Python for DevOps

Module 4.

Modules

Creeating and using own modules

```
#hey.py
def sayhey(name):
    print(f"Hey {name} !!")


#main.py
import hey

hey.sayhey("Igor")
```

from … import

```
#greetings.py
def polite(name):
    print(f"Dear {name},")

def notsopolite(name):
    print(f"Hi {name},")


#main.py
from greetings import polite

polite(“Igor”)
```

import as

```
#greetings.py
def polite(name):
    print(f"Dear {name},")

def notsopolite(name):
    print(f"Hi {name},")


#main.py
from greetings import polite as formal
from greetings import notsopolite as unformal

formal("Igor")

unformal("Igor")
```

Where modules live

- Script run path
- PYTHONPATH
- Check with sys.path:

```
>>> print('\n'.join(sys.path))

/usr/local/Cellar/python@3.9/3.9.6/Frameworks/Python.framework/Versions/3.9/lib/python39.zip
/usr/local/Cellar/python@3.9/3.9.6/Frameworks/Python.framework/Versions/3.9/lib/python3.9
/usr/local/Cellar/python@3.9/3.9.6/Frameworks/Python.framework/Versions/3.9/lib/python3.9/lib-dynload
/usr/local/lib/python3.9/site-packages
```

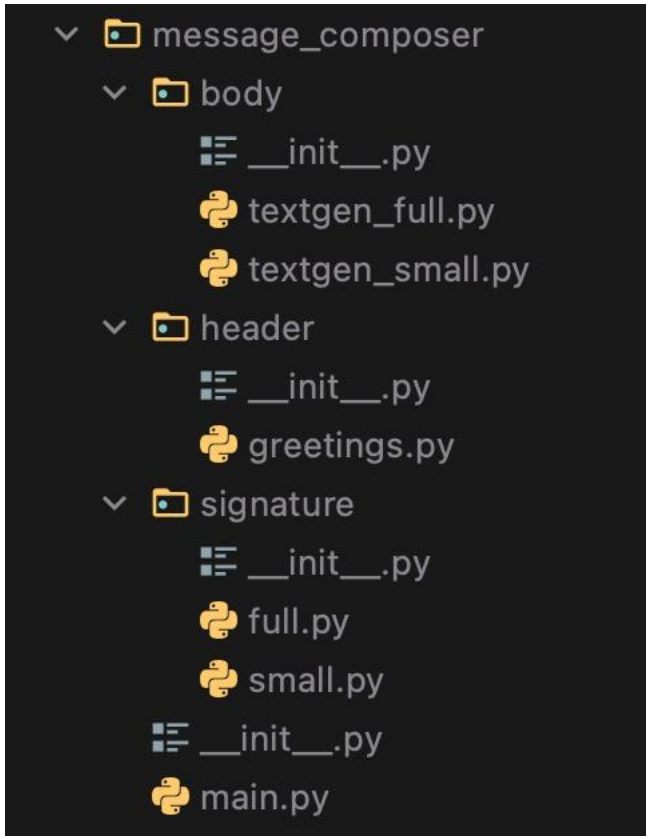
Module’s content

```
import greetings

print(dir(greetings))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'notsopolite', 'polite']
```

Python Packages



The __init__.py files are required to make Python treat directories containing the file as packages.

```
import message_composer.header.greetings as header
from message_composer.body import textgen_small as body
import message_composer.signature.small as signature

name = 'Bob'
print(f"{header.polite(name)} {body.mailbody('We need to go for a cup of coffee')} {signature.sig('Igor')} ")
```

```
Dear Bob,
We need to go for a cup of coffee
BR, Igor
```

Functions: Deep dive

LEGB logic

L – Local. is the code block or body of any Python function
E – Enclosing. Special scope that only exists for nested functions.
G – Global. Scope contains all of the names that you define at the top level of a program or a module. Names in this Python scope are visible from everywhere in your code.
B – Build-in . Special Python scope that's created or loaded whenever you run a script or open an interactive session. Preassigned in python

LOCAL -> ENCLOSING-> GLOBAL -> BUILT-IN

Example

```
>>> result = 0

>>> def sum(num1, num2):
        result = num1 + num2
        print(f"function result is {result}")
        return result
...
>>> sum(2,4)
function result is 6
6

>>> print(f”We are outside of the function: {result}")
We are outside of the function: 0
```

First Class Functions

- It is an instance of an Object type
- Functions can be stored as variable
- Pass First Class Function as argument of some other functions
- Return Functions from other function
- Store Functions in lists, sets or some other data structures.

```
def power(base_num, exponent):
    return base_num ** exponent


def my_math(method, exponents):
    result = list()
    for item in exponents:
        result.append(method(2,item))
    return result

answer_list = my_math(power, [3,4,5,6,7,8])
print(answer_list)
[8, 16, 32, 64, 128, 256]
```

Outer and inner functions

```
def outer_function():
    message = 'Hi!!'

    def inner_function():
        print(message)
    return inner_function()

outer_function()
Hi!!


inner_function()
NameError: name 'inner_function' is not defined
```

Decorators

Why? To add new functionality to an existing object without modifying its structure.

```
def decorator_function(original_function):
    def wrapper_function():
        print(f”Hi, I'm wrapper")
        return original_func()
    return wrapper_function

@decorator_function
def display():
    print("display some text")

display()
```

Decorators with parametres

```
def decorator_function(original_func):
    def wrapper_function(*args, **kwargs):
        print(f"I'm wrapper")
        return original_func(*args, **kwargs)
    return wrapper_function

@decorator_function
def display():
    print("display some text")

display()


@decorator_function
def person_info(name, age):
    print(f"The name is {name}, age is {age}")

person_info('Vasya', 30)
```

Class

3 principals of Object-oriented programing

Encapsulation is achieved when each object keeps its state private, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.

Inheritance. Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy.

Polymorphism. Objects are designed to share behaviours and they can take on more than one form.

Creating a class

```
class Car:

    color = 'gray'

    def __init__(self, manufacturer, model, year, vendor):
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.vendor = vendor

mycar = Car('Ford', 'Focus', 2012, 'Horns and hooves')
othercar = Car('Mazda', '3', 2014, 'Hooves and Horns')

print(mycar.vendor)
print(othercar.year)
print(othercar.color)
```

Class methods. *@staticmethod* and *@classmethod*

```
class Car:
    def __init__(self, manufacturer, model, year, vendor):
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.vendor = vendor

    @staticmethod
    def car_type(type):
        if type == 'sport':
            print("it's time for race")
        else:
            print("you need to ride calm")

Car.car_type('SUV')
```

```
class Car:
    def __init__(self,year, vendor):
        self.year = year
        self.vendor = vendor

    def getinfosold(self):
        print(f"your car sold by {self.vendor} in {self.year}")

    @classmethod
    def get_solf_info(cls, info):
        for car in info:
            year = carinfo[car][0]
            vendor = carinfo[car][1]
        return cls(year, vendor)

carinfo = {'car1': [2020, 'ABC']}

infromation = Car.get_solf_info(carinfo)
infromation.getinfosold()
```