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](#_qimlcdrewn0i)

[\_\_init\_\_](#_9n5web6bt7rx)

[\_\_del\_\_](#_5vvmhku9ql16)

[\_\_str\_\_](#_3r54qw56ukw2)

[\_\_add\_\_](#_pd24xbgrfa7w)

[\_\_setattr\_\_](#_xx22fpzedmc8)

[\_\_getitem\_\_](#_45m7n33vay1c)

[\_\_call\_\_](#_ry6yl3gzsaf)

[\_\_eq\_\_](#_2hdcwfexwzkx)

[\_\_lt\_\_](#_6o5iibgiotp4)

[\_\_iadd\_\_](#_l6dwvu2vel45)

[Переопределение методов](#_1hysx2slo209)

[Интерфейсы](#_dj7ar8b5p0p)

[Интерфейс итерации](#_54hcpeqmtu5y)

[Создание собственных объектов-итераторов](#_1chonsk963kk)

[Декоратор @property](#_ka4thdfwbcl2)

[Композиция](#_jl6m1kbirdiy)

[Особенности ООП в Python](#_ws5bem1lx0vh)

[Преимущества ООП](#_8eq7allnpbgx)

[Недостатки ООП](#_n6a17ichgl9h)

[Важное по ООП в Python](#_e06gaibit2m7)

[Практическое задание](#_n8q7yzi9yj5r)

[Глоссарий](#_3df3ft4q4wqr)

[Дополнительные материалы](#_jik6wf4zqy8t)

[Используемая литература](#_tnflastqfeho)

# 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](https://pythonworld.ru/osnovy/peregruzka-operatorov.html).

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](https://geekbrains.ru/go/02ms72).

# 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 | | --- | --- | | 37 | 43 | | 51 | 86 | | | 3 | 5 | 32 | | --- | --- | --- | | 2 | 4 | 6 | | -1 | 64 | -8 | | | 3 | 5 | 8 | 3 | | --- | --- | --- | --- | | 8 | 3 | 7 | 1 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

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.

1. 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**.

1. 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](https://pythonworld.ru/osnovy/peregruzka-operatorov.html).

# 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](https://pythonworld.ru/osnovy/peregruzka-operatorov.html).
2. [Переопределение методов в Python](http://magicofpython.blogspot.com/2015/12/python_17.html).
3. [Изучаем декораторы в Python](https://python-scripts.com/decorators).
4. [Абстрактные классы и интерфейсы в Python](https://habr.com/ru/post/72757/).

# Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. [Язык программирования Python 3 для начинающих и чайников](https://pythonworld.ru/).
2. [Программирование в Python](https://python-scripts.com/).
3. [Учим Python качественно (habr)](https://habrahabr.ru/post/150302/).
4. [Самоучитель по Python](http://pythonworld.ru/samouchitel-python).
5. [Лутц М. Изучаем Python. — 4-е изд. — М.: Символ-Плюс, 2011](http://www.proklondike.com/books/python/lutz_python_2011.html).