Языки программирования Лекция №11-12

Куникеев Айдын

Модули и пакеты

- Разделение частей кода по разным файлам или папкам
  - Пакеты - папки где хранятся python файлы
  - Модули - python файлы
- Импортируются с ключевым словом import:
  - import math
  - import <module name> (если модуль находится в одной папке)
  - from <package name> import <module name> (выгрузка модуля из папки)
    - или же
    - import <package name>.<package name>.<module name> (многоуревневая папка)
  - dir(<module name>) – выдает все функций и поля модуля
- Желательно чтобы каждый класс находился в разных модулях

ООП

- Объектно-ориентированное программирование – методология программирования, основанная на представлений блоков программы как объекты

- Основные понятия ООП:
  - Класс
  - Объект
  - Наследование
  - Инкапсуляция
  - Полиморфизм

Класс

- Универсальный тип данных
- Состоит из атрибутов (поля) и функций (методы)
- Класс является шаблоном объекта
- Классы создаются с ключевым словом ***class***

Класс в Python

- class <название> [([класс1] [, класс2], …)]:
  аттрибуты (свойства)
  методы (функций)

```
class BirClass:
    def _init_(self): #Конструктор
        self.x = 10 #Только с помощью self делаются атрибуты
        y = 100 #Без self это просто временная переменная
    def adis(self): #Метод обязательно имеет self, если не имеет self, то она вообще нигде не видна
            print(self.x**2)

bc = BirClass() #Объект
```

Создание классов

- class Person:
  - def run(self, speed):
    - print(speed)
  - def walk(self, speed):
    - print(speed)
  - def eat(self, food):
    - print(food)

__init__() и __del__()

- __init__(self) –конструктор, self обязателен, кроме self разрешается задавать другие параметры:
  - def __init__(self, x,y,z):
    - self.x = x #виден во всем классе
    - y = y #виден только в конструкторе
    - self.z = z
- __del__(self): - деконструктор, вызывается автоматический при удалений объекта. Если был создан вами, то выполнится действие при удалений конструктора
  - def __del__(self):
    - <әрекет>
- Если конструктор __init__() не был создан, то Python создаст по умолчанию

self

- Ключевое слово ***self*** позволяет обращаться к сомому объекту, созданная от кокого-либо класса
- При вызове методов ***self*** как параметр не указывается
- Обрашение к методам и полям внутри класса:

- Class Person:
  - def __init__(self):
    - self.name = None
    - self.birthday = None
    - self.gender = None
    - self.hobby = “Camping”

Метод __init__(конструктор) или как создать объект с параметрами

- Class Person:
  - def __init__(self):
    - self.name = None
    - self.birthday = None
    - self.gender = None
    - Age = 10
  - def run(self, speed):
    - print(speed)
  - def walk(self, speed):
    - print(speed)
  - def eat(self, food):
    - print(food)
- ***init*** позволяет объектам заранее присвоить

Несколко параметров в функциях *args и **args

- def go(**args):
  - Возможность задавать любое количество параметров, но сама функция не меняется
  - Например когда нет параметров go() -> прогулка со скоростью 5km/h
  - 1 параметр go(20)-> пробежка на скорости 20km/h
  - 2 параметра go(“Bike”, 30) -> езда на велике со скоротью 30km/h

- * - возвращает в виде tuple
- ** - возвращает в виде словаря

Класс разновидными параметрами

```
class Cup:
        def __init__(self, **argv):
                self.color = None
                self._content = None
                if len(argv)==1:
                        self.color = argv['color']
                elif len(argv)==2:
                        self.color = argv['color']
                        self._content = argv['content'] # protected variable
        def fill(self, beverage):
                self._content = beverage
        def empty(self):
                self._content = None
cup = Cup()
cup1 = Cup(“Black”)
cup2 = Cup(“Red”, “tea”)
print(cup.color, cup._content) #None None
print(cup1.color, cup1._content) #Balck None
print(cup2.color, cup2._content) #Red tea
```

- В данном классе разрешается задавать от 0 до 2 параметров

Объект

- Эземпляр класса

- Класс:
  - Человек
    - Свойства ФИО
    - Свойства Дата рождения
    - Свойства Пол
    - Функция Бегать
    - Функция Ходить
    - Функция Кушать

- Объект:
  - Серик

Создание Объекта

- serik = Person() #Эземпляр класса
- serik.run(20) #Присвоение значения к параметрам метода и вызов метода
- serik.walk(10) #Присвоение значения к параметрам метода и вызов метода
- serik.eat(“Bes barmaq”) #Присвоение значения к параметрам метода и вызов метода

Функций объекта

- getattr(<Объект>, <атрибут> [, <значение по умолчанию>]) – возвращает значение атрибута, или присваивает значение атрибуту и возвращает
- setattr(<Объект>, <атрибут>, < значение по умолчанию >) – присваение значения атрибуту или создает атрибут если задается не существующий атрибут
- delattr(<Объект>, <атрибут>) – удаляет атрибут
- hasattr(<Объект>, <атрибут>) – проверяет существует ли атрибут

Инкапсуляция

- Возможность класса, который позволяет скрывать методы или же поля от объектов, от наследуемых классов

- Public – общедоступный, и объекты и наследуемые классы
  - Методы и поля которые не имеют в начале символ “__” или “_” являются public
  - Часто используют для хранения статичных данных от внешнего изменения
- Private – доступен только классу
  - Методы и поля которые имеют в начале символ “__” являются private

Private

- Class Person:
  - def __init__(self):
    - self.name = None
    - self.birthday = None
    - self.gender = None
    - self.__specialty = “Developer” #private
  - def run(self, speed):
    - print(speed)
  - def walk(self, speed):
    - print(speed)
  - def eat(self, food):
    - print(food)
  - def go(self, speed):
    - if speed>15:
      - self.run(speed)
    - else:
      - self.walk(speed)

Видимость

- По умолчанию все public
- Private переменные имеют __ (двойной) символ перед названием переменной(атрибут) или функций(метод)
  - Они не будут видимы для наследуемых классов

- def __init__(self):
  - self.__p = 100
- def __adis1(self, x,y):
  - print(x+y)

```
class BirClass:
    def __init__(self,x,y):
        self.x = x
        self.y = y
        print(x**y)
    def __adis(self):
        print(self.x**2)
```

```
class TagyBirClass(BirClass):
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
        super().__init__(self.x,self.y)
    def todo(self):
        super().__adis()
        print(self.x+self.y+self.z)
tbc = TagyBirClass(10,20,30)
tbc.todo()
```

Полиморфизм

- Использования одного и того же объекта, функции, класса, переменной разными путями

- Также Полиморфизм означает одинаковое название в разных классах

Полиморфизм

- Также Полиморфизм означает одинаковое название в разных классах

- class Person:
  - def go(self):
    - print(“People walk with two legs”)
- class Dog:
  - def go(self):
    - print(“Dogs walk with four legs”)

- def walk(creation):
  - creation.go()

- per = Person()
- dog = Dog()

- walk(per) #People walk with two legs
- walk(dog) #Dogs walk with four legs

Наследование

- Возможность класса, где класс может унаследовать свойства (методы и поля) существуюшего класса

- Наследуемый класс – базовый или родительский или же супер класс
- Класс который наследует – потомок или производный класс
- Наследование в python можно делать из нескольких классов
- Наследуемые классы указываются в скобках

Наследование

- class <название>([класс1] [,класс2],…)

- Можно тремя способами вызвать конструктор родительского класса:
  - <Родительский класс>.__init__(self)
  - super().__init__(self)
  - super(< Родительский класс>, self).__init__()
- Будьте осторожны с одинаковыми названиями классов и методов

```
class BirClass:
    def __init__(self,x,y):
        self.x = x
        self.y = y
        print(x**y)
    def adis(self):
        print(self.x**2)



class TagyBirClass(BirClass):
    def __init__(self,x,y,z):
```

Функция super()

- Позволяет обращаться к параметрам родительского класса

Функция super()

```
class Person:
    def __init__(self, name, birthday, gender):
        self.name = name
        self.birthday = birthday
        self.gender = gender
    def walk(self,speed):
        print(self.name, "walking on speed ", speed)
    def talk(self, about):
        print(self.name, "talking about ", about)
```

```
class Developer(Person):
    def __init__(self, language, level, name, birthday, gender):
        self.language = language
        self.level = level
```

Наследования

- Если наследник повторяет название методов от родительских – то это называется перезапись метода (у наследника будет отличаться от родительских)
- Если у родителей одинаковые название методов, то наследник возмет тот который стоит первый в скобках
- Если родитель имеет private поля или методы, она не будет видна для потомка, но к ним можно обрашаться:
  - _<название класса>__<название метода или полей>

Private метод и потомок

```
class Person:
    def __init__(self, name, birthday, gender):
        self.name = name
        self.birthday = birthday
        self.gender = gender
    def walk(self,speed):
        print(self.name, "walking on speed ", speed)
    def talk(self, about):
        print(self.name, "talking about ", about)
    def __playingChess(self):
        print("Playing chess")

class Developer(Person):
    def __init__(self, language, level, name, birthday, gender):
        self.language = language
        self.level = level
        super(Developer, self).__init__(name, birthday, gender)
    def coding(self):
        print(self.name, "coding on ", self.language)
    def level(self):
        if level>5:
```

Private метод и потомок

```
class Person:
    def __init__(self, name, birthday, gender):
        self.name = name
        self.birthday = birthday
        self.gender = gender
    def walk(self,speed):
        print(self.name, "walking on speed ", speed)
    def talk(self, about):
        print(self.name, "talking about ", about)
    def __playingChess(self):
        print("Playing chess")

class Developer(Person):
    def __init__(self, language, level, name, birthday, gender):
        self.language = language
        self.level = level
        super(Developer, self).__init__(name, birthday, gender)
    def coding(self):
        print(self.name, "coding on ", self.language)
```

Специальные функций: __call__()

- __call__() – позволяет обращаться к класссу как к функций

- class BirClass:
- def __init__(self,x,y):
- self.x =
- self.y = y
- print(x**y)
- def adis(self):
- print(self.x**2)
- def __call__(self):
- print(self.x, self.y)

- bc = BirClass(10,20)
- bc() #10 20

Специальные функций :

- __getattr__(self, <атрибут>) – Вызывается при обращений к атрибуту
  - BirClass :
    - def __geattr__(self, attr):
      - print(“something”)
  - print(bc.ppp)
- __getattribute__(self, <атрибут>): Вызывается при обращений к существующему атрибуту
- __setattr__(self, <атрибут>, <мән>) – Вызывается при попытке изменения значения существующего атрибута. Использование: self.__dict__[<атрибут>] = <значение>
  - def __setattr__(self, attr, val):
    - self.attr = val
    - print(self.__dict__)
  - bc.x = 10000

Специальные функций :

- __delattr__(self, <attribute>) – вызывается при попытке удаления атрибута
  - def __delattr__(self, attr):
  - print("Something")

  - bc = BirClass(10,2)
  - delattr(bc, "x")
- __len__(self) – len() – вызываетя при использований фукнций len на экземпляр класса
  - def __len__(self):
    - return 10
  - len(bc)

Специальные функций :

- __bool__(self), __int__(self), __float__(self), __complex__(self) – вызвается при попытке изменения типа объекта
- __round__(self, n) – вызывается при попытке использования функции round() к объекту
- __str__(self) – вызывается при попытке использования функции print() () к объекту

Специальные функций

- a объект
- a+b – a.__add__(b)
  - def __add__(self, attr):
  - 	return self.x+attr
  - bc = BirClass(10,2)
  - print(bc+1000) #1010

  - def __add__(self, attr):
  - 	d = BirClass(self.x+attr.x, self.y+attr.y)
  - 	return d
  - bc = BirClass(10,2)
  - dc = BirClass(20,30)
  - print((bc+dc).y)
- b+a – a.__radd__(b) – сумма с правой стороны

Определение операторов для класса

- a+=b –> a.__iadd__(b)
- a-b –> a.__sub__(b)
- b-a -> a.__rsub__(b)
- a-=b -> a.__isub__(b)
- a*b -> a.__mul__(b)
- b*a -> a.__rmul__(b)
- a*=b -> a.__imul__(b)
- a/b -> a.__truediv__(b)
- b/a -> a.__rtruediv__(b)
- a/=b -> a.__itruediv__(b)
- a//b -> a.__floordiv__(b)
- b//a -> a.__rfloordiv__(b)
- a//=b -> a.__ifloordiv__(b)

Определение операторов для класса

- a == b -> a.__eq__(b)
- a != b -> a.__ne__(b)
- a < b -> a.__lt__(b)
- a > b -> a.__gt__(b)
- a <= b -> a.__le__(b)
- a >= b -> a.__ge__(b)
- b in a -> a.__contains__(b)

static методы

- Разрешается вызывать эти функций без объявления экземпляра класса
- @staticmethod
- def func(x,y):
  - return x*y

- BirClass.func(10,20) -> 200

abstract методы

- Данные методы в обязательном порядке должны быть реализованы в наследуемых классах
- @abstractmethod
- def func(x,y):
  - return x*y

- @abstractmethod
- @staticmethod
- def func1(x,y):
  - pass # для создания пустых действии в функций,
    # при вызове func1 ничего не произойдёт

getter, setter, deleter или свойства (property) объекта

- Очень удобно для скрытия private атрибутов
- <свойство> = property(<get>[,<set>[,<del>]])
- атрибут: __x
- “a” – свойство

- class BirClass: