Введение в объектно-ориентированное программирование (ООП) на Python

Пояснительная записка

Элективный курс по информатике "Введение в объектно-ориентированное программирование на Python" представляет собой вводный курс, знакомящий с особенностями ООП (классах, объектах, наследовании, полиморфизме и др.).

Место курса "Введение в объектно-ориентированное программирование на Python" в составе образовательной программы дополнительного образования детей "Лаборатория юного линуксоида" — второй год обучения.

Курс рассчитан примерно на 12 часов.

Цели и задачи курса

Основной целью данного элективного курса является формирование базовых понятий объектно-ориентированного программирования, развитие системного мышления обучающихся. Курс не является учебником по Python.

Примечания
Курс мягко говоря сыроват.

План изложения отличается от большинства учебников по программированию, где описывают ООП. Обычно сначала вводятся такие понятия как класс, объект, наследование, инкапсуляция и полиморфизм. Возможно, это слишком большой объем абстрактной информации, которую детям сложно усвоить сразу.

В данном курсе сначала вводятся понятия класса и объекта, потом, постепенно, рассматриваются вопросы наследования, полиморфизма и др. При этом далеко не абстрактно, а на конкретных примерах. И только на предпоследнем уроке конкретизируются основные идеи объектно-ориентированного программирования.

Что касается инкапсуляции, то в python ее как таковой нет. По сути то, что есть – это не совсем инкапсуляция; вводить сложное понятие, которое в данном случае еще и немного другое – достаточно трудно. Поэтому урок про инкапсуляцию в элективном курсе опущен.

- Общее представление об объектно-ориентированном программировании. Понятия класса и объекта. Урок 1
- Создание классов и объектов. Урок 2
- Конструктор класса — метод __init__. Урок 3
- Наследование в ООП на Python. Урок 4
- Полиморфизм и переопределение методов в ООП на Python. Урок 5
- Композиционный подход в объектно-ориентированном программировании. Урок 7

- Модули и их импорт. Урок 8
- Строки документации исходного кода на Python. Урок 9
- Перегрузка операторов в ООП. Урок 10
- Особенности объектно-ориентированного программирования. Урок 11
- Пример объектно-ориентированной программирования на языке Python. Урок 12

Общее представление об объектно-ориентированном программировании. Понятия класса и объекта. Урок 1

Урок - лекция

Элективный курс: Введение в объектно-ориентированное программирование на Python

Уровень: Программирование для начинающих

Циклы, ветвления, функции — все это элементы так называемого структурного программирования (директивная парадигма программирования). Для написания небольших программ возможностей структурного программирования обычно достаточно. Однако крупные проекты, работу над которыми ведут группы людей, намного рациональней выполнять используя парадигму объектно-ориентированного программирования. Почему? Это мы выясним позднее. Сначала разберемся в общих чертах, что из себя представляет объектно-ориентированное программирование.

ООП — аббревиатура (сокращенное название) объектно-ориентированного программирования.

Истоки ООП начинаются с 60-х годов XX века. Однако окончательное оформление и популяризацию можно отнести к 80-м годам XX века. Особую роль сыграл Алан Кей, сформулировавший основные принципы ООП. Наверное в настоящее время большинство проектов реализуются в стиле ООП. Хотя в программировании операционных систем (системном программировании) большую роль играет язык С (это не ОО язык).

Итак, что же такое ООП? Судя по названию ключевую роль в этой парадигме играет некий объект, а точнее множество объектов. Реальный мир состоит из объектов и их взаимодействий между собой. В результате взаимодействий объекты могут изменяться сами или изменять другие объекты. Поэтому, можно сказать, что ООП является более естественным в каком-то смысле.

В мире можно условно выделять различные системы, реализующие определенные цели (изменяющиеся из одного состояния в другое). Например, группа на занятии. Это система, состоящая из таких объектов как дети, учитель, столы, компьютеры, проектор и др. У этой

системы можно выделить основную цель - увеличение доли знаний детей на некую величину. Чтобы добиться этого, объекты системы должны определенным образом выполнить взаимодействие между собой.

Пример с занятием - это своего рода программа. Допустим, что какому-то «глобальному программисту» нужно было, чтобы на планете люди обладали обширными знаниями. Для этого он придумал специальную программу, которая вбирает на входе людей с N-количеством знаний, а на выходе возвращает с N+1(возможно чуть меньше-больше)-количеством знаний. Он наделил определенными способностями объекты этой системы, чтобы можно было добиться результата. Так, грубо говоря, «объекты-дети» способны воспринимать информацию, «объект-учитель» - ее транслировать, «объекты-предметы» помогать воспринимать и транслировать. Несмотря на то, что все ученики так или иначе способны воспринимать информацию, они различны по своим свойствам (по скорости и объему восприятия, способам обработки знаний и т.п.). Выполнение программы может происходить примерно таким образом: «объект-учитель», используя «объект-доска», «объект-компьютер», «объекты-картинки», передает информацию «объектам-детям». Те в свою очередь принимают информацию и изменяют свои свойства (допустим, количество знаний в голове). На выходе мы получаем «объектов-детей» с новыми свойствами (хотя бывает, что программа дает сбои по разным причинам).

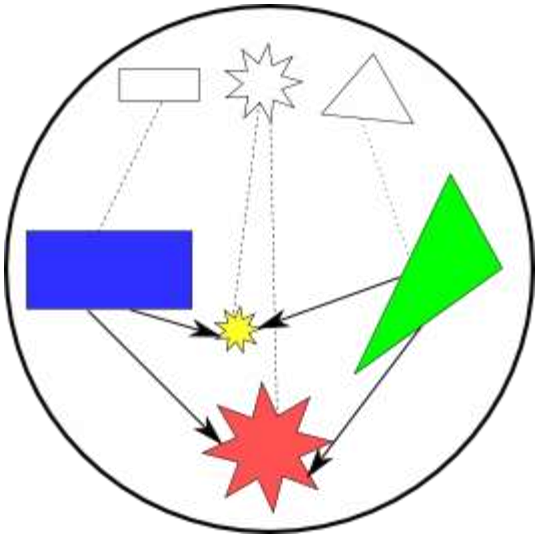
Следует понимать, существенную разницу между программой написанной с структурном «стиле» и программой в «стиле» ООП. В первом случае, на первый план выходит логика, понимание последовательности выполнения выражений (действий) для достижения целей. Во-втором — важно системное мышление, умение видеть систему в целом, с одной стороны, и понимание роли ее частей (объектов), с другой.

В свое время Алан Кей сформулировал для разработанного им языка программирования Smalltalk несколько принципов. Они прекрасно описывают принципы ООП. Так например, утверждается, что объектно-ориентированная программа состоит из объектов, которые посылают друг другу сообщения. Каждый объект может состоять из других объектов (а может и не состоять). Каждый объект принадлежит определенному классу (типу), который задает поведение объектов, созданных на его основе.

Что такое класс или тип? В реальном мире стол — это объект. Но когда его изготавливают, то руководствуются определенным описанием (знанием), что такое стол? Я могу сказать «стол» не имея ввиду никакой конкретный, но большинство поймут, о чем идет речь, т.к знают особенности этого предмета (крышка, четыре ножки и т. п.).

Класс — это описание объектов определенного типа. В каком-то смысле - это абстракция без материального воплощения, которая позволяет систематизировать объекты той или иной системы.

На основе классов создаются объекты. Может быть множество объектов, принадлежащих одному классу. С другой стороны, может быть класс без объектов, реализованных на его основе.



Посмотрите на рисунок. Допустим внешняя окружность — это программа. Она состоит из объектов (цветные фигуры) и классов (белые фигуры). Так объекты «красная_крупная_звезда» и «желтая_мелкая_звезда» могут обрабатывать (видоизменять) объекты «зеленый_треугольник» и «синий_прямоугольник». Звезды — разные объекты, хотя и принадлежат к одному классу. Поэтому можно заподозрить, что обрабатывают они объекты немного по разному. В данном случае объект-прямоугольник и объект-треугольник можно представить исключительно как данные. Кстати, в Python даже число — это объект, принадлежащий классу (типу) integer или float (или другому числовому типу).

Приведем более реальный программный пример (а не «из жизни про занятие»). Допустим нужно создать программу по обработке текстовой информации. Эта программа должна получать от пользователя данные, обрабатывать определенным способом, а затем выдавать на экран. Причем нам сказали написать эту программу, используя парадигму ООП. Выделим объекты системы: пусть это будут «приемщик», «обработчик №1», «обработчик №2» и «отображатель». Итак, пользователь передает «приемщику» текст и информацию каким «обработчиком» обрабатывать. «Приемщик» может взять все это (у него предусмотрены такие функции) и, предварительно оценив, что дают, может передать тому «обработчику», который выбрал пользователь. Выбранный «обработчик» видоизменяет текст и передает его «отображателю». Тот, в свою очередь, специфически форматирует текст и выводит на экран. Также в данной программе может быть предусмотрена возможность напрямую передавать текст от «приемщика» к «отображателю» минуя «обработчики».

Придумайте свою систему взаимодействующих объектов.

Многие современные языки поддерживают несколько парадигм программирования (например, директивное, функциональное, объектно-ориентированное). Такие языки являются смешанными. К ним относится и Python.

Создание классов и объектов. Урок 2

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Итак, программа, написанная с использованием парадигмы объектно-ориентированного программирования, должна состоять из

- объектов,
- классов (описания объектов),
- взаимодействий объектов между собой, в результате которых меняются их свойства.

Объект в программе можно создать лишь на основе какого-нибудь класса. Поэтому, первым делом, ООП должно начинаться с проектирования и создания классов. Классы могут располагаться или вначале кода программы, или импортироваться из других файлов-модулей (также в начале кода).

Создание классов:

Для создания классов предусмотрена **инструкция class**. Это составная инструкция, которая состоит из строки заголовка и тела. Заголовок состоит из ключевого слова class, имени класса и, возможно, названий суперклассов в скобках. Суперклассов может и не быть, в таком случае скобки не требуются. Тело класса состоит из блока различных инструкций. Тело должно иметь отступ (как и любые вложенные конструкции в языке Python).

Схематично класс можно представить следующим образом:

```
1.  class ИМЯКЛАССА:
2.        ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
3.        ….
4.        def ИМЯМЕТОДА(self, ...):
5.              self.ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
6.              …
7.        ….
```

Данная схема не является полной. Например, в заголовке после имени класса могут быть указаны суперклассы (в скобках), а методы могут быть более сложными.

Следует помнить, что методы в классах — это те же функции, за одним небольшим исключением. Они принимают один обязательный параметр — **self** (с англ. можно перевести как "собственная личность"). Он нужен для связи с конкретным объектом.

Атрибуты класса — это имена переменных вне функций и имена функций. Эти атрибуты наследуются всеми объектами, созданными на основе данного класса. Атрибуты обеспечивают свойства и поведение объекта. Объекты могут иметь атрибуты, которые создаются в теле метода, если данный метод будет вызван для конкретного объекта.

Создание объектов:

Объекты создаются так:

```
ПЕРЕМЕННАЯ = ИМЯКЛАССА()
```

Здесь скобки обязательны! После такой инструкции в программе появляется объект, доступ к которому можно получить по имени переменной, связанной с ним. При создании объект получает атрибуты его класса, т. е. объекты обладают характеристиками, определенными в их классах.

Количество объектов, которые можно создать на основе того или иного класса, не ограничено.

Объекты одного класса имеют схожий набор атрибутов, а вот значения атрибутов могут быть разными. Другими словами, объекты одного класса похожи, но индивидуально различимы. Чтобы понять это, можно сравнить отношения объектов одного класса в программировании со следующем высказыванием: "Все млекопитающие принадлежат одному классу и обычно имеют по два глаза, однако у каждого животного (объекта) глаза имеют свои особенности".

Self:

Можно сказать, что методы класса — это небольшие программки, предназначенные для работы с объектами. Методы могут создавать новые свойства (данные) объекта, изменять существующие, выполнять другие действия над объектами.

Методу необходимо "знать", данные какого объекта ему предстоит обрабатывать. Для этого ему в качестве первого (а иногда и единственного) аргумента передается имя переменной, связанной с объектом (можно сказать, передается сам объект). Чтобы в описании класса указать передаваемый в дальнейшем объект, используется параметр **self**. (Посмотрите на схему класса вверху.)

С другой стороны, вызов метода для конкретного объекта в основном блоке программы выглядит следующим образом:

```
ОБЪЕКТ.ИМЯМЕТОДА(…)
```

Здесь под словом ОБЪЕКТ имеется в виду переменная, связанная с ним. Это выражение преобразуется в классе, к которому относится объект, в

```
ИМЯМЕТОДА(ОБЪЕКТ, …)
```

Т. е. конкретный объект подставляется вместо параметра self.

Первая ОО-программа:

Попробуем на основе имеющихся уже знаний написать небольшую ОО-программу. Допустим, это будет класс с одним атрибутом вне метода и одним методом, который выводит с небольшим изменением значение этого атрибута на экран:

```
1.  class First:
2.       color = "red"
3.       def out(self):
4.            print (self.color + "!")
```

Теперь создадим пару объектов данного класса:

```
1.  obj1 = First()
2.  obj2 = First()
```

Оба этих объекта (obj1 и obj2) имею два одинаковых атрибута: color (в виде свойства) и printer (в виде метода). Это легко проверить:

```
1.  print (obj1.color)
2.  print (obj2.color)
3.  obj1.out()
4.  obj2.out()
```

В результате выполнения данного скрипта получается вывод двух надписей red и двух red!. Первые две надписи red – это результат применения встроенной функции print по отношению к свойствам объектов. Вторые две надписи red! - результат применения метода out к объектам.

Усложняем программу:

В предыдущей программе оба созданных объекта абсолютно одинаковы. Класс, на основе которого они созданы слишком прост и не предполагает того, что объекты могут иметь различные значения свойств. Исправим это.

Пусть теперь в классе с помощью атрибутов вне функции устанавливаются два свойства объектов: красный цвет и круглая форма. А методы могут менять эти свойства в зависимости от пожеланий тех, кто создает объекты.

```
1.  class Second:
2.       color = "red"
3.       form = "circle"
4.       def changecolor(self,newcolor):
5.            self.color = newcolor
6.       def changeform(self,newform):
7.            self.form = newform
8.
9.  obj1 = Second()
10. obj2 = Second()
11.
12. print (obj1.color, obj1.form) # вывод на экран "red circle"
13. print (obj2.color, obj2.form) # вывод на экран "red circle"
14.
15. obj1.changecolor("green") # изменение цвета первого объекта
16. obj2.changecolor("blue")  # изменение цвет второго объекта
17. obj2.changeform("oval")   # изменение формы второго объекта
18.
19. print (obj1.color, obj1.form) # вывод на экран "green circle"
20. print (obj2.color, obj2.form) # вывод на экран "blue oval"
```

В данной программе по умолчанию любой созданный объект имеет красный цвет и круглую форму. Однако в дальнейшем с помощью методов данного класса можно поменять и цвет и форму любого объекта. В результате объекты перестают быть одинаковыми (красными и круглыми), хотя сохраняют тот же набор свойств (цвет и форму).

Как же происходят изменения? Дело в том, что методы помимо параметра self, могут иметь и другие параметры, в которых передаются данные для обработки их этим методом. Так, в примере выше, метод changecolor имеет дополнительный параметр newcolor, с помощью которого, в метод можно передать данные о желаемом цвете фигуры. Далее метод меняет цвет с помощью соответствующих инструкций.

Практическая работа:

1. Напишите два скрипта представленных выше. Посмотрите, как они работают. Во второй программу добавьте еще одно свойство и один метод, позволяющий его менять. Создайте третий объект и измените все его свойства.

2. Напишите программу в стиле ООП, удовлетворяющую следующим условиям: в программе должны быть два класса и два объекта, принадлежащих разным классам; один объект с помощью метода своего класса должен так или иначе обрабатывать данные другого объекта: obj1.МЕТОД(obj2.СВОЙСТВО).

Конструктор класса — метод __init__. Урок 3

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Большинство классов имеют специальный метод, который автоматически при создании объекта создает ему атрибуты. Т.е. вызывать данный метод не нужно, т.к. он сам запускается при вызове класса. (Вызов класса происходит, когда создается объект.) Такой метод называется **конструктором класса** и в языке программирования Python носит имя **__init__**. (В начале и конце по два знака подчеркивания.)

Первым параметром, как и у любого другого метода, у **__init__** является **self**, на место которого подставляется объект в момент его создания. Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса.

Рассмотрим два класса: в одном будет использоваться конструктор, а в другом нет. Требуется создать два атрибута объекта.

```
1.  class YesInit:
2.      def __init__(self,one,two):
3.      self.fname = one
4.      self.sname = two
5.
6.  obj1 = YesInit("Peter","Ok")
7.
8.
9.  print (obj1.fname, obj1.sname)
10.
11. class NoInit:
12.     def names(self,one,two):
13.     self.fname = one
14.     self.sname = two
15.
16. obj1 = YesInit()
```

```
17.  obj1.names("Peter","Ok")
18.
19.  print (obj1.fname, obj1.sname)
```

Вывод интерпретатора в обоих случаях:

```
1.   Peter Ok
```

В обоих программах у объекта появляются два атрибута: fname и sname. Однако в первом случае они инициализируются при создании объекта и должны передаваться в скобках при вызове класса. Если какие-то атрибуты должны присутствовать у объектов класса обязательно, то использование метода **__init__** - идеальный вариант. Во второй программе (без использования конструктора) атрибуты создаются путем вызова метода names после создания объекта. В данном случае вызов метода names необязателен, поэтому объекты могут существовать без атрибутов fname и sname.

Обычно метод **__init__** предполагает передачу аргументов при создании объектов, однако аргумент может не быть передан. Например, если в примере выше создать объект так: obj1 = YesInit(), т.е. не передать классу аргументы, то произойдет ошибка. Чтобы избежать подобных ситуаций, можно в методе **__init__** присваивать параметрам значения по умолчанию. Если при вызове класса были заданы аргументы для данных параметров, то хорошо — они и будут использоваться, если нет — еще лучше — в теле метода будут использованы значения по умолчанию. Пример:

```
1.   class YesInit:
2.        def __init__(self,one="noname",two="nametoo"):
3.             self.fname = one
4.             self.sname = two
5.
6.   obj1 = YesInit("Sasha","Tu")
7.   obj2 = YesInit()
8.   obj3 = YesInit("Spartak")
9.   obj4 = YesInit(two="Harry")
10.
11.  print (obj1.fname, obj1.sname)
12.  print (obj2.fname, obj2.sname)
13.  print (obj3.fname, obj3.sname)
14.  print (obj4.fname, obj4.sname)
```

Вывод интерпретатора:

```
1.   Sasha Tu
2.   noname nametoo
3.   Spartak nametoo
4.   noname Harry
```

В данном случае, второй объект создается без передачи аргументов, поэтому в методе **__init__** используются значения по умолчанию ("noname" и "nonametoo"). При создании третьего и четвертого объектов передаются по одному аргументу. Если указывается значение не первого аргумента, то следует явно указать имя параметра (четвертый объект).

Метод **__init__** может содержать параметры как без значений по умолчанию, так и со значениями по умолчанию. В таком случае, параметры, аргументы которых должны быть обязательно указаны при создании объектов, указываются первыми, а параметры со значениями по умолчанию — после. Например, ниже вторая программа с ошибкой:

```
1.  class fruits:
2.       def __init__(self,w,n=0):
3.            self.what = w
4.            self.numbers = n
5.
6.  f1 = fruits("apple",150)
7.  f2 = fruits("pineapple")
8.
9.  print (f1.what,f1.numbers)
10. print (f2.what,f2.numbers)
11.
12. class fruits:
13.      def __init__(self,n=0,w): #ERROR
14.           self.what = w
15.           self.numbers = n
16.
17. f1 = fruits(150,"apple")
18. f2 = fruits("pineapple")
19.
20. print (f1.what,f1.numbers)
21. print (f2.what,f2.numbers)
```

Напишем более существенную программу с использованием конструктора. Допустим это будет класс, значение начальных атрибутов (из метода **__init__**) которого зависит от переданных аргументов при создании объектов. Далее эти свойства объектов, созданных на основе данного класса, можно менять с помощью обычных методов.

```
1.  class Building:
2.       def __init__(self,w,c,n=0):
3.            self.what = w
4.            self.color = c
5.            self.numbers = n
6.            self.mwhere(n)
7.
8.       def mwhere(self,n):
9.            if n <= 0:
10.                self.where = "отсутствуют"
```

```
            elif 0 < n < 100:
                self.where = "малый склад"
            else:
                self.where = "основной склад"

    def plus(self,p):
        self.numbers = self.numbers + p
        self.mwhere(self.numbers)
    def minus(self,m):
        self.numbers = self.numbers - m
        self.mwhere(self.numbers)

m1 = Building("доски", "белые",50)
m2 = Building("доски", "коричневые", 300)
m3 = Building("кирпичи","белые")

print (m1.what,m1.color,m1.where)
print (m2.what,m2.color,m2.where)
print (m3.what,m3.color,m3.where)

m1.plus(500)
print (m1.numbers, m1.where)
```

В данном примере значение атрибута where объекта зависит от значения атрибута numbers.

Практическая работа:

1. Спишите представленные выше скрипт с классом Building. Запустите программу, объясните как она работает. В какой момент создается атрибут where объектов? Зачем потребовалось конструкцию if-elif-else вынести в отдельную функцию, а не оставить ее в методе __init__?
2. Самостоятельно придумайте класс, содержащий конструктор. Создайте на его основе несколько объектов.

Наследование в ООП на Python. Урок 4

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Одной из важнейших особенностей ООП является возможность наследования объектами атрибутов классов, а также наследование одними классами атрибутов других классов. На самом деле с наследованием мы уже сталкивались, когда создавали любой объект в Python: объекты наследуют атрибуты класса, хотя могут иметь и индивидуальные.

```
1.  class Things:
2.       def __init__(self,n,t):
3.            self.namething = n
4.            self.total = t
5.
6.  th1 = Things("table", 5)
7.  th2 = Things("computer", 7)
8.
9.  print (th1.namething,th1.total) # Вывод: table 5
10. print (th2.namething,th2.total) # Вывод: computer 7
11.
12. th1.color = "green" # новое свойство объекта th1
13.
14. print (th1.color) # Вывод: green
15. print (th2.color) # ОШИБКА: у объекта th2 нет свойства color!
```

Здесь оба объекта имеют свойства namething и total, однако только у первого объекта есть свойство color. Все просто: атрибуты класса наследуются объектами, созданными на его основе; однако атрибуты конкретного объекта не зависят от атрибутов других объектов и представляют собственное пространство имен объекта. Последнее позволяет объектам одного класса иметь различные значения атрибутов, а если потребуется и различный набор атрибутов.

Задание. *Спишите код, выполните его с помощью интерпретатора Python. Как можно исправить код, чтобы не было ошибки? Исправьте.*

На самом деле, наследование более широкое понятие, чем просто взаимосвязь между классами и объектами. Один класс может быть подклассом другого, дополняя его. Пояснить это можно проведя аналогию с реальным миром. Например, все столы имеют общие характерные черты («класс»), при этом они имеют разное назначение («подклассы»), хотя продолжают наследовать общие черты. В результате того, что есть такой механизм как наследование можно избежать избыточность кода, просто описав общие свойства и методы в надклассах.

По поводу терминологии. Классы, атрибуты которых наследуются другими классами, могут называть как надклассами так и суперклассами. Классы, которые наследуют атрибуты других классов, часто называют подклассами.

Класс, являющийся надклассом по отношению к одному классу, сам может быть подклассом по отношению к другому. Другими словами, может существовать целая цепочка наследования.

При обращении к атрибуту объекта (obj.prop) сначала просматривается на наличие этого атрибута сам объект, затем его класс, на основе которого он создан. Если в классе не будет найден атрибут, то его поиск продолжится в суперклассе, к которому относится класс.

Суперклассы класса указываются в скобках в заголовке инструкции class.

Рассмотрим такой пример:

```
1.  class Table:
2.      def __init__(self,l,w,h):
3.          self.long = l
4.          self.width = w
5.          self.height = h
6.      def outing(self):
7.          print (self.long,self.width,self.height)
8.
9.  class Kitchen(Table):
10.     def howplaces(self,n):
11.         if n < 2:
12.             print ("It is not kitchen table")
13.         else:
14.             self.places = n
15.     def outplases(self):
16.         print (self.places)
17.
18. t_room1 = Kitchen(2,1,0.5)
19. t_room1.outing()
20. t_room1.howplaces(5)
21. t_room1.outplases()
22.
23. t_2 = Table(1,3,0.7)
24. t_2.outing()
```

```
25.  t_2.howplaces(8) # ОШИБКА
```

Здесь создается два класса: Table и Kitchen. Второй является подклассом первого и наследует все его атрибуты (методы __init__ и outing). Далее создаются два объекта: t_room1 и t_2. Первый объект принадлежит к классу Kitchen и наследует атрибуты этого класса и его суперкласса. Второй объект принадлежит классу Table; к классу Kitchen он никакого отношения не имеет и поэтому не может обращаться к методам howplaces и outplases. В данном примере также можно увидеть, что объекты можно создавать как на основе классов так и суперклассов.

Задание. *Расширьте программу, представленную выше, создав второй подкласс класса Table (например, Worker), содержащий пару методов, отличающихся от методов класса Kitchen().*

Класс может иметь не один, а несколько суперклассов, которые перечисляются друг за другом в скобках в строке заголовка. Такое наследование называется множественным. Потребность во множественном наследовании возникает в случае, если объекты класса предполагают использование свойств и методов различных суперклассов.

Практическая работа

Напишите программу, где класс «геометрические фигуры» (figure) содержит свойство color с изначальным значением white и метод для изменения цвета фигуры, а его подклассы «овал» (oval) и «квадрат» (square) содержат методы __init__ для задания начальных размеров объектов при их создании.

Полиморфизм и переопределение методов в ООП на Python. Урок 5

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Полиморфизм

Парадигма объектно-ориентированного программирования помимо наследования включает еще одну важную особенность — полиморфизм. Слово «полиморфизм» можно перевести как «много форм». В ОО программировании этим термином обозначают возможность использования одного и того же имени операции или метода к объектам разных классов, при этом действия, совершаемые с объектами, могут существенно различаться. Поэтому можно сказать, что у одного и того же слова много форм. Например, два разных класса могут содержать метод total, однако инструкции в методах могут предусматривать совершенно разные операции: так в классе T1 – это прибавление 10 к аргументу, а в T2 – подсчет длины строки символов. В зависимости от того, к объекту какого класса применяется метод total, выполняются те или иные инструкции.

```
1.  class T1:
2.       n=10
3.       def total(self,N):
4.            self.total = int(self.n) + int(N)
5.
6.  class T2:
7.       def total(self,s):
8.            self.total = len(str(s))
9.
10. t1 = T1()
11. t2 = T2()
12. t1.total(45)
13. t2.total(45)
14. print (t1.total) # Вывод: 55
15. print (t2.total) # Вывод: 2
```

Задание. *Напишите программу, запрашивающую у пользователя ввод числа. Если число принадлежит диапазону от -100 до 100, то создается объект одного класса, во всех остальных случаях создается объект другого класса. В обоих классах должен быть метод-конструктор __init__, который в первом классе возводит число в квадрат, а во-втором - умножает на два.*

Ответ:

```
class One:
     def __init__(self,a):
          self.a = a ** 2

class Two:
     def __init__(self,a):
          self.a = a * 2

a = input ("введите число ")
a = int(a)
if -100 < a < 100:
     obj = One(a)
else:
     obj = Two(a)

print (obj.a)
```

Переопределение методов

Использование полиморфизма при наследовании классов позволяет переопределять методы суперклассов их подклассами. Например, может возникнуть ситуация, когда все подклассы реализуют определенный метод из суперкласса, и лишь один подкласс должен иметь его другую реализацию. В таком случае метод переопределяется в подклассе. Пример:

```
class Base:
     def __init__(self,n):
          self.numb = n
     def out(self):
          print (self.numb)

class One(Base):
     def multi(self,m):
          self.numb *= m

class Two(Base):
     def inlist(self):
          self.inlist = list(str(self.numb))
     def out(self):
          i = 0
```

```
            while i < len(self.inlist):
                print (self.inlist[i])
                i += 1

obj1 = One(45)
obj2 = Two('abc')

obj1.multi(2)
obj1.out() # Вывод числа 90

obj2.inlist()
obj2.out() # Вывод в столбик букв a, b, c
```

В данном случае объект obj1 использует метод out из суперкласса Base, а obj2 – из своего класса Two. Атрибуты ищутся «снизу вверх»: сначала в классах, затем суперклассах. Поскольку для obj2 атрибут out уже был найден в классе Two, то из класса Base он не используется. Другими словами, класс Two переопределят атрибут суперкласса Base.

Расширение методов

При ООП может возникнуть ситуация, когда метод суперкласса в принципе подходит для реализации того или иного действия с объектами класса, однако требует небольших изменений. В таком случае можно использовать так называемое расширение метода, когда из тела метода класса вызывается метод суперкласса и дописываются дополнительные инструкции. В примере ниже в методе класса Subclass вызывается метод другого класса (в данном случае его суперкласса; однако может вызываться метод, не принадлежащий собственному суперклассу):

```
class Base:
    def __init__(self,N):
        self.numb = N
    def out(self):
        self.numb /= 2
        print (self.numb)

class Subclass(Base):
    def out(self):
        print ("\n----")
        Base.out(self)
        print ("----\n")

i = 0
while i < 10:
    if 4 < i < 7:
        obj = Subclass(i)
    else:
        obj = Base(i)
    i += 1
```

```
21.       obj.out()
```

Вывод

Полиморфизм в объектно-ориентированном программировании дает возможность реализовывать так называемые единые интерфейсы для объектов различных классов. Имеется ввиду, что если есть методы с одинаковыми названиями (или операции, обозначаемая одинаковыми знаками, как будет показано в уроке №7) для всех объектов, то это позволяет писать более очевидный исходный код. Например, разные классы могут предусматривать различный способ вывода той или иной информации объектов. Однако единое название для всех объектов метода «вывода» позволит не запутать программу, сделать ее более очевидной.

Переопределение методов в подклассах (а также их расширение) позволяет специализировать ранее написанный исходный код, не меняя его в суперклассах, где обычно требуется оставить код в неизменном виде для других подклассов.

Практическая работа

Напишите небольшую объектно-ориентированную программку, демонстрирующую такие свойства ООП как наследование и полиморфизм.

Композиционный подход в объектно-ориентированном программировании. Урок 7

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Еще одной особенностью объектно-ориентированного программирования является возможность реализовывать так называемый композиционный подход. Заключается он в следующем: есть класс-контейнер, который включает в себя вызовы других классов. В результате получается, что создавая объект класса-контейнера, мы одновременно создаем и объекты включенных в него классов.

Чтобы понять зачем нужна композиция в программировании, можно как всегда провести аналогию с реальным миром. Так подавляющее большинство природных, биологических и технических объектов состоят из других более простых частей, по своей сути, также являющихся объектами. Например, человек состоит из различный органов (сердце, кожа и др.), компьютер — из различного "железа" (процессор, ОЗУ, диск и т.д.).

Следует понимать, что "композиция" и "наследование" - достаточно разные свойства реальных и виртуальных систем. Наследование предполагает принадлежность к какой-то общности (похожесть), а композиция — формирование целого из частей.

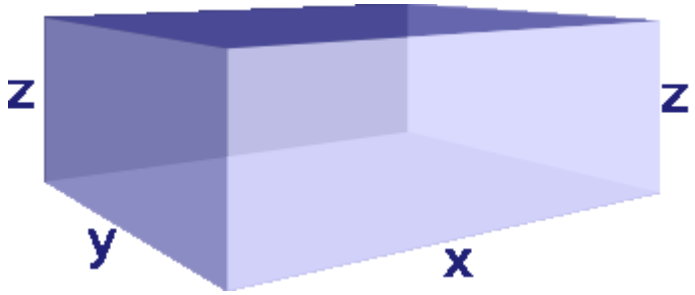
Еще раз: при создании объекта, принадлежащего классу-контейнеру, автоматически создаются объекты-части, из которых он как бы состоит. Свойства и методы объектов частей определяются в их классах. Программисты могут создавать целые коллекции встраиваемых классов.

Рассмотрим использование композиции при программировании на Python с помощью конкретного примера.

Описание задачи

Допустим, нам требуется написать программу, которая вычисляет площадь обоев для оклеивания комнаты определенных пользователем размеров. При этом необходимо учитывать, что окна, двери, пол и потолок оклеивать не надо.

Для начала решим данную задачу логически. 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.



Обои клеятся только на стены, следовательно площади верхнего и нижнего прямоугольников нам не нужны. Из рисунка можно заключить, что площадь одного прямоугольника равна x * z, второго – y * z. Противоположные прямоугольники равны, значит общая площадь четырех прямоугольников будет равна S = 2xz + 2yz = 2z(x+y). Потом из этой площади надо будет вычесть общую площадь дверей и окон. Двери и окна — это прямоугольники (как вычислить их площадь должно быть понятно).

Создание классов-частей

Теперь приступим к созданию программы. В соответствие с изучаемой темой написать ее надо используя объектно-ориентированную парадигму программирования, да еще и применяя "композиционный подход" (насколько он здесь уместен не обсуждается :b).

Можно заметить, что фактически у нас есть три типа объектов - это объекты-окна, объекты-двери и объекты-комнаты. Получается три класса. Окна и двери являются частями помещения, а значит могут создаваться внутри класса «комнаты». Кроме того, для данной задачи существенное значение имеют только два свойства: длина и ширина. Поэтому классы «окна» и «двери» можно объединить в один. Понятно, что если для задачи были бы важны другие свойства (например, толщина стекла, материал), то возможно следовало бы создать два класса.

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

Здесь при вызове класса Win_Door будет автоматически создан атрибут square объекта, являющийся ссылкой на значение площади объекта.

Создание класса-контейнера

Можно по-разному реализовать класс-контейнер. Есть подозрение, что многое зависит от задачи, решаемой программистом, его мастерства и вкуса. Классы-части можно вызывать в методе __init__, тем самым объекты-части будут автоматически создаваться при создании объекта-контейнера. Однако в данной задаче мы пойдем другим путем: окна и двери будут создаваться специальным для этих целей методом (будем считать, что так интересней). Также класс должен содержать метод для вычисления площади требуемых обоев (wallpapers). В конце можно добавить метод, в котором реализован вывод тех или иных данных.

```
1.  class Room:
2.      def __init__(self,x,y,z):
3.          self.square = 2 * z * (x + y)
4.      def win_door(self, d,e, f,g, m=1,n=1):
5.          self.window = Win_Door(d,e)
6.          self.door = Win_Door(f,g)
7.          self.numb_w = m
8.          self.numb_d = n
9.      def wallpapers(self):
10.         self.wallpapers = self.square - \
11.             self.window.square * self.numb_w \
12.             - self.door.square * self.numb_d
13.     def printer(self):
14.         print ("Площадь стен комнаты равна "\
15.         ,str(self.square)," кв.м")
16.         print ("Оклеиваемая площадь равна: ", \
17.             str(self.wallpapers), " кв.м")
```

В методе __init__ создается атрибут square объекта представляющий собой площадь стен комнаты. Метод принимает три аргумента: длину, ширину и высоту помещения.

В методе win_door создаются два объекта: window и door, а также атрибуты numb_w и numb_d (в последних будут содержаться значения о количестве окон и дверей). Если при вызове данного метода в программе не будет указано количество окон и дверей, то по умолчанию будут подставлены значения равные 1.

Метод wallpapers вычисляет площадь_требуемых_обоев = площадь_комнаты — площадь_окна * количество_окон — площадь_двери * количество_дверей. В коде данная строка разбита на несколько строчек с помощью знака \ (так делают, если строка очень длинная). Также обратите внимание, как происходит обращение к свойствам square объектов-частей: указывается объект класса Room (в классе его заменяет self), далее объект-часть, и наконец, сам атрибут (свойство) объекта-части.

Метод printer просто выводит данные.

Создание объектов

После того, как классы созданы, посмотрим как это все работает.

1. Создаем объект класса Room:

```
1.  labor34 = Room(5,4,2)
```

2. Создаем в помещении labor34 окна и двери:

```
1.  labor34.win_door(1.5,1.5, 2,1, 2)
```

Обратите внимание, что количество дверей не указано, а значит их будет ровно 1.

3. Вычисляем метры обоев:

```
1.  labor34.wallpapers()
```

4. Просим вывести, что получилось:

```
1.  labor34.printer()
```

В результате работы метода printer интерпретатор должен выдать что-то вроде этого:

```
1.  Площадь комнаты равна 36 кв.м
2.  Оклеиваемая площадь равна: 29.5 кв.м
```

Может показаться, что в программе всего один реальный объект — labor34. Однако это не так. Там есть еще объекты labor34.window и labor34.door. Чтобы в этом убедиться достаточно обратиться к их свойствам.

```
1.  print(labor34.window.square)
2.  print(labor34.door.square)
```

Практическая работа

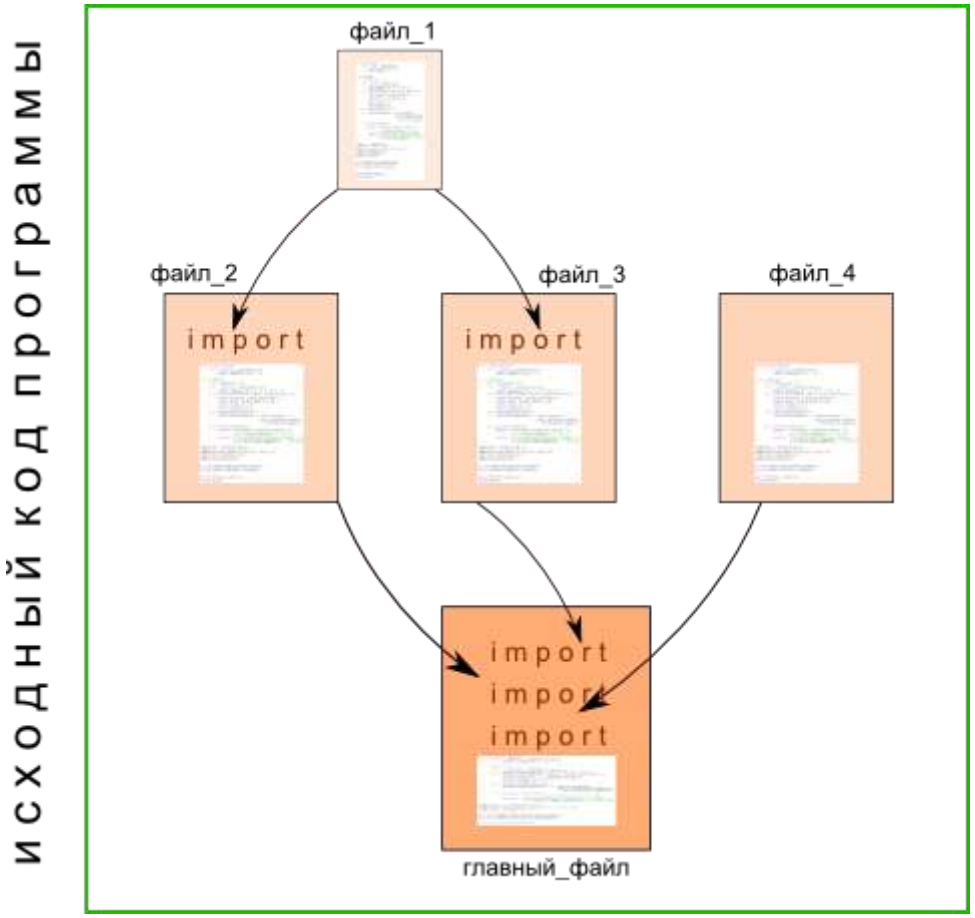
Попробуйте самостоятельно придумать задачу, для решения которой можно использовать композиционный подход. Напишите программу на Python.

Модули и их импорт. Урок 8

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

На прошлом уроке нами была написана "серьезная" программа, которую могут использовать другие программисты. Однако как? Просто копировать код и вставлять в свои скрипты? Или есть более экономный способ (в смысле уменьшения объема кода и удобства его использования)?

При создании крупных программ оказался выгодным так называемый модульный принцип организации, когда есть основной файл с частью кода программы, к которому подсоединяется (в который импортируется) содержимое других файлов. Когда исходный код основного файла транслируется в машинный код, то импортируемые файлы также выполняются как и код основного файла.



Такой способ организации программы позволяет изолировать часто используемый код в файл-модуль, а затем импортировать его в другие файлы без копирования кода. Но это далеко не единственное преимущество модульного принципа организации программы.

Так как же импортировать содержимое одного файла в другой в языке программирования Python? Существует два основных способа: инструкция **import** и инструкция **from**. Первая инструкция запускает (интерпретирует) файл-модуль полностью, при этом для доступа к переменным (атрибутам) модуля из основного файла следует указывать имя модуля впереди требуемого атрибута: module.attribute (так называемая, точечная нотация). Инструкция **from** передает интерпретатору лишь указанные имена из файла-модуля, однако при доступе к этим переменным не надо указывать имя модуля. Первый способ хорош, если предстоит пользоваться содержимым почти всего модуля, второй — если будут востребованы одна-две функции или класс из модуля. В примере данного урока мы воспользуемся инструкцией **import**.

Импорт в языке программирования Python осуществляется следующим образом: после слова **import** пишется имя импортируемого модуля. Модуль и файл в Python понятия почти не различимые. Файлы с кодом на языке Python обычно имеют расширение .py, однако в инструкции

import расширение не указывается. Например, если мы имеем файл-модуль scale.py, то импортировать его в другой файл следует так: import scale.

Где должен располагаться модуль? В принципе, где угодно, т.к. можно "вручную" настроить интерпретатор так, что он будет искать там, где пожелает программист. Однако, если ничего не настраивать, то интерпретатор Python найдет файлы, если их расположить например, в каталоге, куда установлен Python или в том же каталоге, где и файл, в который осуществляется импорт. Этим последним вариантом мы и воспользуемся.

Итак, у нас есть файл с кодом, позволяющим вычислять оклеиваемую площадь помещения (урок №7). Пусть он называется rooms.py. Кроме того, удалим из него "код тестирования" …

```
1.  labor34 = Room(5,4,2)
2.  labor34.win_door(1.5,1.5, 2,1, 2)
3.  labor34.wallpapers()
4.  labor34.printer()
5.
6.  print(labor34.window.square)
7.  print(labor34.door.square)
```

… и предположим, что классы этого модуля будут использоваться в другом (основном) файле. Допустим, этот основной файл предоставляет интерфейс пользователю для ввода данных и получения результата. Основной файл должен быть сохранен в том же каталоге, что и файл rooms.py.

Первым делом, импортируем содержимое файла rooms.py

```
1.  import rooms
```

Далее организуем запрос данных у пользователя, одновременно преобразовав данные в целочисленный тип (функция int):

```
1.  print ("Введите размеры помещения (в метрах) ...")
2.  l = int(input ("длина: "))
3.  w = int(input ("ширина: "))
4.  h = int(input ("высота: "))
5.
6.  print ("Введите данные об оконных проемах (в метрах) ...")
7.  h_w = int(input ("высота: "))
8.  w_w = int(input ("ширина: "))
9.  m = int(input ("количество: "))
10.
11. print ("Введите данные о дверных проемах (в метрах) ...")
12. h_d = int(input ("высота: "))
13. w_d = int(input ("ширина: "))
```

```
14.  n = int(input ("количество: "))
```

Теперь создаем объект класса Room. Описание класса находится в модуле rooms, который был импортирован инструкцией **import** (а не from), поэтому, чтобы получить доступ к классу Room и его атрибутам, следует при создании объекта указать модуль, в котором он находится:

```
1.   uroom = rooms.Room(l,w,h)
```

А теперь можно пользоваться атрибутами класса из модуля сколько влезет:

```
1.   uroom.win_door(h_w, w_w, h_d, w_d, m,n)
2.   uroom.wallpapers()
3.   uroom.printer()
```

Практическая работа:

1. Создайте скрипт, импортирующий модуль с классом Room и использующий его (как показано в данном уроке).
2. Допишите предыдущую программу, расширив ее возможности: можно по-желанию получить дополнительные сведения (площадь окна и двери).
3. Переделайте программу таким образом, чтобы она не запрашивала у пользователя данные, а предлагала выбор из пяти готовых решений: на экран выводятся характеристики различных помещений, — пользователю остается только выбрать.

Строки документации исходного кода на Python. Урок 9

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих
Версия интерпретатора python: 3.-.-

__doc__ - строки документации

На прошлом уроке мы рассмотрели случай импорта программного кода одного файла в другой. Таким образом, например, один программист может использовать разработки другого. К тому же, существует множество стандартных модулей и библиотек, входящих в установочный пакет интерпретатора python, а также огромное количество модулей сторонних разработчиков. Понятно, что чтобы использовать чужую разработку надо знать, что она делает. Для этого вовсе необязательно анализировать исходный код, поскольку есть инструменты для его документирования, которыми приличные программисты должны регулярно пользоваться. Когда кому-то потребуется узнать, что делает тот или иной скрипт, достаточно будет выполнить специальную команду, которая выдаст описание модуля.

Документировать исходный код на языке программирования Python можно по разному. Иногда бывает достаточно простых комментариев. Еще один способ — это создание **строк документации**. Они представляют собой текст, заключенный в кавычки (тройные, тройные одинарные, обычные). Такой текст может располагаться после заголовков классов, функций (методов), а также в начале файла (модуля). Пример:

```
1.  """ Строка документации в начале файла.
2.       А это ее продолжение с новой строки. """
3.  class … :
4.       ''' Это строка документирования класса.
5.            Причем она заключена в тройные одинарные кавычки...
6.            или одинарные тройные 0_o '''
7.       .....
8.       .....
9.       def … :
10.           """ Это документирование модуля.
11.                Пишите коменты на инглише. """
```

Для получения доступа к такой документации предусмотрена специальный метод **__doc__**. Когда интерпретатор Python обрабатывает исходный код класса или функции и обнаруживает там строки документирования, то делает их значением атрибута **__doc__** данного объекта.

Чтобы посмотреть как это выглядит, рассмотрим реальный пример. В языке программирования Python строки, списки, числа являются по-сути такими же классами как и пользовательские (создаваемые для специализированного проекта). Данные классы автоматически загружаются (интерпретируются), когда запускается программа-интерпретатор python, поэтому не требуется дополнительный импорт модулей, содержащих их описание. Однако сути это не меняет, и получить доступ к их строкам документации можно также как и к пользовательскому классу (функции, модулю), а именно **имяОбъекта.__doc__**. Пример:

```
>>> print (str.__doc__)
str(string[, encoding[, errors]]) -> str

Create a new string object from the given encoded string.
encoding defaults to the current default string encoding.
errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.
```

Таким образом, мы получили данные о классе **str** (описывает атрибуты строк). Однако мы не получили никаких сведений о методах данного класса. Да и вообще: какие методы есть у строк? Тут может помочь встроенная функция **dir**, которая выводит список переменных (атрибутов) переданного ей аргумента.

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Допустим, нас интересует функция метод **find**, но мы точно не знаем, делает ли он то, что нам нужно. Выяснить это можно так:

```
>>> print (str.find.__doc__)
S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found, such that sub is contained within s[start:end].
Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.
```

Становятся известны следующие подробности. Оказывается функция find класса str просто возвращает первый индекс подстроки переданной ей в качестве аргумента, если та встречается в строке. По желанию можно указать откуда и докуда искать в исходной строке. Не в тему, но чтобы было понятно:

```
>>> a = "hello new worlds" # создаем строку
>>> a.find("new") # ищем индекс первого элемента подстроки
6
>>> a.find("l",7) # ищем индекс символа, начиная с 8-го элемента
13
```

Форматирование строк документирования

Обратите внимание, когда мы вызывали метод __doc__ для объектов, то использовали встроенную функцию print. Зачем? Ведь в интерактивном режиме вывод работает и без print. Однако если при вызове метода __doc__ не использовать функцию print, то вывод не отформатируется, а будет выглядеть как в исходном коде:

```
>>> str.__doc__
"str(string[, encoding[, errors]]) -> str\n\nCreate a new string object from the given encoded string.\nencoding defaults to the current default string encoding.\nerrors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'."
```

Обратите внимание на сочетание символов **\n**. Оно обозначает переход на новую строку. Вообще оформление строк документации должно соответствовать определенному стандарту.

Функция help

Еще более интересный способ получить информацию о модуле или его частях — это использование встроенной функции **help**. Она выводит текстовый отчет о переданном ей в качестве аргумента объекте. Этот отчет включает не только строки документации, но и структуру запрошенной части кода. т.е. применив функцию **help** по отношению к объектам мы получим еще и информацию о внутренних функциях и классах.

Практическая работа

1. Напишите строки документации к программе, созданной на 7 уроке (вычисление площади оклеивания) . Поместите следующую информацию:
   - Модуль. Для чего предназначена данная программа.
   - Классы. Для чего предназначены, какие аргументы передаются методу __init__.
   - Методы (кроме __init__). Назначение, описание входных и выходных данных.
2. Используя метод __doc__, вызовите в интерактивном режиме строки документации для различных объектов модуля. Предварительно необходимо импортировать модуль (чтобы импортировать его без проблем, поместите файл в каталог установки python).

3. Примените функцию help для данного модуля. Например, если имя модуля rooms, то вызов справки по нему будет выглядеть так: help (rooms).

Пример документированного исходного кода

```
"It is the program for definition of quantity of wall-paper"

class Win_Door:
     """
Class Win_Door calculates the area of a rectangular.
Method __init__ accepts two parameters: length and width.
     """
     def __init__(self,x,y):
          self.square = x * y

class Room:
     """
Class Room is intended for definition of the pasted over area of a room.
Method __init__ accepts three arguments (length, width and height).
     """
     def __init__(self,x,y,z):
          self.square = 2 * z * (x + y)
     def win_door(self, d,e, f,g, m=1,n=1):
          """
The first pair parameters - the sizes of a window,
the second pair - the sizes of a door,
the fifth and sixth parameters - quantity of windows and doors accordingly
          """
          self.window = Win_Door(d,e)
          self.door = Win_Door(f,g)
          self.numb_w = m
          self.numb_d = n
     def wallpapers(self):
          """
This method calculates the pasted over area
          """
          self.wallpapers = self.square - \
               self.window.square * self.numb_w \
               - self.door.square * self.numb_d
     def printer(self):
          """
Displays the information
          """
          print ("Площадь стен комнаты равна "\
          ,str(self.square)," кв.м")
          print ("Оклеиваемая площадь равна: ", \
               str(self.wallpapers), " кв.м")
```

Перегрузка операторов в ООП. Урок 10

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

До этого мы говорили только о классах и объектах, которые создает программист (пользовательские классы). Однако Python настолько объектно-ориентированный язык, что в нем любые строка, число, список и др. являются по сути объектами, принадлежащими встроенным классам (типам данных): строкам, числам и др. Т. е. типы данных — это встроенные классы, а любые данные — это объекты.

Объектно-ориентированное программирование подразумевает не только наличие объектов, но и их взаимодействие между собой. Поэтому важно, чтобы к объектам разных классов можно было применить одну и туже операцию (например, сложение).

Для пользовательских классов предусмотрены специальные методы, позволяющие объектам данных классов участвовать в таких привычных операциях как сложение, вычитание, умножение, деление (+ - * /), а также во многих других. Другими словами, смысл (то, что он делает) знака + (или любого другого оператора) зависит от того, к каким объектам он применяется. Это называется **перегрузкой операторов**. В классах перегруженные операторы описываются с помощью специальных зарезервированных методов, которые в начале и в конце имеют по два знака подчеркивания. В уроке рассматриваются лишь некоторые из них. Кроме того, ранее был уже рассмотрен один такой метод — конструктор **__init__**, который автоматически вызывается при создании объектов класса.

Рассмотрим пример перегрузки операторов.

```
1.  class Newclass:
2.      def __init__(self, base):
3.          self.base = base
4.      def __add__(self, a):
5.          self.base = self.base + a
6.      def __str__(self):
7.          return "%s !!! " % self.base
8.
9.  a = Newclass(10)
10. a + 20
11. print (a)
12.
13. b = Newclass("yes")
14. b + "terday"
15. print (b)
```

```
16.
17.  c = Newclass([2,6,3])
18.  c + [7, 1]
19.  print (c)
```

В данном примере используется два метода (исключая **__init__**) перегрузки операторов: **__add__** и **__str__**. Метод **__add__** вызывается в том случае, когда объект данного класса участвует в операции сложения (для чисел), конкатенации (для строк) и объединения (для списков). Метод **__str__** вызывается, когда объект передается в качестве аргумента встроенной функции **print** (на самом деле не только ей) и представляет данные в виде строки.

Результат работы скрипта представленного выше будет таким:

```
1.   30 !!!
2.   yesterday !!!
3.   [2, 6, 3, 7, 1] !!!
```

Задание. Спешите пример, посмотрите как он работает. Дополните класс методами **__mul__** (вызывается при использовании объекта в операциях умножения) и **__sub__** (вычитание). Вызовите данные методы с помощью соответствующих операций с объектами. Для каких объектов невозможно использовать метод **__sub__**?

__call__ - перегрузка вызова функции

Метод **__call__** автоматически вызывается, когда к объекту обращаются как к функции. Например, здесь во второй строке произойдет вызов метода **__call__** некогоКласса:

объект = некийКласс()
объект([возможные аргументы])

Другими словами, метод **__call__** позволяет объектам вести себя как функции.

Пример:

```
1.   class Changeable:
2.        def __init__(self, color):
3.             self.color = color
4.        def __call__(self, newcolor):
5.             self.color = newcolor
6.        def __str__(self):
7.             return "%s" % self.color
8.
9.   canvas = Changeable("green")
```

```
frame = Changeable("blue")

canvas("red")
frame("yellow")

print (canvas, frame)
```

В этом примере с помощью конструктора класса при создании объектов устанавливается их цвет. Если требуется его поменять, то достаточно обратиться к объекту как к функции и в качестве аргумента передать новый цвет. Такой обращение автоматически вызовет метод **__call__** (который, в данном случае, изменит атрибут color объекта).

Задание. *Создайте класс с методом **__call__**, принимающим два параметра и производящим над ними те или иные математические операции. Создайте несколько объектов класса и, затем, обратитесь к ним как к функциям.*

Рассмотренные в этом уроке методы перегрузки операторов лишь малая часть из существующих. Фактически все, что можно делать со встроенными типами (числами, словарями и др.), можно реализовать и для пользовательских типов (классов). Можно сказать, что перегрузка операторов обеспечивает единый интерфейс для встроенных и пользовательских типов (классов). Так, в первом примере можно видеть как "складываются" объект-число и объект класса Newclass.

Особенности объектно-ориентированного программирования. Урок 11

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

Общее представление об объектно-ориентированном программировании было рассмотрено на первом уроке. Здесь будет обобщение ранее рассмотренного материала и формулирование принципов, лежащих в основе ООП.

Идеи (принципы) объектно-ориентированного программирования

Во многих учебниках выделяют такие основные идеи ООП как наследование, инкапсуляция и полиморфизм. Заключаются они примерно в следующем:

1. **наследование.** Возможность выделять общие свойства и методы классов в один класс верхнего уровня (родительский). Классы, имеющие общего родителя, различаются между собой за счет включения в них различных дополнительных свойств и методов.
2. **инкапсуляция.** Свойства и методы класса делятся на доступные из вне (опубликованные) и недоступные (защищенные). Защищенные атрибуты нельзя изменить, находясь вне класса. Опубликованные же атрибуты также называют интерфейсом объекта, т. к. с их помощью с объектом можно взаимодействовать. По идеи, инкапсуляция призвана обеспечить надежность программы, т.к. изменить существенные для существования объекта атрибуты становится невозможно.
3. **полиморфизм.** Полиморфизм подразумевает замещение атрибутов, описанных ранее в других классах: имя атрибута остается прежним, а реализация уже другой. Полиморфизм позволяет специализировать (адаптировать) классы, оставляя при этом единый интерфейс взаимодействия.

Преимущества ООП

В связи со своими особенностями объектно-ориентированное программирование имеет ряд преимуществ перед структурным (и др.) программированием. Выделим некоторые из них:

1. Использование одного и того же программного кода с разными данными. Классы позволяют создавать множество объектов, каждый из которых имеет собственные значения атрибутов. Нет потребности вводить множество переменных, т.к объекты получают в свое распоряжение индивидуальные так называемые пространства имен. Пространство имен конкретного объекта формируется на основе класса, от которого он был создан, а также от всех родительских классов данного класса. Объект можно представить как некую упаковку данных.
2. Наследование и полиморфизм позволяют не писать новый код, а настраивать уже существующий, за счет добавления и переопределения атрибутов. Это ведет к сокращению объема исходного кода.

Особенность ООП

ООП позволяет сократить время на написание исходного кода, однако ООП всегда предполагает большую роль предварительного анализа предметной области, предварительного проектирования. От правильности решений на этом предварительном этапе зависит куда больше, чем от непосредственного написания исходного кода.

Особенности ООП в Python

По сравнению с другими распространенными языками программирования у Python можно выделить следующие особенности, связанные с объектно-ориентированным программированием:

1. Любое данное (значение) — это объект. Число, строка, список, массив и др. — все является объектом. Бываю объекты встроенных классов (как те, что перечисленные в предыдущем предложении), а бывают объекты пользовательских классов (тех, что создает программист). Для единого механизма взаимодействия предусмотрены методы перегрузки операторов.

2. Класс — это тоже объект с собственным пространством имен. Это нигде не было указано в данном цикле уроков. Однако это так. Поэтому правильнее было употреблять вместо слова «объект», слово «экземпляр». И говорить «экземпляр объекта», подразумевая под этим созданный на основе класса именно объект, и «экземпляр класса», имея ввиду сам класс как объект.
3. Инкапсуляции в Python не уделяется особого внимания. В других языках программирования обычно нельзя получить напрямую доступ к свойству, описанному в классе. Для его изменения может быть предусмотрен специальный метод. В Python же это легко сделать, просто обратившись к свойству класса из вне. Несмотря на это в Python все-таки предусмотрены специальные способы ограничения доступа к переменным в классе.

Пример объектно-ориентированной программирования на языке Python. Урок 12

Методическая разработка урока
Элективный курс: Введение в объектно-ориентированное программирование на Python
Уровень: Программирование для начинающих

В конце данного курса придумаем и напишем небольшую программу, используя объектно-ориентированную парадигму программирования.

Как уже отмечалось в предыдущем уроке в ООП очень важен этап предварительного проектирования. Вообще можно выделить следующие этапы создании ОО-программы:

1. Формулирование задачи.
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут созданы объекты, а также установление между ними иерархических связей.
4. Определение существенных свойств и методов для задач, которые будут решать объекты на основе проектируемых классов.
5. Создание классов, описание их свойств и атрибутов.
6. Создание объектов.
7. Решение задачи путем взаимодействия объектов.

Первый этап:

Допустим нам надо написать программу по «сценарию» описанному в уроке 1: про занятие, где ученики должны получить некий объем знаний.

Второй этап:

Какие объекты понадобятся? Очевидно, это ученики, учитель, информация, а также возможно вспомогательные предметы такие как проектор и др.

Третий этап:

Классы: учитель, ученик, информация. Учитель и ученик во многом похожи: по идеи оба - люди. Значит классы учитель и ученик могут принадлежать одному суперклассу - человек. Однако не все так просто. Если существенные признаки для решения данной задачи не имеют ничего общего, то выделить что-то в суперкласс просто невозможно.

Четвертый этап:

Выделим важное, чем должны обладать объекты (классы) для решения задачи «увеличить знания».

Ученик должен уметь воспринимать информацию, и превращать ее в знания.

Учитель, по крайней мере, должен уметь выбирать и транслировать информацию.

Информация должна содержать определенный набор строк символов. Должна быть предусмотрена возможность извлечения ее частей.

Пятый этап:

Класс «Информация»:

```
1.  class Information:
2.      def __init__(self,info):
3.          self.info = info
4.      def extract(self,i):
5.          self.current = self.info[i]
6.          return "%s" % self.current
```

Объекты данного класса при создании должны содержать ту или иную информацию (содержание урока), допустим в виде списка. В классе также предусмотрен метод extract, позволяющий извлекать какую-то часть информации и возвращать ее в основную программу.

Класс «Учитель»:

```
1.  class Teacher:
2.      def into(self,phrase):
3.          self.phrase = phrase
```

```
4.        def out(self):
5.             return "%s" % self.phrase
```

Объектам типа «Учитель» в нашей программе позволено лишь вспоминать фразу и громко транслировать.

Класс «Ученик»:

```
1.   class Pupil:
2.        def __init__(self):
3.             self.know = []
4.        def take(self, i):
5.             self.know.append(i)
```

Объекты класса Pupil уже при своем создании обязаны иметь атрибут know, куда будут помещаться знания. Также предусмотрен метод take, обеспечивающий приемку информации.

Создание объектов:

Допустим в программе будет по одному объекту «Информатика» и «Учитель» и пару объектов «Ученик».

```
1.   inform = Information(["> (больше)","< (меньше)","== (равно)", "!= (не равно)"])
2.   t = Teacher()
3.   p1 = Pupil()
4.   p2 = Pupil()
```

Решение задачи с помощью взаимодействия объектов:

Какая в данном случае задача? Научить учеников чему-нибудь. Приступим.

```
1.   t.into(inform.extract(2))
2.   p1.take(t.out())
3.   print ("1-ый ученик пока еще знает только ", p1.know)
4.
5.   t.into(inform.extract(0))
6.   p1.take(t.out())
7.   p2.take(t.out())
8.   print ("1-ый ученик знает, что ", p1.know)
9.   print ("2-ой ученик знает, что ", p2.know)
```

Учитель берет с помощью метода extract объекта inform часть информации. Ученики, используя свой метод take имеют возможность получить информацию, воспроизводимую учителем (метод out объекта t).

В результате работы этой программы атрибут know учеников изменяется (если конечно те использовали метод take).

Практическая работа

1. Напишите программу рассмотренную в этом уроке. Посмотрите как она работает.
2. Создайте еще пару учеников и еще один объект класса Information. Научите новых учеников чему-нибудь.
3. Может ли в данной программе ученик освоить информацию минуя учителя. Если «да», то реализуйте в программе «самостоятельную работу» ученика.