

# ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

# Урок 9

## Введение в ООП

### Содержание

<b>1. Введение в ООП.....</b>	<b>4</b>
1.1. Понятие ООП .....	4
1.2. Полиморфизм .....	12
<b>2. Типы данных, определяемые пользователем.....</b>	<b>13</b>
<b>3. Спецификаторы доступа .....</b>	<b>22</b>
3.1. Уровень доступа Public (общедоступный) .....	22
3.2. Уровень доступа Private (приватный).....	24
3.3. Уровень доступа Protected (внутренний или защищенный).....	27
<b>4. Наследование и инкапсуляция .....</b>	<b>28</b>
4.1. Общедоступный, внутренний и приватный метод .....	37
4.2. Уровень доступа Public (общедоступный) .....	38
4.3. Уровень доступа Private (приватный).....	39
4.4. Уровень доступа Protected (внутренний или защищенный).....	43

4.5. Статические методы и методы класса .....	48
4.6. Статические методы.....	49
4.7. Методы класса .....	54
4.8. Множественное наследование и MRO (порядок разрешения методов).....	65
<b>5. Полиморфизм .....</b>	<b>77</b>
5.1. Перегрузка операторов.....	80
5.2. Magic-методы, конструкторы .....	81
5.3. Реализация магических методов.....	83
<b>6. Создание и управление поведением экземпляров класса.....</b>	<b>90</b>
6.1. Функторы.....	90
6.2. Декораторы.....	96
6.3. Управляемые атрибуты объекта — property() .....	103
6.4. Свойства — property() .....	112
6.5. Дескрипторы .....	121
<b>7. Метаклассы .....</b>	<b>133</b>
7.1. Модель метаклассов .....	133
7.2. Метод конструктор __new__().....	140
7.3. Протоколы в Python .....	148

# 1. Введение в ООП

## 1.1. Понятие ООП

Ранее мы с вами уже знакомились с концепцией функционального программирования, при котором основная логика задачи (алгоритм) представляется в виде набора функций. И каждая из таких функций реализует отдельные шаги общего алгоритма.

Пока мы реализуем простые задачи и нам достаточно имеющихся (встроенных) типов данных, то функционального подхода достаточно. Однако в проектах для предметных областей со сложной структурой, в которой нужно обязательно учитывать особенности взаимодействия (связи) между ее объектами, нам нужна другая методология. Нужен такой подход, который позволит описать связь между данными и функциями их обработки (что невозможно в функциональном программировании). Например, [функция\\_1](#) должна работать только с данными одного типа, [функция\\_2](#) только с другими. И более того, нам нужно создать такую [функцию\\_3](#), которая с данными первого типа работает по одному алгоритму, а для данных второго типа предоставляет совсем другую функциональность.

Рассмотрим эту ситуацию на примере. Предположим, что у нас необходимо оценить уровень успеваемости по критериям «отлично», «хорошо», «удовлетворительно», «неудовлетворительно» для ученика и студента. При этом у студента оценка варьируется в диапазоне от 0 до 100,

а у ученика от 1 до 12, т. е. диапазоны критериев успеваемости абсолютно не совпадают. Для студента «отлично» — это балл от 90 до 100, а для ученика — от 10 до 12. Более этого студенческие баллы со значениями выше 12 вообще являются ошибочными данными для студента.

При использовании функционального подхода нам нужно, как минимум, создавать две отдельные функции с разными именами.

```
def checkStudentSuccess(name, score):
    if score >= 90:
        print("{} has excellent level".format(name))
    elif 75 <= score < 90:
        print("{} has good level".format(name))
    elif 60 <= score < 75:
        print("{} has average level".format(name))
    else:
        print("{} has poor level".format(name))

def checkPupilSuccess(name, score):
    if score >= 10:
        print("{} has excellent level".format(name))
    elif 7 <= score < 10:
        print("{} has good level".format(name))
    elif 4 <= score < 7:
        print("{} has average level".format(name))
    else:
        print("{} has poor level".format(name))

checkStudentSuccess("Jane", 78)
checkPupilSuccess("Bob", 6)
```

А что, если потом в задаче добавится еще и ученик младших классов, у которого оценки от 1 до 5? Тогда придется создать еще одну, третью, функцию.

А как вообще проверить, чьи это баллы? Студента или ученика? По имени «*Jane*» или «*Bob*» это точно не определить. А это нужно знать, чтобы определить, какую функцию вызвать в программе: `checkStudentSuccess()` или `checkPupilSuccess()`? То есть нам необходима связь «объект — его данные — его функциональность». Такой тип связи данных и поведения, функциональное программирование не может обеспечить простым и прозрачным способом.

**Рассмотрим еще один пример:** допустим, что нам нужно реализовать заполнение нового списка на основе элементов имеющегося списка. Есть два списка: один содержит логины пользователей и нужно перевести их в нижний регистр, а второй — годы рождения пользователей и нужно рассчитать их возраст (например, для задачи ограничения доступа к некоторому контенту).

Естественно, что нам придется создать две отдельные функции, каждая из которых принимает список как входной аргумент и на основе его данных заполняет результирующий список. То есть внешне функции выглядят очень похоже и возвращают одинаковый результат — список, причем даже одинаковой длины (ведь количество годов рождения во втором списке совпадает с числом логинов пользователей в первом). Однако, алгоритм работы функций полностью зависит от «смысла» ее входных данных.

```
userLogs=['Admin123','superUSER','GOODstudent']
userBYears=[2000, 2010, 2005]

def listMaker1(myList):
    result=[]
```

```

    for item in myList:
        result.append(item.lower())
    return result

def listMaker2(myList):
    result=[]
    for item in myList:
        result.append(2022-item)
    return result

newList1=listMaker1(userLogs)
newList2=listMaker2(userBYears)

print(newList1)
print(newList2)

```

Результат:

```

['admin123', 'superuser', 'goodstudent']
[22, 12, 17]

```

Рисунок 1

Как мы видим, правильность использования нужной функции зависит только от внимательности и ответственности разработчика.

Ведь от случайной ошибки такого типа:

```

newList1=listMaker1(userBYears)
newList2=listMaker2(userLogs)

```

когда мы просто попутали аргументы функций

**AttributeError** ✕

Рисунок 2

```
'int' object has no attribute 'lower'
```

Рисунок 3

разработчик никак не «застрахован».

А когда таких ситуаций (функций) становится много (что естественно для реальных проектов), то программист может запутаться в этой масштабной неоднозначности (даже при условии понятных имен функций и переменных).

Однако функциональное программирование не является единственным подходом к разработке программного обеспечения. Не менее популярной и даже более распространенной является парадигма объектно-ориентированного программирования (ООП). Ее популярность также вызвана тем фактом, что использование объектно-ориентированного программирования позволяет практически однозначно отобразить сущности (элементы) реального мира (предметной области, задачи) в структуру программы. При этом сохраняются особенности их структуры, связи (зависимости) между ними и детали их поведения. Можно сказать, что ООП помогает нам разрабатывать программу с таким содержимым, которое наиболее похоже на наше естественное представление о задаче (фрагменте реального мира).

**Объектно-ориентированное программирование** (ООП) — это такой подход к разработке программ (парадигма программирования), при котором отдельные компоненты системы (программы) представляются в виде объектов. Конечно, основными концепциями в ООП являются понятия объекта и класса.

Большинство сущностей (компонентов, частей) любой предметной области — это объекты.



Например, клиент банка, у которого есть имя, фамилия, возраст, еще какие-то характеристики, является объектом реального мира. Клиент банка может открыть счет, проверить состояние счета, пополнить счет и т. д., т. е. у него есть поведение (отображающее те задачи, который он должен решать в этой предметной области). С помощью ООП мы можем отобразить (выполнить моделирование) объект из реального мира в программный объект, у которого есть некоторые данные и который может выполнять некоторые функции. И это еще не все: ООП позволяет также моделировать отношения между объектами (например, между компаниями и ее сотрудниками, студентами и преподавателями и т. д.). Такое общее представление, которое включает в себя информацию о характеристиках объекта и об особенностях (алгоритмах) его действий мы будем называть классом.

**Класс** — это некоторый шаблон (макет), описывающий структуру и возможное поведение объекта. Можно сказать, что класс — это чертеж (схема), используя которую можно создать конкретный объект.

Конкретная реализация (воплощение) такого шаблона — это объект (называемый также экземпляром класса). Один объект может отличаться от другого также, как один клиент банка отличается от другого: фамилией, возрастом и т. д.

Рассмотрим на примере животного — кошки. У каждой кошки есть голова, лапы, хвост, уши, шерсть, т. е. описывая какую-то отдельную породу кошки нам необходимо описывать эти характеристики. Также каждая кошка умеет мяукать, прыгать, сидеть. Таким образом,

кошка — это класс с перечисленными свойствами и указанным поведением.

А вот сиамская кошка, у которой голова напоминает треугольник, сужающийся прямыми линиями к утонченной морде, уши — крупные и заостренные, это объект (т. е. экземпляр класса кошка). Таким образом, класс — это пользовательский тип данных, описывающий то, какими свойствами и поведением будут обладать переменные этого типа. А объект — это переменная класса, т. е. экземпляр с конкретным значением этих свойств.

Более детально понятия класс и объект мы обсудим немного позже. А сейчас рассмотрим основные принципы (так называемые столпы) ОО-парадигмы.

**Инкапсуляция** — одна из фундаментальных концепций объектно-ориентированного программирования, которая позволяет скрывать детали внутренней реализации объектов. Можно сказать, что согласно этому принципу, класс должен рассматриваться, как некий черный ящик. Пользователь не знает, что находится внутри (особенности реализации), и взаимодействует с ним только с помощью предоставленного интерфейса.

**Рассмотрим на примере.** Для того, чтобы водить современный автомобиль с автоматической коробкой передач не нужно знать, как устроен и работает его двигатель, что такое бензонасос, где расположена система охлаждения двигателя. Все эти особенности и действия отдельных механизмов автомобиля скрыты от водителя и, в тоже время, разрешают ему крутить руль и нажимать на педали газа или тормоза, не задумываясь, что в это время происходит под капотом.

Такое сокрытие внутреннего устройства и процессов, реализующих работу автомобиля, гарантируют простоту его использования и эффективность управления. Даже для водителей, которые не имеют значительного опыта вождения и не являются профессиональными автомеханиками. Таким образом, инкапсуляция обеспечивает безопасность внутреннего содержимого класса. Ведь когда мы таким способом скрываем от пользователя внутреннее устройство объекта, мы обеспечиваем его надежную работу.

**Наследование** — это возможность использовать данные и функциональность чего-то уже существующего для создания нового (на этой имеющейся основе).

В контексте ООП **наследование** — это способность одного класса получать (наследовать) свойства и поведение другого класса. Наследование является основным механизмом повторного использования кода в ОО-парадигме.

**Производный** (дочерний) класс — это класс, который наследует свойства другого класса, расширяя его функциональность и/или характеристики.

**Базовый** (родительский) класс — это класс, свойства и поведение которого наследуются.

Представим, что мы работники цеха по производству мебели. Наш цех производит стандартные стулья с жестким сидением, которые хорошо себя зарекомендовали у покупателей. Но вот поступил заказ на выпуск стульев с мягким сидением. Конечно, рациональным решением является использование в качестве основы для новой модели уже имеющейся модели стандартного стула, которая пользуется популярностью у клиентов. К тому же наше оборудование уже полностью настроено на выпуск таких стандартных

стульев. Поэтому после изготовления стандартного стула мы добавим мягкую обивку сидения, т. е. наша модификация будет иметь большую часть свойств прежней модели.

В этом примере стандартный стул является базовым классом, а стул с мягким сидением — производным, который наследует все характеристики родительского класса и добавляет новое свойство — мягкая обивка сидения.

## 1.2. Полиморфизм

В буквальном смысле термин полиморфизм буквально обозначает наличие нескольких форм одного и того же. В ОО парадигме полиморфизм предоставляет возможность использовать один и тот же интерфейс функции (с разными особенностями поведения) для разных объектов.

Рассмотрим понятие полиморфизма на примере кнопки включения / выключения для стиральной машины, пульта кондиционера и электрочайника. На всех перечисленных устройствах кнопка выглядит (реализована) по-разному (хотя изображение на ней для стиральной машины и пульта кондиционера может быть похожим). При этом результат ее нажатия один и тот же — прибор включается и начинает работать согласно своей программы функционирования.

В программировании полиморфизм реализуется с помощью механизма перегрузки метода класса. Мы уже знакомы с перегрузкой функций, которая позволяет создавать несколько функций с одинаковыми именами, но разным набором параметров и разной логикой поведения. При этом интерпретатор сам определяет, какой код будет работать, в зависимости от количества или типа аргументов при вызове функции.

## 2. Типы данных, определяемые пользователем

Как мы обсудили ранее, использование классов, позволит нам хранить информацию о сущностях предметной области (особенности структуры, значения характеристик, детали поведения) в соответствии с их представлением в реальном мире. Поэтому можно сказать, что классы используются для создания пользовательских структур данных, описывающих устройство объектов предметной области.

Особенности структуры (какими характеристиками, свойствами обладает объект) описываются перечнем **атрибутов** класса (переменными, определенными внутри класса и принадлежащими ему). А возможное поведение — **методами** класса, которые фактически представляют собой функции (включенные в состав класса), чья логика (алгоритм) реализует особенности этого поведения.

Как мы помним, класс — это только макет для создания объектов (экземпляров класса), который, в основном, не содержит конкретных значений.

Рассмотрим общий синтаксис для создания класса в Python:

```
class className:  
    <classStatement_1>  
    <classStatement_2>
```

```
...
<classStatement_N>
```

Давайте создадим класс, представляющий студента:

```
class Student:
    pass
```

Традиционно имя класса всегда пишется с заглавной буквы (**Student**).

Пока в классе **Student** не определено атрибутов и методов. Однако, согласно правилам синтаксиса Python при создании класса должно быть что-то определено. Поэтому мы использовали оператор **pass**, который часто используется в качестве заполнителя в том месте программы, где позже будет добавлен нужный фрагмент кода. Такой подход избавит нас сейчас от возникновения ошибки (согласно синтаксису Python после строки объявления класса, заканчивающейся символом «:», необходимо описать его состав, который сразу может быть неизвестен).

Теперь давайте попробуем добавить в наш класс несколько характеристик студента, например, возраст и имя. Ведь у каждого студента есть имя и возраст. Однако синтаксис Python не позволяет просто перечислить названия характеристик (имена переменных) внутри класса, им требуется задать значения. Например, так:

```
class Student:
    age=20
    name="Bob"
```

Однако, как мы помним класс — это только макет для создания объекта, который представляет конкретный экземпляр класса (конкретного студента в нашем примере). После того, как мы определили класс, мы можем создавать объекты этого класса.

Для создания объекта (экземпляра класса) используется специальная функция — конструктор класса (которая есть по умолчанию в любом классе, и которая не имеет параметров). Для запуска этой функции (создания объекта) нам нужно просто указать имя нашего класса с последующими круглыми скобками. В результате работы конструктора класса мы получаем объект (экземпляр данного класса).

```
student1=Student()
```

Таким образом мы создали объект `student1`, который является экземпляром класса `Student`. В примере выше мы определили у класса `Student` два атрибута `age` и `name`. Для доступа к атрибуту необходимо указать имя объекта и после символа точка имя нужного атрибута:

```
print(student1.name)
```

Эти атрибуты будут входить в состав каждого экземпляром класса `Student` вместе со своими значениями, т. е. у всех экземпляров класса значение атрибута `age` будет `20`, а значение атрибута `name` — «*Bob*».

Давайте создадим еще один объект (еще одного студента), а потом выведем возраст и имя у обоих:

```
class Student:  
    age=20
```

```

name="Bob"

student1=Student()
student2=Student()

print("Student's 1 info:")
print(student1.name)
print(student1.age)

print("Student's 2 info:")
print(student2.name)
print(student2.age)

```

Результат:

```

Student's 1 info:
Bob
20
Student's 2 info:
Bob
20

```

Рисунок 4

Однако данная ситуация нам не подходит: у каждого студента должны быть свои значения возраста и имени.

Для того, чтобы создавать объект с заданными характеристиками нужно включить в состав класса собственную функцию-конструктор.

Как мы уже знаем, все классы по умолчанию имеют функцию функцию-конструктор. Имя этой функции `__init__()` и у нее нет параметров (опять же, по умолчанию).

Мы будем использовать ее для присвоения значений свойствам объекта, указав их имена в качестве ее параметров.



Основное отличие свойств от атрибутов в том, что свойства могут быть разные у разных объектов (такие, как мы укажем в момент его создания). А вот значения атрибутов (как мы уже показали в нашем примере) будут одинаковыми для всех.

Анализируя наш пример, легко понять, что такие характеристики студента, как его имя и возраст не могут быть атрибутами класса, а являются его свойствами.

Добавим в класс конструктор `__init__()`, чтобы присвоить значения свойствам `age` и `name`.

А общей характеристикой для всех студентов может быть, например, название специальности, на которой они учатся. Эту характеристику, значение которой у всех одинаково (`spec="Computer science"`) нужно добавить в класс в виде атрибута:

```
class Student:
    spec="Computer science"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Вы можете указать в `__init__()` любое количество параметров (будущих свойств объекта), но первым параметром всегда будет переменная с именем `self`. Когда создается новый экземпляр класса, он автоматически передается параметру `self` в `__init__()`, чтобы для объекта можно было определить новые свойства. Фактически переменная `self` — это ссылка на текущий объект. Именно используя ее, мы можем внутри класса обращаться к его атрибутам и свойствам.

**Примечание:** заголовок метода `__init__()` в нашем примере имеет отступ в четыре пробела от начала строки, а его тело — восемь пробелов. Данное смещение важно, т.к. таким образом мы сообщаем о том, что метод `__init__()` принадлежит классу `Student`.

В нашем примере в теле метода `__init__()` мы использовали параметр `self` в двух инструкциях для создания свойств объекта (`name` и `age`) и установки их значений на основе значений второго и третьего параметров метода:

- команда `self.name = name` создает атрибут с именем `name` и присваивает ему значение второго параметра метода — `name`;
- команда `self.age = age` создает атрибут с именем `age` и присваивает ему значение третьего параметра метода — `age`.

Как мы помним, конструктор создает экземпляр класса (объект), поэтому значения свойств, определенные с помощью параметров конструктора, будут относиться к конкретному объекту. Таким образом все экземпляры класса `Student` будут иметь свойства `ame` и `age`, но их значения у каждого объекта будут свои — такие, как мы укажем в момент создания объекта:

```
student1=Student("Bob",20)
student2=Student("Jane", 18)

print(student1.showInfo())
print(student1.showMsg("Hello!"))
```

```

print(student2.showInfo())
print(student2.showMsg("Hi!"))

print("Student's 1 info:")
print(student1.name)
print(student1.age)
print(student1.spec)

print("Student's 2 info:")
print(student2.name)
print(student2.age)
print(student1.spec)

```

Результат:

```

Student's 1 info:
Bob
20
Computer science
Student's 2 info:
Jane
18
Computer science

```

Рисунок 5

Каждый объект имеет следующие характеристики:

- **Состояние** — которое представлено атрибутами объекта и также отражает свойства объекта.
- **Идентичность** — каждый экземпляр класса имеет уникальное имя (уникальное имя переменной, которую мы создаем в момент создания самого объекта), которое позволяет одному объекту взаимодействовать с другими объектами.

- **Поведение** — показывает возможную функциональность объекта (также отражает реакцию объекта на другие объекты), представляется методами класса.

С первыми двумя характеристиками объекта мы уже познакомились. Теперь рассмотрим особенности и процесс создания методов класса.

Методы класса — это функции, которые определены внутри класса и могут быть вызваны только из экземпляра этого класса. Как и в методе — конструкторе `__init__()`, при определении любого метода класса первым параметром всегда нужно указывать `self`. При этом при вызове метода мы не предоставляем ему значение, т. к. его (ссылку на текущий объект) автоматически предоставляет сам интерпретатор Python.

Таким образом, когда мы вызываем некоторый метод `methodName` объекта `myObject` подобной строкой кода:

```
myObject.methodName(argName1, argName2)
```

она автоматически преобразуется Python в такую:

```
myClass.methodName(myObject, argName1, argName 2).
```

Давайте добавим в наш класс `Student` два метода: первый для вывода информации о студенте (значений атрибутов и свойств объекта), а второй для вывода нужного текста приветствия.

```
class Student:
    spec="Computer science"
    def __init__(self, name, age):
        self.name = name
```

```
self.age = age

def showInfo(self):
    return f"Student {self.name} is {self.age}
           years old."

def showMsg(self, msgText):
    return f"Student {self.name}
           says '{msgText}'."

student1=Student("Bob",20)
student2=Student("Jane", 18)

print(student1.showInfo())
print(student1.showMsg("Hello!"))

print(student2.showInfo())
print(student2.showMsg("Hi!"))
```

Результат:

```
Student Bob is 20 years old.
Student Bob says 'Hello!'.
Student Jane is 18 years old.
Student Jane says 'Hi!'.
```

Рисунок 6

## 3. Спецификаторы доступа

В объектно-ориентированных языках программирования используются спецификаторы (модификаторы) доступа, которые позволяют устанавливать уровень (ограничения) доступа к свойствам и методам класса.

Большинство языков программирования (и Python тоже) имеют три вида модификаторов доступа: **public** (общедоступный), **protected** (внутренний или защищенный) и **private** (приватный).

### 3.1. Уровень доступа Public (общедоступный)

Все компоненты (члены) класса, объявленные как **public**, доступны из любой части программы. В некоторых языках программирования, например, в C# и в Java нужно явно указывать вид модификаторов доступа. В Python все свойства и методы класса по умолчанию являются общедоступными, т. е. явно указывать ключевое слово «**public**» нам не нужно.

В нашем классе **Person** все три свойства (имя, фамилия и возраст) имеют уровень доступа **public**. Поэтому они доступны внутри класса (используются в методах **getInfo()** и **getHi()**) и за его пределами. Давайте изменим значение свойства «возраст» у объекта, чтобы проверить это:

```
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
```

```

        self.lastName = lastName
        self.age = age
# public methods
def getInfo(self):
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self.age}."
def getHi(self, msgText):
    return f"{msgText}! I am {self.firstName}."

person1=Person("Joe", "Black", 30)
print(person1.getInfo())
person1.age=35
print(person1.getInfo())

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Person first name - Joe; last name - Black; age - 35.
Hi! I am Joe.

```

Рисунок 7

Как мы видим, изменение возраста у *Joe* прошло успешно. Повторный вызов метода `getInfo()` показал уже обновленное значение свойства.

Public-метод может быть вызван внутри других методов этого же класса. Выше мы уже показали вызов методов класса извне (с помощью его объектов). Теперь давайте добавим в метод `getHi()` вызов метода `getInfo()`:

```

class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName

```

```

        self.lastName = lastName
        self.age = age
    # public methods
    def getInfo(self):
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."
    def getHi(self, msgText):
        personInf=self.getInfo()
        return f"{personInf}\n{msgText}!
                I am {self.firstName}."

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Hi! I am Joe.

```

Рисунок 8

Теперь при вызове метода `getHi()` выводится вся информация о человеке (а мы показали доступность `public`-метода внутри других методов класса).

### 3.2. Уровень доступа Private (приватный)

Приватные члены класса (свойства или методы) доступны только внутри самого класса. Мы не можем использовать `private`-свойства или вызывать `private`-методы за пределами класса.

Реализация `private`-уровня доступа для свойства или метода класса обеспечивается двойным символом подчеркивания «`__`», который нужно поместить непосредственно перед именем свойства (без пробела), например `__privateProp`, `__privateMethod`.



Добавим в наш базовый класс `Person` private-свойство `personID`, которое генерируется диапазоне от 1 до 100 внутри конструктора, и private-метод `getID()`.

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
        self.age = age
        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __getID(self):
        return f"{self.__personID}\n"

    # public methods
    def getInfo(self):
        return f"{self.__getID()}Person first name - {self.firstName}; last name - {self.lastName}; age - {self.age}."
    def getHi(self, msgText):
        personInf=self.getInfo()
        return f"{personInf}\n{msgText}!
            I am {self.firstName}."

person1=Person("Joe","Black",30)
print(person1.getInfo())
```

Результат:

```
72
Person first name - Joe; last name - Black; age - 30.
```

Рисунок 9

Как мы видим, внутри класса (в теле метода `getID()`) мы имеем доступ к `private`-свойству `personID`. И вызов `private`-метода `getID()` также возможен внутри класса (мы выполнили его в теле метода `getInfo()`).

Давайте теперь попробуем вывести значение `personID` и вызвать метод `getID()`, используя объект `person1` (т. е. проверим доступ к `private`-свойству и `private`-методу за пределами класса, в котором они определены).

Строка кода:

```
print(person1.__personID)
```

вызовет исключение

A dark red rectangular box with the text "AttributeError" in white, followed by a white "X" icon.

Рисунок 10

по причине:

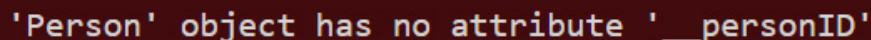
A dark red rectangular box with the text "'Person' object has no attribute '\_\_personID'" in white.

Рисунок 11

Аналогичная ситуация будет и при попытке вызвать `private` — метод за пределами класса:

```
person1.__getID()
```

Возникновение исключения:

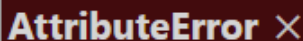
A dark red rectangular box with the text "AttributeError" in white, followed by a white "X" icon.

Рисунок 12

по причине:

```
'Person' object has no attribute '__showID'
```

Рисунок 13

Попытка изменить значение private-свойства `personID` не вызовет исключение

```
person1=Person("Joe", "Black", 30)
print(person1.getInfo())

person1.__personID=100
print(person1.getInfo())
```

однако не даст результата: значение `personID` не изменится:

```
30
Person first name - Joe; last name - Black; age - 30.
30
Person first name - Joe; last name - Black; age - 30.
```

Рисунок 14

Таким образом, private-свойства и методы действительно доступны только из своего класса.

### 3.3. Уровень доступа `Protected` (внутренний или защищенный)

Если нам нужно ограничить доступ к свойствам и методам таким образом, чтобы они были «видны» только внутри самого класса или классов, производных от него, то нам нужен уровень доступа `Protected`. Особенности создания `protected`-свойств и методов мы рассмотрим немного позже — после базовых моментов по реализации наследования в Python.

## 4. Наследование и инкапсуляция

Как мы уже знаем, наследование — это способность одного класса получать (наследовать) свойства и поведение другого класса. В этом механизме у нас всегда есть базовый класс и производный класс (или несколько таких).

Производный (дочерний) класс — это класс, который наследует свойства другого класса, расширяя его функциональность и/или характеристики.

Базовый (родительский) класс — это класс, свойства и поведение которого наследуются. Представим, что нам необходимо разработать такие типы транспортных средств, как автобус, грузовик и автомобиль-седан. Каждый из перечисленных автомобилей имеет четыре колеса, руль, педаль газа. Каждый автомобиль предоставляет возможности (функциональность) просмотра расхода топлива, торможение, поворот передних колес в соответствии с положением руля.

Если нам необходимо выполнить моделирование (реализовать) все перечисленные свойства и поведение программно, то придется написать все эти функции в каждом из трех классов. Представим это схемой (рис. 1):

Как мы видим нам фактически нужно выполнить дублирование кода три раза. Помимо проблемы дублирования кода (избыточности данных) также увеличивается вероятность ошибок при реализации.

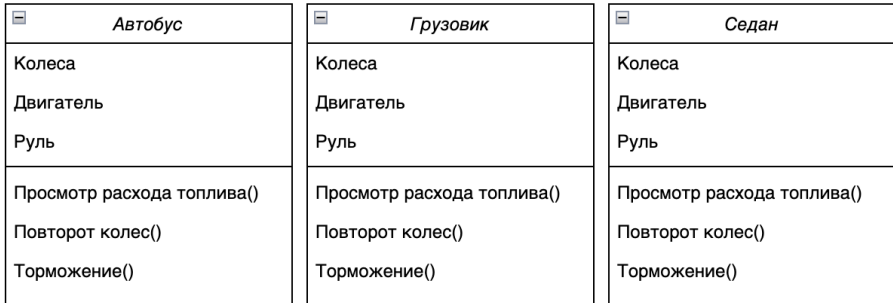


Рисунок 15

Теперь посмотрим, как механизм наследования позволит избежать подобной ситуации.

Вначале мы создадим базовый класс «Транспортное средство», который обладает характеристиками «колеса», «руль», «двигатель» и перечисленными тремя функциями (поведением). Далее остальные классы «Автобус», «Грузовик», «Седан» наследуем от класса «Транспортное средство».



Рисунок 16

Как легко можно заметить, мы можем просто избежать дублирования данных и повысить возможность повторного использования. Теперь давайте рассмотрим, как механизм наследования можно реализовать, используя средства языка Python.

Любой класс может быть родительским классом, поэтому синтаксис создания родительского класса аналогичен созданию любого другого.

Давайте создадим базовый класс «Человек», у которого есть такие свойства, как имя, фамилия, возраст. Также определим в нем два метода: «Поздороваться», который выводит сообщение «Привет! Меня зовут [имя]» и метод «Показать информацию», который выводит всю информацию (имя, фамилия, возраст) о конкретном человеке.

```
class Person:

    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
               last name - {self.lastName};\n\
               age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."
```

Теперь создадим два объекта (экземпляра класса `Person`) и запустим их методы `getInfo()` и `getHi()`:

```

person1=Person("Joe", "Black", 30)
person2=Person("Kate", "Smith", 20)

print(person1.getInfo())
print(person2.getInfo())
print(person1.getHi("Hi"))
print(person2.getHi("Hello"))

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Person first name - Kate; last name - Smith; age - 20.
Hi! I am Joe.
Hello! I am Kate.

```

Рисунок 17

На основе нашего (базового) класса `Person` создадим дочерний класс `Student`, у которого помимо свойств имя, фамилия, возраст будет атрибут «специальность» и новый метод, определяющий по среднему баллу (параметр метода), является ли студент успешным.

Для того чтобы в Python создать класс, наследующий функциональность от другого класса, нужно передать родительский класс в качестве параметра дочернему классу:

```

class Student(Person):
    spec="Computer Science"

    def isSuccessful(self, meanScore):
        return True if meanScore>=75 else False

```

В нашем примере у класса `Student` нет собственного конструктора (функции `__init__()`). Поэтому при создании

его объектов будет вызываться конструктор родительского класса **Person**.

Создадим первого студента — экземпляр класса **Student**. Так как будет использоваться конструктор базового класса **Person** (по причине отсутствия собственного), то нам нужно задавать такой же перечень выходных данных, как и при создании экземпляра класса **Person**:

```
student1=Student("Joe","Black",30)

print(student1.getInfo())
print(student1.getHi("Morning"))

print(f"Is {student1.firstName} successful student?.
      {student1.isSuccessful(85)}")
```

Как мы видим экземпляр класса **Student** может вызывать все методы своего базового класса (**getInfo()** и **getHi()** в нашем примере). И, конечно же, имеет все свойства класса-родителя (мы выводим значение свойства **firstName** в базовом классе при формировании строки с информацией об успешности нашего студента)

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Morning! I am Joe.
Is Joe successful student?.True
```

Рисунок 18

Но что, если нам, кроме атрибутов необходимо также добавить и новые свойства в класс студент? Например, лучшим решением для реализации метода **isSuccessful()**



было бы передавать ему средний балл студента, как свойство самого класса. Для этого нам необходимо добавить в класс студент свойство средний балл. Таким образом, нам нужно переопределить конструктор базового класса так, чтобы в его входные параметры, кроме имени, фамилии, возраста, входил еще и средний балл.

При этом нам нужно не скопировать функциональность (строки кода) конструктора родительского класса, а расширить его. Для этого мы внутри конструктора дочернего класса вначале должны вызвать конструктор родительского класса (чтоб дочерний класс имел те же свойства, что и класс-родитель), а потом добавить новое свойство.

Для обращения к базовому классу из производного используется выражение `super()`. Тогда вызов конструктора родительского класса будет осуществляться таким образом:

```
super().__init__(propName1, propName2,... propNameN),
```

где `propName1, propName2,... propName` перечень имен свойств базового класса.

Внесем нужные изменения в наш класс `Student`:

```
class Student(Person):
    spec="Computer Science"
    def __init__(self, firstName, lastName, age, score):
        super().__init__(firstName, lastName, age)
        self.score = score
    def isSuccessful(self):
        if self.score>=75:
            return True
```

```

        else:
            return False

student1=Student("Joe","Black",30, 78)

print(student1.getInfo())
print(student1.getHi("Morning"))

print(f"Is {student1.firstName} successful student?.
      {student1.isSuccessful()}")

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Morning! I am Joe.
Is Joe successful student?.True

```

Рисунок 19

Однако, как можно заметить из полученного результата, при вызове метода `getInfo()` выводится не вся информация о нашем студенте: нет его среднего балла, что затрудняет объяснение его успешности.

Это произошло потому, что мы не переопределили в дочернем классе данный метод. А в базовом классе информация о среднем балле отсутствовала, поэтому и не выводилась базовым методом `getInfo()`.

Опять же, наш новый (переопределенный) метод `getInfo()` должен включать всю функциональность своей версии из базового класса. Поэтому в теле переопределенного метода мы вначале вызовем базовый метод, а потом напишем строки кода, реализующие дополнительные возможности:

```

class Student(Person):
    spec="Computer Science"

    def __init__(self, firstName, lastName, age, score):
        super().__init__(firstName, lastName, age)
        self.score = score

    def getInfo(self):
        return super().showInfo()+f"score - {self.score}"

    def isSuccessful(self):
        if self.score>=75:
            return True
        else:
            return False

student1=Student("Joe","Black",30, 78)

print(student1.getInfo())

```

Результат:

```

Person first_name - Joe; last name - Black; age - 30.score - 78

```

Рисунок 20

Создадим еще один производный класс **Employee** на основе нашего (базового) класса **Person**.

У нового класса — потомка **Employee** помимо свойств имя, фамилия, возраст будут свойства хранящие информацию о названии должности, стаже работы и размере заработной платы.

Нам необходимо будет переопределить (дополнить) метод базового класса **getInfo()** для учета новой информации. Также у класса будет новый метод **getSickLeavePerc()**,

позволяющий по стажу работы определить процент заработной платы, который будет выплачиваться, если работник пойдет на больничный.

```
class Person:
    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
            last name - {self.lastName};\n\
            age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."

class Employee(Person):
    def __init__(self, firstName, lastName, age,
                 jobTitle, salary, seniority):
        super().__init__(firstName, lastName, age)
        self.jobTitle=jobTitle
        self.salary=salary
        self.seniority=seniority

    def getInfo(self):
        return super().getInfo() +
            f"\njobTitle - {self.jobTitle};\n\
            salary - {self.salary}; seniority -\n\
            {self.seniority}."

    def getSickLeavePerc(self):
        if self.seniority>5:
            return 1
        elif 3<self.seniority<=5:
            return 0.75
```

```

        else:
            return 0.5

employee1=Employee("Kate","Smith",20,"HR",2500,7)

print(employee1.getHi("Hello"))
print(employee1.getInfo())

print("The percentage of salary in case of sick
      leave will be {}%".
      format(employee1.getSickLeavePerc()*100))

```

Результат:

```

Hello! I am Kate.
Person first name - Kate; last name - Smith; age - 20.
jobTitle - HR; salary - 2500; seniority - 7.
The percentage of salary in case of sick leave will be 100%

```

Рисунок 21

## 4.1. Общедоступный, внутренний и приватный метод

Мы уже научились наследовать свойства или поведение любого класса, используя его в качестве базового. Однако, что, если некоторые данные (хранящиеся в свойствах или образующиеся методами класса) не должны наследоваться, т. е. должны быть недоступны из производного класса. В этом случае нам необходим подход для ограничений доступа к некоторым свойствам и методам класса.

Как мы уже знаем, в Python есть три формы модификаторов доступа: **public** (общедоступный), **protected** (внутренний или защищенный) и **private** (приватный).

Используя модификаторы, мы можем указать, будет ли доступно (или нет) свойство (метод) класса только внутри самого класса, для его классов-потомков и для других классов.

Краткое описание уровня доступа, который предоставляет определенный модификатор, приведено в таблице 1.

Таблица 1

Модификатор	Внутри класса (в его методах)	Производные классы	За пределами класса и его потомков
Public	+	+	+
Protected	+	+	-
Private	+	-	-

## 4.2. Уровень доступа Public (общедоступный)

Все компоненты (члены) класса, объявленные как **Public**, доступны из любой части программы. Все свойства и методы класса по умолчанию являются общедоступными.

Ранее мы уже рассмотрели возможности доступа к public-свойствам и методам внутри класса и за его пределами. А что насчет доступа из методов классов — потомков **Person**?

Создадим производный класс **Employee**, который помимо основной информации и базового поведения, полученных от родительского класса, обладает свойством **salary** (размер заработной платы).

Также у него будет метод **isRetiree()**, который на основе данных о возрасте определяет, является ли работник пенсионером. Именно в рамках этого нового метода мы

проверим доступ к public-свойствам базового класса внутри методов производного класса:

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self.age > 60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

employee1 = Employee("Joe", "Black", 30, 3000)
employee1.isRetiree()
```

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Joe is not retiree
```

Рисунок 22

Также в рамках метода `isRetiree()` мы показали наличие доступа к public-методам базового класса и методов производного.

### 4.3. Уровень доступа Private (приватный)

Мы уже знаем, что приватные члены класса (свойства или методы) доступны только внутри класса, в котором они определены. Поэтому мы не можем использовать private-свойства или вызывать private-методы за пределами класса. Однако, это также невозможно и для

классов, производных от него. Эту особенность мы и рассмотрим сейчас.

Давайте вспомним, что реализация private-уровня доступа для свойства или метода класса обеспечивается двойным символом подчеркивания «\_\_», который нужно поместить непосредственно перед именем свойства (без пробела), например `__privateProp`, `__privateMethod`.

В нашем базовом классе `Person` есть private-свойство `personID`, и private-метод `showID()`.

```
import random
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."

    def getHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."
```



Создадим экземпляр класса-потомка `Employee` (который мы создали в примерах ранее). И посмотрим, какую информацию о работнике (объекте класса) мы сможем вывести:

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

employee1=Employee("Joe","Black",30, 3000)
employee1.isRetiree()
```

Результат:

```
44
Person first name - Joe; last name - Black; age - 30.
Joe is not retiree
```

Рисунок 23

Обратите внимание, что информация о `personID` было выведена. Однако это произошло в результате вызова метода базового класса `showInfo()`, который и обеспечил вывод этой информации.

Если же мы напрямую попробуем получить доступ к `private`-свойству `personID`, например, создав в классе-потомке метод `showEmployeeID()` для вывода этой информации:

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

    def showEmployeeID(self):
        print(self.__personID)

employee1=Employee("Joe","Black",30, 3000)
employee1.showEmployeeID()
```

то получим исключение

**AttributeError** ✕

Рисунок 24

по причине:

'Employee' object has no attribute '\_Employee\_\_personID'

Рисунок 25

Давайте попробуем изменить наш новый метод `showEmployeeID()`, вызвав в теле `private`-метода `showID()`:

```
def showEmployeeID(self):
    self.__showID()

employee1.showEmployeeID()
```

Получим такое же исключение

**AttributeError** ✕

Рисунок 26

по причине:

'Employee' object has no attribute '\_Employee\_\_showID'

Рисунок 27

Таким образом, Private-свойства и методы действительно доступны только из своего класса.

#### 4.4. Уровень доступа Protected (внутренний или защищенный)

Если нам нужно ограничить доступ к свойствам и методам таким образом, чтобы они были «видны» только внутри самого класса или классов, производных от него, то нам нужен уровень доступа `protected`.

В Python создание защищенного уровня доступа для свойства или метода класса реализуется с помощью символа подчеркивания «`_`», который нужно поместить непосредственно перед именем свойства (без пробела), например `_propName`, `_methodName`.

Давайте изменим уровень доступа для свойства «возраст» в базовом классе:

```
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protected properties
        self._age = age

    # public methods
    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
            last name - {self.lastName};\n\
            age - {self._age}."

    def getHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."
```

Попробуем изменить это свойство внутри метода класса-потомка (добавив метод `changeAge()`):

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
```

```

        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

employee1=Employee("Joe","Black",30, 3000)
employee1.isRetiree()
employee1.changeAge(65)
employee1.isRetiree()

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Joe is not retiree
Person first name - Joe; last name - Black; age - 65.
Joe is retiree

```

Рисунок 28

А теперь проверим, сможем ли мы получить к нему доступ за пределами базового класса и не в производном:

```

employee1._age=20
print(employee1.showInfo())

```

Результат:

```

Person first name - Joe; last name - Black; age - 20.

```

Рисунок 29

Полученные результаты показывают, что мы можем получить доступ к защищенным членам класса вне класса, более того мы даже можем изменить значение нашего защищенного свойства «возраст».

Теперь проверим, как обстоят дела с защищенными методами. Для этого добавим в наш базовый класс `Person` protected-метод `_getFullName()`:

```
import random
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protected properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #protected methods
    def _getFullName(self):
        return f"Person full name:{self.firstName}
            {self.lastName}"

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
            last name - {self.lastName};
            age - {self._age}."

    def getHi(self, msgText):
        print(self.showInfo())
        return f"{msgText}! I am {self.firstName}."
```

Попробуем вызвать его из объекта класса (т. е. за пределами базового класса):

```
person1=Person("Joe", "Black", 30)
print(person1._getFullName())
```

Результат:

```
Person full name:Joe Black
```

Рисунок 30

Точно также, как и при работе с защищенными свойствами, мы можем получить доступ к защищенным методам вне класса.

Возникает сомнение: а могут ли модификаторы доступа фактически изменять уровень доступа? Или они только добавляют новые правила синтаксиса? На самом деле модификаторы защищенного доступа разработаны таким образом, чтобы ответственный программист мог идентифицировать их по соглашению об именах (наличие символа « » вначале имени свойства или метода) и выполнять требуемую операцию с этими защищенными свойствами только в пределах своего класса или производного от него.

То есть модификатор **protected** — это некоторое соглашение (декларация) о том, что данные свойства и методы не нужно использовать за пределами класса и его потомков, т. к. это может нарушить работу класса. Решение о соблюдении этих правил-договорённостей — это ответственность самого программиста. Например, водитель (даже с солидным опытом вождения) не будет отдельно запускать какие-то компоненты мотора своей

машины (хотя он может открыть капот и получить к ним доступ), т.к. эти действия могут привести к негативным последствиям.

## 4.5. Статические методы и методы класса

В Python можно создавать три вида методов: статические методы, методы класса и методы экземпляра класса.

С **методами экземпляра класса** мы уже знакомы. Во всех рассмотренных ранее примерах использовался именно этот тип методов, которые мы вызвали, используя экземпляр класса (разумеется, уже после его создания).

Например, в нашем классе `Person` у нас было два общедоступных метода: `getInfo()` и `sayHi()`.

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
```



```

        return f"Person first name - {self.firstName};
               last name - {self.lastName};
               age - {self._age}."

    def sayHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."

```

И вот так мы вызвали их, используя объект класса:

```

person1=Person("Joe", "Black", 30)
print(person1.getInfo())

```

Методы экземпляра класса являются самым распространенным типом.

Как мы помним первым параметром таких методов всегда является сам экземпляр класса (обозначаемым как `self`), с помощью которого мы внутри метода получаем доступ ко всем составляющим данного класса: свойства, атрибутам и другим методам.

Таким образом, используя методы экземпляров класса мы можем изменять как состояние конкретного объекта, так и самого класса.

## 4.6. Статические методы

Теперь представим, что нам необходимо создать метод, алгоритм которого не предполагает работать с составляющим данного класса, т.е нам совсем не нужно передавать экземпляр класса (`self`) в качестве параметра.

Давайте попробуем объявить в нашем классе `Person` метод `sayGreetings()` без параметров:

```

import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
            last name - {self.lastName};
            age - {self._age}."

    def sayHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."

    def sayGreetings():
        print("Nice to meet you!")

```

Однако, когда мы попытаемся вызвать этот новый метод традиционным образом, используя экземпляр класса:

```

person1=Person("Joe","Black",30)
person1.sayGreetings()

```

Мы получим ошибку:

**TypeError** ✕

Рисунок 31

Получается, что методы, не имеющие в качестве параметра экземпляр класса, не могут быть вызваны объектом.

Как же тогда создать обычный метод с поведением обычной функции внутри класса?

Конечно, мы можем просто создать отдельную функцию за пределами класса:

```
def sayGreetings():
    print("Nice to meet you!")
```

Однако, если в классе `Person` эта функция должны выводить сообщение «*Nice to meet you*», а в классе `Student` — «*Hello!*». То есть здесь нам все-таки нужна связь «функция — класс».

В Python можно реализовать подобный механизм, используя **статические методы**, описываемые с помощью декоратора `@staticmethod`:

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age
```

```

#private properties
self.__personID=random.randint(1,100)

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

person1=Person("Joe", "Black", 30)
person1.sayGreetings()

```

Результат:

**Nice to meet you!**

Рисунок 32

Рассмотрим еще один пример, показывающий полезность статических методов в классе. Допустим, что нам нужно создать класс, предоставляющий некоторый набор методов обработки. Наш класс будет извлекать все числа из

строки (как целые, так и дробные) и выполнять над ними выбранную операцию. Удобно собрать всю необходимую функциональность в одном классе (а не реализовывать отдельными функциями), т. к. при таком подходе можно в дальнейшем сохранить определение класса в отдельном файле (наподобие внешней библиотеки). Тогда мы будем осуществлять его импорт, когда возникает необходимость использования одного из его методов так, как, например, мы подключаем модуль `random`, когда нужно использовать какой-то метод генерации значений.

У такого класса не будет свойств, только методы, реализующие обработку (или какое-то вычисление) входного значения. Причем этим методам абсолютно не нужен экземпляр класса в качестве параметра, они будут работать только со входным значением, над которым нужно выполнить некоторую операцию.

```
import re
class myOperator:

    @staticmethod
    def incrementer(str):
        numbers=[float(s) for s
                  in re.findall(r'[-?\d+\.\d*', str)]
        result=[]
        for number in numbers:
            result.append(number+1)
        return result

    @staticmethod
    def decrementer(str):
        numbers=[float(s) for s
                  in re.findall(r'[-?\d+\.\d*', str)]
```

```

    result=[]
    for number in numbers:
        result.append(number-1)
    return result

userStr="Extract 500 , 100.45, 23.1 and 1000
        from my string"
print(myOperator.incrementer(userStr))
print(myOperator.decrementer(userStr))

```

Результат:

```

[501.0, 101.45, 24.1, 1001.0]
[499.0, 99.45, 22.1, 999.0]

```

Рисунок 33

Как мы видим метод `incrementer()` извлек все числа из строки и каждое из них увеличил на единицу, после чего добавил измененное число в результирующий список. Такие же действия выполнил метод `decrementer()` с единственным отличием — каждое число было уменьшено на единицу.

Фактически, в Python статические методы являются обычными функциями, которые определены в классе и, соответственно, находятся в пространстве имен этого класса. При этом статические методы не могут изменять состояние ни самого класса, ни его объектов

## 4.7. Методы класса

Как методу экземпляра класса всегда первым параметром передается сам экземпляр, так и методу класса первым параметром всегда передается сам класс (обозначаемым

как *cls*). Именно через этот параметр метод класса получает доступ ко все классу (ко всем его составляющим).

Для того, чтобы создать метод класса нужно использовать декоратор `@classmethod`. Если мы создадим метод класса изменяющий, например, значение атрибута класса, то после вызова данного метода все объекты (экземпляры этого класса) будут иметь обновленный атрибут.

Давайте добавим в наш класс `Person` атрибут «хобби» со значением «`Cooking`». Также создадим метод класса `setDefaultHobby(cls)`, который будет изменять значение на «`Drawing`»:

```
import random
class Person:

    hobby="Cooking"

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
```

```

        return f"Person first name - {self.firstName};
               last name - {self.lastName};
               age - {self._age}."

    def sayHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."

    #static methods
    @staticmethod
    def sayGreetings():
        print("Nice to meet you!")

    #class methods
    @classmethod
    def setDefaultHobby(cls):
        cls.hobby="Drawing"

```

Теперь создадим несколько объектов. Но после создания первого объекта вызовем метод класса `setDefaultHobby(cls)`, чтобы проверить, изменилось ли значение атрибута «хобби» у всех тех объектов, которые были созданы после его вызова.

```

person1=Person("Joe","Black",30)
print(person1.hobby)

Person.setDefaultHobby()

person2=Person("Kate","Smith",20)
print(person2.hobby)

person3=Person("Bob","Gordon",40)
print(person3.hobby)

```



Результат:

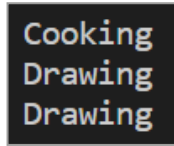


Рисунок 34

То есть методы класса могут изменять состояние класса, что отразится на всех объектах (экземплярах этого класса), но при этом они не могут изменять состояние конкретного объекта.

Теперь рассмотрим еще одну распространенную ситуацию, когда применение методов класса позволяет создавать объекты класса на основе разных данных.

Допустим, что по некоторым людям у нас отсутствует информация о возрасте (которая является обязательным параметром для создания объекта класса [Person](#)). Зато нам известен год рождения этого человека.

Создадим метод класса [basedOnYear\(\)](#), который в результате своей работы вернет экземпляр класса, основываясь на данных об имени, фамилии и годе рождения:

```
import random
from datetime import date
class Person:

    hobby="Cooking"

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
```

```

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

#class methods
@classmethod
def setDefaultHobby(cls):
    cls.hobby="Drawing"

@classmethod
def basedOnBYear(cls,firstName, lastName, bYear):
    personAge=date.today().year - bYear
    return cls(firstName, lastName, personAge)

person1=Person("Joe","Black",30)

```

```
print(person1.getInfo())

person2=Person.basedOnBYear("Kate","Smith",2000)
print(person2.getInfo())
```

Результат:

```
27
Person first name - Joe; last name - Black; age - 30.
64
Person first name - Kate; last _name - Smith; age - 22.
```

Рисунок 35

Теперь предположим, что у нас нет отдельно информации об имени, фамилии и возрасте пользователя, но есть строка, которая содержит все это, например, «*Kate Smith 25*».

Создадим еще один метод класса, который на основе такой строки вернет объект класса [Person](#):

```
import random
from datetime import date

class Person:
    hobby="Cooking"

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age
```

```

        #private properties
        self.__personID=random.randint(1,100)

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

#class methods
@classmethod
def setDefaultHobby(cls):
    cls.hobby="Drawing"

@classmethod
def basedOnBYear(cls,firstName, lastName, bYear):
    personAge=date.today().year - bYear
    return cls(firstName, lastName, personAge)

@classmethod
def basedOnStr(cls,strInf):
    firstName, lastName,age = strInf.split(' ')
    return cls(firstName, lastName,age)

```

```
person3=Person.basedOnStr("Kate Smith 25")
print(person3.getInfo())
```

Результат:

```
74
Person first name - Kate; last name - Smith; age - 25.
```

Рисунок 36

Также методы класса могут оказаться полезными, когда нам нужно изменить логику (алгоритм) работы метода объекта. Предположим, что нам нужно создать метод, который рассчитывает зарплату работника (например, на основе базового оклада и процентной ставки). Размер базового оклада зависит от категории работника: **Junior**, **Middle**, **Senior**, **Lead**.

Создадим класс **Developer** — производный класс от нашего класса **Person**:

```
class Person:
    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
               last name - {self.lastName};\n\
               age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."
```

Мы можем создать в классе `Developer` private-свойство «*rung*» и метод класса `setRung()`, который позволит изменить его и, таким образом, будет «переключать» категорию работника для всех объектов, созданных после его вызова.

```
class Developer (Person):
    def __init__(self, firstName, lastName, age,
                 jobTitle):
        super().__init__(firstName, lastName, age)
        self.jobTitle=jobTitle
        self.__salary=0

    def setBasicSalary(self):
        if self.__rung=='Junior':
            self.__salary=1000
        elif self.__rung=='Middle':
            self.__salary=3000
        elif self.__rung=='Senior':
            self.__salary=5000
        elif self.__rung=='Lead':
            self.__salary=10000

    @classmethod
    def setRung(cls, rungName):
        cls.__rung=rungName

    def calculateSalary(self,perc):
        return self.__salary+self.__salary*perc

    def getInfo(self):
        return super().getInfo() + f"\njobTitle -
            {self.jobTitle};\nrung -
            {self.__rung};\nbasic salary -
            {self.__salary}."
```

Установим ранг разработчика в «Junior»:

```
Developer.setRung("Junior")
```

Теперь создадим несколько объектов, представляющих разработчиков уровня «Junior», установим им базовый оклад, рассчитаем зарплату и выведем информацию о каждом:

```
jun1=Developer("Kate","Smith",22,
               "UI (user interface) designer")
jun2=Developer("Joe","Smith",25,"Back-end developer")

for jun in zip((jun1, jun2), (0.25,0.3)):
    jun[0].setBasicSalary()
    print(jun[0].getInfo())
    print(f"expepected salary:
          {jun[0].calculateSalary(jun[1])}")
```

Результат:

```
Person first name - Kate; last name - Smith; age - 22.
jobTitle - UI (user interface) designer;
rung - Junior;
basic salary - 1000.
expepected salary:1250.0
Person first name - Joe; last name - Smith; age - 25.
jobTitle - Back-end developer;
rung - Junior;
basic salary - 1000.
expepected salary:1300.0
```

Рисунок 37

Далее «переключим» класс в уровень «Middle» и повторим процедуру.

```

Developer.setRung("Middle")

midl1=Developer("Bob","Dilan",32,"Web developer")
midl2=Developer("Anna","Morning",35,"Data scientist")
midl3=Developer("Helen","Adams",42,
               "Systems analyst")

for midl in zip((midl1, midl2, midl3),(0.2,0.37, 0.15)):
    midl[0].setBasicSalary()

    print(midl[0].getInfo())
    print(f"expexted salary:{midl[0].
          calculateSalary(midl[1])}")

```

Результат:

```

Person first name - Bob; last name - Dilan; age - 32.
jobTitle - Web developer;
rung - Middle;
basic salary - 3000.
expexted salary:3600.0
Person first name - Anna; last name - Morning; age - 35.
jobTitle - Data scientist;
rung - Middle;
basic salary - 3000.
expexted salary:4110.0
Person first name - Helen; last name - Adams; age - 42.
jobTitle - Systems analyst;
rung - Middle;
basic salary - 3000.
expexted salary:3450.0

```

Рисунок 38

Таким образом, методы класса могут использоваться для изменения (настройки) некоторых параметров (свойств) класса, которые будут учтены во всех последующих



экземплярах класса (объектов, созданных после вызова данных методов класса). А эти измененные свойства далее повлекут изменение алгоритма поведения этих новых объектов.

## 4.8. Множественное наследование и MRO (порядок разрешения методов)

В рассмотренных ранее примерах все наши производные классы наследовали свойства (методы) только одного базового класса. Однако в Python также предусмотрены средства для реализации множественного наследования.

**Множественное наследование** — это возможность производного класса наследовать особенности и функциональность более, чем одного базового класса. Таким образом, при множественном наследовании абсолютно все свойства и все методы всех классов-родителей наследуются классом-потомком.

Синтаксис множественного наследования не отличается от синтаксиса одиночного — мы просто перечисляем в круглых скобках после имени производного класса имена всех базовых классов. Допустим, что у нас есть два базовых класса `baseClass1` и `baseClass2`, а нам необходимо создать производный класс `derivedClass`, который будет наследовать всю функциональность обоих классов (рис. 39).

Общий синтаксис множественного наследования в этом случае будет иметь вид:

```
class baseClass1:  
    pass
```

```
class baseClass2:
    pass

class derivedClass (baseClass1, baseClass2):
    pass
```

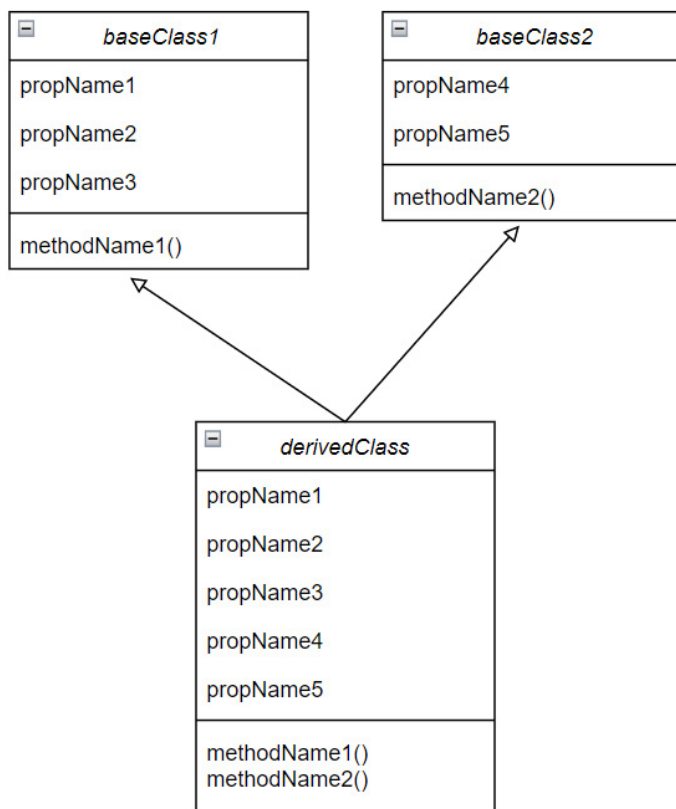


Рисунок 39

Рассмотрим на примере. Допустим, что у нас есть класс «**Книга**» и класс «**Файл**». На их основе нам нужно создать производный класс «**Электронная книга**» (рис. 40):

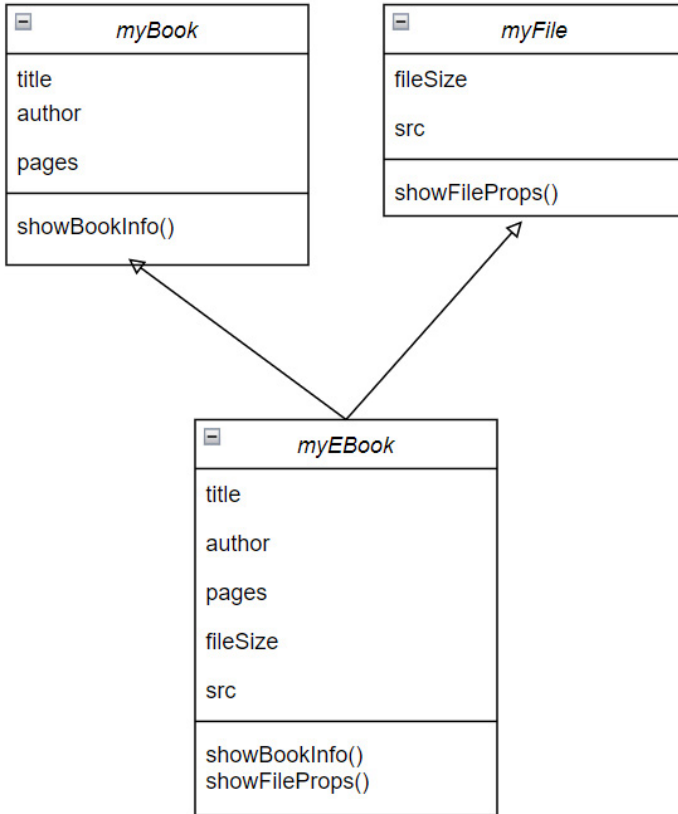


Рисунок 40

```

class myBook:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showBookInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))
  
```

```
class myFile:
    def __init__(self, fileSize, src):
        self.fileSize = fileSize
        self.src = src

    def showFileProps(self):
        print("File size: {}".format(self.fileSize))
        print("File source: {}".format(self.src))

class myEBook(myBook, myFile):
    pass
```

Теперь создадим экземпляр нашего производного класса `myEBook`:

```
eBook1=myEBook()
```

Однако, эта строка кода создаст нам исключение:

**TypeError** ✕

Рисунок 41

И причина его такова:

```
myBook.__init__() missing 3 required positional arguments: 'title', 'author', and 'pages'
```

Рисунок 42

Давайте разбираться, в чем тут дело.

В нашем примере тело нашего производного класса отсутствует, в нем не определен даже его конструктор. В этом случае будет вызван конструктор того базового класса, имя которого мы указали первым при определении

производного класса (т. е. будет вызван конструктор класса `myBook`). А как следует из определения класса `myBook`, его конструктор имеет три параметра: `title`, `author`, `pages`, которые и перечислены выше в тексте сообщения об ошибке.

Давайте добавим в код необходимые для конструктора аргументы и вызовем метод `showBookInfo()` для отображения информации о названии, авторе книги и о количестве страниц в ней:

```
eBook1=myEBook("Python Crash Course","Eric Matthes", 624)
eBook1.showBookInfo()
```

```
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
```

Рисунок 43

Однако, нам также необходимо вызвать конструктор класса `myFile` для заполнения тех свойств электронной книги, которые связаны с файлом.

Ведь если мы попробуем вызвать метод `showFileProps()`, чтобы проверить, что находится в соответствующих свойствах:

```
eBook1.showFileProps()
```

То эта строка (рис. 44) кода вызовет исключение (рис. 45)

**AttributeError** ✕

Рисунок 44

**'myEBook' object has no attribute 'fileSize'**

Рисунок 45

Таким образом, нам нужно создать в нашем производном классе конструктор, в котором последовательно вызвать конструкторы классов-родителей:

```
class myEBook(myBook,myFile):

    def __init__(self,title, author, pages,fileSize, src):
        myBook.__init__(self,title, author, pages)
        myFile.__init__(self,fileSize, src)

eBook1= myEBook("Python Crash Course","Eric Matthes",
               624, 1024,"https://www.amazon.co.uk/
               dp/1593276036/?tag=adnruk-21")

eBook1.showBookInfo()
eBook1.showFileProps()
```

Результат:

```
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
File size: 1024
File source: https://www.amazon.co.uk/dp/1593276036/?tag=adnruk-21
```

Рисунок 46

Теперь рассмотрим ситуацию, когда в каждом из классов-предков есть метод с одинаковым названием. Изменим названия методов `showBookInfo()` и `showFileProps()` на `showInfo()`, после чего вызовем данный метод нашего объекта `eBook1`.

```

class myBook:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

class myFile:
    def __init__(self, fileSize, src):
        self.fileSize = fileSize
        self.src = src

    def showInfo(self):
        print("File size: {}".format(self.fileSize))
        print("File source: {}".format(self.src))

class myEBook(myBook, myFile):
    def __init__(self, title, author, pages, fileSize, src):
        myBook.__init__(self, title, author, pages)
        myFile.__init__(self, fileSize, src)

eBook1 = myEBook("Python Crash Course", "Eric Matthes",
                 624, 1024, "https://www.amazon.co.uk/
                 dp/1593276036/?tag=adnrk-21")
eBook1.showInfo()

```

Результат:

```

Title: Python Crash Course
Author: Eric Matthes
Pages: 624

```

Рисунок 47

Как мы видим, фактически был вызван метод `showInfo()` класса `myBook`. Это произошло в соответствии с линеаризацией класса `myEBook`.

**Линеаризация класса** это механизм формирования списка имен классов при наследовании, который будет задавать порядок поиска свойств и методов при обращении к ним. Для нашего класса `myEBook` данный список будет такой: `[myEBook, myBook, myFile, object]`. Такой порядок тоже известен, как **порядок разрешения методов** (*Method Resolution Order, MRO*).

Таким образом при вызове метода `showInfo()` для экземпляра класса `myEBook` интерпретатор вначале выполнит поиск в текущем классе (`myEBook`). Если поиск не даст результата, то он будет продолжен в базовых классах в том порядке, как они были указаны при определении производного класса (т. е. вначале будет выполнен поиск в классе `myBook`). Так как на этом этапе результат поиска будет успешным, то произойдет выполнение кода метода `showInfo()` класса `myBook`.

Для того, чтобы проверить, в каком порядке будут проинспектированы базовые классы, воспользуемся методом класса `mro()`:

```
print(myEBook.mro())
```

```
[<class '__main__.myEBook'>, <class '__main__.myBook'>, <class '__main__.myFile'>, <class 'object'>]
```

Рисунок 48

Итак, мы с вами рассмотрели базовую (и в тоже время наиболее распространенную) ситуацию множественного наследования.



А что, если у каждого из базовых классов, от которых наследуется наш производный класс, тоже есть классы-предки? Причем это класс-предок у базовых классов общий? Эта ситуация известна как проблема ромбовидного (алмазного) наследования (*diamond problem*), общий вид которого приведен на рисунке 49:

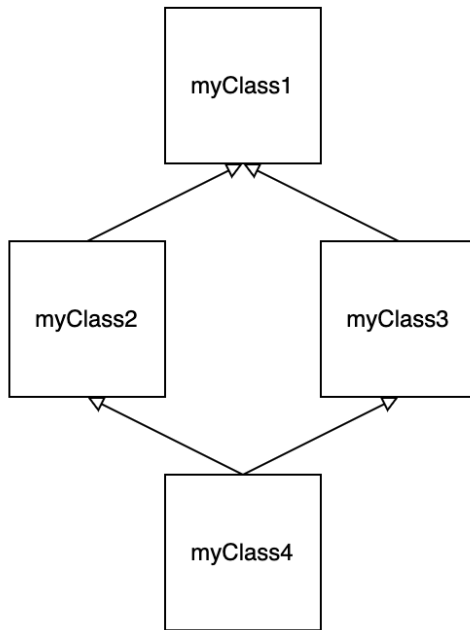


Рисунок 49

```
class myClass1:
    def sayHi(self):
        print("Hi from Class1")

class myClass2(myClass1):
    def sayHi(self):
        print("Hi from Class2")
```

```
class myClass3(myClass1):
    def sayHi(self):
        print("Hi from Class3")

class myClass4(myClass2, myClass3):
    pass
```

Если есть некоторый метод `sayHi()`, который переопределен в одном из классов `myClass2` и `myClass3` или в обоих сразу, то возникает неоднозначность, какой из вариантов методов `sayHi()` должен наследовать `myClass4`.

```
myObj = myClass4()
myObj.sayHi()
```

Результат:

```
Hi from Class2
```

Рисунок 50

Как мы видим, был вызван метод `sayHi()` из класса `myClass2`, т. к. в списке MRO этот класс-предок идет ранее, чем `myClass3`, и в нем определен нужный нам метод `sayHi()`.

Теперь, давайте удалим из класса `myClass2` метод `sayHi()`:


```
class myClass1:
    def sayHi(self):
        print("Hi from Class1")

class myClass2(myClass1):
    pass
```

```
class myClass3(myClass1):
    def sayHi(self):
        print("Hi from Class3")

class myClass4(myClass2, myClass3):
    pass
```

Результат:



```
Hi from Class3
```

Рисунок 51

В это раз был вызван метод `sayHi()` из класса `myClass3`, т. к. несмотря на то, что в списке MRO этот класс — предок идет ранее, чем `myClass3`, в нем не нашлось метода `sayHi()`, а вот в определении базового класса `myClass3`, который идет следующим списке MRO, этот метод был найден.

Главное преимущество множественного наследования — это возможность наследовать особенности и функциональность более, чем одного базового класса.

Но его существенный недостаток — это неоднозначность (путаница), которая возникает в ситуации, когда несколько базовых классов (или более того, и производный класс тоже) реализуют метод с одним и тем же именем (как в наших примерах выше). В связи с этим множественное наследование может затруднить понимание и поддержку кода, в том числе и из-за сложных отношений между классами.

Поэтому разрешение (синтаксисом языка) множественного наследования серьезно усложняет правила перегрузки методов, что приводит к значительному повышению

требований к уровню программистов — разработчиков таких классов.

В некоторых языках программирования, таких как Java, JavaScript, вообще не поддерживается механизм множественного наследования. В C++ и в Python эта возможность существует. Более того, Python имеет хорошо продуманный и четко контролируемый подход к множественному наследованию. Однако, правильное использование этих средств требует высокого уровня Python-разработчика со знаниями всех правил и особенностей механизма множественного наследования.

## 5. Полиморфизм

**Полиморфизм** — это один из основополагающих принципов ООП, который предоставляет возможность использовать один и тот же интерфейс для разных базовых форм.

В полиморфизме метод (оператор) может обрабатывать объекты разным способом в зависимости от типа класса или типа данных. Вначале давайте рассмотрим эту возможность на простых примерах, чтобы лучше понять все нюансы.

Мы знаем, что оператор `+` является одним из наиболее распространенных в программировании. Однако в Python у него не одно функциональное назначение. Для численных данных этот оператор реализует арифметическое сложение, а для строковых — конкатенацию (объединение строк). Однако, несмотря на разные типы данных суть выполняемой операции примерно одинакова — сложение операндов.

```
number1=2
number2=5.7
print(number1+number2) #7.7

str1="Hello!"
str2="Python"
print(str1+str2) #Hello!Python
```

Встроенная функция `len()` вычисляет длину объекта в зависимости от типа его класса. Если объект является

строкой, то `len()` возвращает количество символов, а если объект является списком — количество элементов в списке. Если же мы передаем `len()` в качестве аргумента словарь, то результат ее работы — это количество ключей словаря. То есть, как и в случае с «+», смысл результатов одинаков — количество.

```
print(len("Student")) #7
print(len(["Python", "C#", "JavaScript"])) #3
print(len({"firstName": "Jane", "lastName": "Smith"})) #2
```

Концепция полиморфизма активно используется при создании методов класса. Python позволяет различным классам иметь методы с одинаковыми именами. Именно благодаря этой особенности мы сможем потом обобщить вызов этих методов (например, для нескольких объектов, обрабатывая их в цикле), не придавая особого значению объекту, с которым мы работаем на текущем шаге. Давайте рассмотрим на примере. Создадим два класса `Film` и `Book`, которые имеют схожую структуру и метод `showInfo()`.

```
class Film:
    def __init__(self, originalTitle, director, genre):
        self.originalTitle = originalTitle
        self.director = director
        self.genre = genre

    def showInfo(self):
        print("Original title: {}".format(self.originalTitle))
        print("Director: {}".format(self.director))
        print("Genre: {}".format(self.genre))
```

```

class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

film1=Film("The Godfather", "Francis Ford Coppola",
           "Crime, drama")
book1=Book("Python Crash Course", "Eric Matthes", 624)

for item in (film1, book1):
    item.showInfo()

```

Результат:

```

Original title: The Godfather
Director: Francis Ford Coppola
Genre: Crime, drama
Title: Python Crash Course
Author: Eric Matthes
Pages: 624

```

Рисунок 52

Обратите внимание, что в примере мы не связывали эти классы вместе каким-либо образом. Мы просто «упаковали» эти два разных объекта в кортеж и вывели информацию о каждом из объектов. Это возможно именно благодаря полиморфизму.

## 5.1. Перегрузка операторов

Как мы уже заметили, анализируя функциональность оператора `+`, в Python есть операторы, которые выполняют разные операции в зависимости от типа операндов. Это результат перегрузки операторов.

**Перегрузка операторов в Python** — это возможность изменять (варьировать) их функциональность (выполняемые ими операции) в зависимости от контекста (класса), используя специальные методы в классах.

Итак, у нас есть класс `Book` и два объекта (две книги, которые прочитал пользователь).

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 170)
```

Предположим, что нам необходимо узнать общее количество прочитанных пользователем страниц, т. е. сложить две книги.

Однако в результате такой строки кода:

```
print(book1+book2)
```



получим ошибку

**TypeError** ✕

Рисунок 53

`unsupported operand type(s) for +: 'Book' and 'Book'`

Рисунок 54

Интерпретатор Python не знает, как «складывать» два объекта `Book`. Ведь оператор «+» не определен для объектов такого класса, как, например, он определен для чисел и строк. Для решения этой задачи, нужно «научить» оператор + работать с операндами типа `Book`. И для этого мы познакомимся с магическими методами в Python.

## 5.2. Magic-методы, конструкторы

**Magic-методы** — это специальные методы, названия которых обрамлены двумя нижними подчеркиваниями (начинаются и заканчиваются двойным подчеркиванием). Поэтому их часто называют **dunder** методы, как сокращение от фразы «*double underscore*».

Magic-методы отвечают за реализацию некоторых стандартных способностей объектов и автоматически вызываются при использовании этих способностей.

Например, когда в примере выше мы выполняли операцию `number1 + number2`, то фактически это преобразовывалось в вызов метода `number1.__add__(number2)`. То есть magic-метод `__add__` реализует операцию сложения.

На самом деле в Python многие встроенные операторы и функции вызывают magic-методы.

Добавляя (переопределяя) эти magic-методы в наши объекты, мы сможем применять нужные нам встроенные операторы и функции над объектами.

Мы уже использовали один из магических методов — конструктор `__init__()`, с помощью которого мы определяем особенности инициализации наших объектов. Однако, когда мы создаем экземпляр нашего класса:

```
objName = someClassName()
```

то не метод `__init__()` вызывается в первую очередь.

Первым вызывается метод `__new__()`, который и создает экземпляр нашего класса. В качестве аргумента данный метод принимает класс (который принято обозначать как `cls`). Основное назначение данного метода — это именно создание экземпляра класса (объекта). Этот новый объект далее инициализируется с помощью метода `__init__()`.

Давайте детальнее рассмотрим этот процесс на примере, продемонстрировав последовательность вызовов этих двух методов при создании объекта.

```
class Class1:
    def __new__(cls):
        print("Hi! I am __new__ magic method.")
        return super(Class1, cls).__new__(cls)
    def __init__(self):
        print("Hi! I am __init__ magic method.")

obj1=Class1()
```

Результат:

```
Hi! I am __new__ magic method.
Hi! I am __init__ magic method.
```

Рисунок 55

Однако, наибольшим преимуществом магических методов в Python является то, что они предоставляют нам механизм, с помощью которого мы можем «научить» наши объекты поведению встроенных типов. Рассмотрим наиболее популярные из них.

### 5.3. Реализация магических методов

Вернемся к нашему классу `Book`:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
```

Если мы попытаемся передать наш объект `book1` функции `print()`, то результат не будет информативным для пользователя (мы не выведем информации о значениях свойств нашего объекта):

```
book1=Book("Python Crash Course","Eric Matthes", 624)
print(book1)
```

Результат:

```
<__main__.Book object at 0x000001BDD220BB20>
```

Рисунок 56

Если мы определим в классе магический метод `__str__()`, который отвечает за строковое представление объекта, то мы можем решить эту проблему (т. к. данный метод вызывается автоматически при вызове функции `print()`, которая выводит строковое представление переданного ей аргумента, т. е. результат работы метода `__str__()`).

Метод `__str__()` в качестве аргумента принимает экземпляр класса и возвращает строку, которую мы сформируем в теле метода нужным нам образом.

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return f"Title: {self.title},
                author: {self.author},
                pages: {self.pages}"
print(book1)
```

Результат:

```
Title: Python Crash Course, author: Eric Matthes, pages: 624
```

Рисунок 57

Предположим, что нам нужно сравнить две книги. Если просто попытаться сравнить два объекта напрямую:

```
book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 170)
print(book1==book2) #False
```

Результат всегда будет **False**, т.к. это абсолютно разные сущности. Дело в том, что при создании экземпляра изменяемого типа данных (к которым относятся и пользовательские классы) Python выделяет новый фрагмент памяти и назначает уникальный идентификатор (**id**) для каждого объекта. И сравнение экземпляров изменяемых типов данных по умолчанию происходит именно по их уникальным идентификаторам. Но проблема даже не в этом, а в том, что пользователю сравнение двух книг представляется обычной задачей: он может сравнивать их по цене, по автору, по количеству страниц. Главное для него понимать, что является критерием сравнения. Поэтому в программе, которая «работает» с книгами, он ожидает наличие такой же возможности. Нам нужно реализовать такой механизм сравнения книг, которые в нашей программе представлены объектами, т. е. отобразить эти особенности в поведении объектов - книг.

В Python есть множество магических методов, предназначенных для реализации интуитивно понятных операций сравнения для объектов, которые позволяют реализовывать сравнение с помощью привычных операторов: **==**, **<**, **>**, **<=**, **>=**, **!=**.

Таблица 2

Magic – метод	Оператор сравнения, чье поведение определяется методом
<code>__eq__</code>	<code>==</code>
<code>__ne__</code>	<code>!=</code>
<code>__lt__</code>	<code>&lt;</code>
<code>__le__</code>	<code>&lt;=</code>
<code>__gt__</code>	<code>&gt;</code>
<code>__ge__</code>	<code>&gt;=</code>

Предположим, что книги совпадают, если совпадают их названия и автор. Одна книга больше другой, если количество страниц у нее больше.

Добавим эти возможности в наш класс `Book`:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"Title: {self.title},\n        author: {self.author},\n        pages: {self.pages}"

    def __eq__(self, otherObj):
        if self.author==otherObj.author and\n            self.title==otherObj.title:
            return True
        else:
            return False

    def __gt__(self, otherObj):
        if self.pages>otherObj.pages:
            return True
        else:
            return False

book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",\n        "Douglas Crockford", 170)
book3=Book("Python Crash Course","Eric Matthes", 700)

print(book1==book3)
print(book1>book2)
```

Результат:

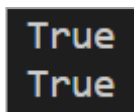


Рисунок 58

Допустим, что у нас имеется информация о количестве отзывов на книгу в течении последнего года. Добавим в наш класс свойство `feedbacksN` — список, содержащий двенадцать целочисленных значений (сколько отзывов было на книгу в определенный месяц, т. е. позиция значения в списке соответствует номеру месяца).

Также внесем изменения в метод `__str__()`, чтобы информация об отзывах также отображалась, как и другие свойства.

```
class Book:
    def __init__(self, title, author, pages, feedbacksN):
        self.title = title
        self.author = author
        self.pages = pages
        self.feedbacksN=feedbacksN

    def __str__(self):
        return f"Title: {self.title},
                author: {self.author},
                pages: {self.pages},
                feedbacksN:{self.feedbacksN}"
```

Было бы удобно, чтобы у объекта (определенной книги) была возможность получать доступ к информации (количеству отзывов) за конкретный месяц с помощью привычного индексирования, например, так `book1[monthNum]`.

Для решения этой задачи нам нужно определить в нашем классе `Book` метод `__getitem__(self, key)`, который определяет поведение при доступе к элементу, используя нотацию `self[key]`.

```
import random

class Book:
    def __init__(self, title, author, pages, feedbacksN):
        self.title = title
        self.author = author
        self.pages = pages
        self.feedbacksN=feedbacksN

    def __str__(self):
        return f"Title: {self.title},\n        author: {self.author},\n        pages: {self.pages},\n        feedbacksN:{self.feedbacksN}"

    def __eq__(self, otherObj):
        if self.author==otherObj.author and\n            self.title==otherObj.title:\n            return True\n        else:\n            return False

    def __gt__(self, otherObj):\n        if self.pages>otherObj.pages:\n            return True\n        else:\n            return False

    def __getitem__(self,ind):\n        if 0<=ind<=11:\n            return self.feedbacksN[ind]\n        else:\n            return -1
```



```
PythonFeedbacks=[random.randint(50,300) for i
                  in range(12)]
book1=Book("Python Crash Course", "Eric Matthes",
          624, PythonFeedbacks)
print(book1[2]) #265
```

Если нам нужно обеспечить также и обновление (запись информации) о количестве отзывов за определенный месяц (в формате `book1[someInd] = newInf`), то в этом поможет магический метод `__setitem__(self, key, value)`.

Дополнительную информацию по магическим методам в Python можно посмотреть по [ссылке](#).

## 6. Создание и управление поведением экземпляров класса

### 6.1. Функторы

Ранее мы уже познакомились с понятием, назначением и особенностями применения замыканий в Python.

Давайте вспомним основные нюансы на примере:

```
def doExercise1(var1):  
    def doExercise2(var2):  
        return var1**var2  
    return doExercise2
```

Функция-замыкание `doExercise2()` использует в своем теле только переменную `var1` (аргумент охватывающей функции `doExercise1()`). Поэтому значение переменной `var1` (число 2) будет «запомнено» (несмотря на то, что выполнение функции `doExercise1(2)` было завершено).

```
case1=doExercise1(2)  
  
print(case1(5)) #32  
print(case1(10)) #1024
```

В строке вызова «`case1(5)`» программа использует для вычисления «запомненное» значение переменной `var1` (2), а значение 5 используется как аргумент функции-замыкания `doExercise2()`.

Также и при вызове «`case1(10)`» используется «запомненное» значение переменной `var1` (2) и аргумент для функции-замыкания `doExercise2()` — значение 10. Основное преимущество и назначение этого подхода — это «запоминание» необходимых данных. В объектно-ориентированной парадигме такое «запоминание» также может оказаться полезным.

Например, нам нужно создать класс `UserPlayer`, в котором предполагается сохранять и обновлять состояние его «кошелька» в игре. Разумеется, мы можем просто создать свойство «`wallet`» (в этом случае лучше будет использовать модификатор доступа `private`) и метод для его обновления `updateWallet()`.

```
class UserPlayer:
    def __init__(self, name):
        self.name=name
        self.__wallet=100

    def updateWallet(self,coins):
        self.__wallet+=coins

    def showWallet(self):
        print(f"You have {self.__wallet} coins now.")

user1=UserPlayer("Joe")
user1.updateWallet(50)
user1.showWallet()
```

Результат:

```
You have 150 coins now.
```

Рисунок 59

Однако, что если на следующем шаге нам нужно создать класс игрока-бота `BotPlayer`, для которого также необходимо сохранять и обновлять «кошелек» по такому же алгоритму? Также нам может понадобится возможность устанавливать начальное состояние «кошелька». Проблема в том, что свойство «`wallet`» является `private`, поэтому придется создать отдельный метод для этого и вызвать его для установки начального состояния «кошелька».

Удобнее реализовать алгоритм обновления состояния «кошелька» отдельно, например, в виде отдельного класса. Более того, такой подход позволит нам также скрыть реальную реализацию алгоритма обновления «кошелька». В такой задаче очень полезными оказываются функторы.

**Функторы** — это объекты (экземпляры классов-функторов), которые можно вызывать, как обычные функции.

В Python любой класс, в котором определен специальный «магический» метод `__call__()` (позволяющий перегрузить оператор `()`) является функтором. Основное преимущество функторов заключается в том, что они могут хранить информацию о состоянии — то, что нам сейчас и нужно.

Создадим функтор для установки начального состояния «кошелька» и его обновления на указанное число монет:

```
class WalletFunctor:

    def __init__(self, startCoins=100):
        self.__startCoins = startCoins

    def __call__(self, coins=0):
        return self.__startCoins+coins
```

Наш метод `__call__()` должен обязательно возвращать новое значение `__startCoin` (а не просто обновлять его), т.к. изначально в нашей идее функтор `WalletFuncтор` — это состояние (память) «кошелька» (вызвали функтор — получили состояние).

Теперь создадим объекты — экземпляры этого класса — функтора для игрока — пользователя и игрока — бота (с разным начальным состоянием «кошелька»). И обновим «кошельки» бота и пользователя.

```
userWallet=WalletFuncтор()
print(f"Start state of user wallet:
      {userWallet()} coins")
print(f"State of user wallet after updating to 50 coins:
      {userWallet(50)} coins")

botWallet=WalletFuncтор(50)
print(f"Start state of bot wallet:
      {botWallet()} coins")
print(f"State of user bot after updating to 20 coins:
      {botWallet(20)} coins")
```

Результат:

```
Start state of user wallet: 100 coins
State of user wallet after updating to 50 coins: 150 coins
Start state of bot wallet: 50 coins
State of user bot after updating to 20 coins: 70 coins
```

Рисунок 60

Теперь создадим класс `UserPlayer` с двумя `private`-свойствами: «кошельком» — `__wallet` и «настройщиком кошелька», экземпляром класса — функтора `WalletFuncтор`,

который задает алгоритм начальной настройки и дальнейшего обновления «кошелька».

```
class UserPlayer:
    def __init__(self, name):
        self.name=name
        self.__walletSetter=WalletFunctor()
        self.__wallet=self.__walletSetter()

    def updateWallet(self,coins=0):
        self.__wallet=self.__walletSetter(coins)

    def showWallet(self):
        print(f"{self.name}!
              You have {self.__wallet} coins now.")
```

Теперь создадим объект игрока, покажем начальное состояние его «кошелька», а потом, после обновления на 50 монет, еще раз проверим «кошелек» игрока.

```
user1=UserPlayer("Joe")
user1.showWallet()
user1.updateWallet(50)
user1.showWallet()
```

Результат:

```
Joe! You have 100 coins now.
Joe! You have 150 coins now.
```

Рисунок 61

Полная последовательность всех выполненных шагов представлена на рисунке 62.

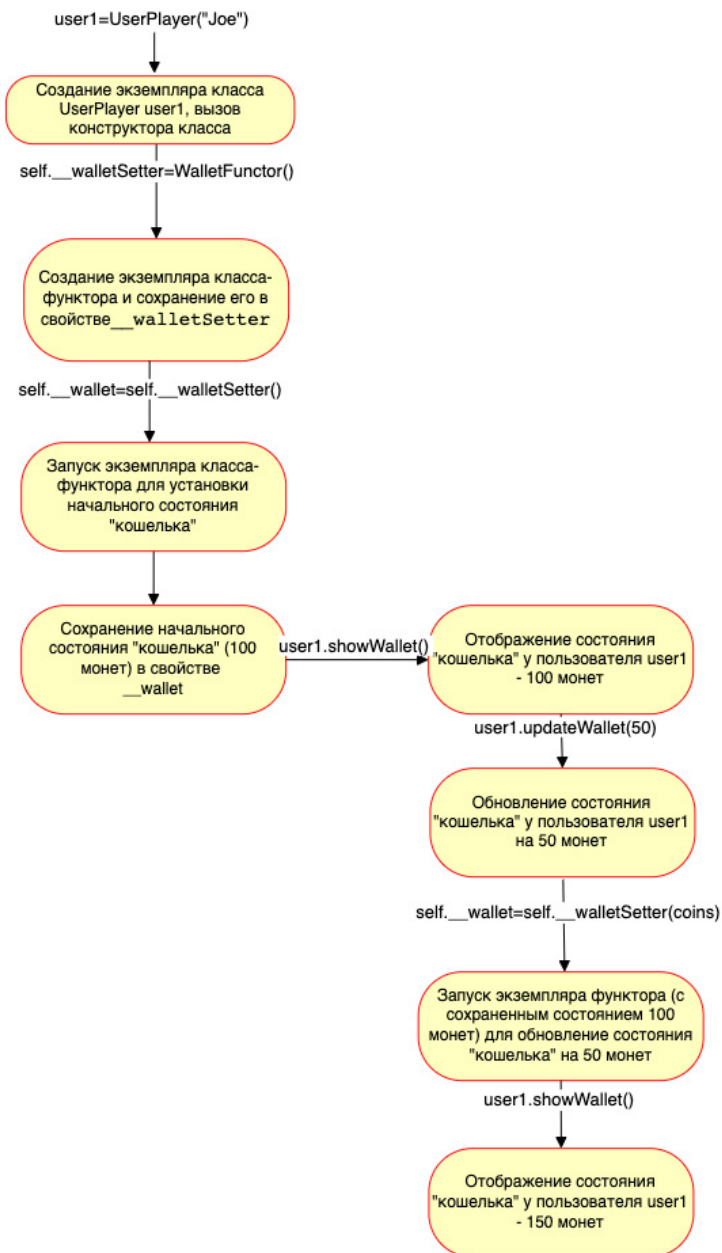


Рисунок 62

## 6.2. Декораторы

Мы с вами уже знакомы с декораторами — функциями, такими функциями — «обёртками», которые позволяют нам расширить функциональность существующей функции без прямого изменения кода в ее теле. Давайте вспомним особенности их применения на примере.

Допустим, что у нас есть список с ценами на товары в долларах. И функция, которая переводит цену товара в долларах в соответствующий гривневый эквивалент:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)

USDrate=27.5

def toPriceNew(priceList):
    return list(map(lambda x: x*USDrate, priceList))
```

Однако сейчас на все товары действует скидка (например, 15%) и нам нужно перевести цены в гривны и еще дополнительно учесть скидку.

Скидка — непостоянная характеристика товара, поэтому изменять код функции `toPriceNew()` нам нет смысла. Мы создадим декоратор, который «применит» скидку после перевода цены в другую валюту:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)

def changePriceDecorator_v1(myFunction):
    print("Hello! Let's change your prices...")
    def simpleWrapper(argList):
```



```

    print("I've got list of prices with {}
          elements. Function starts working...".
          format(len(argList)))
    resutl=myFunction(argList)
    resutlwithDisc=list(map(lambda x: x*(1-0.15),
                           resutl))
    print("Let's set a discount..")
    return resutlwithDisc
return simpleWrapper

pricesToGRN = changePriceDecorator_v1(toPriceNew)
print(toPriceNew(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts working...
Let's set a discount..
[2345.4474999999998, 818.125, 1589.26625, 596.0625] _

```

Рисунок 63

Мы также знаем и второй способ «декорирования» функций — это использование инструкции «**@имя\_декоратора**», которую нужно поместить над объявлением «декорируемой» функции.

```

@changePriceDecorator_v1
def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))

print(toPriceNew(pricesUSD))

```

В Python декораторами могут быть не только функции, но и классы. Мы уже заметили, что методы в Python

фактически являются обычными функциями, у которых первым аргументом должна быть ссылка на экземпляр класса (объект — [self](#)). Таким образом, мы можем декорировать и метод класса, расширив его функциональность без изменения исходного кода метода.

Рассмотрим **декорирование методов класса** на примере нашего класса [Book](#):

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))
```

Допустим, что нам перед выводом информации о книге с помощью метода [showInfo\(\)](#) нужно вывести фразу «[General information:](#)». Для этого создадим соответствующую функцию декоратор:

```
def methodDecorator(method_to_decorate):

    def wrapper(self):
        print("General information:")
        method_to_decorate(self)
    return wrapper
```

И перед объявлением метода [showInfo\(\)](#) в классе [Book](#) добавим инструкцию [@methodDecorator](#):

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    @methodDecorator
    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

book1=Book("Python Crash Course","Eric Matthes", 624)
book1.showInfo()
```

Результат:

```
General information:
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
```

Рисунок 64

А можно ли создать **класс-декоратор**? Ведь мы уже знакомы с функторами и знаем, что объекты (экземпляры классов-функторов) можно вызывать, как обычные функции, значит, можно и «поручить» им процесс декорирования.

Рассмотрим эту задачу на простом примере. Допустим, что у нас есть функция, выполняющая сложение двух чисел и возвращающая полученный результат. Давайте рассмотрим сценарий, в котором мы хотели бы добавить к этой функции дополнительную функциональность

(возвести возвращаемое значение в квадрат), не меняя исходный код. Мы можем добиться этого с помощью класса-декоратора.

Для этого мы определим в классе-декораторе два метода: `__init__()` и `__call__()`. Когда мы декорируем функцию классом, то функция автоматически передается в качестве аргумента конструктору `__init__()`. Мы создадим в нашем классе private-свойство для хранения этой функции.

Также декоратор должен иметь возможности вызываемой функции, которая принимает функцию, которую необходимо декорировать и возвращает ее декорированную версию (которую можно запустить). Поэтому нам нужно определить в классе — декораторе метод `__call__()`, в теле которого мы как раз и добавим нужную нам новую функциональность.

```
class myDecorator:
    def __init__(self, fn):
        self.fn=fn
    def __call__(self, num1, num2):
        return self.fn(num1, num2)**2

@myDecorator
def addNums(x, y):
    return x+y

print(addNums(2,3)) #25
```

В методе `__call__()` мы, кроме первого аргумента `self` (самого объекта) указали еще два аргумента, потому что самой нашей исходной функции, которую мы будем декорировать, необходимо два аргумента. Внутри метода мы вызываем нашу функцию (ссылка на нее установлена

через свойство класса `fn`) с двумя переданными значениями, после чего возводим полученный результат в квадрат.

Далее перед объявлением нашей функции `addNums()` мы добавили инструкцию `@myDecorator`. Теперь в результате вызова нашей функции `addNums(2,3)` мы получим результат работы ее декорированной версии — число 25 ( $(2+3)**2 = 25$ ). Мы помним, что функторы (как и замыкания) позволяют «запоминать» нужные данные, в том числе и результаты предыдущих вызовов функции. Класс-декоратор обладает возможностями функтора (т.к в нем определен метод `__call__()`), поэтому мы можем добавить в наш класс `myDecorator` свойство — список `memoryCall`, который будет накапливать результаты работы декорированной функции. Изначально (при вызове конструктора) `memoryCall` будет пустым списком. А далее (внутри метода `__call__()`) мы будем записывать в него результаты вызовов декорированной функции.

Также нам нужно добавить метод `showMemoryState()` для просмотра состояния «памяти».

```
class myDecorator:
    def __init__(self, fn):
        self.fn=fn
        self.__memoryCall=[]

    def __call__(self, num1, num2):
        self.__memoryCall.append(self.fn(num1, num2)**2)
        return self.fn(num1, num2)**2

    def showMemoryState(self):
        print(f"Current memory state:
              {self.__memoryCall}")
```

```

@myDecorator
def addNums(x, y):
    return x+y

print(addNums(2, 3))
addNums.showMemoryState()

print(addNums(3, 3))
addNums.showMemoryState()

print(addNums(4, 3))
addNums.showMemoryState()

```

Результат:

```

25
Current memory state: [25]
36
Current memory state: [25, 36]
49
Current memory state: [25, 36, 49]_

```

Рисунок 65

Вначале мы запустили декорированную версию функции `addNums(2,3)`, получив результат `25`, который после своего вычисления был добавлен в список `memoryCall` (до его возвращения функцией). Вызов метода `showMemoryState()` вывел текущее содержимое списка `memoryCall` — `[25]`, который на данном этапе содержит только один результат (после одного вызова декорированной версии функции `addNums()`)

Следующий вызов `addNums(3,3)` пополнил список `memoryCall` на результат второго вызова функции, теперь `memoryCall` уже содержит два значения: `[25, 36]`.

И последний (третий) вызов `addNums(4,3)` добавил в `memoryCall` третье значение: `[25, 36, 49]`.

### 6.3. Управляемые атрибуты объекта — `property()`

Как мы знаем, атрибуты объекта используются для описания основных характеристик данного экземпляра класса (конкретного объекта), которые задаются в момент создания этого объекта. Достаточно часто, чтобы не путать атрибуты объекта с атрибутами класса (значения которых одинаковы для всех экземпляров класса) их называют свойствами. И мы тоже практиковали такой подход во многих примерах ранее.

Например, для класса `Product` (товар) такими атрибутами объекта (свойствами) могут быть название товара, его цена, размер скидки, число отзывов, а атрибутами класса (известные также, как переменные класса) — какие-то общие характеристики для всех (или многих) объектов, например, категория товара «бытовая техника».

Совокупность всех атрибутов объекта отображает **состояние** (`state`) объекта, которое во многих задачах нужно анализировать и даже изменять (управлять состоянием объекта). Например, нам нужно изменить размер скидки на определенный товар в зависимости от числа отзывов, т. е. мы проверяем состояние данного объекта (текущее число отзывов о товаре) и изменяем состояние (обновляем размер скидки). Как правило, у нас есть как минимум два способа управления атрибутом: напрямую (обращаясь к нему по имени через соответствующий объект) или с помощью методов класса. Мы уже знакомы с модификаторами доступа `public`, `private` и `protected`

и знаем, как режим доступа влияет на способ работы (взаимодействия) с атрибутом.

Давайте вспомним, что в Python по умолчанию все атрибуты объекта (как и методы) имеют уровень доступа `public`. Нужно отметить, что в большинстве практических случаев мы используем именно этот модификатор. Таким образом, атрибуты становятся частью общедоступного интерфейса (API) для работы с нашим классом, т. е. каждый разработчик получает к ним доступ и сможет изменять их значения непосредственно в своем коде. Проблемы возникают, когда нам нужно модифицировать внутреннюю реализацию данного атрибута объекта, но мы не имеем право нарушить (фактически изменив) эту часть API класса, т.к. это повлияет на стабильность и работоспособность всего кода, который связан с использованием этого атрибута. Рассмотрим эту ситуацию на примере.

Допустим, что у нас есть класс `Product` — товар в онлайн — супермаркете.

```
class Product:
    def __init__(self, name, price, discount=0.25):
        self.name = name
        self.price = price
        self.discount = discount

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.discount)} grn.")

    def toUSD(self, usdExchRate):
        return self.price*(1-self.discount)/
              usdExchRate
```



У каждого товара есть атрибут «**discount**» — размер скидки, представляющий собой число в диапазоне от 0 до 1 (процент скидки уже переведенный в число для сокращения вычислений). Информация о цене товара, в том числе и долларах (для конвертации цены предусмотрен отдельный метод **toUSD()**) выводится с учетом скидки:

```
item1=Product("Lipton Peach Iced", 42, 0.3)
item2=Product("Pure Apple Juice", 28)

item1.showInfo()
print(f"Price in USD$:{item1.toUSD(30)}")
item2.showInfo()
print(f"Price in USD$:{item2.toUSD(30)}")
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
```

Рисунок 66

Мы можем выводить и, при необходимости, изменять скидку на товар (экземпляра класса **Product**):

```
item1.discount=0.5
print(f"New discount for {item1.name} -
      {item1.discount*100}%.")
```

Теперь предположим, что наш важный заказчик пришел с новым требованием — скидка на товар должна храниться в виде процентов (25%, а не 0.25) в свойстве «**discount-Percentage**», которое также должно быть общедоступным.

На данном этапе удаление атрибута «**discount**» и добавление атрибута «**discountPercentage**» приведет к поломке не только двух методов, но и всех строк кода, которые напрямую работали с атрибутом «**discount**». Конечно, нам нужно другое решение этой проблемы, а не удаление атрибута «**discount**».

В некоторых языках программирования (Java и C++) для предотвращения подобных проблем рекомендуется не создавать общедоступные атрибуты, а обеспечивать к ним доступ только через специальные методы: геттеры (*Getters*) — для получения значения атрибута и сеттеры (*Setters*) — для его установки. При таком подходе строк кода, в которых разработчик напрямую обращается к атрибуту, просто не будет в программе. Соответственно, трудностей, связанных со множественным правками кода из-за изменившегося названия атрибута объекта, у нас просто не будет. Давайте попробуем применить этот механизм к нашей задаче.

Вначале создадим класс **Product** с атрибутом объекта «**discount**», а потом рассмотрим, насколько просто нам будет перейти на использование атрибута «**discountPercentage**» со значением скидки в виде процентов.

```
class Product:
    def __init__(self, name, price, discount=0.25):
        self.name = name
        self.price = price
        self._discount = discount

    def getDiscount(self):
        return self._discount
```

```

def setDiscount(self,value):
    self._discount=value

def showInfo(self):
    print (f"name:{self.name}, price with discount:
           {self.price*(1-self.getDiscount())}
           grn.")

def toUSD(self,usdExchRate):
    return self.price*(1-self.getDiscount())/
           usdExchRate

item1=Product("Lipton Peach Iced", 42, 0.3)
item2=Product("Pure Apple Juice", 28)

item1.showInfo()
print(f"Price in USD$:{item1.toUSD(30)}")
item2.showInfo()
print(f"Price in USD$:{item2.toUSD(30)}")

item1.setDiscount(0.5)
print(f"New discount for {item1.name} -
      {item1.getDiscount()*100}%.")

```

Результат:

```

name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
New discount for Lipton Peach Iced - 50.0%.

```

Рисунок 67

Обратите внимание, что мы создали наш атрибут объекта «**discount**» уровня доступа «**protected**» (указав перед именем атрибута один символ нижнего подчеркивания

«\_»), чтобы оно было доступно производным классам, но при этом разработчик понимал, что напрямую к этому атрибуту обращаться нельзя, т. е. строк кода типа:

```
item1._discount=0.5
print(f"New discount for {item1.name} -
      {item1._discount*100}%.")
```

у нас в программе не будет.

Теперь, когда у нас появляется требование об удалении атрибута «**discount**» и добавлении атрибута «**discountPercentage**», то эти изменения коснутся только самого класса **Product**:

```
class Product:
    def __init__(self, name, price,
                  discountPercentage = 25):
        self.name = name
        self.price = price
        self._discountPercentage = discountPercentage

    def getDiscount(self):
        return self._discountPercentage/100

    def setDiscount(self,value):
        self._discountPercentage=value*100

    def showInfo(self):
        print (f"name:{self.name},
              price with discount:
              {self.price*(1-self.getDiscount())}grn.")

    def toUSD(self,usdExchRate):
        return self.price*(1-self.getDiscount())/
              usdExchRate
```

а не всех строк кода, которые работали с указанным атрибутом:

```
item1=Product("Lipton Peach Iced", 42, 30)
item2=Product("Pure Apple Juice", 28)

item1.showInfo()
print(f"Price in USD$:{item1.toUSD(30)}")
item2.showInfo()
print(f"Price in USD$:{item2.toUSD(30)}")

item1.setDiscount(0.5)
print(f"New discount for {item1.name} -
      {item1.getDiscount()*100}%.")
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
New discount for Lipton Peach Iced - 50.0%.
```

Рисунок 68

Мы даже можем добавить в сеттер проверку корректности значения атрибута объекта «`discountPercentage`»: например, чтобы в качестве нового значения атрибута принималось только число в диапазоне от 10 до 75 (т. е. размер скидки на товар меньше 10% и более 75% устанавливать нельзя):

```
def setDiscount(self,value):
    if 0.1<=value<=0.75:
        self._discountPercentage=value*100
```

```
else:
    print(f"Discount value less than 10%
          or greater than 75% is not possible!")
```

Тогда подобная строка кода (с попыткой установить размер скидки 80%):

```
item1=Product("Lipton Peach Iced", 42, 30)
item1.showInfo()
item1.setDiscount(0.8)
```

вызовет сообщение о недопустимости такой скидки для продукта:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Discount value less than 10% or greater than 75% is not possible!
```

Рисунок 69

Однако, мы помним, что модификатор «protected» — это всего лишь соглашение о том, что напрямую обращаться к **protected** — атрибутам нельзя. Фактически наличие этого модификатора не мешает нам или другим программистам обращаться к данным атрибутам:

```
item1._discountPercentage=90
item1.showInfo()
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
name:Lipton Peach Iced, price with discount:4.199999999999999 grn.
```

Рисунок 70

При этом наличие сеттера в классе никак не поможет в исключении этой недопустимой ситуации.

Кроме этого, мы можем установить недопустимое значение размера скидки и при вызове конструктора (например, просто забыв, что мы уже перешли от «`discount`» к «`discountPercentage`» с другими единицами измерения и что при вызове конструктора нужно указывать % скидки, т. е. значение от 10 до 75):

```
item2=Product("Pure Apple Juice", 28, 0.8)
item2.showInfo()
```

Здесь мы получаем сразу две ошибки: не те единицы измерения для скидки (число вместо процентного значения) и недопустимый размер скидки (фактически мы же хотели установить 80% скидки, что не предусмотрено для наших товаров). Ни одна из этих проблемных ситуаций не была «обнаружена» сеттером:

```
name:Pure Apple Juice, price with discount:27.776 grn.
```

Рисунок 71

И, наконец, мы все еще не выполнили требование нашего заказчика об общедоступности атрибута объекта «`discountPercentage`» для любого кода, который будет «работать» с классом `Product`.

Именно в связи с указанными проблемами (вернее, с их «не решением» сеттерами и геттерами) указанный подход не является распространенным в Python.

Однако, более существенным недостатком является тот факт, что все строки кода, в которых программист

традиционно мог использовать для обращения к атрибуту выражения типа `obj.PropName` должны быть изменены на `obj.getPropName`. Точно также и вместо `obj.PropName = value` придется использовать `obj.setPropName(value)`. Это неудобство может привести к значительному количеству ошибок, т. к. нарушается традиционный механизм для работы с атрибутами объектов (причем при работе с `public`-атрибутами мы придерживаемся стандартного подхода, что еще больше усугубит путаницу).

## 6.4. Свойства — `property()`

В отличие от Java и C++ в Python существует способ удобного решения вышеуказанной проблемы — это `property()`. Данный подход позволяет создавать методы, которые «ведут себя» как свойства, не нарушая традиционного подхода работы с ними. Фактически функция `property()` позволяет создавать **управляемые атрибуты объекта**, избегая необходимости работы с геттерами и сеттерами. Данная функция является встроенной, поэтому нам не нужно выполнять никаких шагов, связанных с импортом.

Общий синтаксис `property()`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Первые два аргумента принимают функциональные объекты (чаще всего имена методов текущего класса), которые будут выполнять роль геттера (`fget`) и сеттера (`fset`). Таким образом, с помощью аргумента `fget` мы задаем имя метода, который будет вызываться при использовании выражения типа `obj.PropName`, а аргумент `fset` задает имя метода, вызываемого выражением `obj.PropName = value`.



Таким образом, с помощью `property()` мы можем прикрепить геттер и сеттер к нужному свойству класса, скрывая таким образом внутреннюю логику (реализацию) этого свойства и обеспечивая стабильный API для любой необходимой модификации.

Используя `property()` мы также можем указать способ обработки процесса удаления атрибута и предоставить соответствующую строку документации для него.

Не всегда удобно использовать такое длинное название термина, как «управляемые атрибуты объекта», поэтому для сокращения их часто называют просто «свойствами» (из-за использования функции `property()` для их создания).

Рассмотрим их использование на нашем примере с классом `Product`.

```
class Product:
    def __init__(self, name, price, discountPercentage=25):
        self.name = name
        self.price = price
        self.__discountPercentage = discountPercentage

    def getDiscount(self):
        return self.__discountPercentage

    def setDiscount(self, value):
        if 0.1<=value<=0.75:
            self.__discountPercentage=value*100
        elif 10<=value<=75:
            self.__discountPercentage=value
        else:
            print(f"Discount value less than 10%
                  or greater than 75% is not
                  possible!")
```

```

def showInfo(self):
    print (f"name:{self.name},
           price with discount:
           {self.price*(1-self.getDiscount()/100)}
           grn.")

def toUSD(self, usdExchRate):
    return self.price*(1-self.getDiscount())/
           usdExchRate

discountPercentage = property(
    fget=getDiscount,
    fset=setDiscount
)

```

Вначале мы изменили тип модификатора на `private`, для того, чтобы избежать ситуаций прямого обращения к свойству, при котором не происходит вызова метода `getDiscount()` — проблема, с которой мы сталкивались при некорректном обращении с `protected`-свойствами.

Также мы обеспечили требование общедоступности свойства «`discountPercentage`», т. е. возможность использования выражение типа `obj.discountPercentage` или `obj.discountPercentage=value`:

```

item1=Product("Lipton Peach Iced", 42, 30)
item1.showInfo()
print(item1.discountPercentage)

item1.discountPercentage=0.25
print(item1.discountPercentage)

item1.discountPercentage=60
print(item1.discountPercentage)

```

```
item1.discountPercentage=85
print(item1.discountPercentage)
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
30
25.0
60
Discount value less than 10% or greater than 75% is not possible!
60
```

Рисунок 72

Как мы видим, независимо от единиц измерения (0.25 или 60) размер скидки устанавливается верно. Также программой (вызываемым неявно в выражении `obj.discountPercentage=value` методом `setDiscount()`) обнаруживается некорректное значение (85) размера скидки, которое не устанавливается нашему товару (сохранено предыдущее значение скидки в 60%).

Однако, при использовании некорректного значения для аргумента — скидки при вызове конструктора:

```
item2=Product("Pure Apple Juice", 28, 0.5)
```

вызов метода `setDiscount()` автоматически не происходит.

Для устранения этого момента, который может привести к установке некорректного значения размера скидки внесем изменения в конструктор класса:

```
def __init__(self, name, price, discountPercentage=25):
    self.name = name
    self.price = price
```

```
self.__discountPercentage = discountPercentage*100
    if 0<discountPercentage<1
    else discountPercentage
```

Теперь размер скидки будет верно устанавливаться конструктором как при использовании процентного значения (например, 30), так и при использовании числа (например, 0.5, т. е. 50% скидки):

```
item2=Product("Pure Apple Juice", 28, 0.5)
item2.showInfo()
print(item2.discountPercentage)
```

```
name:Pure Apple Juice, price with discount:14.0 grn.
50.0
```

Рисунок 73

Нам осталось познакомиться еще с двумя аргументами функции `property()`: `fdel` и `doc`.

Иногда нам нужно удалить существующий атрибут объекта, причем именно у конкретного экземпляра класса, а не у самого класса. Например, у товара «Coffee» нет скидки вообще. И чтобы исключить возможность (например, случайной) установки скидки для этого товара мы можем просто удалить атрибут «`discountPercentage`» у конкретного объекта — товара «Coffee» (уже после его создания с помощью конструктора класса).

С помощью аргумента `fdel` мы можем определить метод класса, который будет вызываться в случае удаления свойства через выражение `del obj.имя_свойства`.

А аргумент `doc` может быть полезен для создания описания свойства, которое можно увидеть при использовании встроенной функции `help()`.

Для свойства «`discountPercentage`» создадим метод `delDiscount()`, который будет выводить сообщение о невозможности удаления данного свойства. Для вызова метода `delDiscount()` при использовании выражения `del obj.discountPercentage` установим его в качестве значения аргумента `fdel` в функции `property()` для управляемого атрибута `discountPercentage`.

```
class Product:
    def __init__(self, name, price,
discountPercentage=25):
        self.name = name
        self.price = price
        self.__discountPercentage =
            discountPercentage*100
            if 0<discountPercentage<1
            else discountPercentage

    def getDiscount(self):
        return self.__discountPercentage

    def setDiscount(self,value):
        if 0.1<=value<=0.75:
            self.__discountPercentage=value*100
        elif 10<=value<=75:
            self.__discountPercentage=value
        else:
            print(f"Discount value less than 10% or
greater than 75% is not possible!")

    def delDiscount(self):
        print("It is impossible to delete this property!")
```

```

def showInfo(self):
    print (f"name:{self.name}, price with discount:
           {self.price*(1-self.getDiscount()/100)}
           grn.")

def toUSD(self,usdExchRate):
    return self.price*(1-self.getDiscount())/
           usdExchRate

discountPercentage = property(
    fget=getDiscount,
    fset=setDiscount,
    fdel=delDiscount,
    doc="Product discount property."
)

item1=Product("Lipton Peach Iced", 42,
              "Super iced tea!",30)
del item1.discountPercentage
help(Product.discountPercentage)

```

Результат:

```

It is impossible to delete this property!
Help on property:

    Product discount property.

```

Рисунок 74

Таким образом, использование механизма управляемых атрибутов `property()` позволяет легко вносить изменения во внутреннюю реализацию свойств класса, при этом не нарушая стабильности общедоступного API класса.

Помимо применения встроенной функции `property()` для создания управляемых атрибутов в Python предусмотрен

альтернативный способ — использование декоратора `@property`.

Подход с использованием декоратора для создания управляемых атрибутов требует вначале (первым) определить метод — геттер. Перед его определением (строкой выше) нужно использовать декоратор `@property`

При этом имя метода-геттера должно совпадать с именем создаваемого управляемого атрибута. Того самого атрибута, для которого будет возможен общий доступ при использовании класса дальше.

После создания метода — геттера нужно создать метод — сеттер, имя которого должно точно совпадать с именем определенного выше метода — геттера. А перед его определением нужно указать декоратор следующего вида: `@propertyName.setter`.

Давайте перепишем наш класс `Product` с использованием декораторов:

```
class Product:
    def __init__(self, name, price, discountPercentage=25):
        self.name = name
        self.price = price
        self.__discountPercentage =
            discountPercentage* 100
            if 0<discountPercentage<1
            else discountPercentage

    @property
    def discountPercentage(self):
        return self.__discountPercentage

    @discountPercentage.setter
    def discountPercentage(self,value):
```

```

if 0.1<=value<=0.75:
    self.__discountPercentage=value*100
elif 10<=value<=75:
    self.__discountPercentage=value
else:
    print(f"Discount value less than 10% or
          greater than 75% is not
          possible!")

@discountPercentage.deleter
def discountPercentage(self):
    print("It is impossible to delete this
          property!")

def showInfo(self):
    print (f"name:{self.name}, price with discount:
           {self.price*(1-self.__discountPercentage/
                       100)} grn.")

def toUSD(self,usdExchRate):
    return self.price*(1-self.__discountPercentage)/
           usdExchRate

item1=Product("Lipton Peach Iced", 42, 30)
del item1.discountPercentage

```

Результат:

**It is impossible to delete this property!**

Рисунок 75

Как мы видим, при таком способе нам не нужно создавать методы с именами типа наших предыдущих `getDiscount()`, `setDiscount()`, `delDiscount()` и далее реализовывать их «связку» с соответствующими параметрами



функции `property()`. Теперь у нас есть три метода с одним и тем же ясным и описательным именем, которое совпадает с именем создаваемого управляемого атрибута `discountPercentage`.

## 6.5. Дескрипторы

В предыдущих примерах мы рассмотрели использования функции `property()` и декоратора `@property` для создания управляемых атрибутов объекта (или, как мы их называли, свойств — `property`).

Управляемые атрибуты объекта — это свойства объекта, которые имеют «связанное» поведение, т. е. это такие свойства, при обращении к которым выполняются некоторые алгоритмы (реализованные внутри методов класса).

Как мы уже знаем, в Python предусмотрено три варианта обращения к свойству:

- получение значения свойства, используя выражение `objName.propertyName`;
- изменение (установка) значения свойства: `objName.propertyName = value`;
- удаление атрибута: `del objName.propertyName`.

Каждый из перечисленных способов работы со свойством можно «связать» с нужным методом класса, т. е. переопределить поведение, связанное с определенным вариантом доступа.

В Python такую «привязку» методов класса к свойству мы можем реализовать или с помощью функции `property()`, или с помощью декоратора `@property`. Для того, чтобы

понять, как же все это работает внутри, нам нужно познакомиться с понятием дескриптора.

Именно дескрипторы являются основой таких механизмов и возможностей, как функция `property()`, декораторы `@staticmethod` (статические методы) и `@classmethod` (методы класса), а также функции `super()`.

Фактически каждое из управляемых свойств уже является дескриптором. Давайте разберемся, почему это так.

**Дескриптор** — это объект, для которого определен хотя бы один из методов `__get__()`, `__set__()` или `__delete__()`.

Когда этот объект (фактически ссылку на него) мы присваиваем свойству класса, то в результате получаем управляемый атрибут объекта (свойство со «связанным поведением»). Весь этот процесс реализуется через механизм **протокола дескрипторов**.

В Python предусмотрено три метода, из которых и состоит протокол дескриптора (это соответствует количеству способов обращения к атрибуту, перечисленным выше):

- `__set__(self, obj, value)` — данный метод вызывается при изменении значения свойства с помощью выражения `objName.propertyName = value`
- `__get__(self, obj, type=None)` — вызывается при получении значения свойства, т. е. при выражении `objName.propertyName`
- `__delete__(self, object)` — вызывается каждый раз, когда используется оператор `del` для удаления данного свойства.

Класс, в котором определен только метод `__get__`, называется **дескриптором без данных**, а если класс реализует

методы `__get__` и `__set__`, то это — дескриптор данных. Давайте создадим простой класс-дескриптор и на примере разберемся, что из себя представляют параметры в методах протокола дескриптора.

```
class MyDescriptor1:
    def __init__(self, name=""):
        print("Descriptor's __init__ was started...")
        self.name = name

    def __get__(self, obj, objtype):
        print(f"Descriptor's __get__ (instance={obj},
              objtype={objtype}) was started...")
        return "{} was processed".format(self.name)

    def __set__(self, obj, name):
        print(f"Descriptor's __set__ (instance={obj},
              name={name}) was started...")

        if isinstance(name, str):
            self.name = name
        else:
            print("Name should be string")
```

Наш дескриптор `MyDescriptor1` — это дескриптор данных, который устанавливает и возвращает значения в обычном режиме (как сеттеры и геттеры, рассмотренные нами ранее). Также предусмотрен вывод сообщения, отображающих процесс запуска определенного метода протокола дескриптора и его параметров.

Теперь создадим класс `User`, в котором будет управляемый атрибут объекта `userName`, поведение которого «контролируется» дескриптором `MyDescriptor1`.

```
class User:
    userName = MyDescriptor1()

user1 = User()
user1.userName = "Jack"
print(user1.userName)
```

Результат:

```
Descriptor's __init__ was started...
Descriptor's __set__(instance=<__main__.User object
    at 0x10b763880>, name=Jack) was started...
Descriptor's __get__(instance=<__main__.User object
    at 0x10b763880>, objtype=<class '__main__.User'>)
    was started...
Jackjtfas processed
```

Рисунок 76

Итак, когда мы создали экземпляр класса `User` (объект `user1`) в процессе инициализации управляемого атрибута объекта `userName` был запущен конструктор дескриптора данных `MyDescriptor1`.

Далее в ходе установки нового значения для `userName` был запущен метод `__set__()` нашего дескриптора данных. Здесь мы можем видеть, что второй параметр метода `obj` содержит соответствующий экземпляр класса `User` (ссылку на объект `user1`), а последний параметр `name` — значение, которое мы присваиваем в `userName`.

И последним у нас был запущен метод дескриптора `__get__()`, когда мы выводим значение `userName` в консоль. Как и у метода `__set__()` второй параметр метода `obj` содержит ссылку на объект `user1`, а вот третий параметр

(objtype) — это класс, экземпляром которого является объект `user1`.

Давайте попробуем установить в качестве значения для `userName` число:

```
user1 = User()

user1.userName = "Jack"
print(user1.userName)

user1.userName = 111
print(user1.userName)
```

Результат:

```
Descriptor's __init__ was started...
Descriptor's __set__(instances__main__.User object
    at 0x10e63c8b0>, name=Jack) was started...
Descriptor's __get__(instances__main__.User object
    at 0x10e63c8b0>, objtype=<class '__main__.User'>)
    __was__ started...
Jack was processed
Descriptor's __set__(instances__main__.User object at
    0x10e63c8b0>, name=111) was started...
Name should be string
Descriptor's __get__(instances__main__.User object at
    0x10e63c8b0>, objtype=<class '__main__.User'>)
    was started...
Jack was processed
```

Рисунок 77

Как мы видим, наш дескриптор корректно обработал эту ошибочную ситуацию. Теперь предположим, что нам нужно добавить в наш класс `User` информацию о

возрасте пользователя. Конечно, нам будет также необходимо контролировать ввод значений для этого свойства (например, пусть диапазон допустимых значений для возраста пользователя будет [18,75]). Для «управления» взаимодействием с этим свойством создадим второй дескриптор данных `MyDescriptor2` и внесем соответствующие изменения в класс `User`.

```
class MyDescriptor1:
    def __init__(self, name=""):
        print("Descriptor's __init__ was started...")
        self.name = name

    def __get__(self, obj, objtype):
        print(f"Descriptor's __get__ (instance={obj},
              objtype={objtype}) was started...")
        return "{} was processed".format(self.name)

    def __set__(self, obj, name):
        print(f"Descriptor's __set__ (instance={obj},
              name={name}) was started...")
        if isinstance(name, str):
            self.name = name
        else:
            print("Name should be string")

class MyDescriptor2:
    def __init__(self, age = 18):
        self.age = age

    def __set__(self, obj, age):
        if not 18 <= age <= 75:
            print('Valid age must be in [18, 75]')
        else:
            self.age = age
```

```

def __get__(self, obj, objtype):
    return self.age

class User:
    userName = MyDescriptor1()
    userAge=MyDescriptor2()

user1 = User()
user1.userName = "Jack"
print("User name: {}".format(user1.userName))
print("User age: {}".format(user1.userAge))

user1.userAge = 4
print("User age: {}".format(user1.userAge))
user1.userAge = 100
print("User age: {}".format(user1.userAge))
user1.userAge = 25
print("User age: {}".format(user1.userAge))

```

Результат:

```

Descriptor's __init__ was started...
Descriptor's __set__ (instance=<__main__. User object
    at 0x1073868e0>, name=Jack) was started...
Descriptor's __get__ (instances__main__.User object
    at 0x1073868e0>, objtype=<class '__main__.User'>)
    was started...,
User name: Jack was processed
User age: 18
Valid age must be in [18, 75]
User age: 18
Valid age must be in [18, 75]
User age: 18
User age: 25

```

Рисунок 78

Как мы видим обе попытки установить некорректное значение для возраста пользователя были остановлены дескриптором.

Однако, возникают два существенных вопроса:

1. А чем, собственно, использование дескрипторов лучше, чем использование функции `property()` или декоратора `@property`? Ведь, используя их механизмы, мы ранее уже контролировали корректность значений свойств.
2. Можно ли «привязать» работу конструктора класса (с контролируемыми свойствами) к дескриптору? У нашего класса `User` пока нет конструктора вообще и это существенно понижает удобство его использования.

Основное преимущество дескрипторов по сравнению с `property()` (или `@property`) — это управление работой нескольких свойств класса, которые имеют похожее поведение (или ограничения на значения).

Допустим, что у класса `User` есть два отдельных поля для хранения имени: `firstName`, `lastName`. Каждое из этих полей должно не может быть пустым или меньше определенной длины (3 символа для имени и 4 для фамилии). Давайте реализацию этого класса, используя декоратор `@property`:

```
class User:
    def __init__(self, firstName, lastName):
        self.__firstName = firstName
        self.__lastName = lastName

    @property
    def firstName(self):
        return self.__firstName
```



```

@firstName.setter
def firstName(self, value):
    if not isinstance(value, str):
        print('The first name must be a string!')
    elif len(value) == 0:
        print('The first name cannot be empty!')
    elif len(value) < 3:
        print('Too short value for the first name!')
    self.__firstName = value

@property
def lastName(self):
    return self.__lastName

@lastName.setter
def lastName(self, value):
    if not isinstance(value, str):
        print('The last name must be a string!')
    elif len(value) == 0:
        print('The last name cannot be empty!')
    elif len(value) < 4:
        print('Too short value for the last name!')
    else:
        self.__lastName = value

```

В данной реализации «геттеры» возвращают значения свойств `firstName` и `lastName`, а «сеттеры» проверяют их новые значения перед присвоением свойствам.

```

user1=User("Joe", "Black")
print(f"User1: {user1.firstName} {user1.lastName}")

user1.firstName=""
user1.firstName="AB"
user1.lastName=""
user1.lastName="ABC"

```

Код выполняет свои функции.

```
User1: Joe Black
The first name cannot be empty!
Too short value for the first name!
The last name cannot be empty!
Too short value for the last name!
```

Рисунок 79

Однако, мы можем легко заметить избыточность в коде: не только логика, но и ее реализация у «сеттеров» свойств, как и «геттеров» полностью совпадает.

А теперь представим, что у нас много свойств строкового типа. Тогда уровень избыточности кода еще больше возрастет.

Более верным решением будет создать дескриптор, реализующий управление строковыми свойствами:

```
class StrDescriptor:
    def __init__(self, minLen, name=""):
        self.name=name
        self.minLen=minLen

    def __get__(self, obj, objtype):
        return self.name

    def __set__(self, obj, value):
        if not isinstance(value, str):
            print('The value must be a string!')
        elif len(value) == 0:
            print('The value cannot be empty!')
        elif len(value) < self.minLen:
            print('Too short value!')
        else:
            self.name = value
```

И далее использовать его при определении управляемых атрибутов класса `User`.

```
class User:
    firstName=StrDescriptor(3)
    lastName=StrDescriptor(4)

    def __init__(self, firstName, lastName):
        self.firstName = firstName
        self.lastName = lastName

user1=User("Joe", "Black")
print(f"User1: {user1.firstName} {user1.lastName}")

user1.firstName=""
user1.firstName="AB"

user1.lastName=""
user1.lastName="ABC"

user2=User("AB", "ABC")
```

Результат:

```
User1: Joe Black
The value cannot be empty!
Too short value!
The value cannot be empty!
Too short value!
Too short value!
Too short value!
```

Рисунок 80

Как мы видим, по полученному результату, не только строки кода, содержащие явные обращения к свойствам `firstName` и `lastName`, обрабатываются соответствующими

дескрипторами — были «остановлены» ситуации задания некорректных значений (пустая и слишком короткая строки). Также дескрипторы не допустили установку таких же неверных значений конструктором (`User("AB", "ABC")`). Это происходит потому, что внутри тела конструктора находятся выражения типа `objName.propertyName = value`, вызывающие запуск метода `__set__()` протокола дескриптора, который как раз и содержит соответствующие проверки.

Таким образом, в ситуациях, когда в классе есть несколько управляемых свойств, логика работы которых (или проверки их значений) одинакова, то использование дескрипторов поможет сделать программу более структурированной и позволит избежать дублирования кода.

# 7. Метаклассы

## 7.1. Модель метаклассов

Наверное, каждый из нас не раз слышал вопрос «А могут ли программы сами генерировать код (составлять другие программы) вместо программистов?» И, конечно, мы знаем, что ответ на этот вопрос утвердительный. Более того, существует отдельный подход — метапрограммирование.

**Метапрограммирование** — это вид программирования, который позволяет нам разрабатывать код, управляющий другим кодом, в том числе и генерировать его.

Фреймворки, «интеллектуальные» функции различных библиотек, упрощающие работу с кодом и его генерацию, активно используют приемы метапрограммирования. И мы с вами также уже использовали одну из его форм — **декораторы**, которые позволяют нам расширить функциональность уже существующей функции без прямого изменения кода в ее теле.

Помимо декораторов в Python предусмотрена еще одна форма метапрограммирования — метаклассы.

Нужно сказать, что метаклассы поддерживаются не всеми объектно-ориентированными языками программирования. Причем те языки программирования, которые обеспечивают механизмы метаклассов, значительно различаются по способу их реализации.

**Метакласс** — это специальный тип класса, экземплярами (объектами) которого являются классы. Поэтому можно сказать, что метакласс определяет (задает)

поведение класса таким же образом, как класс задает поведение объектов (своих экземпляров).

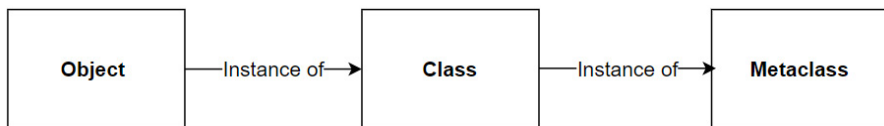


Рисунок 81

Мы знаем: все в Python является объектом. И тот факт, что класс также является объектом, еще раз доказывает истинность этого утверждения.

Для того, чтобы лучше понять особенности, процессы создания и работы метаклассов, давайте еще раз вспомним основополагающие аспекты обычных классов, их объектов и отношений между ними.

Предположим, у нас есть класс `MyClass1` (без свойств и методов) и его экземпляр `myObj1`:

```
class MyClass1():
    pass

myObj1 = MyClass1()
print(myObj1)
```

Вывод объекта `myObj1`, показывает, что он действительно является экземпляром класса `MyClass1`:

```
<__main__.MyClass1 object at 0x00000216BC4D81F0>
```

Рисунок 82

Также мы знаем, что информацию о типе данных можно получить с помощью встроенной функции `type()`.

Попробуем использовать ее для определения типа нашего объекта (ведь класс — это пользовательский тип данных):

```
print(type(myObj1))
```

Полученный результат содержит название класса, который создал данный экземпляр, т. е. тип данных переменной `myObj1` — `MyClass1`:

```
<class '__main__.MyClass1'>
```

Рисунок 83

Значит, согласно логике, представленной на рисунке выше, мы можем использовать функцию `type()` для определения типа нашего класса `MyClass1` (метакласса, экземпляром которого он является):

```
print(type(MyClass1))
```

или таким образом:

```
print(type(type(myObj1)))
```

Результат достаточно неожиданный:

```
<class 'type'>
```

Рисунок 84

Мы увидели название встроенного метакласса (`type`), который создал объект с идентификатором `MyClass1`, когда мы использовали ключевое слово `class` для его объявления. А что насчет метакласса для встроенных типов (классов `str`, `int`, `list`)? Давайте еще поэкспериментируем:

```

myNumber=10
myStr="Admin"
myList=[1,2,3]

print(type(myNumber))
print(type(type(myNumber)))

print(type(myStr))
print(type(type(myStr)))

print(type(myList))
print(type(type(myList)))

```

Результат:

```

<class 'int'>
<class 'type'>
<class 'str'>
<class 'type'>
<class 'list'>
<class 'type'>

```

Рисунок 85

Полученные результаты показывают, что `type` является метаклассом для классов `int`, `str`, `list`, которые являются встроенными классами в Python. Таким образом, `type` является метаклассом по умолчанию.

Выше мы уже использовали встроенную функцию `type()` для определения типа объекта, который мы передавали в качестве аргумента. Однако, функциональность `type` намного шире: она может также возвращать новый тип класса (фактически метакласс). Для этого нужно вызвать ее не с одним, а с тремя аргументами.



Общий синтаксис:

```
type(ClassName, (ParentClassesTuple), propDict)
```

- **ClassName** — имя нового класса (строка);
- **ParentClassesTuple** — кортеж, содержащий имена базовых классов, от которых наследуется создаваемый класс **ClassName** (в случае отсутствия наследования — пустой **()**);
- **propDict** — словарь, содержащий имена атрибутов и методов создаваемого класса.

Давайте разберемся, каким образом это происходит на примере:

```
MyClass = type('MyClass', (BaseClass), clsDict)
```

Объект **MyClass** является классом, который является производным от класса **BaseClass**, а его свойства и методы указаны в словаре **clsDict** (определенном, например, ранее).

Определенный нами ранее класс **MyClass1** можно создать, используя **type()**. Давайте также создадим экземпляры объектов в обоих случаях (для класса **MyClass1**, созданного с помощью ключевого слова **class** и для класса **MyClass1**, полученного в результате работы функции **type()**) и проверим их типы:

```
class MyClass1():
    pass

myObj1 = MyClass1()
print(myObj1)
print(type(myObj1))
```

Результат:

```
<__main__.MyClass1 object at 0x0000016FB38BBB20>
<class '__main__.MyClass1'>
```

Рисунок 86

```
MyClass1=type("MyClass1", (), {})

myObj1 = MyClass1()
print(myObj1)
print(type(myObj1))
```

Результат:

```
<__main__.MyClass1 object at 0x0000025A65E9BB80>
<class '__main__.MyClass1'>
```

Рисунок 87

В обоих случаях созданные объекты являются экземплярами класса `MyClass1`. Обратите внимание на тот факт, что `MyClass1` является и переменной для хранения ссылок на класс и именем класса («`MyClass1`»). Если в классе нужно определить свойства и методы, то нужно заполнить третий параметр — словарь ключами и значениями

Давайте добавим в наш класс `MyClass1` метод. Для этого необходимо ранее (до вызова функции `type()`) определить функцию, которая будет реализовать необходимую логику будущего метода класса. Синтаксис объявления этой функции полностью аналогичен синтаксису объявления метода класса — первым параметром данной функции должен быть стандартный параметр `self` (ссылка на объект).

```
def method1(self):
    print(self.prop1)

MyClass2=type("MyClass2", (),
              {"prop1":"Hello", "method1":method1})

myObj2 = MyClass2()
print(myObj2.prop1)
myObj2.method1()
```

Результат:

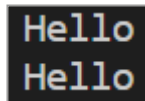


Рисунок 88

Как мы видим, добавление метода в класс практически ничем не отличается от добавления свойства. После определения нужной функции мы просто назначили ее в качестве значения свойства — имени метода (в третьем параметре — словаре при вызове функции `type()`).

Более того, уже после создания класса мы можем добавлять в него атрибуты и методы, если это необходимо:

```
MyClass2.prop2=100

myObj3 = MyClass2()
print(myObj2.prop2) #100
```

Так как классы в Python являются объектами, то, значит, можно динамически создавать класс и изменять его содержимое. Однако, именно это (процесс создания класса метаклассом `type`) и происходит во время выполнения

оператора `class`, который мы использовали ранее. В чем же суть? В том, что, разобравшись сейчас во всех деталях этого механизма, мы можем его кастомизировать — создавать собственные метаклассы.

## 7.2. Метод конструктор `__new__()`

Итак, мы уже знаем, как с помощью функции `type()` динамически создавать классы с нужными особенностями структуры и поведения. Однако, в примерах выше мы изменяли классы (добавляя к ним свойства и методы) уже после его создания.

Тем не менее, существуют ситуации, когда необходимо реализовать автоматическое изменение класса во время его создания или процесс создания класса зависит от выполнения каких-то условий. Например, нам нужно создавать классы с обязательным атрибутом «`id`». И если в определении класса нет такого атрибута, то добавлять его автоматически. Или перед созданием класса нужно проверить, что число методов в его определении не превышает некоторый порог. И только классы, удовлетворяющие этому условию, могут быть созданы в программе.

Для реализации описанных ситуаций (автоматически, с помощью программы, а не просмотром с последующими «ручными» коррекциями кода) нужно создать пользовательский метакласс, производный от метакласса `type`. И в этом пользовательском метаклассе переопределить уже известный нам «магический» метод `__new__()`.

Далее нам необходимо настроить процесс создания класса по нужному нам шаблону, особенности которого описаны в метаклассе. Для этого нужно задать параметру

`metaclass` (в определении класса) имя нашего пользовательского метакласса в качестве значения.

```
class MyMetaClass1(type):
    pass

class MyClass1(metaclass=MyMetaClass1):
    pass
```

Давайте проверим тип созданного метакласса `MyMetaClass1` и класса `MyClass1`:

```
print(type(MyMetaClass1))
print(type(MyClass1))
```

Результат:

```
<class 'type'>
<class '__main__.MyMetaClass1'>
```

Рисунок 89

Теперь перейдем к деталям процесса создания пользовательского метакласса.

Базовый метакласс (метакласс по умолчанию) `type` определяет «магические» методы, которые в новых пользовательских метаклассах (производных от `type`) могут быть переопределены для реализации нужных особенностей:

- `__new__()` — создает экземпляр класса, основанного на данном метаклассе. Переопределение данного метода позволяет управлять процессом создания класса. Например, можно настроить параметр-словарь для функции `type`, который задает свойства и методы соз-

даваемого класса. Или определить (задать ограничения) на параметр — кортеж, который хранит базовые классы в случае, когда создаваемый класс наследуется. Возвращаемый методом `__new__()` результат обычно является экземпляром класса (традиционно определяется как `cls`).

- `__init__()` — данный метод обычно вызывается после создания объекта для его инициализации.
- `__call__()` — переопределение данного метода позволяет настроить поведение процесса, происходящего при вызове конструктора создаваемого класса.

Любой пользовательский метакласс, должен определить свой метод `__new__()` и/или метод `__init__()`, чтобы возвращать необходимый пользовательский класс, на создание (настройку, изменение) которого он ориентирован. Более традиционным считается переопределение метода `__new__()`.

**Рассмотрим на примере:** переопределим в нашем метаклассе `MyMetaClass1` метод `__new__()` и добавим в него вывод информационного сообщения, чтобы увидеть момент срабатывания метода.

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        print("'Hello' from __new__()!")
        print("Type of the class being created ", cls)
        print("Name of the class being created:", name)
        print("Tuple of base classes: ", bases)
        print("Dictionary of attr.: ", dict)

        return type.__new__(cls, name, bases, dict)
```

```
class MyClass1(metaclass=MyMetaClass1):
    attr=100
```

Результат:

```
'Hello' from __new__()!
Type of the class being created <class '__main__.MyMetaClass1'>
Name of the class being created: MyClass1
Tuple of base classes: ()
Dictionary of attr.: {'__module__': '__main__', '__qualname__':
                      'MyClass1', 'attr': 100}
```

Рисунок 90

Метод `MyMetaClass1.__new__()` был вызван в момент определения класса `MyClass1` (с помощью ключевого слова `class`). В теле нашего переопределенного метода `__new__()` мы вызвали метод базового метакласса `type.__new__()` для того, чтобы создать наш пользовательский метакласс `MyMetaClass1`. Однако, данная строка кода не совсем корректна для ООП (мы знаем, что методы базовых классов в классах-потомках традиционно вызываются с помощью `super()`). Давайте внесем необходимые коррективы в наш метакласс `MyMetaClass1`:

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        print("'Hello' from __new__()!")
        print("Type of the class being created ", cls)
        print("Name of the class being created:", name)
        print("Tuple of base classes: ", bases)
        print("Dictionary of attr.: ", dict)
        return super().__new__(cls, name, bases, dict)
```

На данный момент наш метакласс `MyMetaClass1` не имеет особой пользы (кроме демонстрации момента вызова переопределенного метода `__new__()`). Давайте добавим функциональности: предположим, что каждый определяемый в коде класс должен содержать свойство «`id`».

На самом деле, решение очень простое — мы добавим в метод `MyMetaClass1.__new__()` проверку: есть ли среди в словаре `dict` такой ключ:

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class '{name}'!")
        else:
            print(f"Class '{name}' is creating...")
            return super().__new__(cls, name, bases, dict)

class MyClass1(metaclass=MyMetaClass1):
    attr=100

class MyClass2(metaclass=MyMetaClass1):
    attr=100
    id=1
```

Результат:

```
No 'id' attr. in the class 'MyClass1'!
Class 'MyClass2' is creating...
```

Рисунок 91

Давайте добавим еще требование о количестве методов: не более трех в каждом определяемом классе:



```

class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class '{name}'!")
        else:
            nMethods = {key: value for key, value
                          in dict.items()
                          if callable(value)}
            if len(nMethods)>3:
                print(f"More than 3 methods in
                      the class '{name}'!")
            else:
                print(f"Class '{name}' is creating...")
                return super().__new__(cls, name,
                                         bases, dict)

class MyClass1(metaclass=MyMetaClass1):
    attr=100

class MyClass2(metaclass=MyMetaClass1):
    attr=100
    id=1
    def method1(self):
        pass
    def method2(self):
        pass
    def method3(self):
        pass

class MyClass3(metaclass=MyMetaClass1):
    id=2
    def method1(self):
        pass
    def method2(self):
        pass
    def method3(self):
        pass

```

```
def method4(self):
    pass
```

Результат:

```
No 'id' attr. in the class 'MyClass1'!
Class 'MyClass2' is creating...
More than 3 methods in the class 'MyClass3'!
```

Рисунок 92

Мы можем также добавлять нужные атрибуты в класс (например, в случае их отсутствия в определяемом классе, как в примере с «`id`»):

```
class MyMetaClass2(type):
    def __new__(cls, name, bases, dict):
        resultCls=super().__new__(cls, name, bases, dict)
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class
                  '{name}'! Let's add it.")
            resultCls.id=0
        return resultCls

class User(metaclass=MyMetaClass2):
    attr=100

obj=User()
print(obj.id)
```

Результат:

```
No 'id' attr. in the class 'User'! Let's add it.
0
```

Рисунок 93

А что, если нам нужно добавлять разные атрибуты для разных классов? Для этого необходимо передать их в метакласс с помощью параметров, используя **\*\*kwargs**:

```
class MyMetaClass3(type):
    def __new__(cls, name, bases, dict, **kwargs):
        resultCls=super().__new__(cls, name, bases, dict)
        if kwargs:
            for key, value in kwargs.items():
                setattr(resultCls, key, value)
        return resultCls

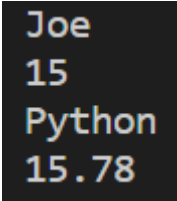
class User(metaclass=MyMetaClass3, firstName='Joe', age=15):
    attr=100

class Book(metaclass=MyMetaClass3, title='Python',
           price=15.78):
    attr=100

obj1=User()
print(obj1.firstName)
print(obj1.age)

obj2=Book()
print(obj2.title)
print(obj2.price)
```

Результат:



```
Joe
15
Python
15.78
```

Рисунок 94

На основе каждого из переданных в определении классов аргументов (со значениями) в эти классы в момент их создания были добавлены соответствующие атрибуты.

Таким образом, основное назначение пользовательских метаклассов, созданных с помощью «магического» метода `__new__()` — это изменение (преобразование) или проверка класса в момент его создания для того, чтобы он удовлетворял некоторому контексту.

### 7.3. Протоколы в Python

Достаточно часто при обсуждении языков с динамической типизацией упоминается термин «утиная типизация» (Duck Typing).

**Утиная типизация** (*Duck Typing*) — это концепция (подход к системе типизации), который используется в языках программирования с динамической типизацией (например, Python, PHP, Javascript и т.д.).

При таком подходе конкретный тип данных (или класс) менее важен, чем свойства и методы, которые в нем определены. Более того, используя концепцию Duck Typing, мы вообще не контролируем типы переменных (объектов). Вместо этого проверяется наличие у объекта нужного метода или свойства.

Название концепции произошло от фразы *«If it looks like a duck and quacks like a duck, it's a duck»* («Если что-то выглядит как утка и крякает как утка, то это есть утка»). То есть нас совершенно не интересует (для решения наших задач) внутренне устройство и особенности данного вида птицы. Мы делаем заключение, что это утка

(и мы будем «обращаться» с этой птицей, как с уткой), основываясь на внешних характеристиках и особенностях поведения.

Рассмотрим применение утиной типизации в Python на простых примерах:

```
number1=2
number2=5.7
print(number1+number2) #7.7

str1="Hello!"
str2="Python"
print(str1+str2) #Hello!Python
```

Оператор `+` для численных данных реализует арифметическое сложение, а для строковых — конкатенацию (объединение строк). Интерпретатор Python выполняет проверку типов (поддерживается ли такой оператор и что он должен реализовывать) во время выполнения кода, в отличие от языков со статической типизацией (таких как C++, Java), которые выполняют ее во время компиляции.

Многократно используемая нами функция `len()` вычисляет длину объекта в зависимости от типа его класса.

```
print(len("Student")) #7
print(len(["Python", "C#", "JavaScript"])) #3
print(len({"firstName": "Jane", "lastName": "Smith"})) #2
```

Функции `len()` не важен тип ее аргумента, существенным является лишь факт, что у аргумента (объекта) можно вызвать метод `__len__()`.

Таким образом, мы можем определить в теле нашего класса метод `__len__()` вызов функция `len()` для объектов этого класса станет возможным:

```
class MyClass:
    def __len__(self):
        return 1000

obj=MyClass()
print(len(obj)) #1000
```

Данный подход обеспечивает нам возможности полиморфизма, повышает гибкость кода, позволяя работать с экземплярами разных классов (абсолютно не связанных друг с другом) с помощью одного интерфейса (набора общих свойств и методов). Однако, наряду с положительными факторами данный подход имеет и недостаток: очень сложно (а иногда и невозможно) обнаружить ошибку некорректного использования объекта.

Например:

```
number=1000
print(len(number))
```

Результат:

**object of type 'int' has no len()**

Рисунок 95

Иногда обнаружение подобной ошибки возможно только на этапе тестирования (т. е. в момент выполнения кода).

К счастью, в Python существует класс `Protocol`, с помощью которого мы можем не только создавать интерфейсы, но и неявно проверять, удовлетворяют ли им объекты.

Давайте вначале разберемся с самим понятием «протокол». Традиционно под протоколом понимается некоторый принятый или установленный набор (кодекс) процедур или правил поведения в любой группе, организации или ситуации.

В программировании протокол — это набор правил, регулирующих взаимодействие сущностей (например, обмен или передачу данных). Устройства и программы обмениваются друг с другом данными только по определенным правилам — протоколам.

Например, между сотрудником и организацией, в которой он работает, заключается трудовой договор. Такой контракт фактически является протоколом, который регулирует взаимодействие (сотрудничество) между работодателем и работником. Сотрудник должен выполнить закрепленные за ним трудовые обязательства, работодатель гарантирует выплачивать заработную плату, обеспечивать социальный пакет и надлежащие условия труда. Если работник не выполняет свои функциональные обязательства, то может, например, не получить заработную плату в полном объеме или его могут даже уволить с занимаемой должности. Точно так же и работодатель за невыполнение каких-то условий (пунктов) протокола может быть оштрафован и т.д.

Именно с помощью протоколов мы реализуем интерфейс, который является механизмом (средством) этого взаимодействия сущностей. Мы уже использовали

понятие «интерфейс» при знакомстве с принципом инкапсуляции в ООП.

Фактически интерфейс представляет собой описание некоторой программной структуры (плана). И если, например, класс соответствует этой структуре, то к его экземплярам можно применять определенные свойства или методы. О таком классе говорят, что для него определен указанный интерфейс. Можно сказать, что с точки зрения реализации интерфейс — это класс «без кода», который определяет, каким будет поведение у его экземпляров, но без реализации особенностей этого поведения. Но об этом чуть позже.

Для чего же нам нужно разрабатывать интерфейсы? Создавая интерфейсы (называемые также пользовательскими протоколами), мы обеспечиваем возможность проверять совместимость типов (экземпляров классов) на основе структуры этих типов. Данный подход представляет собой некоторую вариацию утиной типизации — *«compile time duck typing»*, т. е. анализ совместимости типов происходит до выполнения программы. Таким образом, этот механизм дает нам возможность обнаружить ошибки на более ранних этапах разработки (до тестирования).

Для создания таких пользовательских интерфейсов (классов интерфейсов) нам нужно создать производный класс от класса `Protocol`. Класс `Protocol` был добавлен в Python 3.8 и определен в модуле `typing`.

Рассмотрим необходимость использования протоколов на примере.

Допустим, что у нас есть класс `Book` (книга), в котором определены свойства цена и скидка:



```
class Book:
    def __init__(self, title, author, price, discount):
        self.title = title
        self.author = author
        self.price=price
        self.discount=discount
```

Также мы разработали функцию `calculateTotalPrice()`, которая принимает список объектов `Book` (например, набор книг в «Корзине» покупателя) и возвращает общую стоимость покупки с учетом скидок.

```
def calculateTotalPrice(objs):
    sum=0
    for obj in objs:
        sum+=obj.price*(1-obj.discount/100)
    return sum

book1=Book("Python Crash Course","Eric Matthes", 150, 25)
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 178, 30)
print (calculateTotalPrice([book1,book2])) #237.1
```

Вполне вероятно, что возможность, предоставляемая функцией `calculateTotalPrice()` может быть полезна и для других категорий товаров, реализованных с помощью других классов.

Например, у нас есть класс `Product` (*Товар*), объекты которого также могут «наполнять» пользовательскую «Корзину», и нам может понадобиться вызов функции `calculateTotalPrice()` для расчета общей стоимости покупки. Однако, если в классе `Product` нет хотя бы одного из свойств, используемых функцией `calculateTotalPrice()`,

`price` или `discount`, то данную ошибку несовместимости мы, как и в примере с `len()` можем обнаружить только на этапе тестирования.

Поэтому мы создадим класс-протокол `Item`, производный от класса `Protocol`, который будет описывать нужные функции `calculateTotalPrice()` интерфейс: наличие у класса свойств `price` и `discount`. Таким образом, наш пользовательский класс-протокол `Item` будет содержать два атрибута: `price` и `discount`.

```
from typing import Protocol

class Item(Protocol):
    price:float
    discount:int
```

Атрибуты и методы, указанные в классе-протоколе `Item` должны присутствовать во всех классах (`Book` и `Product` в нашем примере), которые соответствуют данному протоколу. Обратите внимание, что после имени свойства в классе-протоколе следует указать символ двоеточия (`:`) и тип данных значений этого свойства. Наш (объявленный выше) класс `Book` соответствует протоколу `Item`.

Также нам нужно внести изменения в нашу функцию `calculateTotalPrice()`: указать, что ее параметр должен быть списком объектов типа `Item`:

```
def calculateTotalPrice(objs:List[Item]):
    sum=0
    for obj in objs:
        sum+=obj.price*(1-obj.discount/100)
    return sum
```

Вызов функции для покупки, представляющей собой список книг вернет корректный результат:

```
purchase1 = calculateTotalPrice([
    Book("JavaScript: The Good Parts",
        "Douglas Crockford", 178, 30),
    Book("Python Crash Course", "Eric Matthes", 150, 25)
])

print(purchase1) #237.1
```

Теперь давайте создадим класс `Product`, который не будет соответствовать протоколу `Item` (свойство, содержащее информацию о размере скидки, называется «`disc`»).

```
class Product:

    def __init__(self, name, price, disc, producer):
        self.name = name
        self.price = price
        self.disc=disc
        self.producer=producer
```

Однако, если в нашей новой покупке мы вместо одной из книг поместим продукт (экземпляр класса `Product`) и передадим этот список товаров функции `calculateTotalPrice()`:

```
purchase2 = calculateTotalPrice([
    Product('Tea', 100, 15, 'Lipton'),
    Book("Python Crash Course", "Eric Matthes", 150, 25)
])

print(purchase2)
```

то получим ошибку:

**'Product' object has no attribute 'discount'**

Рисунок 96

Это связано с тем, что в состав покупки `purchase2` вошел экземпляр класса `Product ("Tea")`, который не поддерживает протокол `Item` (отсутствует свойство «`discount`»).

Причина появления ошибки только на этапе выполнения программы заключается в том, что протоколы в основном применяются для моделирования статической типизации в `Duck Typing`, а не используются явно во время выполнения программы. То есть с помощью протоколов мы создаем описание (план), которому разработчики классов (для которых предполагается соответствие протоколу) должны следовать. В нашем примере разработчику класса `Product`, зная, что класс должен следовать протоколу `Item`, следовало использовать имя атрибута «`discount`», а не «`disc`» для хранения размера скидки. Тем не менее, хотелось бы иметь возможность автоматизировать проверку соответствия объекта протоколу перед его использованием функцией.

Нам известна функция `isinstance()`, с помощью которой мы могли бы проверить, а соответствует ли объект `obj` протоколу `Item`. Однако, по умолчанию данная функция не работает с протоколами.

Для того, чтобы обеспечить данную возможность, нам нужно перед объявлением класса-протокола указать декоратор `@runtime_checkable` (который предварительно надо импортировать из библиотеки `typing`):

```
from typing import Protocol, List, runtime_checkable

@runtime_checkable
class Item(Protocol):
    price:float
    discount:int
```

Изменим логику функции `calculateTotalPrice()`, чтобы она включала проверку соответствия объекта `obj` протоколу `Item` с помощью функции `isinstance()`:

```
def calculateTotalPrice(objs:List[Item]):
    sum=0
    for obj in objs:
        if isinstance(obj, Item):
            sum+=obj.price*(1-obj.discount/100)
        else:
            print("Incompatible type")
    return sum

purchase3 = calculateTotalPrice([
    Book("JavaScript: The Good Parts",
        "Douglas Crockford", 178, 30),
    Product('Tea', 100, 15,'Lipton'),
    Book("Python Crash Course","Eric Matthes", 150, 25)
])

print(purchase3) #237.1
```

Как мы видим, оба товара — книги были учтены при расчете общей стоимости покупки, а товар — продукт — нет, т. к. он не соответствует протоколу `Item`.



## Урок 9

# Введение в ООП

© STEP IT Academy, [www.itstep.org](http://www.itstep.org)

© Анна Егошина

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии с законодательством о свободном использовании произведения без согласия его автора (или другого лица, имеющего авторское право на данное произведение). Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования. Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника. Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством.