Python | Main course

Session 11

- With statement (Context Manager)
- Generators
- Iterators
- Variable-Length Arguments
- Decorators

by Mohammad Amin H.B. Tehrani - Reza Yazdani

www.maktabsharif.ir

With statement

With statement

Intro

A context manager is an object that defines the runtime context to be established when executing a with statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the with statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

Syntax:

```
with my_context_manager [as variable]:
    # statements
    ...
```

With statement

enter & exit methods

___enter___ method

Enter the runtime context related to this object. The with statement will bind this method’s return value to the target(s) specified in the as clause of the statement, if any.

___exit___ method

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be None.

Return: If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

With statement

Example

```python
class Product:
   def __init__(self, name, inventory):
       self.name = name
       self.inventory = inventory

   def __str__(self):
       return f"Product '{self.name}':" \
              f" inventory = {self.inventory}"
```

Now run the code below:

```python
p = Product("Saboon", inventory=10000)
print(p)
with Order(p, 200) as new_order:
   print(p)
   new_order.payment('12312312', None, None)

print(p)
```

```python
class Order:
   def __init__(self, product: Product, amount: int):
       self.product = product
       self.amount = amount

   def payment(self, card_no, cvv2, password):
       pass # For the payment stage

   def __enter__(self):
       self.product.inventory -= self.amount
       return self

   def __exit__(self, exc_type, exc_val, exc_tb):
       if exc_type:  # When exception Raised!
           self.product.inventory += self.amount
           print(f">>> Error: {exc_type=} : {exc_val=}")
       else:
           print("Congratulations!")  # successful!
       return True # For ignore raising exceptions!
```

Iterators & Generators

Iterators in python

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are hidden in plain sight.
Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

so, You can simply make an object (or class), **iterable** by implement **__iter__** method

Some of built-in iterators in python:

```
from typing import Iterator

print(isinstance(map(..., []), Iterator))
print(isinstance(filter(..., []), Iterator))
print(isinstance(zip([], []), Iterator))
print(isinstance(enumerate([]), Iterator))
```

Custom Iterable object

You can make an object iterable by implementing **__iter__** and **__next__**

```python
class Class1:

   def __iter__(self):
       self.n = -1
       return self

   def __next__(self):
       self.n += 1
       if self.n < 10:
           return self.n ** 2
       raise StopIteration("N < 10")
```

```python
x = iter(Class1())

print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))

print('For:')
for i in x:
   print(i)
```

Generators

Yield Keyword

yield is a keyword in Python that is used to return from a function without destroying the states of its local variable.
When the function is called, the execution starts from the **last** yield statement.
Any function that contains a yield keyword is termed as **generator**. Hence, yield is what makes a generator. yield keyword in Python is less known off but has a greater utility which one can think of.
Generator type is a subclass of Iterator type

Syntax:

```
def a_generator_func():
    ...
    yield ...
    ...
```

```
def generator_example():
    print("First part of code...")
    yield 1

    print("Second part of code...")
    yield 2

    print("Third part of code...")
    yield 3


g = generator_example()
print(next(g))
print(next(g))
print(next(g))
print(next(g))  # ???
```

Generator Example

```
from typing import Generator, Iterator

def generator(n):
   for i in range(n):
       yield i ** 2

g1 = generator(100)
g2 = (i ** 2 for i in range(100))  # Generator comprehensions!

print(isinstance(g1, Generator))
print(isinstance(g2, Generator))

print(isinstance(g1, Iterator))
print(isinstance(g2, Iterator))

print(list(g1) == list(g2))
```

Generators

Example: primals

```python
def is_primal(n: int):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```python
def primals_gen(n: int):
    for i in range(2, n + 1):
        if is_primal(i):
            yield i
```

```python
def primals_func(n: int):
    res = []
    for i in range(2, n + 1):
        if is_primal(i):
            res.append(i)
    return res
```

Compare results:

```python
for i in primals_func(100):
    print(i, end=', ')

print("\nTry it with a bigger number:")

for i in primals_func(100000):
    print(i, end=', ')
```

```python
for i in primals_gen(100):
    print(i, end=', ')

print("\nTry it with a bigger number:")

for i in primals_gen(100000):
    print(i, end=', ')
```

Exercise: range() generator

range generator

Create a **generator** named ‘range’ to re-declare python range() utility.

- Add type hint for arguments and the return
- Check and Validate input arguments using Exceptions (**TypeError**)

Note: test your code using for loop and print the result.

```
# Prototype
def range(start: int, end: int = None, step: int = 1):
    pass
```

Variable-Len Arguments
&
Keyword Arguments

Intro

You may need to process a function for more arguments than you specified while defining the function.
You can use variable-length arguments or keyword arguments to pass multiple unspecified arguments to your function.

*args

The special syntax *args in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

**kwargs

The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments.

Args & kwargs

Example

```
def some_function(a, b, c, *var_len_args, **keyword_args):
   print("Normal aruments:", a, b, c)
   print("Var-Len arguments (*args): ", var_len_args)
   print("Keyword arguments (*kwargs): ", keyword_args)

   print("\nargs is tuple:", isinstance(var_len_args, tuple))
   print("kwargs is dict:", isinstance(keyword_args, dict))


some_function(1,2,3,4,5,6,g=7,h=8)
```

```
Normal aruments: 1 2 3
Var-Len arguments (*args):  (4, 5, 6)
Keyword arguments (*kwargs):  {'g': 7, 'h': 8}

args is tuple: True
kwargs is dict: True
```

Unpack *args & **kwargs

Use * to unpack args:

```
*args    -> (arg1, arg2, arg3, ...)
```

Use ** to unpack kwargs:

```
**kwargs -> (key1=value1, key2=value2, key3 = value3, ...)
```

```
args = [1, 2, 3, 4, 5, 6]
kwargs = {'g': 7, 'h': 8}
some_function(*args, **kwargs)
```

Output = previous output

```
args = ("===", 'Hello', 'World', '!')
kwargs_dict = {'end': '===', 'sep': ' - '}
print(*args, **kwargs_dict)
```

```
=== - Hello - World - !===
```

Decorator

Decorator

Intro

First class Objects:
In Python, functions are first class objects that means that functions in Python can be used or passed as arguments.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Briefly: You can use functions as a value like another types of values like int, float and ...

Decorator

Example

```python
def my_string_customer(s: str):
    return s.swapcase().replace(' ', '-')

# NOTE: don't use parentheses next to it, otherwise you wrongly Called it!!!
f = my_string_customer
print(isinstance(f, Callable))
print("f=", f)
print("f('akbar')=", f('akbar'))
print("f('Hello world')=", f('Hello world'))
print("f('AbcDEfg 1234')=", f('AbcDEfg 1234'))
```

```
True
f= <function my_string_customer at 0x000001F0CBFC6A60>
f('akbar')= AKBAR
f('Hello world')= hELLO-WORLD
f('AbcDEfg 1234')= aBCdeFG-1234
```

Decorator

Example: Pass function to another function

```python
sample_text = """Adipisci voluptatem sed
voluptatem. Aliquam sit quiquia
consectetur ipsum. Velit eius sed
dolore. Etincidunt ut tempora non.
Dolorem sit non amet dolor.
"""
```

```python
def my_string_customer(s: str):
   return s.swapcase().replace(' ', '-')


def get_lines(text, custom_function):
   res = []
   for i in text.split('.'):
       res.append(custom_function(i))
   return res
```

Function call:

```python
a_func = my_string_customer
lines = get_lines(sample_text, a_func)
print(lines)
```

Output:

```
['aDIPISCI-VOLUPTATEM-SED-VOLUPTATEM',
'-aLIQUAM-SIT-QUIQUIA-CONSECTETUR-IPSUM'
, '-vELIT-EIUS-SED-DOLORE',
'-eTINCIDUNT-UT-TEMPORA-NON',
'-dOLOREM-SIT-NON-AMET-DOLOR']
```

Decorator

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 very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

Technically:

- ➔ Decorators are **functions** that,
- ➔ **get another function** as a parameter,
- ➔ finally **return a new function**

Decorator

Example

```
def make_upper(func: Callable[[], str]) -> Callable:
    def inner_function():
        res = func()
        return res.upper()

    return inner_function


def hello_world():
    return "Hello World!"


old_function = hello_world
new_function = make_upper(hello_world)

print('old:', old_function())
print('new:', new_function())
```

Output???

Decorator

Example

```python
def make_upper(func: Callable[[], str]) -> Callable:
    def inner_function():
        res = func()
        return res.upper()

    return inner_function


def hello_world():
    return "Hello World!"


old_function = hello_world
new_function = make_upper(hello_world)

print('old:', old_function())
print('new:', new_function())
```

```
old: Hello World!
new: HELLO WORLD!
```

Example: Decorator Symbol

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example:

```python
def make_upper(func: Callable[[], str]) -> Callable:
    def inner_function():
        res = func()
        return res.upper()

    return inner_function


@make_upper  # Decorating...
def hello_world():
    return "Hello World!"

print(hello_world())
```

```
HELLO WORLD!
```

Decorator

Example: Chaining Decorators

```python
# Decorators
def validate_phone(func):
    """Decorator to validate phone number"""
    def inner_function(*args, **kwargs):
        p: str = func(*args, **kwargs)
        phone_number = p[-10:]
        assert phone_number.isnumeric(), "Invalid phone number: numeric"
        assert len(phone_number) == 10, "Invalid phone number: len"
        assert phone_number.startswith('9'), "Invalid phone number: start"
        return p

    return inner_function

def prefix(func):
    """Add ‘+98’ prefix to phone number"""
    def inner_function(*args, **kwargs):
        phone_number: str = func(*args, **kwargs)
        phone_number = phone_number[-10:]
        return '+98' + phone_number

    return inner_function
```

Decorator

Example: Chaining Decorators

```
# User class
class User:

   def __init__(self, id, name, phone):
       self.__id = id
       self.__name = name
       self.__phone = phone

   ...

   @property
   @prefix
   @validate_phone
   def phone(self):
       return self.__phone
```

```
# Calling

users = [User(1, 'Akbar', '+989123456781'),
        User(2, 'Asqar', '09123456782'),
        User(3, 'Reza', '9123456783'),
        User(4, 'Mamad', '1234'),
        User(5, 'Bagher', '0123412381'),
        ]

print(users[0].phone)
print(users[1].phone)
print(users[2].phone)
print(users[3].phone)
print(users[4].phone)
```

Output???

Advanced topics

- *** Decorator class**
- *** Generator comprehension**
- Date & Time in python