Тема 5. ООП

1 Классы, объекты, атрибуты

1.1 Понятие класса

Класс в ООП — чертёж объекта. Если проводить аналогию с объектами реального мира, то, например, автомобиль — это объект, а чертёж, описывающий структуру автомобиля, его параметры и функции, — класс. Таким образом,Понятие «Машина» будет соответствовать классу. Понятие класса Объектами этого класса будут марки автомобилей с различными характеристиками (атрибутами) и функциональными возможностями (методами), например, Audi, Lexus, Mercedes.

Для определения класса применяется ключевое слово **class**. За ним следует имя класса. Имя класса, в соответствии со стандартом PEP-8, должно начинаться с большой буквы. Далее с новой строки начинается тело класса с отступом в четыре пробельных символа.

Пример:

In [48]:
```
class Auto:
    # атрибуты класса
    auto_name = "Lexus"
    auto_model = "RX 350L"
    auto_year = 2019

    # методы класса
    def on_auto_start(self):
        print(f"Заводим автомобиль")

    def on_auto_stop(self):
        print("Останавливаем работу двигателя")
```

В представленном примере создаётся класс **Auto** с атрибутами **auto_name**, **auto_model**, **auto_year** и методами **on_auto_start()** и **on_auto_stop()**.

В приведённом выше примере используется служебное слово **self**, которое, в соответствии с соглашением в Python, определяет ссылку на объект (экземпляр) класса. Переменная **self** связывается с объектом класса, к которому применяются методы класса. Через переменную **self** можно получить доступ к атрибутам объекта. Когда методы класса применяются к новому объекту класса, то **self** связывается с новым объектом. Через эту переменную осуществляется доступ к атрибутам нового объекта.

1.2 Понятие объекта

Ранее мы разобрались, что класс — чертёж, на основе которого создаётся некоторый объект. Для создания объекта (экземпляра класса) необходимо в отдельной строке указать имя класса с открывающей и закрывающей круглыми

скобками. Эту инструкцию можно связать с некоторой переменной, которая будет содержать ссылку на созданный объект.

Создадим экземпляр для класса, описанного выше.

Результат:

In [49]:
```
a = Auto()
print(a)
print(type(a))
print(a.auto_name)
a.on_auto_start()
a.on_auto_stop()
```

```
<__main__.Auto object at 0x000001D6473A4610>
<class '__main__.Auto'>
Lexus
Заводим автомобиль
Останавливаем работу двигателя
```

В первой строке примера создаётся экземпляр класса **Auto**, ссылка на который связывается с переменной **a**. Содержимое этой переменной выводится во второй строке. В третьей строке проверяется тип переменной **a** — это класс **Auto**. В четвёртой строке осуществляется получение доступа к одному из атрибутов класса, а в пятой и шестой — запуск методов класса.

1.3 Понятие атрибута

Согласно методологии ООП, выделяют атрибуты классов и экземпляров. Атрибуты класса доступны из всех экземпляров класса. Атрибуты экземпляров относятся только к объектам класса. Атрибуты класса объявляются вне любого метода, а атрибуты экземпляра — внутри любого метода. Разберёмся на примере:

Пример:

In [50]:
```
class Auto:

    # атрибуты класса
    auto_count = 0

    # методы класса
    def on_auto_start(self, auto_name, auto_model, auto_year):
        print("Автомобиль заведен")
        self.auto_name = auto_name
        self.auto_model = auto_model
        self.auto_year = auto_year
        Auto.auto_count += 1
```

В приведённом примере создаётся класс **Auto**, содержащий один атрибут класса **auto_count** и три атрибута экземпляра класса: **auto_name**, **auto_model** и **auto_year**. В классе реализован один метод **on_auto_start()** с указанными атрибутами экземпляра. Их значения передаются в виде параметров методу **on_auto_start()**. Внутри этого метода значение атрибута **auto_count** класса увеличивается на единицу.

Важно отметить, что внутри методов атрибуты экземпляра идентифицируются ключевым словом **self** перед именем атрибута. При этом атрибуты класса идентифицируются названием класса перед именем атрибута.

Результат:

```
In [51]: a = Auto()
         a.on_auto_start("Lexus", "RX 350L", 2019)
         print(a.auto_name)
         print(a.auto_count)

         Автомобиль заведен
         Lexus
         1
```

В этом примере выводятся значения атрибута экземпляра класса (**auto_name**) и атрибута класса (**auto_count**).

Теперь, если создать ещё один экземпляр класса **Auto** и вызвать метод **on_auto_start()**, результат будет следующим:

Результат:

```
In [52]: a_2 = Auto()
         a_2.on_auto_start("Mazda", "CX 9", 2018)
         print(a_2.auto_name)
         print(a_2.auto_count)

         Автомобиль заведен
         Mazda
         2
```

Теперь значение атрибута **auto_count** равняется двум, из-за того, что он — атрибут класса и распространяется на все экземпляры. Значение атрибута **auto_count** в экземпляре **a** увеличилось до 1, а его значение в экземпляре **a_2** достигло двух.

2 Конструкторы, методы

2.1 Понятие конструктора

Конструктором в ООП называется специальный метод, вызываемый при создании экземпляра класса. Этот метод определяется с помощью конструкции **__init__**.

Пример:

In [53]:
```
class Auto:
    # атрибуты класса
    auto_count = 0

    # методы класса
    def __init__(self):
        Auto.auto_count += 1
        print(Auto.auto_count)
```

В примере создаётся класс **Auto** с одним атрибутом **auto_count** уровня класса. В классе реализован конструктор, увеличивающий значение **auto_count** на единицу и выводящий на экран итоговое значение.

Теперь при создании экземпляра класса **Auto** вызывается конструктор, значение **auto_count** увеличивается и отображается на экране. Создадим несколько экземпляров класса:

Результат:

In [54]:
```
a_1 = Auto()
a_2 = Auto()
a_3 = Auto()
```

```
1
2
3
```

В результат запуска выводятся значения 1, 2, 3, так как для каждого экземпляра значение атрибута **auto_count** возрастает и выводится на экран. На практике конструкторы используются для инициализации значений атрибутов. Это важно при создании объекта класса.

2.2 Понятие метода

Ранее мы уже познакомились с методами в ООП, то есть, функциями, получающими в качестве обязательного параметра ссылку на объект и выполняющими определённые действия с атрибутами объекта. Мы уже создали

методы **on_auto_start()** и **on_auto_stop()** для класса **Auto**. Вспомним ещё раз, как создаётся метод.

Пример:

```
In [56]: class Auto:

             def get_class_info(self):
                 print("Детальная информация о классе")
```

Результат:

```
In [57]: a = Auto()
         a.get_class_info()

         Детальная информация о классе
```

3 Локальные и глобальные переменные

3.1 Локальные переменные

Понятие области видимости переменных используется и в методологии ООП. Локальная переменная в классе доступна только в рамках части кода, где она определена. Например, если определить переменную в пределах метода, не выйдет получить к ней доступ из других частей программы.

Пример:

```
In [60]: class Auto:
             def on_start(self):
                 info = "Автомобиль заведен"
                 return info
```

В представленном примере создаётся локальная переменная **info** в рамках метода **on_start()** класса **Auto**. Проверим работу кода, создав экземпляр класса **Auto,** и попытаемся получить доступ к локальной переменной **info**.

Результат:

```
In [61]: a = Auto()
         print(a.info)

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         ~\AppData\Local\Temp/ipykernel_480/3779664100.py in <module>
               1 a = Auto()
         ----> 2 print(a.info)

         AttributeError: 'Auto' object has no attribute 'info'
```

Ошибка возникает из-за отсутствия возможности получения доступа к локальной переменной вне блока, в котором переменная определена.

3.2 Глобальные переменные

Глобальные переменные, в отличие от локальных, определяются вне различных блоков кода. Доступ к ним возможен из любых точек программы (класса).

Пример:

```
In [62]: class Auto:
             info_1 = "Автомобиль заведён"

             def on_start(self):
                 info_2 = "Автомобиль заведён"
                 return info_2
```

Результат:

```
In [66]: a = Auto()
         print(a.info_1)
```

```
Автомобиль заведён
```

В примере создаётся глобальная переменная **info_1,** и на экран выводится её значение. При этом ошибка не возникает.

4 Модификаторы доступа

Механизмы использования модификаторов позволяют изменять области видимости переменных. В Python ООП доступны три вида модификаторов:

– Public (публичный).
– Protected (защищённый).
– Private (приватный).

Для переменных с модификатором публичного доступа есть возможность изменения значений за пределами класса. Для публичных переменных префиксы (подчеркивания) не применяются.

Защищённая переменная создаётся с помощью добавления одного знака подчеркивания перед именем переменной. При использовании защищённых переменных их значения могут меняться только в пределах одного и того же пакета.

Приватная переменная идентифицируется с помощью двойного подчёркивания перед именем переменной. Значения приватных переменных могут изменяться только в пределах класса.

Пример:

```
In [67]: class Auto:
             def __init__(self):
                 print("Автомобиль заведен")
                 self.auto_name = "Mazda"
                 self._auto_year = 2019
                 self.__auto_model = "CX9"
```

В примере создаётся класс **Auto** с конструктором и тремя переменными: **auto_name**, **auto_model**, **auto_year**. Переменная **auto_name** — публичная, а переменные **auto_year** и **auto_model** — защищённая и приватная соответственно.

Создадим экземпляр класса **Auto** и проверим доступность переменной **auto_name**.

Результат:

```
In [68]: a = Auto()
         print(a.auto_name)

         Автомобиль заведен
         Mazda
```

Переменная **auto_name** обладает публичным модификатором. Доступ к ней возможен не из класса. Мы это увидели выше.

Теперь попробуем обратиться к значению переменной **auto_model**.

```
In [69]: print(a.auto_model)

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         ~\AppData\Local\Temp/ipykernel_480/1023684930.py in <module>
         ----> 1 print(a.auto_model)

         AttributeError: 'Auto' object has no attribute 'auto_model'
```

После запуска примера мы получили сообщение об ошибке.

5 Инкапсуляция

В Python инкапсуляция реализуется только на уровне соглашения, которое определяет общедоступные и внутренние характеристики. Одиночное подчёркивание в начале имени атрибута или метода свидетельствует о том, что атрибут или методы не предназначены для использования вне класса. Они доступны по этому имени.

Пример:

```
In [70]: class MyClass:
             _attr = "значение"
             def _method(self):
                 print("Это защищенный метод!")
```

Результат:

```
In [71]: mc = MyClass()
         mc._method()
         print(mc._attr)
```

```
Это защищенный метод!
значение
```

Использование двойного подчёркивания перед именем атрибута и метода делает их недоступными по этому имени.

Пример:

```
In [73]: class MyClass:
             __attr = "значение"
             def __method(self):
                 print("Это приватный метод!")
```

Результат:

```
In [74]: mc = MyClass()
         mc.__method()
         print(mc.__attr)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_480/3617187343.py in <module>
      1 mc = MyClass()
----> 2 mc.__method()
      3 print(mc.__attr)

AttributeError: 'MyClass' object has no attribute '__method'
```

Но и эта мера не обеспечивает абсолютную защиту. Обратиться к атрибуту или методу по-прежнему можно, используя следующий подход: _ИмяКласса__ИмяАтрибута.

Пример:

```
In [75]: class MyClass:
             __attr = "значение"
             def __method(self):
                 print("Это защищённый метод!")
```

Результат:

```
In [76]: mc = MyClass()
         mc._MyClass__method()
         print(mc._MyClass__attr)

         Это защищённый метод!
         значение
```

6 Наследование

6.1 Простое наследование

Сущность этого понятия соответствует его названию. Речь идёт о наследовании некоторым объектом характеристик другого объекта-родителя. Объект называется дочерним и обладает не только характеристиками родителя, но и собственными свойствами. Благодаря наследованию можно избежать дублирования кода.

Суть принципа наследования заключается в том, что класс может перенимать (наследовать) параметры другого класса. Класс, наследующий характеристики другого класса, называется дочерним, а класс, предоставляющий свои характеристики, — родительским.

Пример:

```
In [75]: # Класс Transport
         class Transport:
             def transport_method(self):
                 print("Это родительский метод из класса Transport")

         # Класс Auto, наследующий Transport
         class Auto(Transport):
             def auto_method(self):
                 print("Это метод из дочернего класса")
```

В представленном примере создаются два класса: **Transport** (родитель), **Auto** (наследник). Для реализации наследования нужно указать имя класса-родителя внутри скобок, следующих за именем класса-наследника. В классе **Transport** реализован метод **transport_method()**, а в дочернем классе есть метод

auto_method(). Класс **Auto** наследует характеристики класса **Transport,** то есть все его атрибуты и методы.

Проверим работу механизма наследования:

Результат:

```
In [79]: a = Auto()
         a.transport_method()  # Вызываем метод родительского класса

         Это родительский метод из класса Transport
```

В примере создаётся экземпляр класса **Auto**. Для экземпляра класса вызывается метод **transport_method()**. Важно, что в классе **Auto** отсутствует метод с названием **transport_method()**. Так как класс **Auto** унаследовал характеристики класса **Transport**, то экземпляр класса **Auto** работает с методом **transport_method()** класса **Transport**.

6.2 Множественное наследование

Механизм наследования может быть реализован с использованием нескольких родителей у одного класса. И наоборот, один класс-родитель будет передавать свои характеристики нескольким дочерним классам.

Несколько дочерних классов у одного родителя

Пример:

```
In [82]: # класс Transport
         class Transport:
             def transport_method(self):
                 print("Родительский метод класса Transport")

         # класс Auto, наследующий Transport
         class Auto(Transport):
             def auto_method(self):
                 print("Дочерний метод класса Auto")

         # класс Bus, наследующий Transport
         class Bus(Transport):
             def bus_method(self):
                 print("Дочерний метод класса Bus")
```

В этом примере у нас есть класс-родитель **Transport**, наследуемый дочерними классами **Auto** и **Bus**. В обоих дочерних классах возможен доступ к методу **transport_method()** класса-родителя. Для запуска скрипта создадим экземпляры класса.

Результат:

In [83]:
```
a = Auto()
a.transport_method()
b = Bus()
b.transport_method()
```

```
Родительский метод класса Transport
Родительский метод класса Transport
```

Рассмотрим ещё один пример, в котором класс-родитель **Shape** определяет атрибуты. Эти атрибуты могут быть характерны для всех классов-наследников. Например, цвет фигуры, ширина и высота, основание и высота.

Здесь в конструкторах классов-наследников инициализируются параметры. Часть их — собственные атрибуты классов-наследников, а некоторые наследуются от родителей. Чтобы работать с унаследованными атрибутами, нужно их перечислить, например, **super().__init__(color, param_1, param_2)**. Тем самым мы показываем, что хотим иметь возможность работы с атрибутами класса-родителя. Если атрибуты не перечислить, то при попытке обращения к ним через экземпляр класса-наследника возникнет ошибка.

Пример:

In [85]:
```
class Shape:
    def __init__(self, color, param_1, param_2):
        self.color = color
        self.param_1 = param_1
        self.param_2 = param_2

    def square(self):
        return self.param_1 * self.param_2

class Rectangle(Shape):
    def __init__(self, color, param_1, param_2, rectangle_p):
        super().__init__(color, param_1, param_2)
        self.rectangle_p = rectangle_p

    def get_r_p(self):
        return self.rectangle_p

class Triangle(Shape):
    def __init__(self, color, param_1, param_2, triangle_p):
        super().__init__(color, param_1, param_2)
        self.triangle_p = triangle_p

    def get_t_p(self):
        return self.triangle_p
```

Результат:

In [86]:
```
r = Rectangle("white", 10, 20, True)
print(r.color)
print(r.square())
print(r.get_r_p())
t = Triangle("red", 30, 40, False)
print(t.color)
print(t.square())
print(t.get_t_p())
```

```
white
200
True
red
1200
False
```

Несколько родителей у одного класса

Пример:

In [87]:
```
class Player:
    def player_method(self):
        print("Родительский метод класса Player")

class Navigator:
    def navigator_method(self):
        print("Родительский метод класса Navigator")

class MobilePhone(Player, Navigator):
    def mobile_phone_method(self):
        print("Дочерний метод класса MobilePhone")
```

В этом примере создаются классы: **Player**, **Navigator**, **MobilePhone**. Причём классы **Player** и **Navigator** — родительские для класса **MobilePhone**. Поэтому класс **MobilePhone** имеет доступ к методам классов **Player** и **Navigator**. Проверим это.

Результат:

In [88]:
```
m_p = MobilePhone()
m_p.player_method()
m_p.navigator_method()
```

```
Родительский метод класса Player
Родительский метод класса Navigator
```

Возможна ситуация, когда у классов-родителей совпадают имена атрибутов и методов. В этом случае обращение к такому атрибуту или методу через «наследник» будет адресовано к атрибуту или методу того класса-родителя, который значится первым.

Пример:

In [89]:
```
class Shape:
    def __init__(self, param_1, param_2):
        self.param_1 = param_1
        self.param_2 = param_2

    def get_params(self):
        return f"Параметры Shape: {self.param_1}, {self.param_2}"

class Material:
    def __init__(self, param_1, param_2):
        self.param_1 = param_1
        self.param_2 = param_2

    def get_params(self):
        return f"Параметры Material: {self.param_1}, {self.param_2}"

class Triangle(Shape, Material):
    def __init__(self, param_1, param_2):
        super().__init__(param_1, param_2)
        pass
```

Результат:

In [90]:
```
t = Triangle(10, 20)
print(t.get_params())
```

```
Параметры Shape: 10, 20
```

7 Полиморфизм

Дословный перевод этого понятия — «имеющий многие формы». В методологии ООП — это способность объекта иметь различную функциональность. В программировании полиморфизм проявляется в перегрузке или переопределении методов классов.

Перегрузка методов

Реализуется в возможности метода отражать разную логику выполнения в зависимости от количества и типа передаваемых параметров.

Пример:

```
In [91]: # класс Auto
         class Auto:
             def auto_start(self, param_1, param_2=None):
                 if param_2 is not None:
                     print(param_1 + param_2)
                 else:
                     print(param_1)
```

В этом примере возможны несколько вариантов логики метода **auto_start()**. Первый вариант — при передаче в метод одного параметра. Второй — при передаче двух параметров. В первом случае будет выведено значение переданного параметра, во втором — сумма параметров.

Результат:

```
In [92]: a = Auto()
         a.auto_start(50)
         a = Auto()
         a.auto_start(10, 20)

         50
         30
```

7.2 Переопределение методов

Переопределение методов в полиморфизме выражается в наличии метода с одинаковым названием для родительского и дочернего классов. При этом логика методов различается, но названия идентичны.

Пример:

```
In [93]: # класс Transport
         class Transport:
             def show_info(self):
                 print("Родительский метод класса Transport")

         # класс Auto, наследующий Transport
         class Auto(Transport):
             def show_info(self):
                 print("Родительский метод класса Auto")

         # класс Bus, наследующий Transport
         class Bus(Transport):
             def show_info(self):
                 print("Родительский метод класса Bus")
```

В примере классы **Auto** и **Bus** наследуют характеристики класса **Transport**. В этом классе реализуется метод **show_info()**, переопределенный классом-потомком. Теперь, если вызвать метод **show_info()**, результат будет зависеть от объекта, через который осуществляется вызов метода.

Результат:

```
In [94]: t = Transport()
         t.show_info()

         a = Auto()
         a.show_info()

         b = Bus()
         b.show_info()
```

```
Родительский метод класса Transport
Родительский метод класса Auto
Родительский метод класса Bus
```

В этом примере методы **show_info()** вызываются с помощью производных классов одного общего базового класса. Но дочерние классы переопределяются через метод класса-родителя, методы обладают разной функциональностью.

8 Перегрузка операторов

Под перегрузкой операторов понимается изменение логики работы различных операторов языка с использованием специальных методов. Эти методы идентифицируются двойным подчеркиванием до и после имени метода.

Под операторами имеются в виду знаки +, -, *, /, отвечающие за выполнение привычных математических операций, а также особенности

синтаксиса языка, обеспечивающие создание объекта, вызова его как функции, получение доступа к элементу объекта по индексу и т. д. К перегружаемым операторам также относятся >, <, **≤**, **≥**, ==, **!**=, +=, -=. При перегрузке каждого из этих операторов происходит вызов соответствующего магического метода, например:

Таблица 8.1 – Методы перезагрузки операторов

Методы	Описание
__init__()	Соответствует конструктору объектов класса, срабатывает при создании объектов
__del__()	Соответствует деструктору объектов класса, срабатывает при удалении объектов
__str__()	Срабатывает при передаче объекта функциям **str()** и **print()**, преобразует объект к строке
__add__()	Срабатывает при участии объекта в операции сложения в качестве операнда с левой стороны, обеспечивает перегрузку оператора сложения
__setattr__()	Срабатывает при выполнении операции присваивания значения атрибуту объекта
__getitem__()	Срабатывает при извлечении элемента по индексу
__call__()	Срабатывает при обращении к экземпляру класса как к функции
__gt__()	Соответствует оператору>
__lt__()	Соответствует оператору <
__ge__()	Соответствует оператору ≥
__le__()	Соответствует оператору ≤
__eq__()	Соответствует оператору ==
__iadd__()	Соответствует операции «Сложение и присваивание» +=
__isub__()	Соответствует операции «Вычитание и присваивание» -=

Перегрузка операторов относится к редко используемым на практике механизмам. На деле разработчику чаще всего приходится сталкиваться с перегрузкой в конструкторе. Но в рамках концепции ООП эта тема важна. В списке выше приведена только часть методов, используемых при реализации перегрузки операторов в Python. С полным списком можно ознакомиться по ссылке.

Благодаря механизму перегрузки операторов пользовательские классы встают в один ряд со встроенными, поскольку все встроенные типы в Python

относятся к классам. В итоге все объекты класса получают одинаковый интерфейс.

__init__

Выполним перегрузку конструктора. Напомним, что конструктор класса отвечает за создание объекта класса.

Пример:

In [95]:
```
class MyClass:
    def __init__(self, param):
        self.param = param
```

Результат:

In [96]:
```
mc = MyClass("text")
print(mc.param)
```

```
text
```

__del__

В Python разработчик может участвовать как в создании, так и в удалении объекта.

Пример:

In [97]:
```
class MyClass:
    def __init__(self, param):
        self.param = param

    def __del__(self):
        print(f"Удаляем объект {self.param} класса MyClass")
```

Результат:

In [98]:
```
mc = MyClass("text")
del mc
```

```
Удаляем объект text класса MyClass
```

Деструктор на практике может применяться в тех случаях, когда требуется явное освобождение памяти при удалении объектов.

__str__

Вызывается функциями **str**, **print** и **format**. Возвращает строковое представление объекта.

Пример:

In [2]:
```
class MyClass:
    def __init__(self, param_1, param_2):
        self.param_1 = param_1
        self.param_1 = param_2

    def __str__(self):
        return f"Объект с параметрами ({self.param_1}, {self.param_2})"
```

Результат:

In [3]:
```
mc = MyClass("text_1", "text_2")
print(mc)
```

```
Объект с параметрами (text_1, text_2)
```

__add__

Срабатывает при участии объекта в операции сложения в качестве операнда с левой стороны, обеспечивает перегрузку оператора сложения:

Пример:

In [4]:
```
class MyClass:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def __add__(self, other):
        return MyClass(self.width + other.width, self.height + other.height)

    def __str__(self):
        return f"Объект с параметрами ({self.width}, {self.height})"
```

Результат:

In [5]:
```
mc_1 = MyClass(10, 20)
mc_2 = MyClass(30, 40)
print(mc_1 + mc_2)
```

```
Объект с параметрами (40, 60)
```

__setattr__

Срабатывает при выполнении операции присваивания значения атрибуту объекта

Пример:

```
In [6]: class MyClass:
            def __setattr__(self, attr, value):
                if attr == "width":
                    self.__dict__[attr] = value
                else:
                    print(f"Атрибут {attr} недопустим")
```

Результат:

```
In [7]: mc = MyClass()
        mc.height = 40
```

```
Атрибут height недопустим
```

__getitem__

Срабатывает при извлечении элемента по индексу

Пример:

```
In [8]: class Class1:
            def __init__(self, param):
                self.param = param

            def __str__(self):
                return str(self.param)

        class Class2:
            def __init__(self, *args):
                self.my_list = []
                for el in args:
                    self.my_list.append(Class1(el))
```

Результат:

```
In [9]: my_obj = Class2(10, True, "text")
        print(my_obj.my_list[1])
```

```
True
```

Рисунок - Пример 1

Пример:

```
In [10]: class Class1:
             def __init__(self, param):
                 self.param = param

             def __str__(self):
                 return str(self.param)

         class Class2:
             def __init__(self, *args):
                 self.my_list = []
                 for el in args:
                     self.my_list.append(Class1(el))
```

Результат:

```
In [11]: my_obj = Class2(10, True, "text")
         print(my_obj.my_list[1])

         True
```

Рисунок - Пример 2

В этом примере описан класс **Class2**, в котором происходит заполнение списка **my_list** экземплярами класса **Class1**. Для получения элемента списка можно обратиться по индексу к элементу **my_list.**

Теперь рассмотрим второй пример, в котором элемент извлекается по индексу не из атрибута экземпляра класса, а из самого объекта.

Пример:

In [12]:
```
class Class1:
    def __init__(self, param):
        self.param = param

    def __str__(self):
        return str(self.param)

class Class2:
    def __init__(self, *args):
        self.my_list = []
        for el in args:
            self.my_list.append(Class1(el))

    def __getitem__(self, index):
        return self.my_list[index]
```

Результат:

In [13]:
```
my_obj = Class2(10, True, "text")
print(my_obj.my_list[0])
print(my_obj[1])
print(my_obj[2])
```

```
10
True
text
```

Во втором примере показано, как объекты пользовательского класса становятся похожими на объекты встроенных классов-последовательностей (строк, списков, кортежей).

__call__

Срабатывает при обращении к экземпляру класса как к функции

Пример:

In [15]:
```
class MyClass:
    def __init__(self, param):
        self.param = param

    def __call__(self, newparam):
        self.param = newparam

    def __str__(self):
        return f"Значение параметра - {self.param};"
```

Результат:

In [16]:
```
obj_1 = MyClass("width")
obj_2 = MyClass("height")

obj_1("length")
obj_2("square")

print(obj_1, obj_2)
```

```
Значение параметра - length; Значение параметра - square;
```

__eq__

Соответствует оператору ==

Пример:

In [17]:
```
class MyClass:
    def __init__(self):
        self.x = 40

    def __eq__(self, y):
        return self.x == y
```

Результат:

In [18]:
```
mc = MyClass()
print("Равно" if mc == 40 else "Не равно")
print("Равно" if mc == 41 else "Не равно")
```

```
Равно
Не равно
```

__lt__

Соответствует оператору **≤**

Пример:

In [19]:
```
class Salary:
    val = 50000

    def __lt__(self, other):
        print("Оклад меньше премии?")
        return self.val < other.val

class Prize:
    val = 5000

    def __lt__(self, other):
        print("Премия меньше оклада?")
        return self.val < other.val
```

Результат:

In [20]:
```
s = Salary()
p = Prize()

check = (s < p)
print(check)
```

```
Оклад меньше премии?
False
```

__iadd__

Соответствует операции «Сложение и присваивание» +=

Пример:

In [21]:
```
class MyClass:
    def __init__(self, val):
        self.val = val

    def __iadd__(self, other):
        self.val += other
        return self
```

Результат:

In [22]:
```
mc = MyClass(100)
print(mc.val)
mc += 200
print(mc.val)
```

```
100
300
```

9 Переопределение методов

Мы уже познакомились с одним из основных принципов ООП — наследованием. Пришло время познакомиться с таким важным подходом, используемым при реализации наследования, как переопределение методов.

Например, в программе реализован класс-родитель, от которого предполагается наследовать характеристики для другого класса-потомка. В классе-родителе предусмотрен некий метод, с определенной функциональностью. Но для класса-потомка ее недостаточно и требуется дополнительная логика. Вариант решения проблемы — полностью переписать код метода из класса-родителя для класса-потомка. Это ведет к избыточности кода, поэтому такое решение не оптимально.

Существует специальный механизм, позволяющий использовать метод класса-родителя в классе-потомке с добавлением некоторой функциональности.

Пример:

In [23]:
```
class ParentClass:
    def __init__(self):
        print("Конструктор класса-родителя")

    def my_method(self):
        print("Метод my_method() класса ParentClass")

class ChildClass(ParentClass):
    def __init__(self):
        print("Конструктор дочернего класса")
        ParentClass.__init__(self)

    def my_method(self):
        print("Метод my_method() класса ChildClass")
        ParentClass.my_method(self)
```

Результат:

In [24]:
```
c = ChildClass()
c.my_method()
```

```
Конструктор дочернего класса
Конструктор класса-родителя
Метод my_method() класса ChildClass
Метод my_method() класса ParentClass
```

Допустимо также не ссылаться явно на класс-родитель. Для этого используется специальный метод **super()**.

Пример:

In [25]:
```
class ParentClass:
    def __init__(self):
        print("Конструктор класса-родителя")

    def my_method(self):
        print("Метод my_method() класса ParentClass")


class ChildClass(ParentClass):
    def __init__(self):
        print("Конструктор дочернего класса")
        super().__init__()

    def my_method(self):
        print("Метод my_method() класса ChildClass")
        super().my_method()
```

Результат:

In [26]:
```
c = ChildClass()
c.my_method()
```

```
Конструктор дочернего класса
Конструктор класса-родителя
Метод my_method() класса ChildClass
Метод my_method() класса ParentClass
```

Результат выполнения полностью совпадает с результатом запуска скрипта, реализованного выше.

10 Интерфейсы

Под интерфейсом в ООП понимается описание поведения объекта, т. е., совокупность публичных методов объекта, которые могут применяться в других частях программы для взаимодействия с ним.

Рассмотрим подробнее понятие интерфейса в привязке к абстрактным классам, которые реализуются в Python с помощью встроенного в стандартную библиотеку модуля abc (Abstract Base Classes). Абстрактные классы позволяют контролировать поведение классов-наследников, например, проверять, что они обладают одинаковым интерфейсом.

Пример:

In [27]:
```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_method_1(self):
        pass
    @abstractmethod
    def my_method_2(self):
        pass

class MyClass(MyAbstractClass):
    pass

mc = MyClass()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_480/1213968424.py in <module>
     12     pass
     13
---> 14 mc = MyClass()

TypeError: Can't instantiate abstract class MyClass with abstract methods my_method_1, my_method_2
```

В этом примере создается абстрактный класс **MyAbstractClass** и в случае наследования от него во всех классах-потомках необходимо реализовать два базовых метода, т. е., все классы-потомки наследуют интерфейс родителя. Соответственно, логику класса **MyClass** в примере выше необходимо изменить:

Пример:

In [28]:
```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_method_1(self):
        pass
    @abstractmethod
    def my_method_2(self):
        pass

class MyClass(MyAbstractClass):
    def my_method_1(self):
        print("Метод my_method_1()")

    def my_method_2(self):
        print("Метод my_method_2()")
```

Результат:

In [30]:
```
mc = MyClass()
mc.my_method_1()
```

```
Метод my_method_1()
```

11 Интерфейс итерации

Под итераторами понимаются специальные объекты, обеспечивающие пошаговый доступ к данным из контейнера. В привязке к итераторам работают циклы перебора (**for in**), встроенные функции (**map()**, **filter()**, **zip()**), операция распаковки. Эти инструменты способны работать с любыми объектами, поддерживающими интерфейс итерации.

Рассмотрим небольшой пример:

Пример:

```
In [31]: my_list = [30, 105.6, "text", True]

30
105.6
text
True
```

Результат:

```
In [32]: for el in my_list:
             print(el)

30
105.6
text
True
```

Рассмотрим подробнее, как выполняется код выше.

1. Вызов метода **__iter__()** для итерируемого объекта (списка **my_list**): **my_list.__iter__()**. Метод **__iter__()** возвращает объект с методом **__next__()**.
2. Цикл **for in** в ходе каждой итерации запускает метод **__next__()**, который при каждом вызове возвращает очередной элемент итератора.
3. Когда элементы итераторы исчерпаны, метод **__next__()** завершает свою работу и генерирует исключение **StopIteration**. Цикл **for in** перехватывает данное исключение и завершает свою работу.

Итак, итератор в Python — объект, реализующий метод **__next__()** без аргументов, возвращающий очередной элемент или исключение **StopIteration**.

12 Создание собственных объектов-итераторов

Рассмотрим пример создания объекта с поддержкой интерфейса итерации.

Пример:

In [33]:
```
class Iterator:
   # Объект-итератор
    def __init__(self, start=0):
        self.i = start

    # У итератора есть метод __next__
    def __next__(self):
        self.i += 1
        if self.i <= 5:
            return self.i
        else:
            raise StopIteration

class IterObj:
    #Объект, поддерживающий интерфейс итерации (итерируемый объект)
    def __init__(self, start=0):
        self.start = start - 1

    def __iter__(self):
        # Метод __iter__ должен возвращать объект-итератор
        return Iterator(self.start)
```

В этом примере в виде класса **IterObj** реализован объект, поддерживающий итерирование, а в виде класс **Iterator** — сам итератор? возвращающий очередной элемент итерируемого объекта. В данном случае это числа, начиная от значения параметра **start** (его значение определяется при создании экземпляра класса **IterObj**) до 5 (включительно).

Проверим работу примера:

Результат:

In [34]:
```
obj = IterObj(start=2)
for el in obj:
    print(el)
```

```
2
3
4
5
```

Усовершенствуем пример. Реализуем возможности итератора и итерируемого объекта в рамках общего класса:

Пример:

```
In [36]: class Iter:
             def __init__(self, start=0):
                 self.i = start - 1

             # Метод __iter__ должен возвращать объект-итератор
             def __iter__(self):
                 return self

             def __next__(self):
                 self.i += 1
                 if self.i <= 5:
                     return self.i
                 else:
                     raise StopIteration
```

Результат:

```
In [37]: obj = Iter(start=2)
         for el in obj:
             print(el)

2
3
4
5
```

В первом варианте реализации примера экземпляры класса **IterObj()** возвращают объект-итератор. Во втором варианте объекты **Iter()** сами по себе являются итераторами и пройти по ним можно только один раз. После вызова метода **__next__()** итератор запоминает свое состояние. Для выполнения повторной итерации по итерируемому объекту нужно получить новый объект-итератор, в данном случае создать новый объект **Iter()**.

13 Декоратор @property

Под декоратором в Python подразумевается функция (или класс), расширяющая логику работы другой функции. У разработчика существует возможность написания собственных декораторов или использования существующих. Рассмотрим декоратор **@property**. Символ **@** позволяет идентифицировать объект как декоратор и установить его для некоторой функции (или метода класса).

Встроенный декоратор **@property** позволяет работать с методом некоторого класса как с атрибутом.

Проверим работу примера:

Пример:

```
In [38]: class MyClass:
             def __init__(self, param_1, param_2):
                 self.param_1 = param_1
                 self.param_2 = param_2

             @property
             def my_method(self):
                 return f"Параметры, переданные в класс:" \
                     f" {self.param_1}, {self.param_2}"
```

Проверим работу примера

Результат:

```
In [39]: mc = MyClass("text_1", "text_2")

         print(mc.param_1)
         print(mc.param_2)

         print(mc.my_method)
```

```
text_1
text_2
Параметры, переданные в класс: text_1, text_2
```

В результате преобразования метода в свойство доступ к нему осуществляется с помощью обычной точечной нотации.

Рассмотрим еще один пример с декоратором **@property**.

Для обеспечения контролируемого доступа к данным класса в Python применяются модификаторы доступа и свойства. Рассмотрим, что это такое, на примере. Представим, что нам нужно проверить, что модель автомобиля должна быть выпущена в пределах 2000-2019 гг. Если пользователь введет значение года выпуска модели меньше 2000, то значение параметра года выпуска установится в 2000. При указании значения выше 2019 значение параметра должно установиться в эту цифру. Если введено корректное значение (в пределах 2000-2019 гг.), то значение нужно оставить неизменным.

Пример:

In [40]:
```
# класс Auto
class Auto:

    # конструктор класса Auto
    def __init__(self, year):
        # Инициализация свойств.
        self.year = year

    # создаем свойство года
    @property
    def year(self):
        return self.__year

    # сеттер для создания свойств
    @year.setter
    def year(self, year):
        if year < 2000:
            self.__year = 2000
        elif year > 2019:
            self.__year = 2019
        else:
            self.__year = year

    def get_auto_year(self):
        return f"Автомобиль выпущен в {str(self.year)} году"
```

Свойство обладает тремя важными аспектами. Первым делом необходимо определить атрибут — год выпуска автомобиля. Далее необходимо определить свойство атрибута с помощью декоратора **@property**. Третий шаг — создать установщик свойства (сеттер), применив декоратор для параметра года: **@year.setter**.

Теперь, если попытаться указать значение выше 2019, то результат будет:

Результат:

In [41]:
```
a = Auto(2090)
print(a.get_auto_year())
```

```
Автомобиль выпущен в 2019 году
```

Для значения меньше 2000 результат:

Результат:

In [42]:
```
a = Auto(2000)
print(a.get_auto_year())
```

```
Автомобиль выпущен в 2000 году
```

14 Композиция

В концепции ООП существует возможность реализации композиционного подхода, в соответствии с которым создается класс-контейнер, включающий вызовы других классов. Таким образом, при создании экземпляра класса-контейнера создаются экземпляры входящих в него классов. Композиция часто встречается применительно к объектам реального мира. Например, персональный компьютер состоит из комплектующих: процессора, памяти, видеокарты.

Рассмотрим реализацию композиции на примере вычисления площади обоев, необходимых для оклеивания комнаты. Оклеивать пол, потолок, двери и окна не требуется. Комната является прямоугольным параллелепипедом, состоящим из шести прямоугольников. Площадь комнаты формируется на основе суммы площадей прямоугольников, входящих в параллелепипед. Площадь каждого прямоугольника вычисляется как произведение его длины и высоты.

Т. к. обои необходимо клеить только на стены, площади верхнего и нижнего прямоугольников исключаются из расчетов. Представим, что площади двух смежных стен вычисляются по формулам **len_1 * height** и **len_2 * height**, соответственно. Ввиду равенства противоположных стен (прямоугольников), общая площадь четырех прямоугольников вычисляется по формуле: **S = 2 * (len_1 * height) + 2 * (len_2 * height) = 2 * height * (len_1 + len_2)**. Далее из вычисленной площади необходимо вычесть площадь окон и дверей, т. к. они не требуют поклейки обоев.

Перенесем параметры задачи на концепцию ООП. Выделим три класса: комнаты, окна, двери. Последние два класса относятся к комнате, поэтому они будут входит в состав объекта-комнаты. Для текущей задачи важны только свойства: длина и высота, поэтому классы окна и двери можно объединить.

Пример:

```
In [43]: class WindowDoor:
             def __init__(self, wd_len, wd_height):
                 self.square = wd_len * wd_height
```

Контейнером для окон и дверей является класс **«Комната»**, который должен содержать вызовы описанного выше класса **«ОкноДверь»**.

Пример:

In [46]:
```
class Room:
    def __init__(self, len_1, len_2, height):
        self.square = 2 * height * (len_1 + len_2)
        self.wd = []
    def add_win_door(self, wd_len, wd_height):
        self.wd.append(WindowDoor(wd_len, wd_height))
    def common_square(self):
        main_square = self.square
        for el in self.wd:
            main_square -= el.square
        return main_square
```

Проверим работу кода на примере:

Результат:

In [47]:
```
r = Room(7, 4, 3.7)
print(r.square)
r.add_win_door(2, 2)
r.add_win_door(2, 2)
r.add_win_door(2, 2)
print(r.common_square())
```

```
81.4
69.4
```