Пакеты и модули

Классы и объекты

Наследование

Полиморфизм методов

Модуль

- это отдельный файл с программным кодом на языке Python, который создается один раз и далее может быть использован программами многократно.
- Для формирования модуля необходимо создать обычный текстовый файл с расширением *py и записать в него целевые программные инструкции.
- Название файла при этом будет представлять имя модуля, а сам модуль после создания станет доступным для использования либо в качестве независимого сценария, либо в виде расширения, подключаемого к другим модулям, позволяя тем самым связывать отдельные файлы в крупные программные системы.

Все модули в Python можно разделить на три основные категории:

- стандартная библиотека (от англ. *standard library*)

https://docs.python.org/3/library/index.html

- сторонние модули (от англ. *3rd party*) – более 90 000 модулей и пакетов, которые не входят в дистрибутив Python, но могут быть установлены из https://pypi.org/ официального сайта с помощью утилиты pip;
- пользовательские модули

Как импортировать модули в Python?

- Самый простой способ импортировать модуль в *Python* это воспользоваться конструкцией:
- ***import имя_модуля***
- Импорт и использование модуля *math*, который содержит математические функции, будет выглядеть вот так.

```
>>> import math
>>> math.factorial(5)
120
```

Импортирование некоторых объектов

- **from имя_модуля import имя_объекта**

```
>>> from math import cos
>>> cos(3.14)
0.999998731727539
```

- При этом импортируется только конкретный объект (в нашем примере: функция cos), остальные функции недоступны, даже если при их вызове указать имя модуля.

Импортировать все объекты из модуля

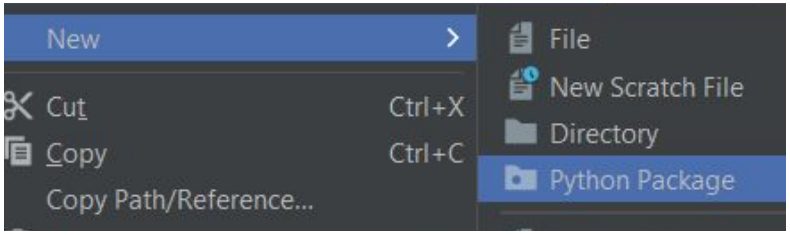
- Если необходимо импортировать все фукнции, классы и т.п. из модуля, то воспользуйтесь следующей формой оператора *from … import …*
- ***from имя_модуля import ****

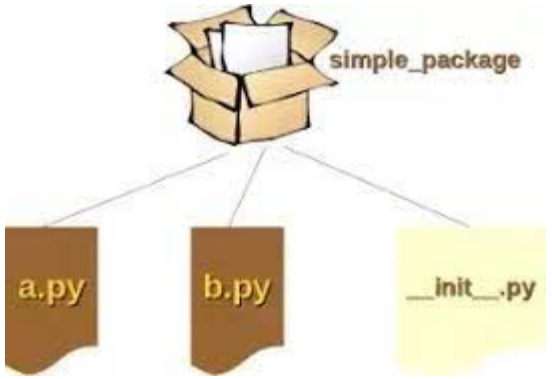
```
>>> from math import *
>>> cos(pi/2)
6.12323399573676e-17
>>> sin(pi/4) 0.707106781186547
>>> factorial(6)
720
```

Импорт собственных модулей в Python.

- https://youtu.be/oSNc_ZRWNzI?si=tVdj4haYkllhAn7o

Пакеты





- Пакет в *Python* – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл *__init__.py*.
- Пакеты используются для формирования пространства имен, что позволяет работать с модулями через указание уровня вложенности (через точку).
- Для импортирования пакетов используется тот же синтаксис, что и для работы с модулями.

Примеры работы с пользовательскими пакетами и модулями

- https://youtu.be/VCRxOdCueqM

Задание 1

- Создать модуль Matrix5 с функциями :
- Создает и заполняет матрицу 5*5 случайными числами.
- Создает матрицу 5*5 состоящую из чисел фибоначчи
- Заполняет матрицу 5*5 по диагонали единицами, все остальное заполняет нулями

Задание 2

- Вызвать функции модуля Matrix5 в модуле Matrix
- Создать пакет Vektor рядом с модулями Matrix5 и Matrix.
- Внутри пакета Vektor создать модуль Vektor10, в котором находится функции:
- Заполняет список случайными числами и выводит на экран
- Заполняет список числами sin(x). X меняется от 0 до 1.
- Вызвать методы модуля Vektor10 в Matrix.

Классы и объекты

- **Класс** — тип, описывающий устройство объектов. **Объект** — это экземпляр класса.
- Класс может состоять из атрибутов и методов.

```
class MyClass:
    # i - атрибут класса,  f – метод класса
    i = 12345
    def   f(self):
        return 'hello world', MyClass.i+5
# `MyObj` - это объект класса
MyObj = MyClass()
print(MyObj.i)
print(MyObj.f())
```

Вывод :

12345

('hello world', 12350)

Конструктор класса

- Конструктор используется для инициализации атрибутов класса. Для этого в классе Python определятся специальный метод "**конструктор класса**" с именем __init__(), который выполняется при создании экземпляра класса.
- Метод .__init__() принимает новый объект в качестве первого аргумента self.

```
class MyClass:
    def __init__(self):
        self.j = 0
        self.i = 1
    def f(self):
        return self.j+self.i

MyObj = MyClass()
print(MyObj.f())   #1
```

Мы можем изменить атрибуты класса в программе

```
class MyClass:
    k = 5
    def __init__(self):
        self.j = 0
        self.i = 1
    def f(self):
        return self.j+self.i

MyObj = MyClass()
MyObj.k = 7
MyObj.j = 10
print(MyObj.f())   #11
print(MyObj.k)    # 7
```

Конструктор класса

- Метод __init__() может иметь аргументы для большей гибкости. В этом случае аргументы, переданные оператору создания класса, передаются в метод __init__().

```
class Animals:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f'имя-{self.name}, возраст-{self.age} ')
rex = Animals("Rex", 2)
rex.info()  #имя- Rex, возраст- 2
```

- конструктор класса .__init__() не должен явно возвращать ничего отличного от None, иначе будет вызываться исключение TypeError.

```
class Animals:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        return   f'имя-{self.name}, возраст-{self.age}‘
rex = Animals("Rex", 2)
```

Traceback (most recent call last):
File "/home/main.py", line 6, in <module>
rex = Animals("Rex", 2)
TypeError: __init__() should return None, not 'str'

Конструктор при наследовании

```
class Person:
    def __init__(self, name, birth_date):
        self.name = name
        self.birth_date = birth_date

class Employee(Person):
    def __init__(self, name, birth_date, position):
        super().__init__(name, birth_date)
        self.position = position

john = Employee("John Doe", "2001-02-07", "Разработчик Python")
print(john.name) # 'John Doe'
print(john.birth_date) # '2001-02-07'
print(john.position) # 'Разработчик Python'
```

Создание экземпляра класса через метод __new__

- До этого мы создавали экземпляр класса обычным способом
- MyObj = MyClass()

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        return super().__new__(cls)
```

Мы можем переопределить метод __new__, добавив что то еще. Его используют когда при создании экземпляра класса необходимо совершить дополнительные действия

Пример простой

```
class A(object):
    def __new__(cls):
        a = int(input("enter the number "))
        b = int(input("enter the number "))
        print(a+b)

        return super(A, cls).__new__(cls)

a = A()
```

Результат:

enter the number 5
enter the number 4
9

Пример - Паттерн синглтон

```
class Singleton(object):
    obj = None    # Атрибут для хранения единственного экземпляра
    def __new__(cls, *dt, **mp):        # класса Singleton.
        if cls.obj is None:             # Если он еще не создан, то
            cls.obj = object.__new__(cls, *dt, **mp)  # вызовем __new__ родительского класса
        return cls.obj                  # вернем синглтон

obj = Singleton()
obj.attr = 12
print(obj.attr)
print(obj)
new_obj = Singleton()
print("аттрибут нового объекта", new_obj.attr)
print(new_obj)
print(new_obj is obj)  # new_obj и obj - это один и тот же объект
```

```
from random import choice

class Pet:
    def __new__(cls):
        # выбираем класс случайным образом
        other = choice([Dog, Cat, Bird])
        # подставляем вместо собственного класса `cls` случайно выбранный `other`
        instance = super().__new__(other)
        print(f"Я {type(instance).__name__}!")
        return    instance
class Dog:
    def communicate(self):
        print("Гав! Гав!")
class Cat:
    def communicate(self):
        print("Мяу! Мяу!")
class Bird:
    def communicate(self):
        print("Чик! Чирик!")

pet1 = Pet()
pet1.communicate()
pet2 = Pet()
pet2.communicate()
```

Пример не простой

Инкапсуляция

- Мы можем ограничить доступ к методам и переменным, что предотвратит модификацию данных — это и есть инкапсуляция. Приватные атрибуты выделяются нижним подчеркиванием: одинарным _ или двойным __.

Инкапсуляция пример

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print(f"Цена продажи: {self.__maxprice}")
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# изменение цены
c.__maxprice = 1000
c.sell()
# используем функцию изменения цены
c.setMaxPrice(1000)
c.sell()
```

Вывод:

Цена продажи: 900
Цена продажи: 900
Цена продажи: 1000

- Мы объявили класс Computer.
- Затем мы использовали метод __init__() для хранения максимальной цены компьютера. Затем мы попытались изменить цену — безуспешно: Python воспринимает __maxprice как приватный атрибут.
- Как видите, для изменения цены нам нужно использовать специальную функцию — setMaxPrice(), которая принимает цену в качестве параметра.

Полиморфизм методов

- **Полиморфизм** — особенность ООП, позволяющая использовать одну функцию для разных форм (типов данных).
- При наследовании полиморфизм осуществляется с помощью переопределения метода **(method overriding)** родительского класса

```
class Animals:
    def __init__(self, name, age):
            self.name = name
            self.age = age
    def voice(self):
            print("голос животного")
    def info(self):
            print(f'имя {self.name} возраст {self.age}')
class Dog(Animals):
    def voice(self):
            print("собака лает")
    def info(self):
            print(f'имя собаки {self.name} возраст {self.age}')
class  Cat(Animals):
    def voice(self):
            print("кошка мяукает")

sharik = Dog("Sharik", 5)
sharik.info()
sharik.voice()
rex = Animals("Rex", 2)
rex.info()rex.voice()
murka = Cat("Murka", 4)
murka.info()
```

Абстрактные классы

- Абстрактный класс не содержит всех реализаций методов, необходимых для полной работы, это означает, что он содержит один или несколько абстрактных методов. Абстрактный метод - это только объявление метода, без его подробной реализации.
- Абстрактный класс предоставляет интерфейс для подклассов, чтобы избежать дублирования кода. Нет смысла создавать экземпляр абстрактного класса.
- Производный подкласс должен реализовать абстрактные методы для создания конкретного класса, который соответствует интерфейсу, определенному абстрактным классом. Следовательно, экземпляр не может быть создан, пока не будут переопределены все его абстрактные методы.

Абстрактные классы

```
from abc import ABC, abstractmethod

class Animals(ABC):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @abstractmethod
    def voice(self):
        print("голос животного")
    @abstractmethod
    def info(self):
        print(f'имя {self.name} возраст {self.age}')
```

```
class Dog(Animals):
    def voice(self):
        print("собака лает")
    def info(self):
        print(f'имя собаки {self.name} возраст {self.age}')
class  Cat(Animals):
    def voice(self):
      print("кошка мяукает")

sharik = Dog("Sharik", 5)
sharik.info()
sharik.voice()
murka = Cat(“Мурка”, 4)
murka.voice()
#TypeError: Can't instantiate abstract class Cat with abstract method info
rex = Animals("Rex", 2)
rex.info()
rex.voice()
#TypeError: Can't instantiate abstract class Cat with abstract method info
```