ООП Python

общие свойства объектов в языке программирования

Class

- данные(свойства)

```
prop = ‘value’
```

- методы(действия)

```
def get_prop(self, x, y):
   self.x = x
   self.y = y
```

Добавить св-во

- Class.prop = ‘value’
- setattr(Class, ‘prop’, ‘value’)

Читать св-во

- c = Class.prop - если нет св-ва, возвращает ошибку
- getattr(Class, prop, False) - если нет св-ва, возвращает третий аргумент

Наличие св-ва

- hasattr(Class, prop) - учитывает наследование

Удалить св-во

- del Class.prop - если нет св-ва, возвращает ошибку
- delattr(Class, prop) - если нет св-ва, возвращает ошибку

Пространство имен

- Class.__dict__

Все свойства экз.класса

- dir(obj)

Описание класса, комментарий

- Class.__doc__

Метод класса

@classmethod - работает с атрибутами класса

```
def method_a(cls, arg):
    return cls.x
```

Статический метод

@staticmethod - независимая, сервисная функция; нет дополнительных параментров, кроме тех, которые указываем; не имеют доступа к атрибутам класса и к атрибутам экз.класса; можем вызыват внутри обычных методов

```
def method_b(x, y):
    return x*x + y*y
```

Режим доступа (механизм инкапсуляции)

- attr - публичное свойство
- _attr - режим доступа protected
- __attr - режим доступа private

pip install accessify - защитить методы от внешнего доступа

Магические методы - методы, которые вызываются интерпретатором для выполнения различных операций над объектами

Метод	Что делает
__new__(cls) return super().__new__(cls)	Создание объекта cls ссылается на текущий класс должен возвращать адрес нового созданного объекта
__init__(self)	Инициализатор объекта
__del__(self)	Финализатор объекта
__setattr__(self, key, value)	Автоматически вызывается при изменении свойства key класса
__getattribute__(self, item)	Автоматически вызывается при получении свойства класса с именем item
__getattr__(self, item)	Автоматически вызывается при получении несуществующего свойства item класса
__delattr__(self, item)	Автоматически вызывается при удалении свойства item (не важно существует оно или нет)
__call__(self, *args, **kwargs)	Позволяет экземплярам класса вести себя так, как будто они функции
__str__(self)	Для отображения информации об объекте класса для пользователей
__repr__(self)	Для отображения информации об объекте класса в режиме отладки
__len__(self)	Позволяет применить функцию len() к экземпляру класса
__abs__(self)	Позволяет применить функцию abs() к экземпляру класса
__add__(self)	Для операции сложения

__sub__(self)	Для операции вычитания
__mul__(self)	Для операции умножения
__truediv__(self)	Для операции деления
__eq__(self)	Для равенства ==
__ne__(self)	Для неравенства !=
__lt__(self)	Для оператора <
__le__(self)	Для оператора <=
__gt__(self)	Для оператора >
__ge__(self)	Для оператора >=
__hash__(self)	Вычисляется хэш экземпляра класса
__bool__(self)	Используется в программах, где требуется описать собственные проверки истинности или ложности объектов
__getitem__(self, item)	Получение значения по ключу item
__setitem__(self, key, value)	Запись значения value по ключу key
__delitem__(self, key)	Удаление элемента по ключу key
__iter__(self)	Получение итератора для перебора объекта
__next__(self)	Переход к следующему значению и его считывание
объект-свойство `property` **1. prop = property (getter, setter) 2. @property def name(self): return self.__name @name.setter def name(self, name): self.__name = name @name.deleter def name(self): del self.__name**	Для работы с приватными локальными свойствами экземпляров классов
коллекция `__slots__` `__slots__ = ('x', 'y')`	Ограничивает допустимый набор имен атрибутов объекта только перечисленными именами - ограничение создаваемых локальных свойств - уменьшение занимаемой памяти, атрибут `__dict__` удаляется - ускорение работы с локальными свойствами

Метакласс:

```
class Meta(type):
     def __new__(cls, name, base, attrs):
         attrs.update({'MAX_NUM': 100, 'MIN_NUM': 0})
         return type.__new__(cls, name, base, attrs)

     #def __init__(cls, name, base, attrs):
         #super().__init__(name, base, attrs)
         #cls.MAX_NUM = 100
         #cls.MIN_NUM = 0

class Name(metaclass=Meta):
      get_info(self):
          return (0, 0)

c = Name()
```

```
print(c.get_info)
print(c.MAX_NUM)
```

```
def create_class(name, base, attrs):
        attrs.update({'MAX_NUM': 100, 'MIN_NUM': 0})
        return type(name, base, attrs)

class Name(metaclass=create_class):
      get_info(self):
          return (0, 0)

c = Name()
print(c.get_info)
print(c.MAX_NUM)
```

Dataclass:

```
class Name:
      def __init__(self, name, price):
          self.name = name
          self.price = price
      def __repr__(self):
          return f'Name: {self.__dict__}'
```

```
from dataclasses import dataclass, field

@dataclass
class Name:
      name: str
      price: float
      dims: list = field(default_factory=list)
      length: float = filed(init=False)

      def __post_init__(self):
          self lengt = (...)

c = Name('Book Python', 500)
c.dims.append(10)
```

Дескриптор:

def __*get*__(self, instance, owner) : return…	Дескриптор не данных только считывают данные имеют приоритет, как и атрибуты класса
def __*get*__(self, instance, owner) : return… def __*set*__(self, instance, value) : … def __*del*__(self) : …	Дескриптор данных - это атрибут объекта со связанным поведением, то есть такой атрибут, при доступе к которому его поведение переопределяется методом протокола дескриптора

```
class Integer:
  @classmethod
  def verify_coord(cls, coord):
    if type(coord) != nt:
      raise TypeError('Координата должна быть целым числом')

  def __set_name__(self, owner, name):
    self.name = '_' + name

  def __get__(self, owner, instance):
#     return instance.__dict__[self.name]
    return getattr(instance, self.name)

  def __set__(self, instance, value):
    self.verify_coord(value)
#     instance.__dict__[self.name] = value
    return setattr(instance, self.name, value)


class Point3D:
  x = Integer()
  y = Integer()
  z = Integer()

  def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z

p = Point3D(1, 2, 3)
print(p.__dict__)
```