, 4))           #8.8
print(hello_world())              # hello world!!
```

function execution.lo

```
2020-05-01 13:55:53.059315  multiply
2020-05-01 13:55:53.060312  hello_world
2020-05-01 13:55:53.060314  multiply
2020-05-01 13:55:53.060323  hello_world
```

PER SCHOLAS

Summary Review

- Comprehensions.
- Iterators.
- Generators.
  - yield statement
- *lambda* Functions.
  - map, filter, reduce functions
- Closures.
- Decorators.
- Exercises.

Questions?