**Модуль 1. Понятие класса. Инкапсуляция  (4 ак. ч.)**

[Анкета 1](#_Toc146836307)

[Введение 1](#_Toc146836308)

[Парадигмы программирования 7](#_Toc146836309)

[Класс и экземпляр класса 10](#_Toc146836310)

[Конструктор класса 12](#_Toc146836311)

[Данные экземпляра, методы экземпляра и свойства экземпляра 16](#_Toc146836312)

[Понятие атрибута 16](#_Toc146836313)

[Свойства экземпляра класса: инкапсуляция. 19](#_Toc146836314)

[Атрибут класса 21](#_Toc146836315)

[Классные методы (методы класса) 23](#_Toc146836316)

[Практикум 24](#_Toc146836317)

[Решение 25](#_Toc146836318)

[Выводы 26](#_Toc146836319)

[Дополнительный материал 26](#_Toc146836320)

[Финализатор объекта класса 37](#_Toc146836321)

[Магический метод \_\_new\_\_. Пример паттерна Singleton 38](#_Toc146836322)

[Пример паттерна Singleton (учебный) 40](#_Toc146836323)

[Модуль accessify 42](#_Toc146836324)

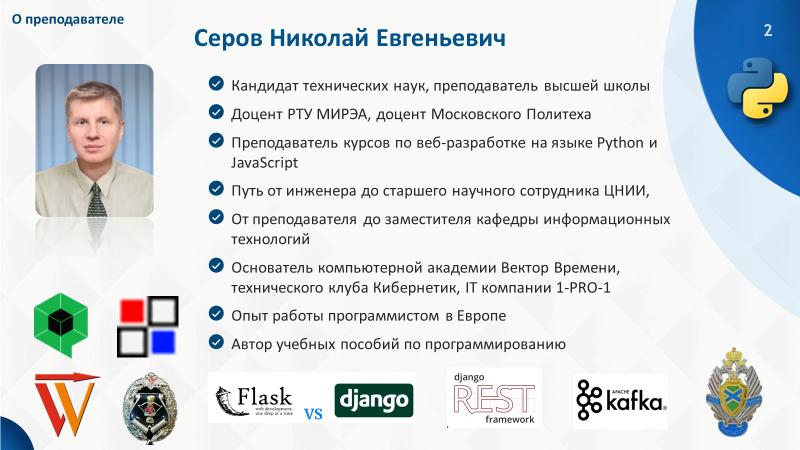
[Магические методы \_\_setattr\_\_, \_\_getattribute\_\_, \_\_getattr\_\_ и \_\_delattr\_\_ 43](#_Toc146836325)

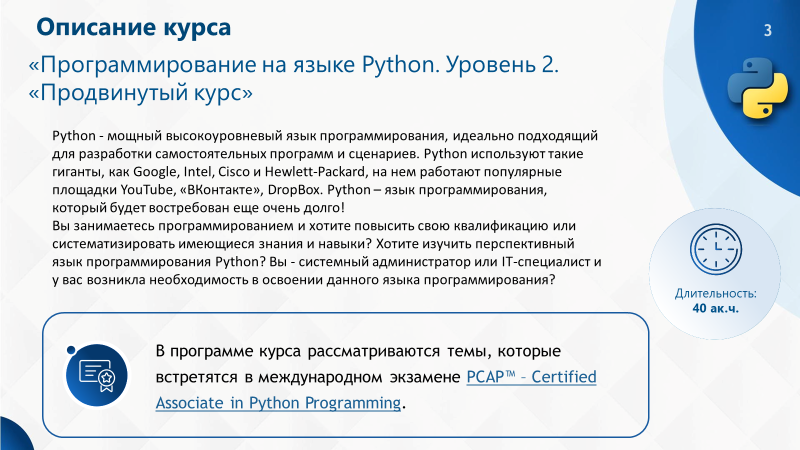
[Практикум: Простые задачи. 45](#_Toc146836326)

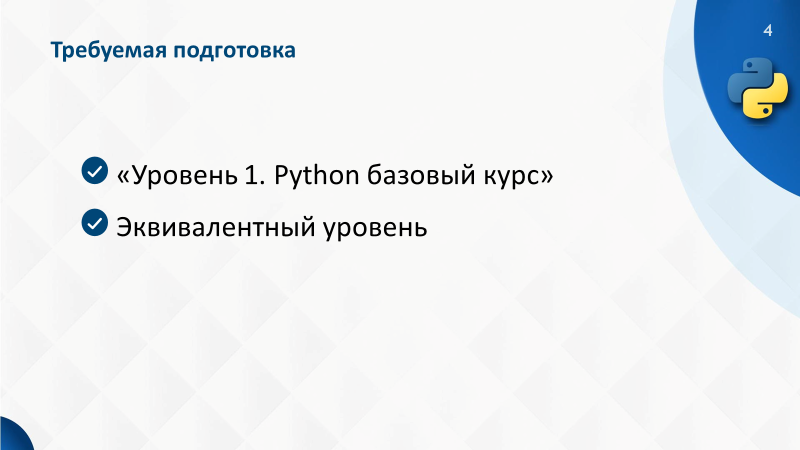
# Анкета

1. Имя, отчество \_\_\_\_ . Место работы (не обязательно) \_\_\_\_\_
2. Я изучаю Python для \_\_\_\_\_\_\_\_\_\_\_
3. Я имею опыт программирования на языках: \_\_\_\_\_\_
4. Прочий IT-опыт (графика, фрейморки, девопс, сисад и т.п) \_\_\_\_\_\_

# Введение



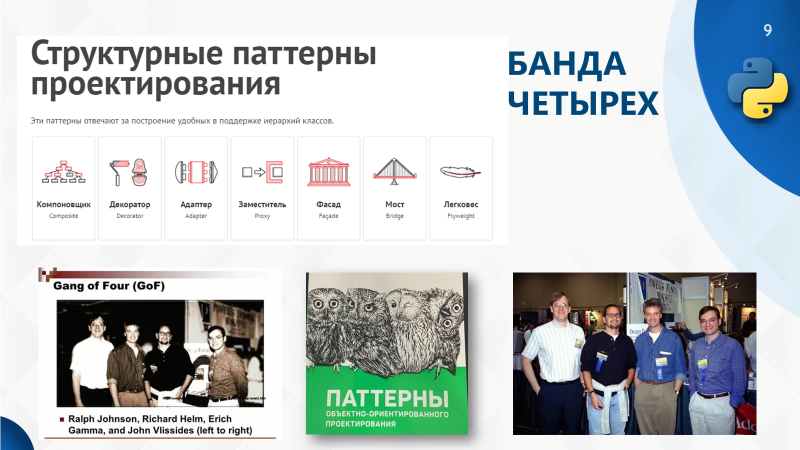


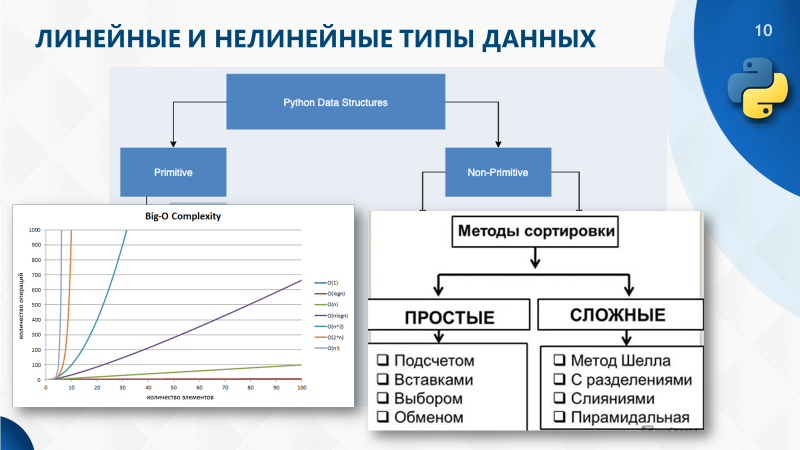


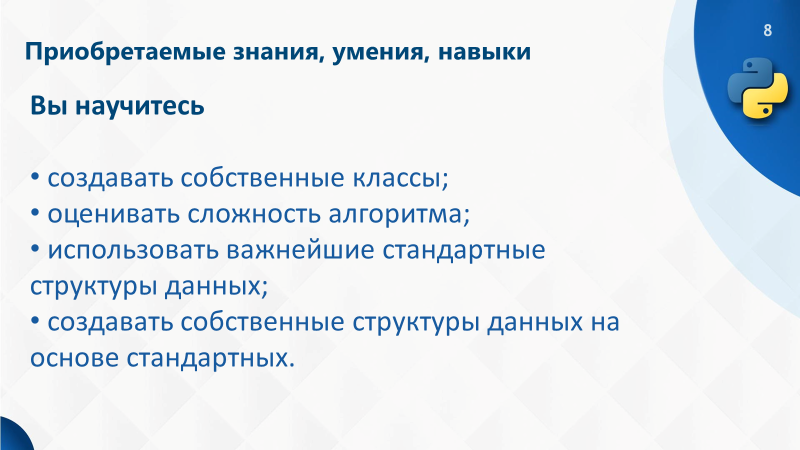


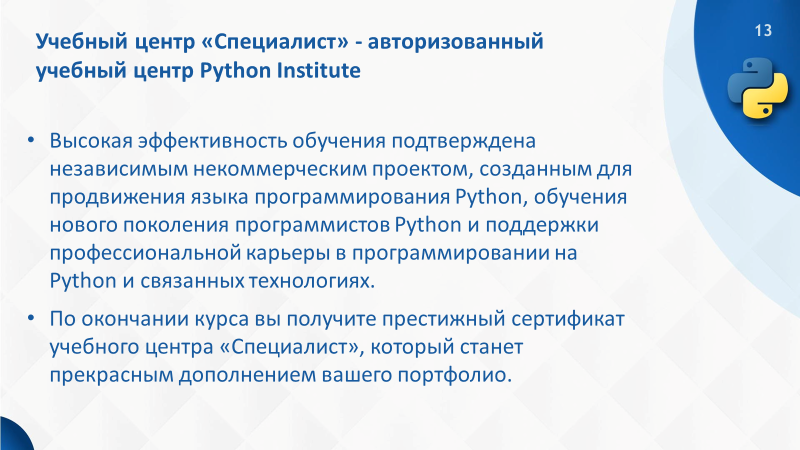


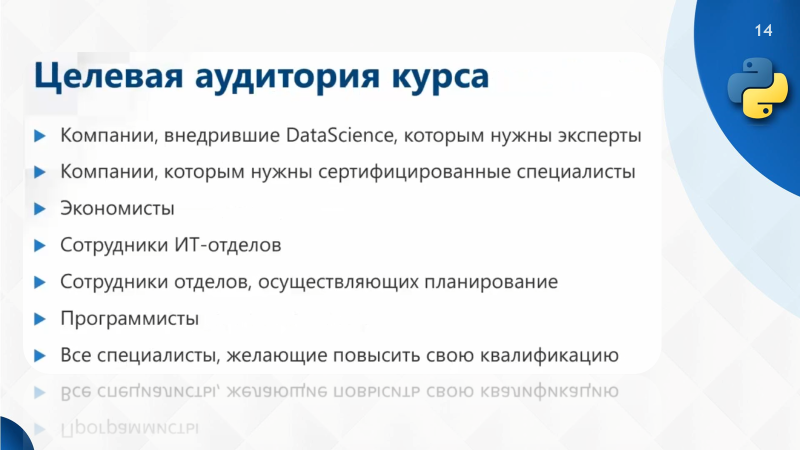




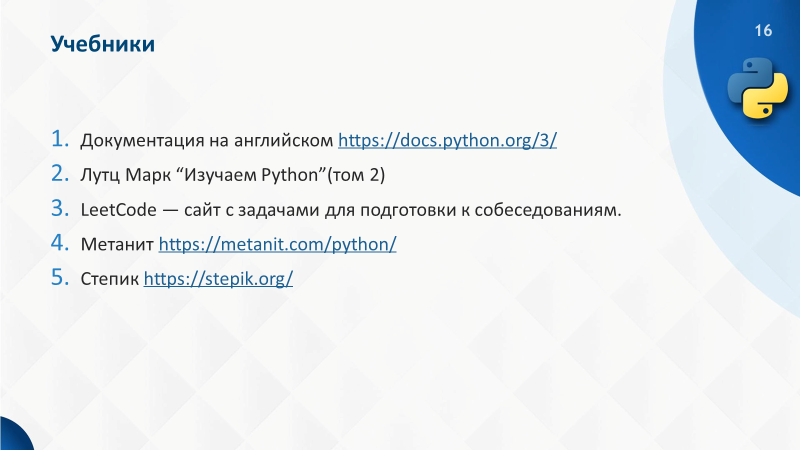


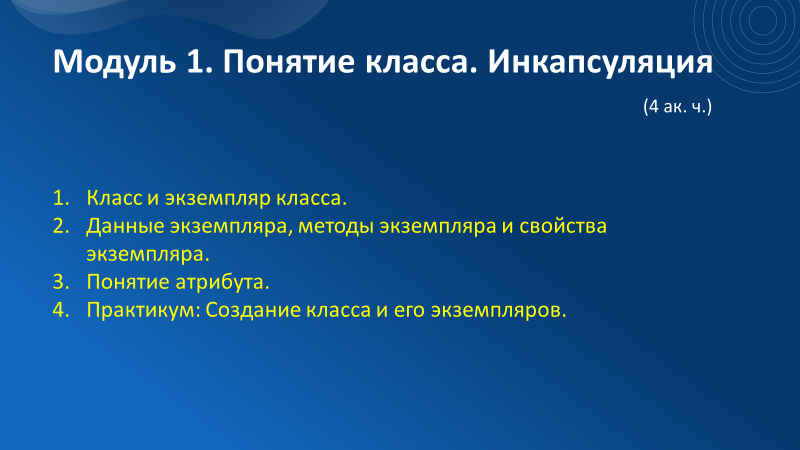








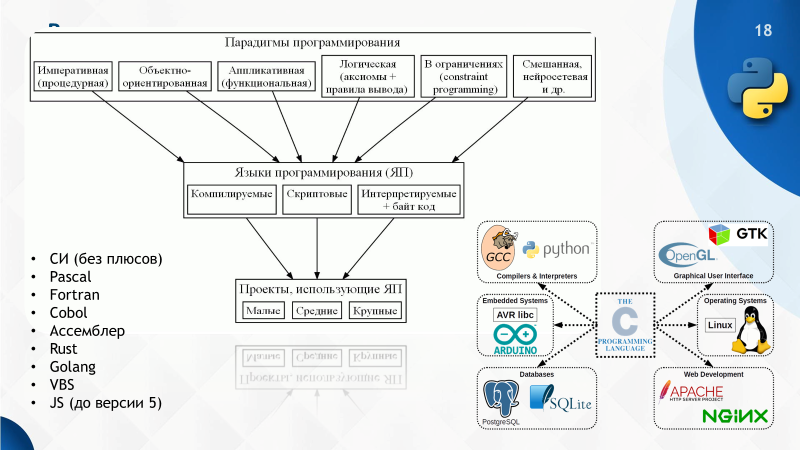




## Парадигмы программирования

Существует три основных парадигмы: структурное, объектно-ориентированное и функциональное. Интересно, что сначала было открыто функциональное, потом объектно-ориентированное, и только потом структурное программирование, но применяться повсеместно на практике они стали в обратном порядке.

Хрестоматийный феномен.



Структурное программирование было открыто Дейкстрой в 1968 году. Он понял, что goto – это зло, и программы должны строиться из трёх базовых структур: последовательности, ветвления и цикла.

Объектно-ориентированное программирование было открыто в 1966 году.

Функциональное программирование открыто в 1936 году, когда Чёрч придумал лямбда-исчисление. Первый функциональный язык LISP был создан в 1958 году Джоном МакКарти.

Каждая из этих парадигм убирает возможности у программиста, а не добавляет. Они говорят нам скорее, что нам не нужно делать, чем то, что нам нужно делать.

Концепция ООП зародилась в 1960-x гг. когда. Но стоит отметить, что вплоть до начала 1990-х программисты могли свободно обходиться без ООП, пока оно не стало доминирующим направлением и внедрено в самый популярный (на тот момент) язык программирования С++. Так что же это такое и почему сейчас знать ООП должен каждый, уважающий себя, начинающий программист?

Для нас важно вот что: речь будет идти не просто о классах и объектах, а о том, как концепция классов и объектов реализуется в языке Python.

Почему это важно?

Важно потому, что сама по себе тема ООП и, более конкретно, классов и объектов, обычно достаточно сложна для понимания даже для тех, кто имеет опыт программирования.

А в случае с языком Python проблемы, скорее всего, возникнут не только у новичков, но и у программистов, знакомых с методами ООП на примере таких языков, как C++, Java или С#.

Для тех, кто знаком с другими объектно-ориентированными языками: в Python класс сам является объектом.

Это интригующее обстоятельство имеет весьма далеко идущие последствия.

Более того, как мы уже знаем, переменные в Python не объявляются, а вводятся в программу путем присваивания значения. Это же правило остается справедливым при работе с классами и объектами.

Отсюда получается, что процедура объявления полей, стандартная для многих языков программирования, в Python просто теряет смысл.

Аналогично, многие привычные (по языкам программирования С++, Java и С#) в ООП моменты окажутся чуждыми для языка Python. Короче говоря, в экзотике недостатка не будет.

Введение

Заказчик требует создать программу чтение с клавиатуры – вывод в терминал



Далее добавить чтение из файла

Результат:

* Грязно
* Много условных выражений
* Сложно понять
* Что делать – вводить объект

Вывод – нас спасет ООП

Вокруг нас все объекты – человек, машина, здания, дома, компьютеры.

И объекты между собой общаются, взаимодействуют.

В программировании общаются между собой только передачей сообщений.

В общем базовая концепция ООП состоит из двух пунктов – всё есть объект и объекты общаются между собой путем передачи сообщений.

Причем Python не 100% ООП-ориентированный язык.

Если есть объект, то его нужно как-то классифицировать, как-то описать.

обычно это сравнивают с чертежом, чтобы построить дом надо сделать чертеж.

Или если вспомнить зоологию, корова – копытные есть — значит класс парнокопытные, класс млекопитающих парнокопытные.

И в ООП такие же классы

class ИмяКласса:  
 pass  
  
type(ИмяКласса)

Мы видим результат <class 'type'>. На самом деле класс – это объект для описания других объектов

В блоке ИмяКласса мы и будем описывать атрибутику, поведение, методы. Вообще всё, что он из себя представляет. Пока он пустой, но мы уже можем создавать объекты этого класс.

Конвеницально принято называть имена классов с большой буквы.

Обращение также как как функции, но не забываем про большую букву

объект = ИмяКласса()  
print (type(объект))

Запустим

<class '\_\_main\_\_.ИмяКласса'>

Как проверить, является ли наш объект экземплярам конкретного класса

isinstance(объект, ИмяКласса)

это функция – предикат, возвращает булев тип. Ну и соответственно возвращает

print (isinstance(объект, list))

# Класс и экземпляр класса

Перейдем к практическому примеру:

Создадим класс Lamp как светильник

class Lamp:  
 pass  
  
lamp1 = Lamp()  
lamp2 = Lamp()  
  
print(isinstance(lamp1, Lamp))  
print(isinstance(lamp2, Lamp))

#True

Лампочки уже есть, но они ничего не могут делать. Зачем они тогда нужны.

Вспомним концепцию ООП – все лампочки общаются сообщениями. Или другой вариант – у объектов есть поведение. Человек может ходить, автомобиль ехать. Лампочка включаться-выключаться.

Человек общается с человеком передачей сообщений. Для этой цели в ООП используются методы. По сути класс – это набор методов.

А методы – ничто иное как обычные функции, просто которые описаны в классе. Все-что мы знаем о функциях здесь легко применить.

Добавим функции включения-выключение лампочки

class Lamp:  
 def switch\_on():  
 print('Лампочку включили')  
  
 def switch\_off():  
 print('Лампочку выключили')

и пошлем сообщение “Лампочка включись”

lamp1.switch\_on()  
lamp1.switch\_off()

но мы видим ошибку

TypeError: Lamp.switch\_on() takes 0 positional arguments but 1 was given

Ошибка типа: Lamp.switch\_on() принимает 0 позиционных аргументов, но был задан 1.

Где это параметр, я ничего не передал. Используем принцип наименьшего удивления

В чем причина? У нас два объекта, а метод у них один. А интерпретатор внутри класса не понимает у какого объекта нужно дернуть метод switch-on/off.

## Конструктор класса

Проблема в области видимости. В глобальной мы всё видим, а класс внутри себя не понимает кто в него стучится.

Поэтому Пайтон всегда неявно сюда первым параметром передает этот объект

lamp1.switch\_on(lamp1)

а метод switch должен принять его

def switch\_on(себяшка):  
 print('Лампочку включили')

т.е. себяшка – обычный аргумент функции. Но по конвенции это дело обычно называют self.

Возникает вопрос – зачем этот self, если он передается во все методы, зачем его передавать, зачем его указывать. Но увы, конвенция, и во всех методах мы обязаны его указывать. Но с другой стороны.

В php и JS таким словом является this. Но причем там оно явно не пишется

Во внешнем мире он lamp или Александр Петрович, а в духовном мире (волшебно-магическом) отец Георгий или self.

Итак перед вами – минимальная концепция ООП, ОО-парадигмы. Открой, закрой, включи.

Обычные функции. Общий принцип

def имя\_метода (self):

pass

объект = Lamp()

объект.имя\_метода()

Технически на этом можно остановиться, но у нас есть засада. Лампочку можно выключать бесконечное множество раз. А каким образом выключать то, что еще не включили.

Значит нам нужен флаг или логическое состоянии. Т.е. кроме поведения, к примеру у человека – бегать, прыгать, есть, говорить, учится, у человека есть состояния – возраст, имя,

Соответственно у объектов тоже есть состоянии. И они называются атрибутами.

Атрибут вроде как обычная переменная. Добавим атрибут состояния в класс Lamp

class Lamp:

def switch\_on(self):

**self.state = True**

print('Лампочку включили')

def switch\_off(self):

**if self.state == True:**

print('Лампочку выключили')

**self.state == False**

**lamp1.switch\_on()**

**lamp1.switch\_off()**

**работает!? Но!**

lamp1.switch\_on()

lamp1.switch\_off()

**lamp1.switch\_off()**

**или просто выключить без включения**

lamp1.switch\_off()

AttributeError: 'Lamp' object has no attribute 'state'

Нет атрибута state у лампы.

Мы можем конечно, что то здесь проверит в switch, поставить if и так далее

Но на самом деле, правильный путь – где-то это состояние проинициализировать.

Инициализацией кода в ООП называется конструктор. Конструктор – специальные метод, который (1) запускается автоматически, (2) запускается при создании объекта

Служебное имя у конструктора \_\_init\_\_. Иногда можно встретить вместо конструктор слова инициализатор, а даже инстанс (в java).

Т.о. создается объект, вызывается его конструктор, записывается в память (как правило это обычная переменная) . Это обычная переменная, все объекты в пайтоне – ссылочного типа.

Здравствуй, волшебство. Привет, дандер-методы.

Эти методы четко определены с помощью двойного подчеркивания до и после имени метода

Как правило все атрибуты создаются, инициализируются в конструкторе

def \_\_init\_\_(self):

self.state = False

т.е по дефолту наша лампочка еще не горит.

Теперь мы можем по-человечески отследить состояние лампочки

def switch\_on(self):

**if not self.state:**

**self.state = True**

**print('Лампочку включили')**

def switch\_off(self):

**if self.state:**

**print('Лампочку выключили')**

**self.state = False**

Поэкспериментируем в консоли. Логика не нарушена, дизайн хороший. Багов и дыр нету?

Краткий итог – self, init

Появление переменных где-то в середине кода – плохой стиль, откуда, чего, куда, зачем. Поэтому и метод init помогает решить эту задачу.

Однако, вдруг нам захочется определить еще какой-нибудь атрибут у лампочки, например этажность

self.floor = 1

Но так явно не совсем удобно, допустим мне бы хотелось явно при создании лампочки, указать этаж, на котором её разместить

lamp1 = Lamp(1)

lamp2 = Lamp(2)

поэтому здесь можно определить стандартный параметр

def \_\_init\_\_(self, floor=0):

self.state = False

self.floor = floor

Таким образом, мы можем напрямую обращаться к его свойствам, атрибутам.

lamp1.state

False

lamp2.floor

2

Однако, если я захочу вывести

print (lamp2)

результат будет ледующий

<\_\_main\_\_.Lamp object at 0x0000025210B5D250>

Но это не очень информативно, но такую возможность упускать не нужно.

print ([1,2,3])

Но нам нужно, чтобы при обращении к лампе команда print вывела состояние лампочки, на каком она этаже, сколько её ресурс и т.п.

И первый раз мы подходим к понятию переопределение методов.

У пайтона есть много служебных методов, и нашем случае мы обращались к методу repr.

От слова representation

def \_\_repr\_\_(self):

return f'Я лампочка на {self.floor} этаже'

на самом деле здесь 2 таких метода, второй метод \_\_str\_\_

и он вызывается только в принте, repr во всех случаях.

def \_\_str\_\_(self):

return f'Я лампочка на {self.floor} этаже'

s = lamp2

print (s)

волшебный дандер метод \_\_repr\_\_ более универсален

Т.о. класс набор методов, которые описывают поведение будущего объекта (экземпляра будущего класса). Также у объекта могут быть атрибуты, состояния. В некоторых книжках очень много акцентов делается на атрибуты. Но атрибуты вторичны. Концепция ООП крутится вокруг методов.

Класс – несколько функций во одном флаконе, коробка в которой хранятся методы и переменные.

Можно легко накосячить и уйти в некуда.

# Погружение в инкапсуляцию

## Данные экземпляра, методы экземпляра и свойства экземпляра

Начнем знакомство с принципом № 1 в ООП – «**Инкапсуляция»**.

У лампочки есть состояние, есть выключатель, и мы не залезаем внутрь выключателя, чтобы понять как он работает.

Не нужно быть сам-себе режиссером – я сам пишу класс и сам его использую. НЕТ!

Класс пишет один человек, а использует его другой. И тот, который использует этот класс, не должен залезать внутрь класса (выключатель эти или телевизор), чтобы понять, что там внутри.

Представьте он на замок закрыт. Нам не должно быть интересно, как работает коробка передач автомобиля. Не лезь – сломаешь

Типичный пример обращения напрямую к состоянию лампочки

lamp1.state

False

lamp1.state = True

lamp1.state

# Понятие атрибута

Существует атрибуты у экземпляров класса и самого класс.

По порядку.

Мы дали напрямую обратиться к атрибуту экземпляра класса. Мы не должны были этого делать – давать доступ напрямую.

В джаве есть поля и свойства (вместо атрибутов). У нас в питоне атрибуты и свойства.

Атрибуты – непосредственно хранят эти свойства. А свойства – это интерфейс доступа к этим значениям.

Как сделать интерфейс доступа к этим значениям. Как это делается в других языках?

Например в Джаве к каждому полю пишется свои методы get state и set state.

def get\_floor(self):

return self.floor

def set\_state(self, f):

self.state = f

Результат

lamp2.get\_floor()

2

lamp2.set\_floor(4)

lamp2.get\_floor()

а если я снова сделаю lamp1.floor я снова получу доступ напрямую к floor.

Для этих целей атрибут можно сделать приватным. Для этого использую ключевые слова, которых в пайтоне нет. Private, protected и public.

А в пайтоне – первое это конвенция, просто начать имя атрибута с подчеркивания.

self.\_floor = floor

это чисто указатель, внутренняя кухня, конвенция.

Если мы хотим действительно сделать свойство закрытым от внешнего мира – физически невозможный доступ к атрибуту, название атрибута нужно начать с двух подчеркиваний. И здесь уже конкретная приватность.

Теперь достучаться до floor из внешнего мира невозможно. Из глобальной видимости его не видно.

Теперь мы можем к нему обратиться с помощью методов … Однако это не удобно, каждый раз обращаться к функциям.

Поэтому в Пайтоне применяется трюк следующего содержания. Есть два способа, два варианта.

Делаем \_\_state приватным. Пишем метод get\_state. По задумке получать состояние можно, изменять состояние нет.

def get\_state(self):

return self.\_\_state

и далее пишем переменную state (вне метода) и с помощью специальной функции property передаю функцию get\_state которая должна выступать в роли геттера.

state = property(get\_state)

Теперь можно получить состояние лампочки

print (lamp1.state)

а поменять состояние нельзя

lamp1.state = True

как это работает? При обращении к lamp1.state в области видимости класса Lamp ищется метод с названием lamp и не находится, но находится переменная state которая через property вызывает функция get\_state, которая возвращает состояние лампы

Это не что иное, как интерфейс доступа. Т.е. атрибуте state имеет атрибут доступа. Но пока у него есть возможность только получить данные.

А теперь тоже самое сделаем с этажностью. Для этого приватим floor. Пишем свойство floor, обращаемся к функции property, внутри которой передаем сначала геттер, потом сеттер. Теперь мы к нашей этажности имеем доступ как на чтение, так и на запись.

def get\_floor(self):

return self.\_\_floor

def set\_floor(self, f):

self.\_\_floor = f

floor = property(get\_floor, set\_floor)

Результат:

lamp2.floor

2

lamp2.floor=4

lamp2.floor

У вас должен возникнуть вопрос. В чем проблема, в чем задача, связанная с этим гетом и сетом.

Зачем я сделал state приватным. По правилам ООП напрямую обращаться к методам нельзя. В данном случае читать можно, изменять нельзя.

Зачем я сделал это с этажом и наградил его сеттером. Ответ и причина кроется в обратной совместимости.

Когда мы ему даем прямой доступ, код от нас идет куда-туда, минуя нас, мы не можем перехватить, что посылается в атрибут состояния или в атрибут этажность. А если туда полетит какая-нибудь абракадабра.

А мы должны данные перехватывать, перед тем как они туда запишутся или уйдут оттуда.

Представьте себе, что-то сломалось или тз поменялось. Допустим этажность нужно писать русскими буквами

lamp1 = Lamp('один')

lamp2 = Lamp('ресепшен')

И тут мы получаем проблему, что новый код, должен писать кириллицу, а старый работает только с целочисленными этажами. А вдруг согласно ТЗ всё нужно писать в верхнем регистре.

Нам нужен в сеттере преобразователь. Т.е. перед тем как записать и отдать нам нужен какой-то код.

Пример из жизни – Я хочу узнать как вас зовут. Я должен вам послать сообщение или спросить – как Вас зовут. Если я к вам обратился – вы ответите Иван Иванович, а если пацан на улице – вы ему Вано скажете, или там отправите его в Монино.

Т.е. если я очень хочу узнать ваше имя, я это не смогу сделать. Не давайте доступа без нужды. Не пишите на одежде как вас зовут.

Подводим итог по первому принципу ООП – инкапсуляция. В Пайтоне она реализована через \_ и \_\_ подчеркивания, а также функции property.

## Свойства экземпляра класса: инкапсуляция.

Общий принцип реализации инкапсуляции.

(параллельно создать еще python-файл, класс c Lamp оставить без изменения).

class Класс:  
  
 def \_\_init\_\_(self, значение = 0):  
 *# нет доступа по соглашению* self.\_protected = значение  
 *# конкретно нет доступа* self.\_\_private = значение  
  
 def get\_private(self):  
 return self.\_\_private  
  
 def set\_private(self, value):  
 self.\_\_private = value  
  
 private = property(get\_private, set\_private)  
  
  
объект = Класс()  
объект.private = 100  
  
print(объект.private)

Пишем объект.private = 100. Ищем его в области видимости методов, потом атрибутов, и находим, а далее через property

объект = Класс()  
объект.private = 100  
  
print(объект.private)

Есть еще вторая возможность. Использовать декораторы для свойств.

class Класс:  
  
 def \_\_init\_\_(self, значение = 0):  
 self.\_\_private = значение  
 *# getter* @property  
 def private(self):  
 return self.\_\_private  
  
 *# setter* @private.setter  
 def private(self, value):  
 self.\_\_private = value

Они используются вот таким образом.

Предположим, что у меня есть атрибут private и я хочу описать для него геттер и сеттер.

Геттер описывается через декоратор property и описываем соответственно, метод с именем, который будет дергаться (вызываться).

Чтобы описать сеттер, декоратор становится именем свойства. @private (имя свойства) точка setter. Таким образом мы имеем два метода с одинаковым именем.

Еще раз посмотрим на код и зафиксируем синтаксис инкапсуляции в Пайтон с помощью декоратора property и имя\_свойства.setter.

@property

def state(self):

return self.\_\_state

@state.setter

def state(self, s):

self.\_\_state = s

Результат:

print (lamp2)

lamp1.state = 6

print (lamp1.state)

lamp1.floor = 2

print (lamp2)

Это все встроенное в Пайтон. Соглашения, конвенция, дзен, философия. Синтаксический сахар.

Однако, мы всё пока крутится вокруг объекта класса, т.е сущности которые класс порождает – лампочки, породы собак, модели автомобилей.

Но в пайтоне все является объектом и даже класс. Поэтому мы можем смело создать атрибуты у самого Класс.

## Атрибут класса

class ИмяКласса:  
 атрибут\_класса = значение\_по\_умолчанию  
  
 def метод(self, параметр):  
 ИмяКласса.атрибут\_класса = параметр

Т.е. они указываются прямо в классе. В чем фича? Атрибут общий для всех порождаемых классом сущностей.

У человека – две руки, у автомобиля – есть кузов, двигатель. Или константа pi, e или число Авогадро.

Или у всех ламочек бренд “Филипс”

class Lamp:

brand = "Philips"

Если мы обращаемся извне, то мы пишем имя класса и атрибут:

объект = ИмяКласса()  
объект.метод(значение)  
  
print(ИмяКласса.атрибут\_класса)

print(lamp2.brand)

а конструкторе (инициализаторе) прописать

def \_\_init\_\_(self, floor=0):

self.\_\_state = False

self.\_\_floor = floor

print(f'Создана лампочка {Lamp.brand}' )

?Оно не принадлежит объекту? Оно унаследовано от …

Для чего еще могу быть использованы атрибуты класс. К примеру, для подсчета общего количества лампочек

class Lamp:

brand = "Philips"

**count = 0**

def \_\_init\_\_(self, floor=0):

self.\_\_state = False

self.\_\_floor = floor

**Lamp.count +=1**

print(f'Создана лампочка {Lamp.brand} № {Lamp.count}' )

Результат

Создана лампочка Philips № 1

Создана лампочка Philips № 2

Создана лампочка Philips № 3

print(f'Всего лампочек - {Lamp.count}')

Всего лампочек – 3

Это довольно удобно.

Кроме того, у класса могут быть не только методы, но и методы. Не что иное как поведение класса.

Причем, у нас есть две возможности – статический и классический (класс) методы.

## Классные методы (методы класса)

class ИмяКласса:  
 атрибут\_класса = 0  
 @staticmethod  
 def static\_method():  
 return ИмяКласса.атрибут\_класса  
  
 @classmethod  
 def class\_method(cls):  
 return cls.атрибут\_класса  
  
объект = ИмяКласса()  
  
объект.static\_method()  
объект.class\_method()

Технический между ними разницы нет. Есть синтаксическая разница. Если мы используем декоратор @staticmethod то обращение идет к классу через точку его метод.

@staticmethod

def static\_method():

return Lamp.count

@classmethod

def class\_method(cls):

return cls.count

print (lamp1.static\_method())

print (lamp1.class\_method())

Если используем класс-метод (в терминах ООП – это тоже классический метод). Но в класс-методе, декоратор автоматом передает имя класс. Некая переменная в которую прилетает имя класса, по соглашению, по конвенции – название её cls ( по аналогии self).

В первом случае имя класс мы пишем руками. Никакого self. Но какая возникает проблема. Подумайте, если мы поменяем имя класса. Что это значит? Нам нужно будет поменять имя класса во всех статических методах.

В случае с класс-методами – этого делать не нужно. Разница в терминах ООП никакой. Кроме того, что необходимо менять название в статик-методах.

Ну и, соответственно, внутри класса мы не можем обращаться к каким-либо объектам. Только к классу. К общему поведению всех объектов. Т.е класс – это тоже самостоятельная единица.

# Практикум

Создайте класс Point, который описывает точку с координатами х и y

В классе необходимо описать:

* конструктор, который принимает в качестве параметров значения для координат x и y
* метод move\_to, который принимает в качестве параметров новые значения для координат x и y
* метод move\_by, который принимает в качестве параметров новые значения для координат x и y относительно текущих значений
* свойства для изменения и получения значений координат x и y

Необходимые условия, которые надо учесть:

* при приведении объекта к строке должна возвращаться строка Я - точка: координата\_x x координата\_y

Как это должно работать

point = Point(10, 20)  
print(point) *# Я - точка: 10 x 20*point.move\_to(100, 200)  
print(point.x, ' : ', point.y) *# 100 : 200*point.move\_by(10, 20)  
print(point.x, ' : ', point.y) *# 110 : 220*point.x = 30  
point.y = 40  
print(point) *# Я - точка: 30 x 40*

# Решение

class Point:

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

def move\_to(self, x, y):

self.\_\_x = x

self.\_\_y = y

def move\_to(self, x, y):

self.\_\_x += x

self.\_\_y += y

def \_\_repr\_\_(self):

return f'Я точка: {self.\_\_x}x {self.\_\_y}'

Продолжение

def get\_x(self):

return self.\_\_x

def set\_x(self, x):

self.\_\_x = x

x = property(get\_x, set\_x)

для y через декораторы

@property

def y (self):

return self.\_\_y

@y.setter

def y(self, y):

self.\_\_y = y

проверяем

p1= Point(1,2)

p1.x = 10

p1.y = 20

print(p1)

Результат:

p1= Point(1,2)

p1.x = 10

p1.y = 20

print(p1)

Ну и до финала, посчитать, сколько у нас создалось таких точек

class Point:

count = 0

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

Point.count += 1

Ну и снова инкапсулируем атрибуты

class Point:

\_\_count = 0

@classmethod

def get\_count\_points(cls):

return cls.\_\_count

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

Point.\_\_count += 1

Сложности начнутся потом, а может не начнутся. Синдром копипаста.

# Выводы

Класс — тип данных описывающий объект

Объект — тип данных, экземпляр класса

Метод объекта — сообщение, с помощью которого объекты общаются друг с другом

Атрибут объекта — хранилище текущего состояния объекта

Свойство объекта — интерфейс доступа к атрибутам объекта

Класс — самостоятельный тип. Может иметь свои методы и атрибуты

# Дополнительный материал

Класс и экземпляр класса

чтобы увидеть все атрибуты класса можно обратиться к специальной коллекции \_\_dict\_\_:

Point.\_\_dict\_\_

Здесь отображается множество служебных встроенных атрибутов и среди них есть наши

Но часто можно встретить, что объект класса и экземпляр класса - это одно и то же. Теперь мы знаем, что это не так.

Для фанатов С++, Java и С# такой терминологический подход может показаться произволом (дело в том, что в этих языках объектом класса обычно называют тот объект, что создается на основе класса). Но мы так поступаем по необходимости и следуем той традиции именования классов и объектов, которая "исторически" сложилась среди разработчиков языка Python и программистов, использующих этот язык.

С помощью функции type мы можем посмотреть тип данных для переменных a или b:

type(a)

Видим, что это класс Point. Эту принадлежность можно проверить, например, так:

type(a) == Point

или так:

isinstance(a, Point)

Point.type\_pt = 'disc'

Или то же самое можно сделать с помощью специальной функции:

setattr(Point, 'prop', 1)

Она создает новый атрибут в указанном пространстве имен (в данном случае в классе Point) с заданным значением. Если эту функцию применить к уже существующему атрибуту:

setattr(Point, 'type\_pt', 'square')

то оно будет изменено на новое значение.

Если же мы хотим прочитать какое-либо значение атрибута, то достаточно обратиться к нему. В консольном режиме это выглядит так:

Point.circle

Но, при обращении к несуществующему атрибуту класса, например:

Point.a

возникнет ошибка. Этого можно избежать, если воспользоваться специальной встроенной функцией:

getattr(Point, 'a', False)

Здесь третий аргумент – возвращаемое значение, если атрибут не будет найден. Эту же функцию можно вызвать и с двумя аргументами:

getattr(Point, 'a')

Но тогда также будет сгенерирована ошибка при отсутствии указанного атрибута. Иначе:

getattr(Point, 'color')

она возвратит его значение. То есть, эта функция дает нам больше гибкости при обращении к атрибутам класса. Хотя на практике ей пользуются только в том случае, если есть опасность обращения к несуществующим атрибутам. Обычно, все же, применяют обычный синтаксис:

Point.color

Наконец, мы можем удалять любые атрибуты из класса. Сделать это можно, по крайней мере, двумя способами. Первый – это воспользоваться оператором del:

**del** Point.prop

Если повторить эту команду и попытаться удалить несуществующий атрибут, возникнет ошибка. Поэтому перед удалением рекомендуется проверять существование удаляемого свойства. Делается это с помощью функции hasattr:

hasattr(Point, 'prop')

Она возвращает True, если атрибут найден и False – в противном случае.

Также удалить атрибут можно с помощью функции:

delattr(Point, 'type\_pt')

Она работает аналогично оператору del.

И, обратите внимание, удаление атрибутов выполняется только в текущем пространстве имен. Например, если попытаться удалить свойство color из объекта b:

**del** b.color

то получим ошибку, т.к. в объекте b не своих локальных свойств и удалять здесь в общем то нечего. А вот в объекте a есть свое свойство color, которое мы с вами добавляли:

a.\_\_dict\_\_

и его можно удалить:

**del** a.color

Смотрите, после удаления локального свойства color в объекте a становится доступным атрибут color класса Point с другим значение ‘black’. И это логично, т.к. если свойство не обнаруживается в локальной области, то поиск продолжается в следующей (внешней) области видимости. А это (для объекта a) класс Point. Вот этот момент также следует хорошо понимать при работе с локальными свойствами объектов и атрибутами класса.

Отмечу, что в любом классе языка Python мы можем прописывать его описание в виде начальной строки, например, так:

**class** Point:

    "Класс для представления координат точек на плоскости"

    color = 'red'

    circle = 2

В результате, специальная переменная:

Point.\_\_doc\_\_

будет ссылаться на это описание. Обычно, при создании больших программ, в ключевых классах создают такие описания, чтобы в последующем было удобнее возвращаться к ранее написанному коду, корректировать его и использовать, не обращаясь к специальной документации.

Практические задания:

Задание 1. Объявите класс с именем DataBase, который бы хранил в себе следующую информацию:

pk: 1

title: "Классы и объекты"

author: "Александр Пушкин"

views: 14356

comments: 12

Имена переменных (атрибутов класса) используйте такие же (pk, title, author, views и comments) с соответствующими значениями.

Решение:

class DataBase:

pk = 1

title = "Классы и объекты"

author = "Александр Пушкин"

views = 14356

comments = 12

Задание 2. Объявите класс с именем Goods и пропишите в нем следующие атрибуты (переменные):

title: "Мороженое"

weight: 154

tp: "Еда"

price: 1024

Затем, после объявления класса, измените его атрибут price на значение 2048 и добавьте еще один атрибут:

inflation: 100

Решение:

class Goods:

title ="Мороженое"

weight = 154

tp = "Еда"

price = 1024

Goods.price = 2048

setattr(Goods, "inflation", 100)

## Данные экземпляра, методы экземпляра и свойства экземпляра

Что же пишут в теле класса? В теле класса, как правило, описывают методы. Метод – это та же функция, только вызываться она будет из экземпляра класса. Мы с методами уже имели дело неоднократно. Но только раньше мы использовали готовые методы, а теперь нам предстоит все это организовать своими руками.

Прежде, чем рассмотреть конкретный пример, отметим одно важное обстоятельство. Формально метод в теле класса описывается как обычная функция.

Но методы, как мы уже знаем, должны вызываться из экземпляра класса.

При описании метода экземпляр класса, из которого будет вызываться метод, должен быть явно указан как первый аргумент метода.

При вызове метода этот аргумент методу явно не передается.

Причина в том, что экземпляр класса, из которого вызывается метод, указывается явно (перед именем метод через точку).

Получается такое своеобразное правило, которое условно можно назвать "минус один аргумент": при вызове метода из экземпляра класса у него на один аргумент меньше, чем это было при описании метода (это если нет аргументов со значением по умолчанию).

Благодаря методам внутри класса можно реализовывать самые разные алгоритмы, то есть методы – это действия. Именно поэтому, в названиях методов используют глаголы, например:

set\_value, get\_param, start, stop, и т.п.

В то время как именами свойств (данных) выступают существительные:

color, size, x, y, и т.п.

Рекомендуется придерживаться этого простого правила.

Давайте, для примера объявим метод set\_coords в классе Point, который будет просто выводить в консоль сообщение «вызов метода set\_coords»:

**class** Point:

    color = 'red'

    circle = 2

**def** set\_coords(self):

**print**("вызов метода set\_coords")

Здесь сразу бросается в глаза вот этот параметр self, который автоматически прописывает интегрированная среда. Зачем он здесь, если мы пока ничего не собираемся передавать этому методу? Давайте его уберем! Пока никаких проблем не возникло. Мало того, мы можем его вызвать из класса Point:

Point.set\_coords()

и все будет работать без ошибок. Здесь мы видим, как вызываются методы класса. Все довольно очевидно. Записываем имя класса (Point), и через точку указываем имя метода. В конце обязательно прописываем круглые скобки, так как это оператор вызова функций. И, так как метод – это функция класса, то для вызова метода используется тот же оператор, что и для вызова функций.

В результате, мы получили класс, в котором два свойства и один метод. Далее, создадим экземпляр этого класса:

pt = Point()

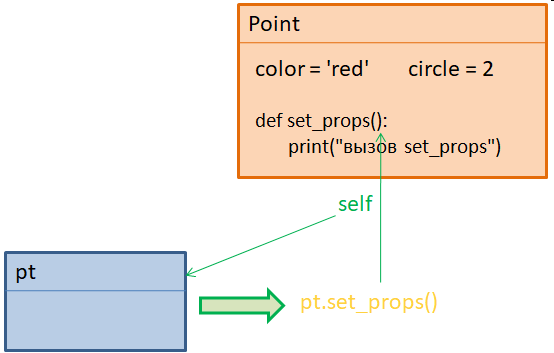
И, как мы с вами говорили, через объект pt можно обращаться ко всем атрибутам класса Point, в том числе и к методу set\_coords:

pt.set\_coords

Этот атрибут ссылается на объект-функцию, которую мы определили в классе Point. Попробуем ее вызвать:

pt.set\_coords()

Видим ошибку, что в метод set\_coords при вызове передается один аргумент, а он у нас определен без параметров. Дело в том, что когда мы вызываем методы класса через его объекты, то интерпретатор Python автоматически добавляет первым аргументом ссылку на объект, из которого этот метод вызывается.



Поэтому, если мы хотим внутри класса определить метод, который можно было бы вызывать из его экземпляров, то дополнительно прописывается первый параметр, обычно, с именем self:

**class** Point:

    color = 'red'

    circle = 2

**def** set\_coords(self):

**print**("вызов метода set\_coords " + str(self))

Еще раз, параметр self будет ссылаться на экземпляр класса, из которого вызывается метод. Зачем это надо? Сейчас узнаете. После этого дополнения мы уже не сможем вызвать данный метод через класс без указания первого аргумента:

но можем через его объекты:

pt.set\_coords()

То есть, когда метод вызывается через класс, то Python автоматически не подставляет никаких аргументов. А когда вызов идет через экземпляры класса, то первый аргумент – это всегда ссылка на экземпляр. Данный момент нужно знать и помнить.

(?) Но мы все же можем вызвать метод set\_coords и через класс, если явно передадим ссылку на объект pt, следующим образом:

Point.set\_coords(pt)

Именно это на автомате делает Python, когда вызов осуществляется через объекты классов.

Так зачем понадобилось такое поведение? Дело в том, что метод класса – это тоже его атрибут и когда создаются экземпляры класса, то метод становится общим для всех объектов и не копируется в них.

Фактически, только благодаря параметру self мы «знаем» какой объект вызвал данный метод и можем организовать с ним обратную связь.

Например, пусть метод set\_coords задает координаты точек для текущего объекта. Тогда, мы пропишем в нем два дополнительных параметра и через self в самом экземпляре класса создадим (либо переопределим) два свойства:

**class** Point:

    color = 'red'

    circle = 2

**def** set\_coords(self, x, y):

        self.x = x

        self.y = y

В результате, при вызове метода:

pt.set\_coords(1, 2)

**print**(pt.\_\_dict\_\_)

в объекте pt будут созданы два свойства x, y со значениями 1 и 2. Вот для чего нужен этот параметр self. Если в программе создать еще один объект:

pt2 = Point()

и через него вызвать тот же самый метод:

pt2.set\_coords(10, 20)

**print**(pt2.\_\_dict\_\_)

То увидим, что свойства x, y со значениями 10 и 20 были созданы только в нем (в его пространстве имен) и никак не связаны с координатами другого объекта pt или классом Point. То есть, через self мы работаем с конкретным объектом, из которого был вызван данный метод.

Конечно, в классах мы можем прописывать произвольное количество методов. Например, определим еще один, который будет возвращать координаты точки в виде кортежа значений:

**class** Point:

    color = 'red'

    circle = 2

**def** set\_coords(self, x, y):

        self.x = x

        self.y = y

**def** get\_coords(self):

**return** (self.x, self.y)

И ниже в программе можем вызвать его:

**print**(pt.get\_coords())

Интересно, что так как имя метода – это атрибут класса, то мы можем обратиться к нему через знакомую нам уже функцию:

res = getattr(pt, 'get\_coords')

**print**(res)

Видим, что это ссылка на объект-функцию. А раз так, то ничто нам не мешает ее здесь вызывать:

**print**(res())

Конечно, так делают очень редко. Обычно используют синтаксис через точку. Я привел это, чтобы еще раз подчеркнуть, что имена методов – это те же самые атрибуты, просто они ведут не как данные, а как функции.

Во всем остальном они схожи с атрибутами-данными класса.

Итак, в этом разделе вы должны были узнать, как определяются простые методы класса, за что отвечает параметр self и как происходит обращение к методам и их вызов.

И в дальнейшем мы обязаны придерживаться соглашение называть при описании метода первый его аргумент self в тех местах программного кода, где у метода объявлен аргумент с названием – self.

И этот аргумент будет означать ссылку на экземпляр класса, из которого вызывается метод.

Однако, ситуация, когда значения полям экземпляров класса присваиваются (явно или посредством специальных методов) уже после создания экземпляра класса не очень удобна.

Особенно она неудобна, если приходится иметь дело со значительным количеством экземпляров. Существует механизм, который позволяет частично (или даже полностью) снять эту проблему. Речь идет об использовании конструктора.

## Инициализатор \_\_init\_\_ и финализатор \_\_del\_\_

Рассмотрим начальную инициализацию объектов в момент их создания и финализации при их удалении.

При создании класса можно описать специальный метод, который называется конструктором экземпляра класса (иногда также называют методом инициализации). Этот метод автоматически вызывается при создании экземпляра класса. Рецепт создания конструктора очень простой: необходимо создать метод с названием \_\_init\_\_() (два символа подчеркивания в начале и два символа подчеркивания в конце). Аргументов у конструктора может быть сколько угодно – но не меньше одного (ссылка на экземпляр, при создании которого вызывается конструктор).

В каждом классе языка Python есть набор предопределенных «магических» методов. Да, это такое общепринятое название. Магические методы начинаются и заканчиваются двумя подчеркиваниями:

\_\_ имя метода\_\_

В частности существуют два таких метода:

* \_\_init\_\_(self) –  инициализатор объекта класса
* \_\_del\_\_(self) – финализатор класса

Первый вызывается сразу после создания экземпляра класса, а второй – перед непосредственным его удалением. Давайте посмотрим, как они работают и зачем нужны.

Снова вернемся к прошлому примеру, где мы определили класс с двумя свойствами color и circle и двумя методами set\_coords и get\_coords:

**class** Point:

    color = 'red'

    circle = 2

**def** set\_coords(self, x, y):

        self.x = x

        self.y = y

**def** get\_coords(self):

**return** (self.x, self.y)

Использовать на практике такой класс не очень удобно, так как после создания объекта:

pt = Point()

координат x, y в нем никаких не будет. Дополнительно для этого нужно еще вызывать метод:

pt.set\_coords(1, 2)

Было бы хорошо сделать эти действия сразу в момент создания экземпляра класса. И поможет нам в этом магический метод \_\_init\_\_. Давайте сначала мы его пропишем в самом простом виде:

**class** Point:

    color = 'red'

    circle = 2

**def** \_\_init\_\_(self):

**print**("вызов \_\_init\_\_")

        self.x = 0

        self.y = 0

    ...

Здесь+ первый параметр self является ссылкой на созданный экземпляр класса, через который мы создаем в этом новом объекте локальные свойства x, y. В результате, создавая экземпляр класса:

pt = Point()

мы увидим в консоли, что, во-первых, произошел вызов этого метода и, во-вторых, в объекте pt были созданы два локальных свойства x, y с нулевыми значениями:

**print**(pt.\_\_dict\_\_)

Детально это работает, следующим образом. Вначале происходит создание объекта в памяти устройства.

Непосредственно перед его созданием вызывается магический метод \_\_new\_\_ (О нем мы еще будем говорить).

Затем, после успешного создания объекта, вызывается магический метод \_\_init\_\_ для начальной инициализации созданного объекта. В результате, у нас появляются два локальных атрибута x и y.

Давайте теперь сделаем так, чтобы в момент создания экземпляра класса мы могли сразу указывать значения координат точки. Для этого в инициализаторе можно прописывать дополнительные параметры:

**def** \_\_init\_\_(self, a, b):

        self.x = a

        self.y = b

И при создании объекта с таким инициализатором мы уже должны в круглых скобках передавать эти два аргумента:

pt = Point(1, 2)

Если этого не сделать, то возникнет ошибка и объект создан не будет.

Обратите внимание, я специально в качестве дополнительных параметров указал имена a и b. При этом, в самом объекте создаются локальные свойства с именами x и y. То есть, то, что мы прописываем после self, то и является именем нового атрибута. Конечно, на практике, обычно, имена параметров в инициализаторе совпадают с именами создаваемых свойств и лучше записать инициализатор в таком виде:

**def** \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Также всегда следует помнить, что метод \_\_init\_\_ - это обычная функция, поэтому мы в качестве дополнительных параметров можем указывать и фактические и формальные параметры. Например, так:

**def** \_\_init\_\_(self, x=0, y=0):

        self.x = x

        self.y = y

Тогда, при создании объектов мы можем вообще не указывать аргументы:

pt = Point()

Либо указать один или два:

pt = Point(10)

pt = Point(10, 20)

То есть, здесь все ровно так, как и с обычными функциями.

## Финализатор объекта класса

Я, надеюсь, вы теперь хорошо себе представляете, как работает инициализатор объектов класса и зачем он нужен. Второй аналогичный магический метод \_\_del\_\_ автоматически вызывается непосредственно перед уничтожением экземпляра класса. Он называется **финализатор** (или деструктор).

Давайте пропишем такой магический метод в классе Point, следующим образом:

**def** \_\_del\_\_(self):

**print**("Удаление экземпляра: "+ str(self))

Здесь по-прежнему self – это ссылка на экземпляр класса, который будет удален.

После запуска программы видим, что объект в конце программы действительно был удален и вызван метод \_\_del\_\_.

Но, здесь у вас может возникнуть вопрос: а когда и в какой момент вообще происходит удаление объектов.

Мы же в программе это явно нигде не указываем? Работает все достаточно просто. Интерпретатор языка Python имеет, так называемый, **сборщик мусора**.

Это алгоритм, который отслеживает объекты и как только они становятся ненужными, удаляет их. Но как он определяет нужный объект или ненужный? Все очень просто. Пока на какой-либо объект ведет хотя бы одна внешняя ссылка, то он считается используемым и сохраняется в памяти. Как только все внешние ссылки пропадают, то сборщик мусора его уничтожает (освобождает память, которую он занимал).

## Магический метод \_\_new\_\_. Пример паттерна Singleton

Познакомимся с еще одним магическим методом \_\_new\_\_, который вызывается непосредственно перед созданием объекта класса. Я напомню, что другой магический метод \_\_init\_\_ вызывается после создания объекта (о нем мы говорили перед этим).

Здесь у вас может сразу возникнуть вопрос, зачем нужно было определять два разных метода, которые последовательно вызываются при создании экземпляров классов? Разве не достаточно одного \_\_init\_\_, чтобы выполнять начальную инициализацию объекта? Конечно, нет.

В практике программирования встречаются самые разнообразные задачи и иногда нужно что-то делать и до создания объектов. Например, реализация известного паттерна Singleton в Python, как раз делается через метод \_\_new\_\_ и мы с ним позже познакомимся.

А для начала нам нужно познакомиться с работой самого метода \_\_new\_\_. Давайте добавим его в наш класс Point. Я его перепишу в сокращенной форме:

**class** Point:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**def** \_\_init\_\_(self, x=0, y=0):

**print**("вызов \_\_init\_\_ для " + str(self))

        self.x = x

        self.y = y

Смотрите, здесь записан метод \_\_new\_\_, у которого первым идет обязательный параметр cls – это ссылка на текущий класс Point, а затем, указываются коллекции из фактических и формальных параметров, которые может принимать данная функция. Это стандартное определение метода \_\_new\_\_ в классах. В теле функции я просто сделал вывод сообщения и переменной cls.

Если теперь попробовать создать экземпляр класса:

pt = Point(1, 2)

то мы в консоли увидим только одно сообщение от метода \_\_new\_\_. То есть, второй метод \_\_init\_\_ не был вызван и, кроме того, если мы распечатаем переменную pt:

**print**(pt)

то увидим значение None, то есть, объект не был создан. Почему так произошло? В Python магический метод \_\_new\_\_ должен возвращать адрес нового созданного объекта.

А в нашей программе он ничего не возвращает, то есть, значение None, что эквивалентно отказу в создании нового объекта. Именно поэтому переменная pt принимает значение None.

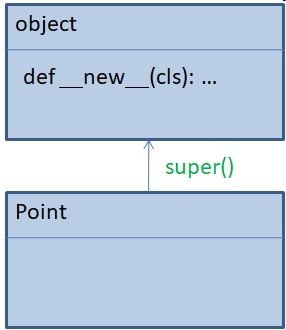
Хорошо, давайте адрес нового объекта. Но откуда мы его возьмем? Для этого можно вызвать аналогичный метод базового класса и делается это, следующим образом:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**return** super().\_\_new\_\_(cls)

Здесь функция super() возвращает ссылку на базовый класс и через нее мы вызываем метод \_\_new\_\_ с одним первым аргументом. Но, подождите! Что это за базовый класс? Мы наш класс Point ни от какого класса не наследовали? Да и вообще еще не изучали тему наследования! Да, поэтому, забегая вперед, скажу, что, начиная с версии Python 3, все классы автоматически и неявно наследуются от базового класса object:



И уже из этого базового класса мы вызываем метод \_\_new\_\_. Кстати, если метод \_\_new\_\_ не прописывать в классе Point, то будет автоматически запускаться версия базового класса.

То есть, этот метод всегда вызывается при создании нового объекта. При необходимости, мы можем его переопределять, добавляя новую логику его работы.

И то же самое относится ко всем магическим методам. Они всегда существуют у всех классов. Но переопределяем мы лишь те, что необходимо, а остальные работают по умолчанию. В этом сила базового класса object. В нем уже существует программный код, общий для всех классов языка Python. Иначе, нам пришлось бы его каждый раз прописывать заново.

Итак, теперь мы знаем откуда берется и вызывается магический метод \_\_new\_\_. Запустим программу и видим в консоли, что были вызваны оба метода \_\_new\_\_ и \_\_init\_\_ нашего класса Point, а также был успешно сформирован новый объект.

Возможно, здесь у вас остался один вопрос: а зачем нужны списки параметров \*args, \*\*kwargs в методе \_\_new\_\_? Мы, вроде, их нигде не используем? В действительности, здесь хранятся дополнительные параметры, которые мы можем указывать при создании объекта. Например, строчка:

pt = Point(1, 2)

создает объект с двумя числовыми значениями, то есть, \*args будет содержать эти два числа. По идее, мы можем реализовать в методе \_\_new\_\_ какую-либо логику с учетом значений этих аргументов.

Но, в данном случае, просто игнорируем. Используем их дальше в методе \_\_init\_\_ при инициализации объекта. То есть, аргументы 1 и 2 передаются и в метод \_\_new\_\_ и в метод \_\_init\_\_.

## Пример паттерна Singleton (учебный)

Думаю, вы в целом теперь представляете себе работу магического метода \_\_new\_\_, но остается вопрос: зачем все же он нужен? В качестве ответа я приведу пример очень известного паттерна проектирования под названием Singleton. Этот паттерн будет представлен в учебном варианте, то есть, мы его реализуем не полностью, т.к. пока отсутствуют достаточные знания.

Итак, давайте предположим, что мы разрабатываем класс для работы с БД. В частности, через него можно будет подключаться к СУБД, читать и записывать информацию, закрывать соединение:

**class** DataBase:

**def** \_\_init\_\_(self, user, psw, port):

        self.user = user

        self.psw = psw

        self.port = port

***def*** *connect(self):*

***print****(f"соединение с БД: {self.user}, {self.psw}, {self.port}")*

***def*** *close(self):*

***print****("закрытие соединения с БД")*

***def*** *read(self):*

***return*** *"данные из БД"*

***def*** *write(self, data):*

***print****(f"запись в БД {data}")*

И далее полагаем, что в программе должен существовать только один экземпляр этого класса в каждый момент ее работы. То есть, одновременно два объекта класса DataBase быть не должно. Чтобы это обеспечить и гарантировать, как раз и используется паттерн Singleton. Реализуем его для класса DataBase.

Я пропишу в нем специальный атрибут (на уровне класса):

\_\_instance = None

который будет хранить ссылку на экземпляр этого класса. Если экземпляра нет, то атрибут будет принимать значение None. А, затем, чтобы гарантировать создание строго одного экземпляра, добавим в класс магический метод \_\_new\_\_:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**if** cls.\_\_instance **is** None:

            cls.\_\_instance = super().\_\_new\_\_(cls)

**return** cls.\_\_instance

Работает этот метод очевидным образом. Мы проверяем атрибут класса \_\_instance. Причем, для обращения к нему используем параметр cls – ссылку на текущий класс. Подробнее я еще освещу этот момент. Далее, проверяем, если значение равно None, то вызываем метод \_\_new\_\_ базового класса и тем самым разрешаем создание объекта. Иначе, просто возвращаем ссылку на ранее созданный экземпляр. Как видите, все достаточно просто.

И пропишем еще один магический метод – финализатор \_\_del\_\_, который будет обнулять атрибут \_\_instance перед уничтожением объекта, чтобы мы могли, при необходимости, создать новый.

Все, простейший вариант паттерна Singleton готов. Правда он имеет один изъян. Смотрите, если попробовать создать два экземпляра:

db = DataBase('root', '1234', 80)

db2 = DataBase('root2', '5678', 40)

**print**(id(db), id(db2))

то их id ожидаемо будут равны. То есть, ссылки db и db2 действительно ведут на один объект. Но, если выполнить метод:

db.connect()

db2.connect()

то увидим значения: 'root2', '5678', 40 – аргументы при повторном создании класса. По идее, если объект не создается, то и локальные свойства его также не должны меняться. Почему так произошло? Все просто. Мы здесь действительно видим первый объект. Но при повторном вызове DataBase() также был вызван магический метод \_\_init\_\_ с новым набором аргументов и локальные свойства изменили свое значение. Конечно, мы можем здесь поставить «костыль» (как говорят в программисты) и дополнительно в классе прописать флаговый атрибут, например:

\_\_is\_exist = False

специально для метода \_\_init\_\_, чтобы не выполнять его если объект уже создан. Но я даже не буду дописывать такую программу. Слишком уж костыльно получается. Правильнее было бы здесь переопределить еще один магический метод \_\_call\_\_, о котором мы еще будем говорить. А пока оставим нашу реализацию паттерна Singleton в таком виде.

Я, надеюсь, что из этого занятия вы поняли, как работает магический метод \_\_new\_\_ и зачем он нужен.

в Python можно относительно легко обратиться и к приватным атрибутам извне. Если распечатать все атрибуты экземпляра:

**print**(dir(pt))

то среди прочих мы увидим, следующие:

'\_Point\_\_x', '\_Point\_\_y'

Это и есть кодовые имена приватных атрибутов, к которым мы можем обратиться через ссылку pt:

**print**(pt.\_Point\_\_x, pt.\_Point\_\_y)

и менять их. Однако, так делать крайне не рекомендуется и двойное подчеркивание должно сигнализировать программисту, что работать с такими атрибутами нужно только через разрешенные интерфейсные методы. Иначе, возможны непредвиденные ошибки.

## Модуль accessify

Если у вас появилась необходимость лучше защитить методы класса от доступа извне, то это можно сделать с помощью модуля accessify. Для его установки нужно выполнить команду:

pip install accessify

И, затем, импортировать из него два декоратора:

**from** accessify **import** private, protected

Далее, нужный декоратор просто применяем к методу и он становится либо приватным (private), либо защищенным (protected):

    @private

    @classmethod

**def** check\_value(cls, x):

**return** type(x) **in** (int, float)

Все, теперь мы можем обратиться к check\_value только внутри класса, но не извне:

pt.check\_value(5)

Я, думаю, из этого занятия вам стало понятно, как реализуются режимы доступа public, protected и private, а также, как правильно обращаться к скрытым атрибутам через интерфейсные методы – сеттеры и геттеры.

## Магические методы \_\_setattr\_\_, \_\_getattribute\_\_, \_\_getattr\_\_ и \_\_delattr\_\_

* \_\_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).

Работают они достаточно просто. Начнем с метода \_\_getattribute\_\_ и с его помощью ограничим доступ к приватным свойствам \_\_x и \_\_y экземпляра. Для простоты я переопределю класс Point, следующим образом:

**class** Point:

    MAX\_COORD = 100

    MIN\_COORD = 0

**def** \_\_init\_\_(self, x, y):

        self.\_\_x = x

        self.\_\_y = y

**def** \_\_getattribute\_\_(self, item):

**print**("\_\_getattribute\_\_")

**return** object.\_\_getattribute\_\_(self, item)

Здесь добавлен новый магический метод \_\_getattribute\_\_. Он автоматически вызывается, когда идет считывание атрибута через экземпляр класса. Например, при обращении к свойству MIN\_COORD:

**print**(pt1.MIN\_COORD)

Или к приватному свойству через специальное имя:

**print**(pt1.\_Point\_\_x)

Но раз это так, то давайте явно запретим считывать такой атрибут из экземпляра класса. Для этого пропишем в методе \_\_getattribute\_\_ проверку:

**def** \_\_getattribute\_\_(self, item):

**if** item == "\_Point\_\_x":

**raise** ValueError("Private attribute")

**else**:

**return** object.\_\_getattribute\_\_(self, item)

То есть, мы смотрим, если идет обращение к приватному атрибуту по внешнему имени \_Point\_\_x, то генерируем исключение ValueError. И, действительно, после запуска программы видим отображение этой ошибки в консоли. Вот так, через магический метод \_\_getattribute\_\_ можно реализовывать определенную логику при обращении к атрибутам через экземпляр класса.

Следующий магический метод \_\_setattr\_\_ автоматически вызывается в момент присваивания атрибуту нового значения. Пропишем формально этот метод в классе Point:

**def** \_\_setattr\_\_(self, key, value):

**print**("\_\_setattr\_\_")

        object.\_\_setattr\_\_(self, key, value)

После запуска видим несколько сообщений «\_\_setattr\_\_». Это связано с тем, что в момент создания экземпляров класса в инициализаторе \_\_init\_\_ создавались локальные свойства \_\_x и \_\_y. В этот момент вызывался данный метод. Также в переопределенном методе \_\_setattr\_\_ мы должны вызывать соответствующий метод из базового класса object, иначе, локальные свойства в экземплярах создаваться не будут.

Давайте теперь для примера через этот магический метод запретим создание локального свойства с именем z. Сделаем это следующим образом:

**def** \_\_setattr\_\_(self, key, value):

**if** key == 'z':

**raise** AttributeError("недопустимое имя атрибута")

**else**:

            object.\_\_setattr\_\_(self, key, value)

Обратите внимание, что внутри метода \_\_setattr\_\_ нельзя менять свойства напрямую:

**def** \_\_setattr\_\_(self, key, value):

**if** key == 'z':

**raise** AttributeError("недопустимое имя атрибута")

**else**:

            self.\_\_x = value

В этом случае метод \_\_setattr\_\_ начнет выполняться по рекурсии, пока не возникнет ошибка достижения максимальной глубины рекурсии. Если нужно сделать что-то подобное, то используйте коллекцию \_\_dict\_\_:

self.\_\_dict\_\_[key] = value

или, если требуется стандартное поведение метода, то вызывайте его из класса object, как это мы прописывали вначале:

object.\_\_setattr\_\_(self, key, value)

Следующий магический метод \_\_getattr\_\_ автоматически вызывается, если идет обращение к несуществующему атрибуту. Добавим его в наш класс:

**def** \_\_getattr\_\_(self, item):

**print**("\_\_getattr\_\_: " + item)

Если ниже обратиться к несуществующему свойству, например:

**print**(pt1.a)

то увидим сообщение «\_\_getattr\_\_: a» и значение None, которое вернул данный метод. Если же прописать существующий атрибут:

**print**(pt1.MAX\_COORD)

то этот магический метод уже не вызывается. Зачем он может понадобиться? Например, нам необходимо определить класс, в котором при обращении к несуществующим атрибутам возвращается значение False, а не генерируется исключение. Для этого записывается метод \_\_getattr\_\_ в виде:

**def** \_\_getattr\_\_(self, item):

**return** False

Наконец, последний магический метод \_\_delattr\_\_ вызывается в момент удаления какого-либо атрибута из экземпляра класса:

**def** \_\_delattr\_\_(self, item):

**print**("\_\_delattr\_\_: "+item)

Добавим новое локальное свойство в экземпляр pt1:

pt1.a = 10

затем выполним команду его удаления:

**del** pt1.a

и видим, что действительно был вызван метод \_\_delattr\_\_, правда, сам атрибут удален не был:

**print**(pt1.\_\_dict\_\_)

Это из-за того, что внутри этого метода нужно вызвать соответствующий метод класса object, который и выполняет непосредственное удаление:

**def** \_\_delattr\_\_(self, item):

        object.\_\_delattr\_\_(self, item)

Я думаю, что из этого вы хорошо себе уяснили, как происходит обращение к атрибутам класса и как можно управлять доступом к ним через магические методы.

## Практикум: Простые задачи.

Объявите пустой класс с именем Car. С помощью функции setattr() добавьте в этот класс атрибуты:

model: "Тойота"

color: "Розовый"

number: "П111УУ77"

Выведите на экран значение атрибута color, используя словарь \_\_dict\_\_ класса Car.

class Car:

pass

setattr(Car, "model", "Тойота")

setattr(Car, "color", "Розовый")

setattr(Car, "number", "О111АА77")

print (Car.\_\_dict\_\_["color"])

Решение 2

class Car:

pass

d = {

'model': "Тойота",

'color': "Розовый",

'number': "О111АА77"

}

[setattr(Car,k,v) for k,v in d.items()]

print(Car.\_\_dict\_\_['color'])

Объявите класс с именем Notes и определите в нем следующие атрибуты:

uid: 1005435

title: "Шутка"

author: "И.С. Бах"

pages: 2

Затем, с помощью функции getattr() прочитайте и выведите на экран значение атрибута author.

**class Notes:**

**uid = 1005435**

**title = "Шутка"**

**author = "И.С. Бах"**

**pages = 2**

**print (getattr(Notes, "author"))**

Объявите класс с именем Dictionary и определите в нем следующие атрибуты:

rus: "Питон"

eng: "Python"

Затем, с помощью функции getattr() прочитайте и выведите на экран значение атрибута rus\_word. Если такого атрибута в классе нет, то функция getattr() должна возвращать булево значение False.

class Dictionary:

rus = "Питон"

eng = "Python"

print (getattr(Dictionary, "rus\_word", False))

Задание 3. Объявите класс с именем Figure и двумя атрибутами:

type\_fig: 'ellipse'

color: 'red'

Создайте экземпляр с именем fig1 этого класса и добавьте в него следующие локальные атрибуты:

start\_pt: (10, 5)

end\_pt: (100, 20)

color: 'blue'

Удалите из экземпляра класса свойство color и выведите на экран список всех локальных свойств (без значений) объекта fig1 в одну строчку через пробел в порядке, указанном в задании.

**class Figure:**

**type\_fig = 'ellipse'**

**color = 'red'**

**fig1 = Figure()**

**setattr (fig1, "start\_pt", (10, 5))**

**setattr (fig1, "end\_pt", (100, 20))**

**setattr (fig1, "color", 'blue')**

**delattr (fig1, 'color')**

**print (\*fig1.\_\_dict\_\_)**

Объявите класс с именем Person и атрибутами:

name: ' Александр Пушкин'

job: 'Программист'

city: 'Москва'

Создайте экземпляр p1 этого класса и проверьте, существует ли у него локальное свойство с именем job. Выведите True, если оно присутствует в объекте p1 и False - если отсутствует.

**class Person:**

**name = ' Александр Пушкин'**

**job = 'Программист'**

**city = 'Москва'**

**p1 = Person()**

**print ("job" in p1.\_\_dict\_\_ )**

**или** print(hasattr(p1.\_\_dict\_\_, 'job'))

**для теста:**

Выберите верное значение атрибута, которое будет выведено на экран при выполнении следующей программы:

class Figures:

type = 'ellipse'

color = 'red'

fig1 = Figures()

print(fig1.color)

Выдаст ошибку, т.к. свойства color нет в объекте fig1

None

False

Red

Установите соответствия между функциями и их описаниями.

Сопоставьте значения из двух списков

|  |  |
| --- | --- |
| setattr()  getattr()  hasattr()  delattr() | позволяет создавать новый атрибут или устанавливать значение уже существующего в классе или объекте  позволяет получать значение указанного атрибута из класса или объекта  позволяет проверять атрибут на наличие в классе или объекте  позволяет удалять атрибут в классе или объекте |

Выберите верные способы удаления атрибута tp из класса Point.

delattr(Point, tp)

Point.tp = None

del Point['tp']

Point.tp = 0

del tp of Point

del Point.tp

delattr('Point', 'tp')

delattr(Point, 'tp')

 В программе объявлен следующий класс:

class Magazine:

name = 'Наука и жизнь'

price = 1101

Что произойдет при выполнении команды:

print(getattr(Magazine, 'id'))

Будет создан новый атрибут с именем id и значением None

Выведет значение None

**Ошибка, так как атрибут id отсутствует в классе Magazine**

Выведет значение False

Объявите класс с именем MediaPlayer с двумя методами:

open(file) - для открытия медиа-файла с именем file (создает локальное свойство filename со значением аргумента file в объекте класса MediaPlayer)  
play() - для воспроизведения медиа-файла (выводит на экран строку "Воспроизведение <название медиа-файла>")

Создайте два экземпляра этого класса с именами: media1 и media2. Вызовите из них метод open() с аргументом "filemedia1" для объекта media1 и "filemedia2" для объекта media2. После этого вызовите через объекты метод play(). При этом, на экране должно отобразиться две строки (без кавычек):

"Воспроизведение filemedia1"  
"Воспроизведение filemedia2"

class MediaPlayer:

def open(self,file):

self.filename = file

def play(self):

print (f"Воспроизведение {self.filename}")

def play(self):

print (f'Воспроизведение {self.filename}')

media1 = MediaPlayer()

media2 = MediaPlayer()

media1.open("filemedia1")

media2.open("filemedia2")

media1.play()

media2.play()

Что называется методом класса?

**Такого термина в ООП нет**

**Любая (не статическая) функция, объявленная внутри класса**

**Переменные и функции внутри класса**

**Любая переменная, объявленная внутри класса**

Что называют атрибутами класса?

**Экземпляры (объекты) класса**

**Только переменные класса**

**Переменные и имена методов (ссылки на методы) класса**

**Только методы класса**

Какую роль играет параметр self в методах класса?

**это просто обязательный (зарезервированный) параметр для методов класса, через который можно передавать произвольные данные**

**это ссылка на класс, которому принадлежит метод**

**это ссылка на объект класса, из которого был вызван метод**

**это ссылка на текущий метод класса**

 Имеется следующий класс:

class Stepik:

    def next\_task(self):

        return "Следующее задание"

И создается объект этого класса:

my\_st = Stepik()

Выберите все верные варианты вызова метода next\_task()

next\_task(Stepik)

next\_task(my\_st)

my\_st.next\_task(Stepik)

**Stepik.my\_st.next\_task()**

**my\_st.next\_task()**

Stepik.next\_task(my\_st)

Дан класс Translator (для перевода с английского на русский), в котором объявлены три метода:

class Translator:

def add(self, eng, rus):

if 'tr' not in self.\_\_dict\_\_:

self.tr = {}

self.tr.setdefault(eng, [])

# здесь продолжайте метод add

def remove(self, eng):

# здесь продолжайте метод remove

def translate(self, eng):

# здесь продолжайте метод translate

В объекте этого класса должны локально (в атрибуте tr) храниться связки между английским и русскими словами в виде следующего словаря:

{'<английское слово>': [<одно или несколько русских слов>], ...}

Методы должны делать следующее:

add(self, eng, rus) - для добавления **в словарь** новой связки английского и русского слова (если английское слово уже существует, то новое русское слово добавляется как синоним для перевода, например, go - идти, ходить, ехать); если связка eng-rus уже существует, то второй раз ее добавлять не нужно, например:  add('go', 'идти'), add('go', 'идти');  
remove(self, eng) - для удаления **из словаря**связки по указанному английскому слову;  
translate(self, eng) - для перевода с английского на русский (метод должен возвращать список из русских слов, соответствующих переводу английского слова, даже если в списке всего одно слово).

Все добавления и удаления связок должны выполняться внутри каждого конкретного объекта класса Translator, т.е. связки хранить локально внутри экземпляров классов класса Translator, используя коллекцию-словарь. (Хранить связки непосредственно в коллекции \_\_dict\_\_ не нужно!)

Создайте экземпляр tr класса Translator и вызовите метод add для следующих связок:

tree - дерево  
car - машина  
car - автомобиль  
leaf - лист  
river - река  
go - идти  
go - ехать  
go - ходить  
milk - молоко

Затем методом remove() удалите связку для английского слова car. С помощью метода translate() переведите слово go. Результат выведите на экран в виде строки из всех русских слов, связанных со словом go:

**Вывод в формате:** идти ехать ходить

class Translator:

WORDS = dict()

def add(self, eng, rus):

self.WORDS.setdefault(eng, []).append(rus)

def remove(self, eng):

del self.WORDS[eng]

def translate(self, eng):

return self.WORDS[eng]

eng\_words = ("tree", "car", "car", "leaf", "river", "go", "go","go", "milk")

rus\_words = ("дерево", "машина", "автомобиль", "лист", "река", "идти", "ехать", "ходить", "молоко")

tr = Translator()

for eng, rus in zip(eng\_words, rus\_words):

tr.add(eng, rus)

tr.remove("car")

print(\*tr.translate("go"))