GeekBrains

Основы Python

Урок 10.

Объектно-ориентированное программирование. Продвинутый уровень

На этом уроке:

1. Научимся осуществлять перегрузку и переопределение методов.
2. Познакомимся с понятием интерфейса и интерфейса итерации.
3. Научимся создавать собственные объекты-итераторы.
4. Узнаем о назначении и особенностях применения декоратора @property.
5. Научимся реализовывать в ООП-проектах механизм композиции.

Оглавление

Перегрузка операторов

Перегрузка операторов — это изменение логики работы операторов языка с использованием специальных методов. Эти методы идентифицируются двойным подчёркиванием до и после имени метода.

Операторы — это знаки **+**, **-**, *****, **/.** Они отвечают за выполнение привычных математических операций. Операторы работают с особенностями синтаксиса языка и обеспечивают создание объекта, вызов его как функции, получение доступа к элементу объекта по индексу и т. д. К перегружаемым операторам относятся: **>**, **<**, **≤**, **≥**, **==**, **!=**, **+ =**, **-=**. При перегрузке каждого из них происходит вызов соответствующего метода, например:

- **__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, когда выполняется операция присваивания значения атрибуту;
- **__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» — **-=**.

Механизм перегрузки операторов на практике используется редко. На деле разработчику чаще всего приходится сталкиваться с перегрузкой в конструкторе. Но в рамках концепции ООП эта тема важна. Выше приведена только часть методов, используемых для 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. Конструктор класса отвечает за создание объекта класса.

Пример:

```
class MyClass:
    def __init__(self, param):
        self.param = param


mc = MyClass("text")
print(mc.param)
```

Результат:

```
text
```

__del__

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

Пример:

```
class MyClass:
    def __init__(self, param):
        self.param = param

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


mc = MyClass("text")
del mc
```

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

__str__

Пример:

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

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


mc = MyClass("text_1", "text_2")
print(mc)
```

Результат:

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

__add__

Пример:

```
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})"


mc_1 = MyClass(10, 20)
mc_2 = MyClass(30, 40)
print(mc_1 + mc_2)
```

Результат:

```
Объект с параметрами (40, 60)
```

__setattr__

Пример:

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


mc = MyClass()
mc.height = 40
```

Результат:

```
Атрибут height недопустим
```

__getitem__

Рассмотрим два примера.

Пример 1:

```
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))


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

Результат:

```
True
```

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

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

Пример 2:

```
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]


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

Результат:

```
10
True
text
```

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

__call__

Пример:

```
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};"


obj_1 = MyClass("width")
obj_2 = MyClass("height")

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

print(obj_1, obj_2)
```

Результат:

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

__eq__

Пример:

```
class MyClass:
    def __init__(self):
        self.x = 40

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


mc = MyClass()
print("Равно" if mc == 40 else "Не равно")
print("Равно" if mc == 41 else "Не равно")
```

Результат:

```
Равно
Не равно
```

__lt__

Пример:

```
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


s = Salary()
p = Prize()

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

Результат:

```
Оклад меньше премии?
False
```

__iadd__

Пример:

```
class MyClass:
    def __init__(self, val):
        self.val = val

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

mc = MyClass(100)
print(mc.val)
mc += 200
print(mc.val)
```

Результат:

```
100
300
```

Переопределение методов

Мы уже познакомились с одним из основных принципов ООП — наследованием. Пришло время рассмотреть новый подход, который используется при реализации наследования, или переопределение методов.

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

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

Пример:

```
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)


c = ChildClass()
c.my_method()
```

Результат:

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

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

Пример:

```
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()


c = ChildClass()
c.my_method()
```

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

Интерфейсы

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

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

Пример:

```
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()
```

Результат:

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

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

```
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()")

mc = MyClass()
mc.my_method_1()
```

Результат:

```
Метод my_method_1()
```

Интерфейс итерации

Итераторы — специальные объекты, которые обеспечивают пошаговый доступ к данным из контейнера. В привязке к ним работают циклы перебора (**for in**), встроенные функции (**map()**, **filter()**, **zip()**) и операция распаковки. Эти инструменты способны работать с любыми объектами, поддерживающими интерфейс итерации.

Рассмотрим небольшой пример:

```
my_list = [30, 105.6, "text", True]
for el in my_list:
    print(el)
```

Результат:

```
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** во время каждой итерации запускает метод **__next__()**, который при каждом вызове возвращает очередной элемент итератора.
3. Когда элементы итератора исчерпаны, метод **__next__()** завершает свою работу и генерирует исключение **StopIteration**. Цикл **for in** перехватывает это исключение и завершает свою работу.

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

Создание собственных объектов-итераторов

Создание объекта с поддержкой интерфейса итерации:

```
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, который поддерживает итерирование. А в виде класса **Iterator** — сам итератор, возвращающий очередной элемент итерируемого объекта. Здесь это числа, начиная от значения параметра **start** до 5 (включительно). **З**начение параметра **start** определяется при создании экземпляра класса **IterObj.**

Проверим работу примера:

```
obj = IterObj(start=2)
for el in obj:
    print(el)
```

Результат:

```
2
3
4
5
```

Можно проверить работу кода ещё раз:

```
print("Еще раз ...")
for el in obj:
    print(el)
```

Результат будет идентичен результату из примера выше.

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

```
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
```

Проверим работу примера:

```
obj = Iter(start=2)
for el in obj:
    print(el)
```

Результат:

```
2
3
4
5
```

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

Декоратор @property

Декоратор в Python — это функция (или класс), расширяющая логику работы другой функции. У разработчика существует возможность написания собственных декораторов или использования существующих. В рамках этого урока 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:

```
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}"

mc = MyClass("text_1", "text_2")

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

print(mc.my_method)
```

Результат:

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

В результате преобразования доступ к методу осуществляется с помощью обычной точечной нотации.

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

Для обеспечения контролируемого доступа к данным класса в Python применяются модификаторы доступа и свойства. Например, нужно проверить, что модель автомобиля должна быть выпущена в пределах 2000–2019 гг. Если пользователь введёт 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 его нужно оставить неизменным.

Пример:

```
# 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"

a = Auto(2090)
print(a.get_auto_year())
```

Свойство обладает тремя важными аспектами. Первым делом необходимо определить атрибут — год выпуска автомобиля. Далее нужно 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, результат будет:

```
Автомобиль выпущен в 2019 году
```

Для значения меньше 2000 результат:

```
Автомобиль выпущен в 2000 году
```

Больше информации о декораторах — в записи вебинара.

Композиция

В концепции ООП возможна реализация композиционного подхода. В соответствии с ним создаётся класс-контейнер, который включает 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, который состоит из шести прямоугольников. Площадь комнаты формируется на основе суммы площадей прямоугольников, входящих в параллелепипед. Вычислить площадь каждого прямоугольника можно через произведение его длины и высоты.

Так как обои необходимо клеить только на стены, площади верхнего и нижнего прямоугольников исключаются из расчётов. Представим, что площади двух смежных стен вычисляются по формулам **len_1 * height** и **len_2 * height** соответственно. Ввиду равенства противоположных стен (прямоугольников), общая площадь четырёх 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, так как они не требуют поклейки обоев.

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

Пример:

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

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

Пример:

```
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:

```
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())
```

Результат:

```
81.4
69.4
```

Особенности ООП в Python

Пришло время подвести промежуточные итоги по концепции ООП в Python. В основе каждого объекта лежит некоторый класс, от которого объект наследует атрибуты. В Python поддерживаются принципы ООП: инкапсуляция, наследование, полиморфизм. Но инкапсуляция, как механизм сокрытия данных, в Python поддерживается только на уровне соглашения, а не синтаксиса языка.

В Python возможна реализация множественного наследования, когда у дочернего класса существует несколько базовых. Благодаря такому подходу дочерний класс может сочетать собственные атрибуты и атрибуты нескольких классов-родителей.

Благодаря полиморфизму объекты в Python могут обладать сходными интерфейсами. Полиморфизм обеспечивается путём определения в классах методов с идентичными названиями. К проявлению полиморфизма также относится перегрузка операторов. Ещё одна особенность ООП — композиция

(агрегирование), когда в классе реализуются вызовы других классов. Далее при создании экземпляра класса-агрегатора генерируются объекты других классов, которые являются элементами агрегатора.

Классы принято помещать в файлы-модули. При этом в одном модуле можно хранить код нескольких классов. Модули объединяются в пакеты. И модули, и пакеты можно импортировать. Чтобы определить директорию в качестве пакета, необходимо создать в ней файл **__init__.py** без кода. Иначе при импорте пакета возникнет ошибка.

Преимущества ООП

Напоследок закрепим достоинства концепции ООП. Прежде всего это возможность использования одного и того же программного кода с разными данными. То есть с помощью ООП мы можем избежать дублирования кода. На основе классов генерируются их объекты с индивидуальными значениями свойств. Для обработки свойств (атрибутов) используются методы. Благодаря наследованию можно использовать код уже существующих классов, добавлять свой функционал.

Недостатки ООП

Требуется значительный анализ предметной области для оптимальной организации её в виде набора классов. На этом этапе важно определить сущности, которые можно использовать в виде классов. Нужно понимать, где допустимо использование наследования. Важно определить набор атрибутов и методов каждого класса. Одна и та же задача в ООП может быть решена по-разному. Но только с опытом приходит понимание, как определить оптимальное решение.

Важное по ООП в Python

1. Всё в Python — это объекты. Строка, число, список, словарь, функция, класс, модуль, пакет — объекты. Даже класс — тоже объект, порождающий другие объекты (экземпляры).
2. В Python все типы данных — классы.
3. Инкапсуляция в Python формальная. В других языках программирования инкапсуляция гарантирует защиту свойства класса от прямого доступа. В Python такой доступ сохраняется.

Практическое задание

1. Реализовать класс **Matrix** (матрица). Обеспечить перегрузку конструктора класса (метод **__init__()**), который должен принимать данные (список списков) для формирования матрицы.

   Подсказка: матрица — система некоторых математических величин, расположенных в виде прямоугольной схемы.

Примеры матриц: 3 на 2, 3 на 3, 2 на 4.

31	22		3	5	32		3	5	8	3
37	43		2	4	6		8	3	7	1
51	86		-1	64	-8					

Следующий шаг — реализовать перегрузку метода __str__() для вывода матрицы в привычном виде.

Далее реализовать перегрузку метода __add__() для сложения двух объектов класса Matrix (двух матриц). Результатом сложения должна быть новая матрица.

Подсказка: сложение элементов матриц выполнять поэлементно. Первый элемент первой строки первой матрицы складываем с первым элементом первой строки второй матрицы и пр.

2. Реализовать проект расчёта суммарного расхода ткани на производство одежды. Основная сущность (класс) этого проекта — **одежда**, которая может иметь определённое название. К типам одежды в этом проекте относятся **пальто** и **костюм**. У этих типов одежды существуют параметры: **размер** (для **пальто**) и **рост** (для **костюма**). Это могут быть обычные числа: **V** и **H** соответственно.

   Для определения расхода ткани по каждому типу одежды использовать формулы: для пальто **(V/6.5 + 0.5)**, для костюма **(2*H + 0.3)**. Проверить работу этих методов на реальных данных.

   Выполнить общий подсчёт расхода ткани. Проверить на практике полученные на этом уроке знания. Реализовать абстрактные классы для основных классов проекта и проверить работу декоратора **@property**.

3. Осуществить программу работы с органическими клетками, состоящими из ячеек. Необходимо создать класс «Клетка». В его конструкторе инициализировать параметр, соответствующий количеству ячеек клетки (целое число). В классе должны быть реализованы методы перегрузки арифметических операторов: сложение (**__add__()**), вычитание (**__sub__()**), умножение (**__mul__()**), деление (**__floordiv__**truediv__**()**). Эти методы должны применяться **только к клеткам** и выполнять увеличение, уменьшение, умножение и округление до целого числа деления клеток соответственно.

   Сложение. Объединение двух клеток. При этом число ячеек общей клетки должно равняться сумме ячеек исходных двух клеток.

Вычитание. Участвуют две клетки. Операцию необходимо выполнять, только если разность количества ячеек двух клеток больше нуля, иначе выводить соответствующее сообщение.

Умножение. Создаётся общая клетка из двух. Число ячеек общей клетки — произведение количества ячеек этих двух клеток.

Деление. Создаётся общая клетка из двух. Число ячеек общей клетки определяется как целочисленное деление количества ячеек этих двух клеток.

В классе необходимо реализовать метод **make_order()**, принимающий экземпляр класса и количество ячеек в ряду. Этот метод позволяет организовать ячейки по рядам.

Метод должен возвращать строку вида *****\n*****\n*****..., где количество ячеек между **\n** равно переданному аргументу. Если ячеек на формирование ряда не хватает, то в последний ряд записываются все оставшиеся.

Например, количество ячеек клетки равняется 12, а количество ячеек в ряду — 5. В этом случае метод **make_order()** вернёт строку: *****\n*****\n****.

Или количество ячеек клетки — 15, а количество ячеек в ряду равняется 5. Тогда метод **make_order()** вернёт строку: *****\n*****\n*****.

Подсказка: подробный список операторов для перегрузки доступен по ссылке.

Глоссарий

1. **Декоратор в Python** — это функция (или класс), расширяющая логику работы другой функции.
2. **Итераторы** — специальные объекты, которые обеспечивают пошаговый доступ к данным из контейнера.
3. **Интерфейс в ООП** — описание поведения объекта.
4. **Операторы** — это знаки **+**, **-**, *, **/.** Они отвечают за выполнение привычных математических операций.
5. **Перегрузка операторов** — это изменение логики работы операторов языка с использованием специальных методов.

Дополнительные материалы

1. Перегрузка операторов.
2. Переопределение методов в Python.
3. Изучаем декораторы в Python.
4. Абстрактные классы и интерфейсы в Python.

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. Язык программирования Python 3 для начинающих и чайников.
2. Программирование в Python.
3. Учим Python качественно (habr).
4. Самоучитель по Python.
5. Лутц М. Изучаем Python. — 4-е изд. — М.: Символ-Плюс, 2011.