
# Учебный курс **«**Основы программирования на **Python»**

# Объектно-ориентированное программирование
# Лекция 3

- Классы и объекты
- Встроенные классы
- Абстрагирование
- Инкапсуляция
- Наследование
- Полиморфизм
- Перегрузка операторов
- Композиции

# Что такое ООП?

- Объе́ктно-ориенти́рованное программи́рование (ООП) —методология (парадигма, набор правил) программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

# Откуда взялось ООП?

Если мы взглянем на реальный мир под тем углом, под которым привыкли на него смотреть, то для нас он предстанет в виде множества объектов, обладающих определенными свойствами и способностями взаимодействовать между собой и изменяться. Эта привычная для взгляда человека картина мира была перенесена в программирование.

# Преимущество ООП

Каждый программист может разрабатывать свою группу объектов. Разработчикам достаточно договориться между собой только о том, как их объекты будут взаимодействовать между собой, то есть об их интерфейсах. Пете не надо знать, как Вася реализует рост коров в результате поедания травы. Ему, как разработчику лужаек, достаточно знать, что когда любая корова прикасается к траве, последней на соответствующей лужайке должно стать меньше.

# Понятие класса

- Реальный мир: совокупность объектов с одинаковыми свойствами и способностями.
- Программирование: совокупность свойств (полей, атрибутов) и способностей (методов, поведения), которыми будут обладать все объекты этого класса.
- Столы, за которыми вы сидите – это объекты класса стол.

# Принципы ООП

- Абстрагирование.
- Наследование.
- Инкапсуляция.
- Полиморфизм.

# Понятие абстрагирования

- Абстрагирование – выделение в моделируемом реальном объекте важных для решения конкретной задачи свойств и методов для создания класса.
- Для задачи грузоперевозок в классе «стол» будет важна масса, а для задачи организации ресторана будет важно число мест за столом.

# Понятие наследования

- Дочерние классы наследуют свойства и методы родительских, однако дополняют или в определенной степени модифицируют их характеристики.
- Когда мы создаем конкретный экземпляр стола, то должны выбрать, к какому классу столов он будет принадлежать.
- Если он принадлежит классу журнальных столов, то получит все характеристики общего класса столов и класса журнальных столов, но не особенности письменных и обеденных.

# Понятие инкапсуляции

- Под инкапсуляцией понимают сокрытие свойств и методов, в результате чего они становятся доступными только в других методах этого же класса и дочерних классов и недоступны в методах других классов.
- Для сокрытия атрибутов в **Python** используется соглашение, согласно которому, если свойство или метод имеют два знака подчеркивания впереди имени, но не сзади, то этот атрибут предусмотрен исключительно для внутреннего пользования.

# Понятие полиморфизма

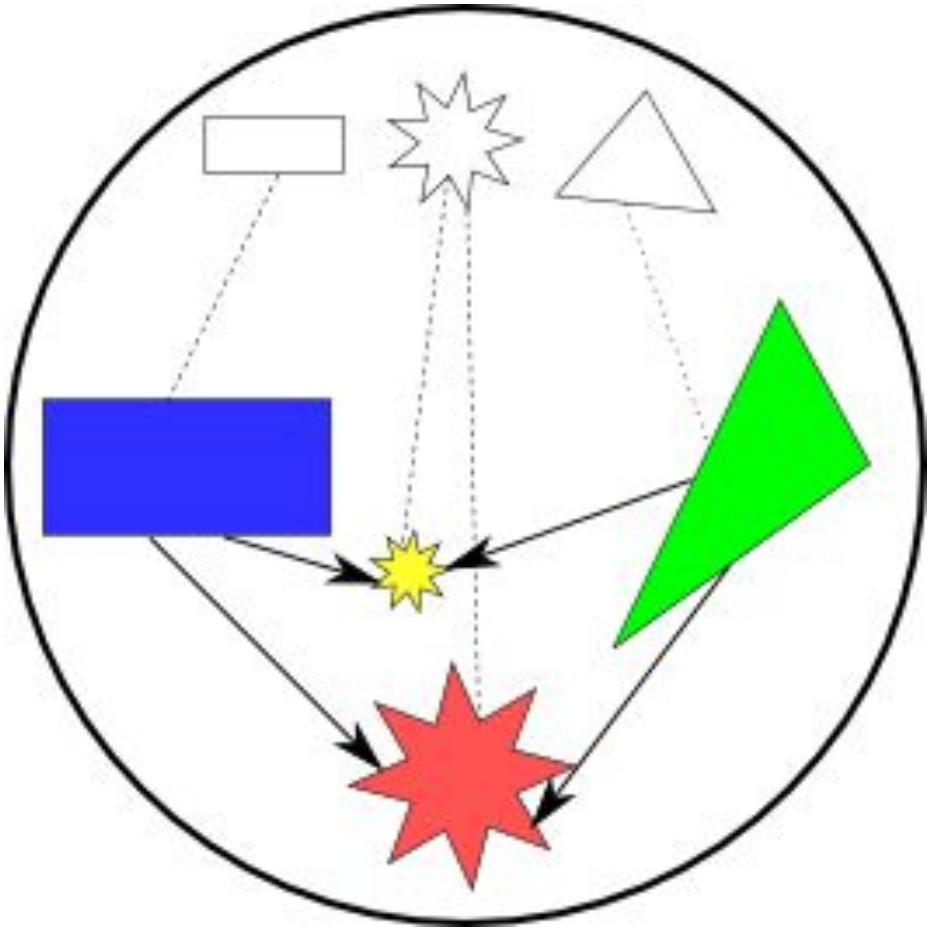
- **Полиморфизм –** это множество внутренних реализаций скрытых одной общей внешней формой.
- Объекты разных классов, с разной внутренней реализацией, то есть программным кодом, могут иметь одинаковые свойства и методы.
- Вы уже сталкивались с полиморфизмом операции **+**: для чисел она обозначает сложение, а для строк – конкатенацию.
- Полиморфизм проявляется во внутренней реализации и результате операции.

# Пример

Рассмотрите схему.

Подумайте над следующими вопросами:

1. Какие фигуры на ней вы бы назвали классами, а какие – объектами?
2. Что обозначают пунктирные линии?
3. Может ли объект принадлежать нескольким классам?
4. Может ли у класса быть множество объектов?
5. Звезды скорее обладают разными свойствами или разным поведением?
6. Могут ли свойства оказывать влияние на поведение?
7. Что обозначено стрелками?



# Пример

1. Классы – прозрачные фигуры, объекты – залитые, обладающие значением свойства «цвет».
2. Принадлежность объекта к классу.
3. Может за счет наследования.
4. Может, обычно так и есть.
5. Разными значениями свойств «размер», «положение», «цвет».
6. Могут, если в методах есть условия, проверяющие значения свойств.
7. Передача прямоугольника и треугольника в качестве параметров при вызове методов звезд.

# Создание классов

- Классы создаются с помощью команды **class,** за которой следует произвольное имя класса**,** после которого ставится двоеточие**,** далее с новой строки и с отступом реализуется тело класса **(**свойства и методы**).**
- Пример**:** класс **«**стол**»** со свойством **«**масса**»:**

```
class Table:
    mass = 0
```

# Создание дочерних классов

- Если класс является дочерним, то родительские классы перечисляются в круглых скобках после имени класса.
- В **Python** допускается наследование от нескольких классов сразу (множественное наследование).

# Создание дочерних классов

- Пример: классы «обеденный стол» и «журнальный стол», наследующие класс «стол»:

```
5   class DinnerTable(Table):
6       places = 0
7
8
9   class JournalTable(Table):
10      storage = 0
```

# Создание объектов

- Объект создается путем вызова класса по его имени. При этом после имени класса обязательно ставятся скобки.
- Пример: создание нового объекта класса «обеденный стол»:

```
13    newTable = DinnerTable()
```

# Конструктор класса – метод **__init__()**

- Конструктором класса (или конструктором объектов класса) называют метод, который автоматически вызывается при создании объектов.
- В **Python** роль конструктора играет метод **__init__()**.
- Может принимать параметры.
- Параметры являются обязательными если нет конструктора без параметров и если параметрам не заданы значения по умолчанию.

# Конструктор класса – метод **__init__()**

- Пример: конструктор класса «стол» для установки массы каждого стола при его создании:

```
1  class Table:
2      __mass = 0
3
4      def __init__(self, mass0):
5          self.__mass = mass0
6
```

# Конструктор класса – метод **__init__()**

- Конструкторы наследуются, как и другие методы.
- Пример: создание списка обеденных столов разной массы:

```
48  newTable = [
49      DinnerTable(10),
50      DinnerTable(20),
51      DinnerTable(30)]
```

# Применение инкапсуляции

- Разрешим задавать массу только в конструкторе – каким стол выпустили, таким он и будет.
- Массу нужно будет считывать при погрузке столов в грузовик для транспортировки.
- Методы для считывания называют геттерами, их названия составляют из слова **get** и названия свойства.
- Методы присвоения называют сеттерами, их названия составляют из слова **set** и названия свойства.

# Применение инкапсуляции

```
1  class Table:
2      __mass = 0
3
4      def __init__(self, mass0):
5          self.__mass = mass0
6
7      def get_mass(self):
8          return self.__mass
9
```

# Ссылка **self**

- Обычно первым параметром методов пишется **self –** ссылка на объект**,** для которого вызывается метод**.**
- Переменная **self** связывается с объектом**,** к которому был применен данный метод **(**стоящему перед точкой перед методом при вызове**),** и через эту переменную мы получаем доступ к атрибутам объекта**.**

# Простое наследование методов родительского класса

В качестве примера рассмотрим разработку класса столов и его двух подклассов – кухонных и письменных столов. Все столы, независимо от своего типа, имеют длину, ширину и высоту. Пусть для письменных столов важна площадь поверхности, а для кухонных – количество посадочных мест. Общее вынесем в класс, частное – в подклассы.

Связь между родительским и дочерним классом устанавливается через дочерний: родительские классы перечисляются в скобках после его имени.

# Простое наследование методов родительского класса

```
class Table:
    def __init__(self, l, w, h):
        self.lenght = l
        self.width = w
        self.height = h

class KitchenTable(Table):
    def setPlaces(self, p):
        self.places = p

class DeskTable(Table):
    def square(self):
        return self.width * self.length
```

# Полное переопределение метода надкласса

Что если в подклассе нам не подходит код метода его надкласса. Допустим, мы вводим еще один класс столов, который является дочерним по отношению к **DeskTable.** Пусть это будут компьютерные столы, при вычислении рабочей поверхности которых надо отнимать заданную величину. Имеет смысл внести в этот новый подкласс его собственный метод **square():**

```
class ComputerTable(DeskTable):
    def square(self, e):
        return self.width * self.length - e
```

# Полное переопределение метода надкласса

При создании объекта типа **ComputerTable** по-прежнему требуется указывать параметры, так как интерпретатор в поисках конструктора пойдет по дереву наследования сначала в родителя, а потом в прародителя и найдет там метод **__init__()**.

Однако когда будет вызываться метод **square()**, то поскольку он будет обнаружен в самом **ComputerTable**, то метод **square()** из **DeskTable** останется невидимым, т. е. для объектов класса **ComputerTable** он окажется переопределенным.

```
>>> from test import ComputerTable
>>> ct = ComputerTable(2, 1, 1)
>>> ct.square(0.3)
1.7
```

# Дополнение, оно же расширение, метода

Если посмотреть на вычисление площади, то часть кода надкласса дублируется в подклассе. Этого можно избежать, если вызвать родительский метод, а потом дополнить его:

```
class ComputerTable(DeskTable):
    def square(self, e):
        return DeskTable.square(self) - e
```

Здесь вызывается метод другого класса, а потом дополняется своими выражениями. В данном случае вычитанием.

Рассмотрим другой пример. Допустим, в классе **KitchenTable** нам не нужен метод, поле **places** должно устанавливаться при создании объекта в конструкторе. В классе можно создать собственный конструктор с чистого листа, чем переопределить родительский:

```
class KitchenTable(Table):
    def __init__(self, l, w, h, p):
        self.length = l
        self.width = w
        self.height = h
        self.places = p
```

Однако это не надкласса. Проще вызвать родительский конструктор, после чего дополнить своим кодом:

```
class KitchenTable(Table):
    def __init__(self, l, w, h, p):
        Table.__init__(self, l, w, h)
        self.places = p
```

# Пример полиморфизма

Например, два разных класса содержат метод **total,** однако инструкции каждого предусматривают совершенно разные операции. Так в классе **T1 –** это прибавление **10** к аргументу, в **T2 –** подсчет длины строки символов. В зависимости от того, к объекту какого класса применяется метод **total,** выполняются те или иные инструкции.

```
class T1:
     n=10
     def total(self, N):
          self.total = int(self.n) + int(N)

class T2:
     def total(self,s):
          self.total = len(str(s))

t1 = T1()
t2 = T2()
t1.total(45)
t2.total(45)
print(t1.total) # Вывод: 55
print(t2.total) # Вывод: 2
```

# Полиморфизм

- У каждого может быть свой метод **__init__()** или **square()** или какой-нибудь другой. Какой именно из методов **square()** вызывается, и что он делает, зависит от принадлежности объекта к тому или иному классу.

- Однако классы не обязательно должны быть связанны наследованием. Полиморфизм как один из ключевых элементов ООП существует независимо от наследования. Классы могут быть не родственными, но иметь одинаковые методы, как в примере выше.

- Полиморфизм дает возможность реализовывать так называемые единые интерфейсы для объектов различных классов. Например, разные классы могут предусматривать различный способ вывода той или иной информации объектов. Однако одинаковое название метода вывода позволит не запутать программу, сделать код более ясным.

# Полиморфизм у методов перегрузки операторов

- Полиморфизм у методов перегрузки операторов проявляется в том, что независимо от типа объекта, его участие в определенной операции, вызывает метод с конкретным именем. В случае **__init()__** операцией является создание объекта.
- Рассмотрим пример полиморфизма на еще одном методе, который перегружает функцию **print()**.
- Если вы создадите объект собственного класса, а потом попробуете вывести его на экран, то получите информацию о классе объекта и его адрес в памяти.

```
>>> class A:
...     def __init__(self, v1, v2):
...             self.field1 = v1
...             self.field2 = v2
...
>>> a = A(3, 4)
>>> print(a)
<__main__.A object at 0x7f840c8acfd0>
```

# Полиморфизм у методов перегрузки операторов

- Если же мы хотим, чтобы, когда объект передается функции **print()**, выводилась какая-нибудь другая более полезная информация, то в класс надо добавить специальный метод **__str__()**. Этот метод должен обязательно возвращать строку, которую будет выводить функция **print()**:

```
class A:
    def __init__(self, v1, v2):
        self.field1 = v1
        self.field2 = v2
    def __str__(self):
        return str(self.field1) + " " + str(self.field2)

a = A(3, 4)
print(a)
```

Вывод:

```
3 4
```

# Полиморфизм у методов перегрузки операторов

Какую именно строку возвращает метод **__str__(),** дело второстепенное**.** Он вполне может строить квадратик из символов**:**

```
class Rectangle:
    def __init__(self, width, height, sign):
        self.w = int(width)
        self.h = int(height)
        self.s = str(sign)
    def __str__(self):
        rect = []
        for i in range(self.h): # количество строк
            rect.append(self.s * self.w) # знак повторяется w раз
        rect = '\n'.join(rect) # превращаем список в строку
        return rect

b = Rectangle(10, 3, '*')
print(b)
```

Вывод:

```
**********
**********
**********
```

# Инкапсуляция

Если класс имеет счетчик своих объектов, то необходимо исключить возможность его случайного изменения из вне.

```
class B:
    count = 0
    def __init__(self):
        B.count += 1
    def __del__(self):
        B.count -= 1

a = B()
b = B()
print(B.count) # выведет 2
del a
print(B.count) # выведет 1
```

Все работает. В чем тут может быть проблема? Проблема в том, что если в основной ветке где-то по ошибке или случайно произойдет присвоение полю **B.count**, то счетчик будет испорчен:

```
…
B.count -= 1
print(B.count) # будет выведен 0, хотя остался объект b
```

# Инкапсуляция

Для имитации сокрытия атрибутов в **Python** используется соглашение **(**соглашение **–** это не синтаксическое правило языка**,** при желании его можно нарушить**),** согласно которому**,** если поле или метод имеют два знака подчеркивания впереди имени**,** но не сзади**,** то этот атрибут предусмотрен исключительно для внутреннего пользования**:**

```
class B:
    __count = 0
    def __init__(self):
        B.__count += 1
    def __del__(self):
        B.__count -= 1

a = B()
print(B.__count)
```

Попытка выполнить этот код приведет к выбросу исключения:

```
  File "test.py", line 9, in <module>
    print(B.__count)
AttributeError: type object 'B' has no attribute '__count'
```

# Инкапсуляция

- То есть атрибут **__count** за пределами класса становится невидимым, хотя внутри класса он вполне себе видимый. Понятно, если мы не можем даже получить значение поля за пределами класса, то присвоить ему значение – тем более.

- На самом деле сокрытие в **Python** не настоящее и доступ к счетчику мы получить все же можем. Но для этого надо написать **B._B__count:**

```
…
print(B._B__count)
```

# Инкапсуляция

Мы защитили поле от случайных изменений. Для получения значения этого поля можно сделать с помощью добавления метода:

```
class B:
    __count = 0
    def __init__(self):
        B.__count += 1
    def __del__(self):
        B.__count -= 1
    def qtyObject():
        return B.__count

a = B()
b = B()
print(B.qtyObject()) # будет выведено 2
```

# Инкапсуляция

То же самое с методами. Их можно сделать "приватными" с помощью двойного подчеркивания:

```
class DoubleList:
    def __init__(self, l):
        self.double = DoubleList.__makeDouble(l)
    def __makeDouble(old):
        new = []
        for i in old:
            new.append(i)
            new.append(i)
        return new

nums = DoubleList([1, 3, 4, 6, 12])
print(nums.double)
print(DoubleList.__makeDouble([1,2]))
```

Результат:

```
[1, 1, 3, 3, 4, 4, 6, 6, 12, 12]
Traceback (most recent call last):
  File "test.py", line 13, in <module>
    print(DoubleList.__makeDouble([1,2]))
AttributeError: type object 'DoubleList' has no attribute '__makeDouble'
```

# Метод **__setattr__()**

В **Python** атрибуты объекту можно назначать за пределами класса**:**

```
>>> class A:
...     def __init__(self, v):
...             self.field1 = v
...
>>> a = A(10)
>>> a.field2 = 20
>>> a.field1, a.field2
(10, 20)
```

# Метод __setattr__()

Если такое поведение нежелательно, его можно запретить с помощью метода перегрузки оператора присваивания атрибуту **__setattr__():**

```
>>> class A:
...     def __init__(self, v):
...             self.field1 = v
...     def __setattr__(self, attr, value):
...             if attr == 'field1':
...                     self.__dict__[attr] = value
...             else:
...                     raise AttributeError
...
>>> a = A(15)
>>> a.field1
15
>>> a.field2 = 30
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __setattr__
AttributeError
>>> a.field2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'field2'
>>> a.__dict__
{'field1': 15}
```

- Поясним, что здесь происходит. Метод **__setattr__(),** если он присутствует в классе, вызывается всегда, когда какому-либо атрибуту выполняется присваивание. Обратите внимание, что присвоение несуществующему атрибуту также обозначает его добавление к объекту.

- Когда создается объект **a,** в конструктор передается число **15.** Здесь для объекта заводится атрибут **field1.** Факт попытки присвоения ему значения тут же отправляет интерпретатор в метод **__setattr__(),** где проверяется соответствует ли имя атрибута строке **'field1'.** Если так, то атрибут и соответствующее ему значение добавляется в словарь атрибутов объекта.

- Нельзя в **__setattr__()** написать просто **self.field1 = value,** так как это приведет к новому рекурсивному вызову метода **__setattr__().** Поэтому поле назначается через словарь **__dict__,** который есть у всех объектов, и в котором хранятся их атрибуты со значениями.

- Если параметр **attr** не соответствует допустимым полям, то искусственно возбуждается исключение **AttributeError.** Мы это видим, когда в основной ветке пытаемся обзавестись полем **field2.**

# Композиция

- Еще одной особенностью объектно-ориентированного программирования является возможность реализовывать так называемый композиционный подход. Заключается он в том, что есть класс-контейнер, он же агрегатор, который включает в себя вызовы других классов. В результате получается, что при создании объекта класса-контейнера, также создаются объекты включенных в него классов.

- Чтобы понять, зачем нужна композиция в программировании, проведем аналогию с реальным миром. Большинство биологических и технических объектов состоят из более простых частей, также являющихся объектами. Например, животное состоит из различный органов (сердце, желудок), компьютер — из различного "железа" (процессор, память).

- Не следует путать композицию с наследованием, в том числе множественным. Наследование предполагает принадлежность к какой-то общности (похожесть), а композиция — формирование целого из частей. Наследуются атрибуты, т. е. возможности, другого класса, при этом объектов непосредственно родительского класса не создается. При композиции же класс-агрегатор создает объекты других классов.

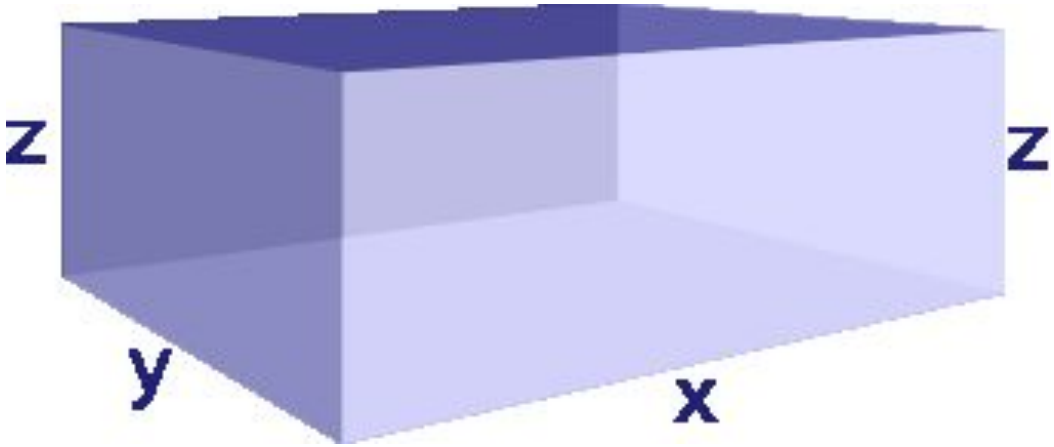
# Композиция

Требуется написать программу, которая вычисляет площадь обоев для оклеивания помещения. При этом окна, двери, пол и потолок оклеивать не надо.

Комната – это прямоугольный параллелепипед, состоящий из шести прямоугольников. Его площадь представляет собой сумму площадей составляющих его прямоугольников. Площадь прямоугольника равна произведению его длины на ширину.

# Композиция

По условию задачи обои клеятся только на стены, следовательно площади верхнего и нижнего прямоугольников нам не нужны. Из рисунка видно, что площадь одной стены равна **xz**, второй – y**z**. Противоположные прямоугольники равны, значит общая площадь четырех прямоугольников равна **S = 2xz + 2**y**z = 2z(x+y).** Потом из этой площади надо будет вычесть общую площадь дверей и окон, поскольку они не оклеиваются.



# Композиция

- Для данной задачи существенное значение имеют только два свойства – длина и ширина. Поэтому классы «окна» и «двери» можно объединить в один. Если бы были важны другие свойства (например, толщина стекла, материал двери), то следовало бы для окон создать один класс, а для дверей – другой. Пока обойдемся одним, и все что нам нужно от него – площадь объекта:

```
class Win_Door:
    def __init__(self, x, y):
        self.square = x * y
```

- Кла
дверей. Он должен содержать вызовы класса "окно_дверь".

# Композиция

- Хотя помещение не может быть совсем без окон и дверей, но может быть чуланом, дверь которого также оклеивается обоями. Поэтому имеет смысл в конструктор класса вынести только размеры самого помещения, без учета элементов "дизайна", а последние добавлять вызовом специально предназначенного для этого метода, который будет добавлять объекты-компоненты в список.

```
class Room:
    def __init__(self, x, y, z):
        self.square = 2 * z * (x + y)
        self.wd = []
    def addWD(self, w, h):
        self.wd.append(WinDoor(w, h))
    def workSurface(self):
        new_square = self.square
        for i in self.wd:
            new_square -= i.square
        return new_square

r1 = Room(6, 3, 2.7)
print(r1.square) # выведет 48.6
r1.addWD(1, 1)
r1.addWD(1, 1)
r1.addWD(1, 2)
print(r1.workSurface()) # выведет 44.6
```

# Контактная информация

Репозиторий на **GitHub.com** с лекциями и заданиями**:**
**https://github.com/Burmakova/Python**