

Курс включает 15 уроков:

	<i>Стр.</i>	<i>Стр. решения практической работы</i>
1 Что такое объектно-ориентированное программирование	4	65
2 Создание классов и объектов	10	67
3 Конструктор класса – метод <code>__init__()</code>	15	72
4 Наследование	18	74
5 Полиморфизм	24	79
6 Инкапсуляция	27	81
7 Композиция	33	82
8 Перегрузка операторов	36	84
9 Статические методы	41	86
10 Итераторы	45	88
11 Генераторы	50	89
12 Модули и пакеты	53	90
13 Документирование кода	57	91
14 Пример объектно-ориентированной программы на Python	60	92
15 Итоги курса "Объектно-ориентированное программирование на Python"	62	-

Урок 1. Что такое объектно-ориентированное программирование

Циклы, ветвления и функции – все это элементы структурного программирования. Его возможностей вполне хватает для написания небольших, простых программ и сценариев. Однако крупные проекты часто реализуют, используя парадигму объектно-ориентированного программирования (ООП). Что оно из себя представляет и какие преимущества дает?

Истоки ООП берут начало с 60-х годов XX века. Однако окончательное формирование основополагающих принципов и популяризацию идеи следует отнести к 80-м годам. Большой вклад внес Алан Кей.

Следует отметить, что хоть и многие, но не все современные языки поддерживают объектно-ориентированное программирование. Так язык C, обычно используемый в системном программировании (создание операционных систем, драйверов, утилит), не поддерживает ООП.

В языке Python ООП играет ключевую роль. Даже программируя в рамках структурной парадигмы, вы все равно пользуетесь объектами и классами, пусть даже встроенными в язык, а не созданными лично вами.

Итак, что же такое объектно-ориентированное программирование? Судя по названию, ключевую роль здесь играют некие объекты, на которые ориентируется весь процесс программирования.

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, взаимодействующих между собой и вследствие этого изменяющимися. Эта привычная для взгляда человека картина мира была перенесена в программирование.



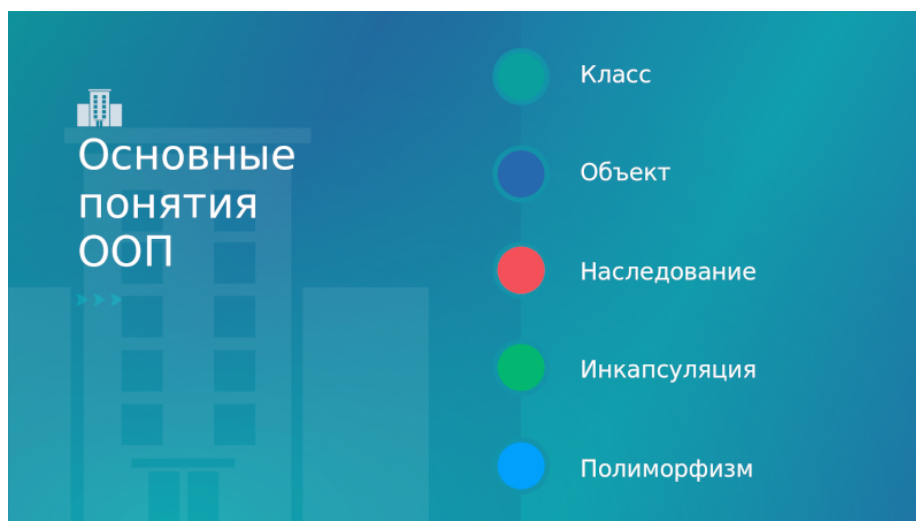
Она потребовала более высокого уровня абстракции от того, как вычислительная машина хранит и обрабатывает данные, потребовала от программистов умения конструировать своего рода "виртуальные миры", распределять между собой задачи. Однако дала возможность более легкой и продуктивной разработки больших программ.

Допустим, команда программистов занимается разработкой игры. Программу-игру можно представить как систему, состоящую из цифровых героев и среды их обитания, включающей множество предметов. Каждый воин, оружие, дерево, дом – это цифровой объект, в котором "упакованы" его свойства и действия, с помощью которых он может изменять свои свойства и свойства других объектов.

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

Ключевую разницу между программой, написанной с структурным стилем, и объектно-ориентированной можно выразить так. В первом случае, на первый план выходит логика, понимание последовательности выполнения действий для достижения поставленной цели. Во-втором – важнее представить программу как систему объектов, взаимодействие которых способно решить ту или иную задачу.

Основными понятиями, используемыми в ООП, являются класс, объект, наследование, инкапсуляция и полиморфизм. В языке Python класс равносителен понятию тип данных.



Класс и объект

Что такое **класс** или тип? Проведем аналогию с реальным миром. Если мы возьмем конкретный стол, то это **объект**, но не класс. А вот общее представление о столах, их назначении – это класс. Ему принадлежат все реальные объекты столов, какими бы они ни были. Класс столов дает общую характеристику всем столам в мире, он их обобщает.



То же самое с целыми числами в Python. Тип `int` – это класс целых чисел. Числа 5, 100134, -10 и т. д. – это конкретные объекты этого класса.

В языке программирования Python объекты принято называть также **экземплярами**. Это связано с тем, что в нем все классы сами являются объектами класса `type`. Точно также как все модули являются объектами класса `module`.

```
>>> type(list), type(int)
(<class 'type'>, <class 'type'>)
>>> import math
>>> type(math)
<class 'module'>
```

Поэтому во избежании путаницы объекты, созданные на основе обычных классов, называют экземплярами. В этом курсе мы чаще будем такие объекты называть объектами, так как данная терминология более универсальная и используется в других языках.

Наследование, инкапсуляция, полиморфизм

Следующее по важности понятие объектно-ориентированного программирования – **наследование**. Вернемся к столам. Пусть есть класс столов, описывающий общие свойства всех столов. Однако можно разделить все столы на письменные, обеденные и журнальные и для каждой группы создать свой класс, который будет наследником общего класса, но также вносить ряд своих особенностей. Таким образом, общий класс будет родительским, а классы групп – дочерними, производными.

Дочерние классы наследуют особенности родительских, однако дополняют или в определенной степени модифицируют их характеристики. Когда мы создаем конкретный экземпляр стола, то должны выбрать, 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.

Основное (но не единственное) преимущество, которое дает концепция наследования в программировании, — это вынос одинакового кода из разных классов в один родительский класс. Другими словами, наследование позволяет сводить на нет повторение кода в разных частях программы.


Инкапсуляция в ООП понимается двояко. Во многих языках этот термин обозначает сокрытие данных, то есть невозможность напрямую получить доступ к внутренней структуре объекта, так как это небезопасно. Например, наполнить желудок едой можно напрямую, положив еду в желудок. Но это опасно. Поэтому прямой доступ к желудку закрыт. Чтобы наполнить его едой, надо совершить ритуал, через элемент интерфейса под названием рот.

В Python нет такой инкапсуляции, хотя она является одним из стандартов ООП. В Python можно получить доступ к любому атрибуту объекта и изменить его. Однако в Питоне есть механизм, позволяющий имитировать сокрытие данных, если это так уж необходимо.

Отсутствие сокрытия данных в Python делает программирование на нем проще, но привносит ряд особенностей, связанных с пространствами имен.


Второй смысл инкапсуляции — объединение описания свойств объектов и их поведения в единое целое, то есть в класс. Инкапсуляция в этом смысле вытекает из самой идеи объектно-ориентированного программирования и, соответственно, имеется во всех ОО-языках.

Полиморфизм можно перевести как множество форм. В ООП под полиморфизмом понимается следующее. Объекты разных классов, с разной внутренней реализацией, то есть программным кодом, могут иметь "одинаковые" методы. На самом деле у методов совпадают только имена, а вложенный в них код (то, что они делают) различен. Вот и получается, что у одного имени как бы множество форм.




ПОЛИМОРФИЗМ


Разные классы, одинаковое название метода, разный код в методе




test



test
! ! ! ! !
! ! ! ! !




test
@@ @@ @@
@@
@



test

#####



test
\$\$\$\$\$\$
\$\$\$\$\$\$
\$\$\$\$\$\$

>>>

Например, для чисел есть операция сложения, обозначаемая знаком $+$. Однако мы можем определить класс, объекты которого также будут поддерживать операцию, обозначаемую этим знаком. Но это вовсе не значит, что объекты должны быть числами, и будет получаться какая-то сумма. Операция $+$ для объектов нашего класса может значить что-то иное. Но интерфейс, в данном случае это знак $+$, у чисел и нашего класса будет одинаков. Полиморфность же проявляется во внутренней реализации и результате операции.

Вы уже сталкивались с полиморфизмом операции $+$. Для чисел она обозначает сложение, а для строк – конкатенацию. Внутренняя реализация кода для этой операции у чисел отличается от реализации таковой для строк.

Полиморфизм полезен не только тем, что дает возможность объектам пользовательских классов участвовать в стандартных операциях. Если у объектов разных классов есть одноименный метод, то коллекция таких разнородных объектов может быть обработана в одном цикле.

Пример объектно-ориентированной программы на Python

Проиллюстрируем рассмотренные концепции на примере небольшой программы. На данном этапе полностью понимать ее код не требуется, о создании классов, объектов и работе методов будет рассказано в следующих уроках.

В программе определены три класса. Класс *A* является родительским по отношению к *B* и *C*. Последние наследуют от *A* поле-переменную *field1* и метод *make_str*, который потом переопределяют.

От каждого класса мы создаем по одному объекту и присваиваем их переменным *a*, *b*, *c*. Метод *make_str* выводит на экран значения полей объекта. У объектов разных классов разный набор полей, поэтому код метода отличается.

```
class A:
    field1 = 1

    def make_str(self):
        print(self.field1)

class B(A):
    field2 = 2

    def make_str(self):
        print(self.field1, self.field2)

class C(A):
    field3 = 3

    def make_str(self):
        print(self.field1, self.field3)

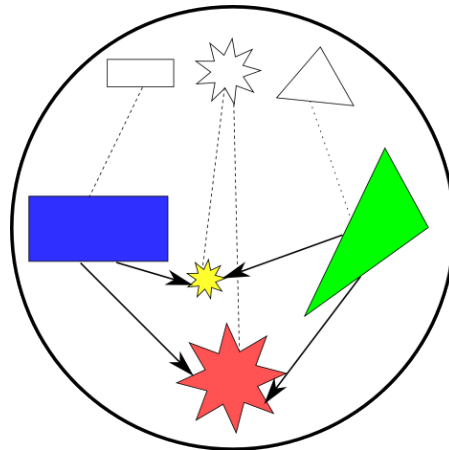
a = A()
b = B()
c = C()
```

```
for i in (a, b, c):  
    i.make_str()
```

Результат выполнения программы:

1
1 2
1 3

Практическая работа



Рассмотрите схему. Подумайте над следующими вопросами:

1. Какие фигуры на ней вы бы назвали классами, а какие – объектами? Что обозначают пунктирные линии?
2. Может ли объект принадлежать множеству классов? Может ли у класса быть множество объектов?
3. Звезды скорее обладают разными свойствами или разным поведением? Могут ли свойства оказывать влияние на поведение?
4. Что могли бы обозначать стрелки?

Урок 2. Создание классов и объектов

В языке программирования Python классы создаются с помощью инструкции `class`, за которой следует произвольное имя класса, после которого ставится двоеточие, далее с новой строки и с отступом реализуется тело класса:

```
class ИмяКласса:
    код_тела_класса
```

Если класс является дочерним, то родительские классы перечисляются в круглых скобках после имени класса.

Объект создается путем вызова класса по его имени. При этом после имени класса обязательно ставятся скобки:

```
ИмяКласса()
```

То есть класс вызывается подобно функции. Однако в случае вызова класса происходит не выполнение его тела, как это происходило бы при вызове функции, а создается объект.

Поскольку в программном коде важно не потерять ссылку на только что созданный объект, то обычно его связывают с переменной. Поэтому создание объекта чаще всего выглядит так:

```
имя_переменной = ИмяКласса()
```

В последствии к объекту обращаются через связанную с ним переменную.

Пример "пустого" класса и двух созданных на его основе объектов:

```
>>> class A:
...     pass
...
>>> a = A()
>>> b = A()
```

Класс как пространство имен

С точки зрения пространства имен класс можно представить подобным модулю. Также как в модуле в классе могут быть свои переменные со значениями и функции. Также как в модуле у класса есть собственное пространство имен, доступ к которому возможен через имя класса:

```
>>> class B:
...     n = 5
...     def adder(v):
...         return v + B.n
...
>>> B.n
5
>>> B.adder(4)
9
```

Однако в случае классов используется особая терминология. Пусть имена, определенные в классе, называются атрибутами этого класса. В примере имена `n` и `adder` – это атрибуты класса

B. Атрибуты-переменные называют **полями** или свойствами (в других языках понятия "поле" и "свойство" не совсем одно и то же). Полем является *n*. Атрибуты-функции называются **методами**. Методом в классе *B* является *adder*. Количество свойств и методов в классе может быть любым.

Класс как шаблон для создания объектов

На самом деле классы – не модули. Они своего рода шаблоны, от которых создаются объекты-экземпляры. Такие объекты наследуют от класса его атрибуты. Вернемся к нашему классу *B* и создадим на его основе два объекта:

```
>>> class B:
...     n = 5
...     def adder(v):
...         return v + B.n
...
>>> a = B()
>>> b = B()
```

У объектов, связанных с переменными *a* и *b*, нет собственного поля *n*. Однако они наследуют его от своего класса:

```
>>> a.n
5
>>> a.n is B.n
True
```

То есть поля *a.n* и *B.n* – это одно и то же поле, к которому можно обращаться и через имя *a*, и через имя *b*, и через имя класса. Поле одно, ссылок на него три.

Однако что произойдет в момент присваивания этому полю значения через какой-нибудь объект-экземпляр?

```
>>> a.n = 10
>>> a.n
10
>>> b.n
5
>>> B.n
5
```

В этот момент у экземпляра появляется собственный атрибут *n*, который перекроет (переопределит) родительский, то есть тот, который достался от класса.

```
>>> a.n is B.n
False
>>> b.n is B.n
True
```

При этом присвоение через *B.n* отразится только на *b* и *B*, но не на *a*:

```
>>> B.n = 100
>>> B.n, b.n, a.n
(100, 100, 10)
```

Иная ситуация нас ожидает с атрибутом *adder*. При создании объекта от класса функция *adder* не наследуется как есть, а как бы превращается для объекта в одноименный метод:

```
>>> B.adder is b.adder
False
>>> type(B.adder)
<class 'function'>
>>> type(b.adder)
<class 'method'>
```

Через имя класса мы вызываем функцию *adder*:

```
>>> B.adder(33)
133
```

Через имя объекта вызываем метод *adder*:

```
>>> b.adder(33)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: adder() takes 1 positional argument but 2 were given
```

В сообщении об ошибке говорится, что *adder* принимает только один аргумент, а было передано два. Откуда появился второй, если в скобках было указано только одно число?

Дело в том, что в отличии от функции **в метод первым аргументом всегда передается объект, к которому применяется этот метод**. То есть выражение `b.adder(33)` как бы преобразовывается в `adder(b, 33)`. Сам же `b.adder` как объект типа **method** хранит сведения, с каким классом он связан и какому объекту-экземпляру принадлежит:

```
>>> b.adder
<bound method B.adder of <__main__.B object at 0x7fcbf1ab9b80>>
```

В нашем случае, чтобы вызывать *adder* через объекты-экземпляры, класс можно переписать так:

```
>>> class B:
...     n = 5
...     def adder(obj, v):
...         return v + obj.n
...
>>> b = B()
>>> b.adder(33)
38
```

В коде выше при вызове метода *adder* переменной-параметру *obj* присваивается объект, связанный с переменной, к которой применяется данный метод. В данном случае это объект, связанный с *b*. Если *adder* будет вызван на другой объект, то уже он присвоится *obj*:

```
>>> a = B()
>>> a.n = 9
>>> a.adder(3)
12
```

В Python принято переменную-параметр метода, которая связывается с экземпляром своего класса, называть именем **self**. Таким образом, более корректный код будет таким:

```
>>> class B:
...     n = 5
...     def adder(self, v):
...         return v + self.n
```

Можем ли мы все также вызывать *adder* как функцию, через имя класса? Вполне. Только теперь в функцию надо передавать два аргумента:

```
>>> B.adder(B, 200)
205
>>> B.adder(a, 200)
209
```

Здесь первым аргументом в функцию передается объект, у которого есть поле *n* лишь только потому, что далее к этому полю обращаются через выражение `self.n`.

Однако если атрибут определен так, что предполагается его работа в качестве метода, а не функции, то через класс его уже не вызывают (нет смысла, логика программы этого не подразумевает).

С другой стороны, в ООП есть понятие "статический метод". По сути это функция, которая может вызываться и через класс, и через объект, и которой первым аргументом не подставляется объект, на который она вызывается. В Python статический метод можно создать посредством использования специального декоратора.

Атрибут `__dict__`

В Python у объектов есть встроенные специальные атрибуты. Мы их не определяем, но они есть. Одним из таких атрибутов объекта является свойство `__dict__`. Его значением является словарь, в котором ключи – это имена свойств экземпляра, а значения – текущие значения свойств.

```
>>> class B:
...     n = 5
...     def adder(self, v):
...         return v + self.n
...
>>> w = B()
>>> w.__dict__
{}
>>> w.n = 8
>>> w.__dict__
{'n': 8}
```

В примере у экземпляра класса *B* сначала нет собственных атрибутов. Свойство *n* и метод *adder* – это атрибуты объекта-класса, а не объекта-экземпляра, созданного от этого класса. Лишь когда мы выполняем присваивание новому полю *n* экземпляра, у него появляется собственное свойство, что мы наблюдаем через словарь `__dict__`.

В следующем уроке мы увидим, что свойства экземпляра обычно не назначаются за пределами класса. Это происходит в методах классах путем присваивания через `self`. Например, `self.n = 10`.

Атрибут `__dict__` используется для просмотра всех текущих свойств объекта. С его помощью можно удалять, добавлять свойства, а также изменять их значения.

```
>>> w.__dict__['m'] = 100
>>> w.__dict__
{'n': 8, 'm': 100}
>>> w.m
100
```

Практическая работа

Напишите программу по следующему описанию. Есть класс "Воин". От него создаются два экземпляра-юнита. Каждому устанавливается здоровье в 100 очков. В случайном порядке они бьют друг друга. Тот, кто бьет, здоровья не теряет. У того, кого бьют, оно уменьшается на 20 очков от одного удара. После каждого удара надо выводить сообщение, какой юнит атаковал, и сколько у противника осталось здоровья. Как только у кого-то заканчивается ресурс здоровья, программа завершается сообщением о том, кто одержал победу.

Урок 3. Конструктор класса – метод `__init__()`

В объектно-ориентированном программировании конструктором класса называют метод, который автоматически вызывается при создании объектов. Его также можно назвать конструктором объектов класса. Имя такого метода обычно регламентируется синтаксисом конкретного языка программирования. Так в Java имя конструктора класса совпадает с именем самого класса. В Python же роль конструктора играет метод `__init__()`.

В Python наличие пар знаков подчеркивания спереди и сзади в имени метода говорит о том, что он принадлежит к группе методов перегрузки операторов. Если подобные методы определены в классе, то объекты могут участвовать в таких операциях как сложение, вычитание, вызываться как функции и др.

При этом методы перегрузки операторов не надо вызывать по имени. Вызовом для них является сам факт участия объекта в определенной операции. В случае конструктора класса – это операция создания объекта. Так как объект создается в момент вызова класса по имени, то в этот момент вызывается метод `__init__()`.

Необходимость конструкторов связана с тем, что нередко объекты должны иметь собственные свойства сразу. Пусть имеется класс `Person`, объекты которого обязательно должны иметь имя и фамилию. Если класс будет описан подобным образом

```
class Person:
    def set_name(self, n, s):
        self.name = n
        self.surname = s
```

то создание объекта возможно без полей. Для установки имени и фамилии метод `set_name()` нужно вызывать отдельно:

```
>>> from test import Person
>>> p1 = Person()
>>> p1.set_name("Bill", "Ross")
>>> p1.name, p1.surname
('Bill', 'Ross')
```

В свою очередь, конструктор класса не позволит создать объект без обязательных полей:

```
class Person:
    def __init__(self, n, s):
        self.name = n
        self.surname = s
```

```
p1 = Person("Sam", "Baker")
print(p1.name, p1.surname)
```

Здесь при вызове класса в круглых скобках передаются значения, которые будут присвоены параметрам метода `__init__()`. Первый его параметр – `self` – ссылка на сам только что созданный объект.

Теперь, если мы попытаемся создать объект, не передав ничего в конструктор, то будет возбуждено исключение, и объект не будет создан:

```
>>> p1 = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'n' and 's'
```

Однако бывает, что надо допустить создание объекта, даже если никакие данные в конструктор не передаются. В таком случае параметрам конструктора класса задаются значения по умолчанию:

```
class Rectangle:
    def __init__(self, w=0.5, h=1):
        self.width = w
        self.height = h

    def square(self):
        return self.width * self.height

rec1 = Rectangle(5, 2)
rec2 = Rectangle()
rec3 = Rectangle(3)
rec4 = Rectangle(h=4)
print(rec1.square())
print(rec2.square())
print(rec3.square())
print(rec4.square())
```

Вывод:

```
10
0.5
3
2.0
```

Если класс вызывается без значений в скобках, то для параметров будут использованы их значения по умолчанию. Однако поля `width` и `height` будут у всех объектов.

Кроме того, конструктору вовсе не обязательно принимать какие-либо параметры, не считая `self`. Значения полям могут назначаться как угодно. Также не обязательно, чтобы в конструкторе происходила установка атрибутов объекта. Там может быть, например, код, который порождает создание объектов других классов.

В других языка программирования, например в Java, классы могут содержать несколько конструкторов, которые между собой отличаются количеством параметром, а также, возможно, их типом. При создании объекта срабатывает тот конструктор, количество и типы параметров которого совпали с количеством и типами переданных в конструктор аргументов.

В Python создать несколько методов `__init__()` в классе можно, однако "рабочим" останется только последний. Он переопределит ранее определенные. Поэтому в Python в классах используется только один конструктор, а изменчивость количества передаваемых аргументов настраивается через назначение значений по-умолчанию.

Практическая работа. Конструктор и деструктор

Помимо конструктора объектов в языках программирования есть обратный ему метод – деструктор. Он вызывается, когда объект не создается, а уничтожается.

В языке программирования Python объект уничтожается, когда исчезают все связанные с ним переменные или им присваивается другое значение, в результате чего связь со старым объектом теряется. Удалить переменную можно с помощью команды языка `del`.

В классах Python функцию деструктора выполняет метод `__del__()`.

Напишите программу по следующему описанию:

1. Есть класс `Person`, конструктор которого принимает три параметра (не учитывая `self`) – имя, фамилию и квалификацию специалиста. Квалификация имеет значение заданное по умолчанию, равное единице.
2. У класса `Person` есть метод, который возвращает строку, включающую в себя всю информацию о сотруднике.
3. Класс `Person` содержит деструктор, который выводит на экран фразу "До свидания, мистер ..." (вместо троеточия должны выводиться имя и фамилия объекта).
4. В основной ветке программы создайте три объекта класса `Person`. Посмотрите информацию о сотрудниках и увольте самое слабое звено.
5. В конце программы добавьте функцию `input()`, чтобы скрипт не завершился сам, пока не будет нажат `Enter`. Иначе вы сразу увидите как удаляются все объекты при завершении работы программы.

В Python деструктор используется редко, так как интерпретатор и без него хорошо убирает "мусор".

Урок 4. Наследование

Наследование – важная составляющая объектно-ориентированного программирования. Так или иначе мы уже сталкивались с ним, ведь объекты наследуют атрибуты своих классов. Однако обычно под наследованием в ООП понимается наличие классов и подклассов. Также их называют супер- или надклассами и классами, а также родительскими и дочерними классами.

Суть наследования здесь схожа с наследованием объектами от классов. Дочерние классы наследуют атрибуты родительских, а также могут переопределять атрибуты и добавлять свои.



Простое наследование методов родительского класса

В качестве примера рассмотрим два класса столов. Класс Table – родительский по отношению к DeskTable (письменные столы). Независимо от своего типа все столы имеют длину, ширину и высоту. Пусть для письменных столов также важна площадь поверхности. Общее вынесем в класс, частное – в подкласс.

Наследственная связь между классами устанавливается через подкласс. При определении дочернего после его имени в скобках указывается родительский.

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

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

```
t1 = Table(1.5, 1.8, 0.75)
t2 = DeskTable(0.8, 0.6, 0.7)
print(t2.square()) # вывод: 0.48
```

В данном случае у класса DeskTable нет своего конструктора, поэтому он наследует его от родителя. При создании объектов передавать аргументы необходимо в обоих случаях. Попытка вызова DeskTable с пустыми скобками приведет к ошибке.

С другой стороны, экземпляры надкласса Table, согласно неким родственным связям, не наследуют метод square своего подкласса.

В этом смысле терминология "родительский и дочерний класс" не совсем верна. Наследование в ООП – это скорее аналог систематизации и классификации наподобие той, что есть в живой природе. Все млекопитающие имеют четырехкамерное сердце, но только носороги – рог.

Полное переопределение метода надкласса

Рассмотрим вариант программы с "цепочкой наследования". Пусть дочерний по отношению к Table класс DeskTable в свою очередь выступит родительским по отношению к ComputerTable (компьютерные столы):

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

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

class ComputerTable(DeskTable):
    def square(self, monitor=0.0):
        return self.width * self.length - monitor

t3 = ComputerTable(0.8, 0.6, 0.7)
print(t3.square(0.3)) # вывод: 0.18
```

Допустим, по задумке разработчиков рабочая поверхность компьютерного стола может вычисляться за вычетом площади, которую занимает монитор. В результате метод square в ComputerTable имеет отличия.

Определив в дочернем классе метод, одноименный методу родительского, мы тем самым **переопределяем** метод родительского класса. При вызове square на экземпляры ComputerTable будет вызываться метод из этого класса, а не из родительского класса DeskTable.

В то же время ComputerTable наследует конструктор класса от своей "бабушки" – класса Table.

Дополнение, оно же расширение, метода

Часто требуется не столько заменить, то есть полностью переопределить, метод родительского класса в дочернем, сколько дополнить, то есть расширить, код метода родительского класса в дочернем. В таких случаях решением является вызов метода надкласса в теле соответствующего метода подкласса. Обычно после этого в теле метода подкласса пишется дополнительный код.

Пусть имеется подкласс кухонных столов, для которых необходимо еще одно поле – количество посадочных мест. Из-за этой детали мы вынуждены в дочернем классе переопределить конструктор родительского.

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

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

```
t4 = KitchenTable(1.5, 2, 0.75, 6)
```

Поскольку существенная часть кода конструктора подкласса является такой же как в надклассе, правильнее будет вызвать метод другого класса, а не дублировать код:

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

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

```
t4 = KitchenTable(1.5, 2, 0.75, 6)
```

Здесь в теле конструктора KitchenTable мы вызываем метод `__init__` через объект-класс Table, а не через объект-экземпляр. Вспомним, что в таких случаях метод вызывается как обычная функция (объект, к которому применяется метод, не передается в качестве первого аргумента). Поэтому в конструктор надкласса мы "вручную" передаем текущий экземпляр (self), записывая его перед остальными аргументами.

У кода выше есть небольшой недостаток. Нам ничего не мешает (при условии совпадения количества параметров) вызвать конструктор другого класса, а не только родительского, указав его имя вместо Table. Кроме того, имя надкласса может измениться, и тогда есть риск неправильных обращений к нему из дочерних классов.

В Python с целью улучшения так называемой обслуживаемости кода можно использовать встроенную в язык функцию `super`. Наиболее распространенным вариантом ее применения является вызов метода родительского класса из метода подкласса:

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

В данном случае аргумент `self` в скобках вызываемого родительского метода указывать явно не требуется.

Параметры со значениями по умолчанию у родительского класса

Рассмотрим случай, когда родительский класс имеет параметры со значениями по умолчанию, а дочерний – нет:

```
class Table:
    def __init__(self, l=1, w=1, h=1):
        self.length = l
        self.width = w
        self.height = h

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

При таком определении классов можно создать экземпляр от `Table` без передачи аргументов в конструктор:

```
t = Table()
```

Можем ли мы создать экземпляр от `KitchenTable`, передав значение только для параметра `p`? Например, вот так:

```
k = KitchenTable(10)
```

Возможно ли, что `p` будет присвоено число 10, а `l`, `w` и `h` получат по единице от родительского класса? Невозможно, будет выброшено исключение по причине несоответствия количества переданных аргументов количеству требуемых конструктором:

```
... k = KitchenTable(10)
TypeError: __init__() missing 3 required
positional arguments: 'l', 'w', and 'h'
```

Когда создается объект от дочернего класса, сначала вызывается его конструктор, если он есть. Интерпретатор еще не знает, что в теле этого конструктора будет вызван конструктор родительского класса. Ведь это не обязательно. Значит, если все параметры дочернего

конструктора не имеют значений по умолчанию, при построении объекта все значения должны передаваться.

Поэтому, если требуется допустить создание объектов от дочернего класса без передачи аргументов, придется назначить значения по умолчанию также в конструкторе дочернего класса.

```
class Table:
    def __init__(self, l=1, w=1, h=1):
        self.length = l
        self.width = w
        self.height = h

class KitchenTable(Table):
    def __init__(self, l=1, w=1, h=0.7, p=4):
        Table.__init__(self, l, w, h)
        self.places = p
```

Параметр `p`, которого нет у родительского класса, мы делаем последним не просто так. Бывает, объекты разных родственных классов создаются или обрабатываются в одном цикле, то есть по одному алгоритму. При этом у них должны быть одинаковые "интерфейсы", то есть одинаковое количество передаваемых в конструктор аргументов.

Поэтому лучше, когда методы родственных классов принимают одинаковое число параметров. А если разное, то у "лишних" должны быть значения по-умолчанию, чтобы при вызове конструктора их можно было бы не передавать. Если такие параметры находятся еще и в конце, передачу аргументов для предстоящих параметров можно выполнять без ключей.

Другой вариант – отказаться от конструктора в дочернем классе, а значение для поля `places` устанавливать отдельным вызовом метода:

```
class Table:
    def __init__(self, l=1, w=1, h=1):
        self.length = l
        self.width = w
        self.height = h

class KitchenTable(Table):
    places = 4

    def set_places(self, p):
        self.places = p
```

Здесь у всех кухонных столов по-умолчанию будет 4 места. Если мы хотим изменить значение поля `places`, можем вызвать метод `set_places()`. Хотя в случае Python можем сделать это напрямую, присвоив полю. При этом у экземпляра появится собственное поле `places`.

```
k = KitchenTable()
k.places = 6
```

Поэтому метод `set_places()` в общем-то не нужен.

В любом случае произвольное количество мест будет устанавливаться не в конструкторе, а отдельно. Если все же требуется указывать места при создании объекта, это можно сделать и в конструкторе родителя:

```
class Table:
    def __init__(self, l=1, w=1, h=1):
        self.length = l
        self.width = w
        self.height = h
        if isinstance(self, KitchenTable):
            p = int(input("Сколько мест: "))
            self.places = p
```

С помощью функции `isinstance()` проверяется, что создаваемый объект имеет тип `KitchenTable`. Если это так, то у него появляется поле `places`.

Мы не используем параметр `p` со значением по умолчанию в заголовке конструктора потому, что, если объектам других родственных классов он не нужен, не происходило путаницы и сложностей с документированием кода.

Практическая работа

Разработайте программу по следующему описанию.

В некой игре-стратегии есть солдаты и герои. У всех есть свойство, содержащее уникальный номер объекта, и свойство, в котором хранится принадлежность команде. У солдат есть метод "иду за героем", который в качестве аргумента принимает объект типа "герой". У героев есть метод увеличения собственного уровня.

В основной ветке программы создается по одному герою для каждой команды. В цикле генерируются объекты-солдаты. Их принадлежность команде определяется случайно. Солдаты разных команд добавляются в разные списки.

Измеряется длина списков солдат противоборствующих команд и выводится на экран. У героя, принадлежащего команде с более длинным списком, увеличивается уровень.

Отправьте одного из солдат первого героя следовать за ним. Выведите на экран идентификационные номера этих двух юнитов.

Урок 5. Полиморфизм

Полиморфизм в объектно-ориентированном программировании – это возможность обработки разных типов данных, т. е. принадлежащих к разным классам, с помощью "одной и той же" функции, или метода. На самом деле одинаковым является только имя метода, его исходный код зависит от класса. Кроме того, результаты работы одноименных методов могут существенно различаться. Поэтому в данном контексте под полиморфизмом понимается множество форм одного и того же слова – имени метода.

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:
    def __init__(self):
        self.n = 10

    def total(self, a):
        return self.n + int(a)

class T2:
    def __init__(self):
        self.string = 'Hi'

    def total(self, a):
        return len(self.string + str(a))

t1 = T1()
t2 = T2()

print(t1.total(35)) # Вывод: 45
print(t2.total(35)) # Вывод: 4
```

В предыдущем уроке мы уже наблюдали полиморфизм между классами, связанными наследованием. 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 связаны 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.

В Python среди прочего полиморфизм находит отражение в методах перегрузки операторов. Два из них мы уже рассмотрели. Это `__init__()` и `__del__()`, которые вызываются при создании объекта и его удалении. 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 `str()`, которую автоматически вызывает функция `print()`.

Если вы создадите объект собственного класса, а потом попытаете вывести его на экран, то получите информацию о классе объекта и его адрес в памяти. Такое поведение функции `str()` по умолчанию по отношению к пользовательским классам запрограммировано на самом верхнем уровне иерархии, где-то в суперклассе, от которого неявно наследуются все остальные.

```
class A:
    def __init__(self, v1, v2):
        self.field1 = v1
        self.field2 = v2
```

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

Вывод:

```
<__main__.A object at 0x7f251ac2f8d0>
<__main__.A object at 0x7f251ac2f8d0>
```

Здесь мы используем переменную `b`, чтобы показать, что функция `print()` вызывает `str()` неявным образом, так как вывод значений обоих переменных одинаков.

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 в свою очередь возвращать функция `str()`, вызываемая функцией `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)
b = str(a)
print(a)
print(b)
```

Вывод:

```
3 4
3 4
```

Какую именно строку возвращает метод `__str__()`, дело десятое. 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:

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

Практическая работа. Метод перегрузки оператора сложения

В качестве практической работы попробуйте самостоятельно перегрузить оператор сложения. Для его перегрузки используется метод `__add__()`. Он вызывается, когда объекты класса, имеющего данный метод, фигурируют в операции сложения, причем с левой стороны. Это значит, что в выражении `a + b` у объекта `a` должен быть метод `__add__()`. Объект `b` может быть чем угодно, но чаще всего он бывает объектом того же класса. Объект `b` будет автоматически передаваться в метод `__add__()` в качестве второго аргумента (первый – `self`).

Отметим, в Python также есть правосторонний метод перегрузки сложения – `__radd__()`.

Согласно полиморфизму ООП, возвращать метод `__add__()` может что угодно. Может вообще ничего не возвращать, а "молча" вносить изменения в какие-то уже существующие объекты. Допустим, в вашей программе метод перегрузки сложения будет возвращать новый объект того же класса.

Урок 6. Инкапсуляция

Под инкапсуляцией в объектно-ориентированном программировании понимается упаковка данных и методов для их обработки вместе, т. е. в классе. В Python инкапсуляция реализуется как на уровне классов, так и объектов. В ряде других языков, например в Java, под инкапсуляцией также понимают сокрытие свойств и методов, в результате чего они становятся приватными. Это значит, что доступ к ним ограничен либо пределами класса, либо модуля.

В Python подобной инкапсуляции нет, хотя существует способ ее имитировать. Перед тем как выяснять, как это делается, надо понять, зачем вообще что-то скрывать.

Дело в том, что классы бывают большими и сложными. В них может быть множество вспомогательных полей и методов, которые не должны использоваться за его пределами. Они просто для этого не предназначены. Они своего рода внутренние шестеренки, обеспечивающие нормальную работу класса.

Кроме того, в других языках программирования хорошей практикой считается сокрытие всех полей объектов, чтобы уберечь их от прямого присвоения значений из основной ветки программы. Их значения можно изменять и получать только через вызовы методов, специально определенных для этих целей.

Например, если надо проверять присваиваемое полю значение на корректность, то делать это каждый раз в основном коде программы будет неправильным. Проверочный код должен быть помещен в метод, который получает данные для присвоения полю. А само поле должно быть закрыто для доступа из вне класса. В этом случае ему невозможно будет присвоить недопустимое значение.

Часто намеренно скрываются поля самого класса, а не его объектов. Например, если класс имеет счетчик своих объектов, то необходимо исключить возможность его случайного изменения из вне. Рассмотрим пример с таким счетчиком на языке Python.

```
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) # выведет 0, хотя остался b
```

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:

```
...
print(B.__count)
AttributeError: type object 'B' has no attribute '__count'
```

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)
```

Таково соглашение. Если в классе есть атрибут с двумя первыми подчеркиваниями, то для доступа извне к имени атрибута добавляется имя класса с одним впереди стоящим подчеркиванием. В результате атрибут как он есть (в данном случае `__count`) оказывается замаскированным. Вне класса такого атрибута просто не существует. Для программиста же наличие двух подчеркиваний перед атрибутом должно сигнализировать, что трогать его вне класса не стоит вообще, даже через `_B__count`, разве что при крайней необходимости.

Хорошо, мы защитили поле от случайных изменений. Но как теперь получить его значение? Сделать это можно с помощью добавления метода:

```
class B:
    __count = 0

    def __init__(self):
        B.__count += 1

    def __del__(self):
```

```
B.__count -= 1

def qty_objects():
    return B.__count

a = B()
b = B()
print(B.qty_objects()) # выведет 2
```

В данном случае метод `qty_object()` не принимает объект (нет `self`а), поэтому вызывать его надо через класс. Хотя правильнее такие методы делать статическими (рассматривается в одном из следующих уроков).

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

```
class DoubleList:
    def __init__(self, l):
        self.double = DoubleList.__make_double(l)

    def __make_double(old):
        new = []
        for i in old:
            new.append(i)
            new.append(i)
        return new

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

Результат:

```
[1, 1, 6, 6, 12, 12]
Traceback (most recent call last):
...
print(DoubleList.__make_double([1, 2]))
AttributeError: type object 'DoubleList' has no attribute '__make_double'
```

В одном из комментариев к предыдущим версиям данного курса был приведен пример, согласно которому скрытые поля при присваивании им становятся открытыми:

```
class Full:
    def __init__(self, field):
        self.__field = field

    def setField(self, field):
        self.__field = field

    def getField(self):
        return self.__field

obj = Full(8)
obj.setField(3)
print(obj.getField())
```

try:

```
print(obj.__field)
except AttributeError:
    print("Нет атрибута __field")

obj.__field = 5
print(obj.__field)
```

Результат выполнения:

```
3
Нет атрибута __field
5
```

На самом деле в данном примере поле экземпляра `__field`, определенное за пределами класса, – это совсем другое поле. Не тот `__field`, который находится в классе и обращаться к которому из вне надо с помощью `_Full__field`. В этом можно убедиться, если вывести на экран содержимое атрибута `__dict__`:

```
...
print(obj.__dict__)
obj.__field = 5
print(obj.__dict__)
print(obj.__field is obj._Full__field)
```

Результат:

```
...
{'_Full__field': 3}
{'_Full__field': 3, '__field': 5}
False
```

Поэтому в коде выше выражение `obj.__field` в блоке `try` приводит к выбросу исключения, так как происходит обращение к еще несуществующему полю, а не потому, что это поле скрыто. Когда же этому полю присваивается значение 5, то у объекта появляется новое поле.

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)
```

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`.

Если объект содержит скрытые поля и к ним происходит обращение из `__setattr__()`, то делать это надо так, как будто обращение происходит не из класса. Следующий код приведет к генерации исключения:

```

class A:
    def __init__(self, x):
        self.__x = x

    def __setattr__(self, attr, value):
        if attr == "__x":
            self.__dict__[attr] = value
        else:
            raise AttributeError

a = A(5)

```

Результат:

```
Traceback (most recent call last):
  File "...2.py", line 12, in <module>
    a = A(5)
  File "...2.py", line 3, in __init__
    self.__x = x
  File "...2.py", line 9, in __setattr__
    raise AttributeError
AttributeError
```

В методе `__setattr__()` параметр `attr` – это имя свойства экземпляра в том виде, в котором оно находится в словаре `__dict__`. Если свойство скрытое, то в `__dict__` оно будет записано через имя класса. Поэтому в данном случае правильный код будет таким:

```
class A:
    def __init__(self, x):
        self.__x = x

    def __setattr__(self, attr, value):
        if attr == "_A__x":
            self.__dict__[attr] = value
        else:
            raise AttributeError
```

Практическая работа

Разработайте класс с "полной инкапсуляцией", доступ к атрибутам которого и изменение данных реализуются через вызовы методов. В объектно-ориентированном программировании принято имена методов для извлечения данных начинать со слова `get` (взять), а имена методов, в которых свойствам присваиваются значения, – со слова `set` (установить). Например, `get_field`, `set_field`.

Урок 7. Композиция

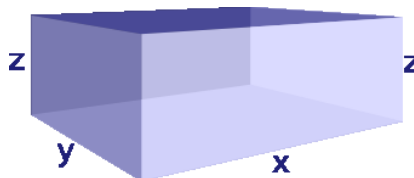
Еще одной особенностью объектно-ориентированного программирования является возможность реализовывать так называемый композиционный подход. Заключается он в том, что есть класс-контейнер, он же агрегатор, который включает в себя вызовы других классов. В результате получается, что при создании объекта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

Рассмотрим на примере реализацию композиции в Python. Пусть, требуется написать программу, которая вычисляет площадь обоев для оклеивания помещения. При этом окна, двери, пол и потолок оклеивать не надо.

Прежде, чем писать программу, займемся объектно-ориентированным проектированием. То есть разберемся, что к чему. 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.



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 — yz . Противоположные прямоугольники равны, значит общая площадь четырех прямоугольников равна $S = 2xz + 2yz = 2z(x+y)$. Потом из этой площади надо будет вычесть общую площадь дверей и окон, поскольку они не оклеиваются.

Можно выделить три типа объектов — окна, двери и комнаты. Получается три класса. Окна и двери являются частями комнаты, поэтому пусть они входят в состав объекта-помещения.

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

Класс "комната" – это класс-контейнер для окон и дверей. Он должен содержать вызовы класса "окно_дверь".

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 add_wd(self, w, h):
        self.wd.append(WinDoor(w, h))

    def work_surface(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.add_wd(1, 1)
r1.add_wd(1, 1)
r1.add_wd(1, 2)
print(r1.work_surface()) # выведет 44.6
```

Практическая работа

Приведенная выше программа имеет ряд недочетов и недоработок. Требуется исправить и доработать, согласно следующему плану.

При вычислении оклеиваемой поверхности мы не "портим" поле `self.square`. В нем так и остается полная площадь стен. Ведь она может понадобиться, если состав списка `wd` изменится, и придется заново вычислять оклеиваемую площадь.

Однако в классе не предусмотрено сохранение длин сторон, хотя они тоже могут понадобиться. Например, если потребуется изменить одну из величин у уже существующего объекта. Площадь же помещения всегда можно вычислить, если хранить исходные параметры. Поэтому сохранять саму площадь в поле не обязательно.

Исправьте код так, чтобы у объектов `Room` были только четыре поля – `width`, `length`, `height` и `wd`. Площади (полная и оклеиваемая) должны вычислять лишь при необходимости путем вызова методов.

Программа вычисляет площадь под оклейку, но ничего не говорит о том, сколько потребуется рулонов обоев. Добавьте метод, который принимает в качестве аргументов длину и ширину одного рулона, а возвращает количество необходимых, исходя из оклеиваемой площади.

Разработайте интерфейс программы. Пусть она запрашивает у пользователя данные и выдает ему площадь оклеиваемой поверхности и количество необходимых рулонов.

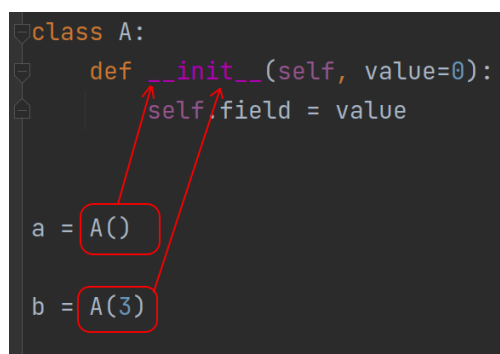
Урок 8. Перегрузка операторов

Перегрузка операторов в Python – это возможность с помощью специальных методов в классах переопределять различные операторы языка. Имена таких методов включают двойное подчеркивание спереди и сзади.

Под операторами в данном контексте понимаются не только знаки +, -, *, /, обеспечивающие операции сложения, вычитания и др., но также специфика синтаксиса языка, обеспечивающая операции создания объекта, вызова объекта как функции, обращение к элементу объекта по индексу, вывод объекта и другое.

Мы уже использовали ряд методов перегрузки операторов. Это

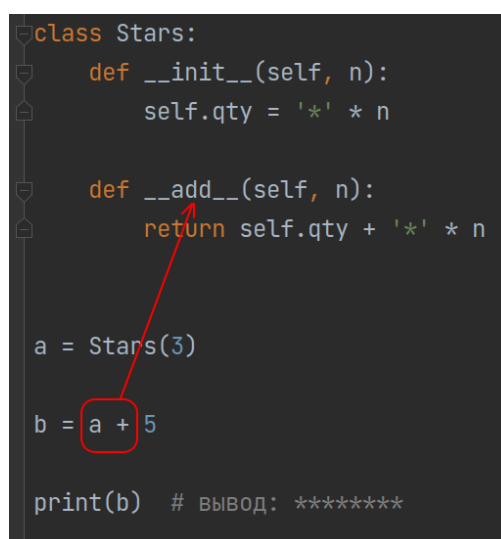
- `__init__()` – конструктор объектов класса, вызывается при создании объектов



```
class A:
    def __init__(self, value=0):
        self.field = value

a = A()
b = A(3)
```

- `__del__()` – деструктор объектов класса, вызывается при удалении объектов
- `__str__()` – преобразование объекта к строковому представлению, вызывается, когда объект передается функциям `print()` и `str()`
- `__add__()` – метод перегрузки оператора сложения, вызывается, когда объект участвует в операции сложения будучи операндом с левой стороны



```
class Stars:
    def __init__(self, n):
        self.qty = '*' * n

    def __add__(self, n):
        return self.qty + '*' * n

a = Stars(3)
b = a + 5

print(b) # вывод: *****
```

- `__setattr__()` – вызывается, когда атрибуту объекта выполняется присваивание

```

class A:
    def __init__(self, v):
        self.field = v

    def __setattr__(self, attr, value):
        if attr == 'field':
            self.__dict__[attr] = value
        else:
            raise AttributeError

a = A(15)
a.field = -3
a.field2 = 10

```

The image shows a code editor with Python code. A class `A` is defined with an `__init__` method that sets `self.field = v` and a `__setattr__` method that checks if the attribute is `'field'`. If it is, it sets `self.__dict__[attr] = value`; otherwise, it raises `AttributeError`. Below the class definition, an instance `a` is created with `a = A(15)`. Then, `a.field` is assigned `-3` and `a.field2` is assigned `10`. Red boxes highlight `self.field = v`, `a.field = -3`, and `a.field2 = 10`. Red arrows point from the `self.field` box in the `__init__` method to the `a.field` box, and from the `a.field2` box to the `__setattr__` method, illustrating how the custom `__setattr__` method is called for non-existent attributes.

В Python много других методов перегрузки операторов. В этом уроке рассмотрим еще несколько.

На самом деле перегрузка операторов в пользовательских классах используется не так часто, если не считать конструктора. Но сам факт наличия такой особенности объектно-ориентированного программирования, требует отдельного рассмотрения темы.

Возможность перегрузки операторов обеспечивает схожесть пользовательского класса со встроенными классами Python. Ведь все встроенные типы данных Питона – это классы. В результате все объекты могут иметь одинаковые интерфейсы. Так если ваш класс предполагает обращение к элементу объекта по индексу, например `a[0]`, то это можно обеспечить.

Пусть будет класс-агрегат B, содержащий в списке объекты класса A:

```

class A:
    def __init__(self, arg):
        self.arg = arg

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

class B:
    def __init__(self, *args):
        self.aList = []
        for i in args:
            self.aList.append(A(i))

```

```
group = B(5, 10, 'abc')
```

Чтобы получить элемент списка, несомненно, мы можем обратиться по индексу к полю `aList`:

```
print(group.aList[1])
```

Однако куда интереснее извлекать элемент по индексу из самого объекта, а не из его поля:

```
class B:
    def __init__(self, *args):
        self.aList = []
        for i in args:
            self.aList.append(A(i))

    def __getitem__(self, i):
        return self.aList[i]

group = B(5, 10, 'abc')
print(group.aList[1]) # выведет 10
print(group[0]) # 5
print(group[2]) # abc
```

Это делает объекты класса B похожими на объекты встроенных в Python классов-последовательностей (списков, строк, кортежей). Здесь метод `__getitem__()` перегружает операцию извлечения элемента по индексу. Другими словами, этот метод вызывается, когда к объекту применяется операция извлечения элемента: `объект[индекс]`.

Бывает необходимо, чтобы объект вел себя как функция. Это значит, если у нас есть объект `a`, то мы можем обращаться к нему в нотации функции, т. е. ставить после него круглые скобки и даже передавать в них аргументы:

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

Метод `__call__()` автоматически вызывается, когда к объекту обращаются как к функции. Например, здесь во второй строке произойдет вызов метода `__call__()` некоего Класа:

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

Пример:

```
class Changeable:
    def __init__(self, color):
        self.color = color

    def __call__(self, newcolor):
        self.color = newcolor

    def __str__(self):
        return "%s" % self.color

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. Такое обращение автоматически вызовет метод `__call__()`, который, в данном случае, изменит атрибут `color` объекта.

В Python кроме метода `__str__()` есть схожий с ним по поведению, но более "низкоуровневый" метод `__repr__()`. Оба метода должны возвращать строку.

Если в классе есть только метод `__str__()`, то при обращении к объекту в интерпретаторе без функции `print()`, он не будет вызываться:

```
>>> class A:
...     def __str__(self):
...         return "This is object of A"
...
>>> a = A()
>>> print(a)
This is object of A
>>> a
<__main__.A instance at 0x7fe964a4cdd0>
>>> str(a)
'This is object of A'
>>> repr(a)
'<__main__.A instance at 0x7fe964a4cdd0>'
```

Попытка преобразования объекта в строку с помощью встроенной функции `str()` также вызывает метод `__str__()`. То есть скорее всего метод `__str__()` перегружает не функцию `print()`, а функцию `str()`. Функция же `print()` так устроена, что сама вызывает `str()` для своих аргументов.

В Python есть встроенная функция `repr()`, которая также как `str()` преобразует объект в строку. Но "сырую" строку. Что это значит, попробуем понять с помощью примера:

```
>>> a = '3 + 2'
>>> b = repr(a)
>>> a
'3 + 2'
>>> b
"'3 + 2'"
>>> eval(a)
5
>>> eval(b)
'3 + 2'
>>> c = "Hello\nWorld"
>>> d = repr(c)
>>> c
'Hello\nWorld'
>>> d
"'Hello\\nWorld'"
>>> print(c)
Hello
World
>>> print(d)
'Hello\nWorld'
```

Функция `eval` преобразует переданную строку в программный код, который тут же выполняется. Функция `print()` выполняет переход на новую строку, если встречается символ `\n`. Функция `repr()`

выполняет действия, направленные на своего рода защиту строки от интерпретации, оставляет ее "сырой", т. е. в исходном виде. Еще раз:

```
>>> c = "Hello\nWorld"
>>> c # аналог print(repr(c))
'Hello\nWorld'
>>> print(c) # аналог print(str(c))
Hello
World
```

Однако для большинства случаев различия между `repr()` и `str()` не важны. Поэтому в классах проще определять один метод `__repr__()`. Он будет вызываться для всех случаев преобразования к строке:

```
>>> class A:
...     def __repr__(self):
...         return "It's obj of A"
...
>>> a = A()
>>> a
It's obj of A
>>> repr(a)
"It's obj of A"
>>> str(a)
"It's obj of A"
>>> print(a)
It's obj of A
```

Если же нужен различающийся вывод данных, тогда в классе следует определить оба метода перегрузки операторов преобразования к строке.

Практическая работа

Напишите класс `Snow` по следующему описанию.

В конструкторе класса иницируется поле, содержащее количество снежинок, выраженное целым числом.

Класс включает методы перегрузки арифметических операторов: `__add__()` – сложение, `__sub__()` – вычитание, `__mul__()` – умножение, `__truediv__()` – деление. В классе код этих методов должен выполнять увеличение или уменьшение количества снежинок на число `n` или в `n` раз. Метод `__truediv__()` перегружает обычное (`/`), а не целочисленное (`//`) деление. Однако пусть в методе происходит округление значения до целого числа.

Класс включает метод `makeSnow()`, который принимает сам объект и число снежинок в ряду, а возвращает строку вида `"*****\n*****\n*****..."`, где количество снежинок между `\n` равно переданному аргументу, а количество рядов вычисляется, исходя из общего количества снежинок.

Вызов объекта класса `Snow` в нотации функции с одним аргументом, должен приводить к перезаписи значения поля, в котором хранится количество снежинок, на переданное в качестве аргумента значение.

Урок 9. Статические методы

Ранее было сказано, с определенным допущением классы можно рассматривать как модули, содержащие переменные со значениями и функции. Только здесь переменные называются полями или свойствами, а функции – методами. Вместе поля и методы называются атрибутами.

Однако в случае классов, когда метод применяется к объекту, этот экземпляр передается в метод в качестве первого аргумента:

```
>>> class A:
...     def meth(self):
...         print('meth')
...
>>> a = A()
>>> a.meth()
meth
>>> A.meth(a)
meth
```

Вызов `a.meth()` на самом деле преобразуется к `A.meth(a)`, то есть мы идем к "модулю А" и в его пространстве имен ищем атрибут `meth`. Там оказывается, что `meth` это функция, принимающая один обязательный аргумент. Тогда ничего не мешает сделать так:

```
>>> b = 10
>>> A.meth(b)
meth
```

В таком "модульном формате" вызова методов передавать объект-экземпляр именно класса А совсем не обязательно. Однако нельзя сделать так:

```
>>> b = 10
>>> b.meth()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'meth'
```

Если объект передается методу в нотации через точку, то этот метод должен быть описан в том классе, которому принадлежит объект, или в родительских классах. В данном случае у класса `int` нет метода `meth()`. Объект `b` классу `A` не принадлежит. Поэтому интерпретатор никогда не найдет метод `meth()`.

Что делать, если возникает необходимость в методе, который не принимал бы объект данного класса в качестве аргумента? Да, мы можем объявить метод вообще без параметров и вызывать его только через класс:

```
>>> class A:
...     def meth():
...         print('meth')
...
>>> A.meth()
meth
>>> a = A()
>>> a.meth()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: meth() takes 0 positional arguments but 1 was given
```

Получается странная ситуация. Ведь `meth()` вызывается не только через класса, но и через порожденные от него объекты. Однако в последнем случае всегда будет возникать ошибка. То есть имеется потенциально ошибочный код. Кроме того, может понадобиться метод с параметрами, но которому не надо передавать экземпляр данного класса.

В ряде языков программирования, например в Java, для таких ситуаций предназначены статические методы. При описании этих методов в их заголовке ставится ключевое слово `static`. Такие методы могут вызываться через объекты данного класса, но сам объект в качестве аргумента в них не передается.

В Python острой необходимости в статических методах нет, так как код может находиться за пределами класса, и программа не начинает выполняться из класса. Если нам нужна просто какая-нибудь функция, мы можем определить ее в основной ветке. В Java это не так. Там, не считая импортов, весь код находится внутри классов. Поэтому методы, не принимающие объект данного класса и играющие роль обычных функций, необходимы. Статические методы решают эту проблему.

Однако в Python тоже можно реализовать подобное, то есть статические методы, с помощью декоратора `@staticmethod`:

```
>>> class A:
...     @staticmethod
...     def meth():
...         print('meth')
...
>>> a = A()
>>> a.meth()
meth
>>> A.meth()
meth
```


Пример с параметром:

```
>>> class A:
...     @staticmethod
...     def meth(value):
...         print(value)
...
>>> a = A()
>>> a.meth(1)
1
>>> A.meth('hello')
hello
```

Статические методы в Python – по-сути обычные функции, помещенные в класс для удобства и находящиеся в пространстве имен этого класса. Это может быть какой-то вспомогательный код. Вообще, если в теле метода не используется `self`, то есть ссылка на конкретный объект, следует задуматься, чтобы сделать метод статическим. Если такой метод необходим только для

обеспечения внутренних механизмов работы класса, то возможно его не только надо объявить статическим, но и скрыть от доступа из вне.

Статические методы



В Python не обязательны.

Экземпляр не передается первым аргументом (метод без self).

`b.adder(33)` → ~~`adder(b, 33)`~~

Одинаково вызывается от класса и экземпляра.

Пусть у нас будет класс "Цилиндр". При создании объектов от этого класса у них заводятся поля высота и диаметр, а также площадь поверхности. Вычисление площади можно поместить в отдельную статическую функцию. Она вроде и относится к цилиндрам, но с другой стороны само вычисление объекта не требует и может быть использовано где угодно.

```
from math import pi

class Cylinder:
    @staticmethod
    def make_area(d, h):
        circle = pi * d ** 2 / 4
        side = pi * d * h
        return round(circle*2 + side, 2)

    def __init__(self, diameter, high):
        self.dia = diameter
        self.h = high
        self.area = self.make_area(diameter, high)

a = Cylinder(1, 2)
print(a.area)

print(a.make_area(2, 2))
```

В примере вызов `make_area()` за пределами класса возможен в том числе через экземпляр. При этом понятно, в данном случае свойство `area` самого объекта `a` не меняется. Мы просто вызываем функцию, находящуюся в пространстве имен класса.

Практическая работа

Приведенный в конце урока пример плохой. Мы можем менять значения полей `dia` и `h` объекта за пределами класса простым присваиванием (например, `a.dia = 10`). При этом площадь никак не будет пересчитываться. Также мы можем назначить новое значение для площади, как простым

присваиванием, так и вызовом функции `make_area()` с последующим присваиванием. Например, `a.area = a.make_area(2, 3)`. При этом не меняются высота и диаметр.

Защитите код от возможных логических ошибок следующим образом:

- Свойствам `dia` и `h` объекта по-прежнему можно выполнять присваивание за пределами класса. Однако при этом "за кулисами" происходит пересчет площади, т. е. изменение значения `area`.
- Свойству `area` нельзя присваивать за пределами класса. Можно только получать его значение.

Подсказка: вспомните про метод `__setattr__()`, упомянутый в уроке про инкапсуляцию.

Урок 10. Итераторы

В английской документации по Python фигурируют два похожих слова – **iterable** и **iterator**. Обозначают они разное, хотя и имеющее между собой связь.

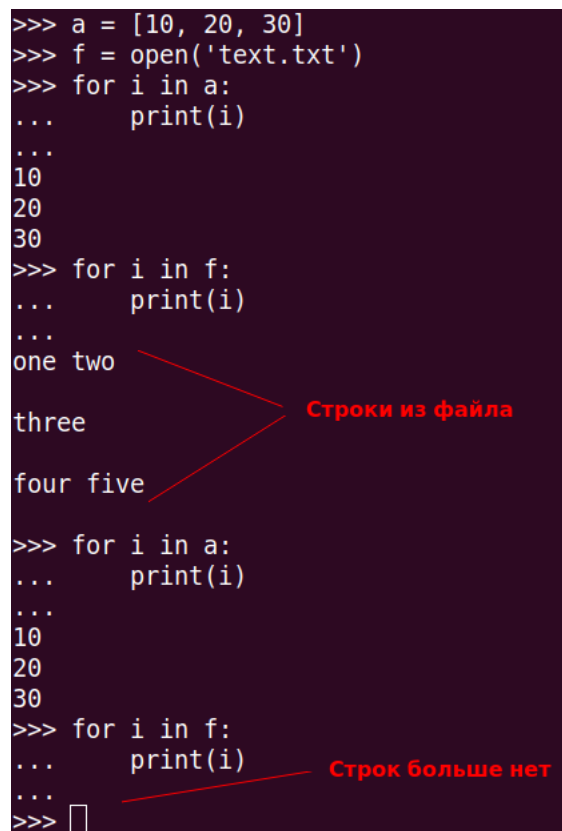
На русский язык **iterable** обычно переводят как итерируемый объект, а **iterator** – как итератор, или объект-итератор. С объектами обоих разновидностей мы уже сталкивались в курсе "Python. Введение в программирование", однако не делали на этом акцента.

Iterable и **iterator** – это не какие-то конкретные классы-типы, наподобие `int` или `list`. Это обобщения. Существует ряд встроенных классов, чьи объекты обладают возможностями **iterable**. Ряд других классов порождают объекты, обладающие свойствами итераторов.

Кроме того, мы можем сами определять классы, создающие итераторы или итерируемые объекты.

Примером итерируемого объекта является список. Примером итератора – файловый объект. Список включает в себя все свои элементы, а файловый объект по-очереди "вынимает" из себя элементы и "забывает" то, что уже вынул. Также не ведает, что в нем содержится еще, так как это "еще" может вычисляться при каждом обращении или поступать извне. Например, файловый объект не знает сколько еще текста в связанном с ним файле.

Из такого описания должно быть понятно, почему один и тот же список мы можем перебирать сколько угодно раз, а с файловым объектом это можно сделать только единожды.



```
>>> a = [10, 20, 30]
>>> f = open('text.txt')
>>> for i in a:
...     print(i)
...
10
20
30
>>> for i in f:
...     print(i)
...
one two
three
four five
>>> for i in a:
...     print(i)
...
10
20
30
>>> for i in f:
...     print(i)
...
>>>
```

Строки из файла

Строк больше нет

Зачем нужны объекты, элементы которых можно получить только один раз? Представьте, что текстовый файл большой. Если сразу загрузить его содержимое в память, то последней может не хватить. Также бывает удобно генерировать значения на лету, по требованию, если они

нужны в программе только один раз. В противовес тому, как если бы они были получены все сразу и сохранены в списке.

У всех итераторов, но не итерируемых объектов, **есть метод `__next__()`**. Именно его код обеспечивает выдачу очередного элемента. Каков этот код, зависит от конкретного класса. У файлового объекта это по всей видимости код, читающий очередную строку из связанного файла.

```
>>> f = open('text.txt')
>>> f.__next__()
'one two\n'
>>> f.__next__()
'three \n'
>>> f.__next__()
'four five\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Когда итератор выдал все свои значения, то очередной вызов `__next__()` должен возбуждать исключение `StopIteration`. Почему именно такое исключение? Потому что на него "реагирует" цикл `for`. Для `for` это сигнал останова.

Судя по наличию подчеркиваний у `__next__()`, он относится к методам перегрузки операторов. Он перегружает встроенную функцию `next()`. То есть когда объект передается в эту функцию, то происходит вызов метода `__next__()` этого объекта-итератора.

```
>>> f = open('text.txt')
>>> next(f)
'one two\n'
>>> next(f)
'three \n'
>>> next(f)
'four five\n'
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Если объект итератором не является, то есть у него нет метода `__next__()`, то вызов функции `next()` приведет к ошибке:

```
>>> a = [1, 2]
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
```

Внутренний механизм работы цикла `for` так устроен, что на каждой итерации он вызывает функцию `next()` и передает ей в качестве аргумента объект, указанный после `in` в заголовке. Как только `next()` возвращает `StopIteration`, цикл `for` ловит это исключение и завершает свою работу.

Напишем собственный класс с методом `__next__()`:

```

>>> class A:
...     def __init__(self, qty):
...         self.qty = qty
...     def __next__(self):
...         if self.qty > 0:
...             self.qty -= 1
...             return '+'
...         else:
...             raise StopIteration
...
>>> a = A(3)
>>> next(a)
'+'
>>> next(a)
'+'
>>> next(a)
'+'
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration

```

Вызов next() работает, но если мы попробуем передать объект циклу for, получим ошибку:

```

>>> b = A(5)
>>> for i in b:
...     print(i)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'A' object is not iterable

```

Интерпретатор говорит, что объект типа A не является итерируемым объектом. Другими словами, цикл for ожидает, что после in будет стоять итерируемый объект, а не итератор. Как же так, если цикл for потом вызывает метод __next__(), который есть только у итераторов?

На самом деле цикл for ожидает, что у объекта есть не только метод __next__(), но и __iter__(). Задача метода __iter__() – "превращать" итерируемый объект в итератор. Если в цикл for передается уже итератор, то метод __iter__() этого объекта должен возвращать сам объект:

```

>>> class A:
...     def __init__(self, qty):
...         self.qty = qty
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.qty > 0:
...             self.qty -= 1
...             return '+'
...         else:
...             raise StopIteration
...
>>> a = A(4)
>>> for i in a:
...     print(i)
...
+
+

```

+

+

Если циклу `for` передается не итератор, а итерируемый объект, то его метод `__iter__()` должен возвращать не сам объект, а какой-то объект-итератор. То есть объект, созданный от другого класса.

Получается, в классах-итераторах метод `__iter__()` нужен лишь для совместимости. Ведь если `for` работает как с итераторами, так и итерируемыми объектами, но последние требуют преобразования к итератору, и `for` вызывает `__iter__()` без оценки того, что ему передали, то требуется, чтобы оба – `iterator` и `iterable` – поддерживали этот метод. С точки зрения наличия в классе метода `__iter__()` итераторы можно считать подвидом итерируемых объектов.

Очевидно, по аналогии с `next()`, цикл `for` вызывает не метод `__iter__()`, а встроенную в Python функцию `iter()`.

Если список передать функции `iter()`, получим совсем другой объект:

```
>>> s = [1, 2]
>>> si = iter(s)
>>> type(s)
<class 'list'>
>>> type(si)
<class 'list_iterator'>
>>> si
<list_iterator object at 0x7f217a583320>
>>> next(si)
1
>>> next(si)
2
>>> next(si)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Как видно, объект класса `list_iterator` исчерпывается как нормальный итератор. Список `s` при этом никак не меняется. Отсюда понятно, почему после обхода циклом `for` итерируемые объекты остаются в прежнем составе. От них создается "копия"-итератор, а с ними самими цикл `for` не работает.



Напишем свой iterable-класс и связанный с ним iterator-класс.

```
class Letters:
    def __init__(self, string):
        self.letters = []
        for i in string:
            self.letters.append(f'-{i}-')

    def __iter__(self):
        return LettersIterator(self.letters[:])

class LettersIterator:
    def __init__(self, letters):
        self.letters = letters

    def __iter__(self):
        return self

    def __next__(self):
        if self.letters == []:
            raise StopIteration
        item = self.letters[0]
        del self.letters[0]
        return item

kit = Letters('aeoui')
print(kit.letters)
```

```
for i in kit:
    print(i)

print(kit.letters)
```

Результат:

```
['-a-', '-e-', '-o-', '-u-', '-i-']
-a-
-e-
-o-
-u-
-i-
['-a-', '-e-', '-o-', '-u-', '-i-']
```

Практическая работа

Напишите класс-итератор, объекты которого генерируют случайные числа в количестве и в диапазоне, которые передаются в конструктор.

Урок 11. Генераторы

Генераторы можно считать подвидом итераторов, а способ их создания – инструментом для создания несложных итераторов.

В отличие от обычных итераторов, генераторы создаются путем вызова функции, а не от класса.

Чтобы функция возвращала объект-генератор, в ее теле должен быть оператор **yield**. Когда любая yield-содержащая функция вызывается, она возвращает объект типа **generator**, а не None или какой-нибудь другой тип данных через оператор return.

У генераторов методы `__next__()` и `__iter__()` создаются средствами самого языка, то есть автоматически. Программисту их определять не надо, что упрощает создание пользовательских типов итераторов.

```
>>> def starmaker(n):
...     while n > 0:
...         yield '*'
...         n -= 1
...
>>> type(starmaker)
<class 'function'>
>>> s = starmaker(3)
>>> type(s)
<class 'generator'>
>>> next(s)
'*'
>>> next(s)
'*'
>>> next(s)
'*'
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

В определенном смысле оператор yield заменяет return с тем исключением, что мы снова возвращаемся в функцию, когда вызывается `next()`. При этом объект-генератор помнит состояние переменных и место, откуда при прошлом вызове произошел выход из функции.

Если мы сделаем нечто подобное

```
>>> def g():
...     yield 1
...
```

то не получим бесконечный генератор, потому что код тела функции полностью выполнится при первом вызове `next()`:

```
>>> a = g()
>>> next(a)
1
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Обратите внимание, что функция `starmaker()` делает то же самое, что класс, описанный в прошлом уроке:

```
>>> class A:
...     def __init__(self, qty):
...         self.qty = qty
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.qty > 0:
...             self.qty -= 1
...             return '+'
...         else:
...             raise StopIteration
...
>>> a = A(3)
>>> for i in a:
...     print(i)
...
+
+
+
```

При этом код функции, создающей итератор, намного короче аналогичного класса. Поэтому классы-итераторы скорее уместны, когда создаются сложные объекты, включающие множество полей и сложную логику их обработки, а не только методы `__iter__()` и `__next__()`.

Генераторные выражения

Существует еще более простой, чем функция с `yield`, способ создания итераторов – генераторные выражения. Они подходят, когда код тела функции можно записать в одно выражение.

Синтаксис генераторных выражений подобен генераторам списков, рассматриваемых в курсе "Python. Введение в программирование". Однако, в отличие от списков, в случае генераторов используются круглые скобки.

Напомним, как выглядят генераторы списков и то, что возвращают они списковый тип данных:

```
>>> a = [i+1 for i in range(10)]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> type(a)
<class 'list'>
>>> import random
>>> b = [random.randint(0,9) for i in range(5)]
>>> b
[2, 5, 5, 2, 9]
>>> c = [i for i in b if i % 2 == 0]
>>> c
[2, 2]
```

Результат выражения, стоящего до `for`, добавляется на каждой итерации цикла в итоговый список. Выполнение выражения генератора списка сразу заполняет список.

В случае генераторных выражений создается объект-генератор, у которого будет вычисляться очередной элемент только при каждом вызове `next()`:

```
>>> a = (i+1 for i in range(10))
>>> a
<generator object <genexpr> at 0x7fa586339f10>
>>> type(a)
<class 'generator'>
>>> next(a)
1
>>> next(a)
2
```

Пример со звездочкой с помощью генераторного выражения будет выглядеть так:

```
>>> d = ('*' for i in range(5))
>>> for i in d:
...     print(i)
...
*
*
*
*
*
```

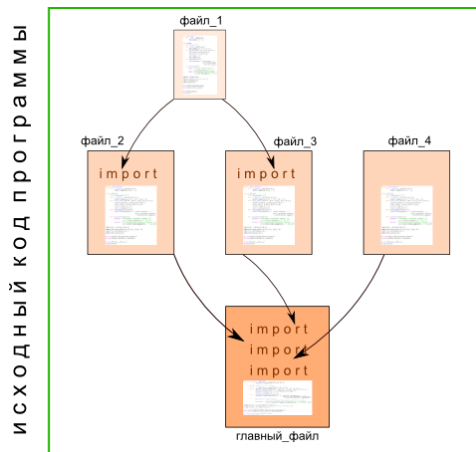
В отличие от генераторных выражений, `yield`-функции более универсальны не только из-за произвольного количества кода в их теле. В них вы можете передавать разные значения аргументов. А значит, одна и та же функция может использоваться для создания нескольких разных генераторов.

Практическая работа

В задании к прошлому уроку требовалось написать класс-итератор, объекты которого генерируют случайные числа в количестве и в диапазоне, которые передаются в конструктор. Напишите выполняющую ту же задачу генераторную функцию. В качестве аргументов она должна принимать количество элементов и диапазон.

Урок 12. Модули и пакеты

Что такое модули, как их импортировать в программу, а также как создавать собственные модули, было описано в [одном](#) из уроков курса "Python. Введение в программирование". Там модули рассматривались с точки зрения обособления функций, которые потом можно было бы импортировать в разные программы. На самом деле модули содержат не столько функции, сколько классы с их методами.



В этом уроке шагнем дальше и рассмотрим, как несколько модулей-файлов могут быть объединены в пакет. Также выясним, что модули могут исполняться как самостоятельные программы.

Пакеты модулей

В программировании связанные модули принято объединять в пакеты. Пакет представляет собой каталог с файлами-модулями. Кроме того, внутри пакета могут быть вложенные каталоги, а уже в них – файлы.

Допустим, мы пишем пакет модулей для вычисления площадей и периметров фигур. Пакет будет состоять из двух модулей. В одном будут описаны классы двумерных фигур, в другом – трехмерных.

Каталог-пакет назовем `geometry`. Один модуль – `planimetry.py`, другой – `stereometry.py`. Пакет следует разместить в одном из каталогов, содержащихся в списке `sys.path`. Первым его элементом является домашний каталог, обозначаемый как пустая строка. Таким образом, пакет проще разместить в том же каталоге, где будет основной скрипт.

Если не планируется писать скрипт, а достаточно протестировать пакет в интерактивном режиме, то в Linux будет проще разместить его в домашнем каталоге.

Содержимое файла `planimetry.py`:

```
from math import pi, pow

class Rectangle:
    def __init__(self, a, b):
        self.w = a
```

```

        self.h = b

    def square(self):
        return round(self.w * self.h, 2)

    def perimeter(self):
        return 2 * (self.w + self.h)

class Circle:
    def __init__(self, radius):
        self.r = radius

    def square(self):
        return round(pi * pow(self.r, 2), 2)

    def length(self):
        return round(2 * pi * self.r)

```

Код файла stereometry.py:

```

from math import pi, pow

class Cuboid:
    def __init__(self, a, b, c):
        self.length = a
        self.width = b
        self.height = c
        self.__sq_sur = 2 * (a * b + a * c + b * c)
        self.__volume = a * b * c

    def S(self):
        return round(self.__sq_sur, 2)

    def V(self):
        return round(self.__volume, 2)

class Ball:
    def __init__(self, radius):
        self.r = radius

    def S(self):
        s = 4 * pi * pow(self.r, 2)
        return round(s, 2)

    def V(self):
        v = (4 / 3) * pi * pow(self.r, 3)
        return round(v, 2)

```

Также в каталоге пакета должен быть файл `__init__.py`, даже если этот файл будет пустым. Его наличие позволяет интерпретатору понять, что перед ним пакет, а не просто каталог. Файл `__init__.py` может быть не пустым, а содержать переменную, в которой перечислены модули, которые будут импортироваться командой `from имя_пакета import *`, а также какой-либо иницизирующий код, например, подключение к базе данных.

Теперь попробуем импортировать модули пакета:

```
>>> import geometry.planimetry, geometry.stereometry
>>> a = geometry.planimetry.Rectangle(3, 4)
>>> b = geometry.stereometry.Ball(5)
>>> a.square()
12
>>> b.v()
523.6
```

Если сделать импорт только пакета, то мы не сможем обращаться к модулям:

```
pl@pl-desk:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import geometry
>>> b = geometry.stereometry.Ball(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'geometry' has no attribute 'stereometry'
```

Тогда возникает вопрос: в чем выгода пакетов, если все равно приходится импортировать модули индивидуально? Основной смысл заключается в структурировании пространств имен. Представьте, что есть разные пакеты, содержащие одноименные модули и классы. В таком случае точечная нотация через имя пакета, подпакета, модуля дает возможность пользоваться в программе одноименными сущностями из разных пакетов. Например, `a.samename` и `b.samename`. Кроме того точечная нотация дает своего рода описание объекту. Например, выражения `geometry.planimetry.House()` или `geometry.stereometry.House()` говорят, что в первом случае будет создан двумерный объект-дом, во-втором – трехмерный. Это куда информативней, чем просто `House()`.

Однако в файле `__init__.py` в переменной `__all__` можно перечислить, какие модули будут импортироваться через `from имя_пакета import *`:

```
__all__ = ['planimetry', 'stereometry']
```

После этого можно делать так:

```
>>> from geometry import *
>>> b = stereometry.Ball(5)
>>> a = planimetry.Circle(5)
```

Выполнение модуля как скрипта

В Python обычный файл-скрипт, или файл-программа, не отличается от файла-модуля почти ничем. Нет команд языка, которые бы "говорили", что вот это – модуль, а это – скрипт. Отличие заключается лишь в том, что обычно модули не содержат команды вызова функций и создания экземпляров в основной ветке. В модуле обычно происходит только определение классов и функций.

Однако возможности языка позволяют в модули помещать код, который будет выполняться, когда файл не импортируется, а сам передается интерпретатору как самостоятельная программа. Выглядит это примерно так:

```
class A:
    def __str__(self):
        return "A"

if __name__ == "__main__":
    print(A())
```

То, что находится в теле if, выполнится только в случае исполнения файла как скрипта. Но не при импорте.

```
pl@pl-desk:~$ python3 test.py
A
```

Встроенный атрибут `__name__`, представляющий собой переменную, есть у каждого файла. При импорте этой переменной присваивается имя модуля:

```
>>> import math
>>> math.__name__
'math'
>>> planimetry.__name__
'geometry.planimetry'
```

Однако когда файл исполняется как скрипт, значение `__name__` становится равным строке `"__main__"`. Это можно увидеть, если в код поместить `print(__name__)` и выполнить файл как скрипт.

Таким образом, если `__name__` равен `"__main__"`, то выполняется код, вложенный в тело условного оператора. Обычно сюда помещают код для тестирования модуля в процессе разработки, а в готовый модуль – примеры, как пользоваться определенными здесь сущностями.

Практическая работа

В практической работе урока 7 "Композиция" требовалось разработать интерфейс взаимодействия с пользователем. Разнесите сам класс и интерфейс по разным файлам. Какой из них выполняет роль модуля, а какой – скрипта? Оба файла можно поместить в один каталог.

Урок 13. Документирование кода

Поскольку в языках программирования, в том числе Python, существует большое количество встроенных и предоставляемых третьими сторонами модулей, то помнить их все, в том числе их функционал, невозможно. Поэтому важно документировать код. Под документированием понимаются не комментарии, а так называемые строки документации, которые принято добавлять в начало модуля, класса, метода или функции.

Основное назначение комментариев – пояснить что делает код, как он работает. Основное назначение строк документации – кратко описать в целом для чего предназначен объект, какие аргументы принимает, и что возвращает.

The infographic is divided into two main sections. The top section, titled 'docstring - строка документации', explains that a docstring is a string literal used as the first expression in a module, class, or function, which is then stored in the `__doc__` attribute. It notes that docstrings are typically enclosed in triple double quotes. Below this, it states that docstrings contain information about the object's purpose and usage. The bottom section, titled 'help()', describes it as a tool for reading documentation, which launches an interactive help system. If an argument is provided, it displays the documentation for that specific object.

docstring – строка документации

Строковый литерал, который располагают как первое выражение модуля, класса, функции. При интерпретации кода помещается в атрибут `__doc__` своего объекта.

Строки документации принято заключать в тройные двойные кавычки.

Содержат информацию о назначении объекта, описывают способы работы с ним.

Инструмент чтения строк документации.

help()

Запускает встроенную справочную систему в интерактивном режиме. Если передается аргумент, в структурированном виде выводит строки документации данного объекта.

В исходном коде Python строки документации заключаются в тройные кавычки и пишутся сразу под заголовком объекта. Пример документированного модуля:

```
"""Модуль содержит классы плоских фигур."""

from math import pi, pow

class Rectangle:
    """Класс Прямоугольник."""

    def __init__(self, a, b):
        """Конструктор принимает длину и ширину."""
        self.w = a
        self.h = b

    def square(self):
        """Метод для вычисления площади."""
        return round(self.w * self.h, 2)
```

```

def perimeter(self):
    """Метод для вычисления периметра"""
    return 2 * (self.w + self.h)

class Circle:
    """Класс Круг."""

    def __init__(self, radius):
        """Конструктор принимает радиус."""
        self.r = radius

    def square(self):
        """Метод для вычисления
        площади круга."""
        return round(pi * pow(self.r, 2), 2)

    def length(self):
        """Метод для вычисления
        длины окружности."""
        return round(2 * pi * self.r)

```

Извлекать строки документации можно двумя способами:

- Через встроенный для каждого объекта атрибут-переменную `__doc__`.
- С помощью встроенной в Python функции `help()`, которая запускает интерактивную справочную систему.

```

>>> import planimetry
>>> planimetry.Rectangle.__doc__
'Класс Прямоугольник.'
>>> print(planimetry.__doc__)
Модуль содержит классы плоских фигур.

```

Если дать команду `help(planimetry)`, то будут выведены все строки документации модуля в структурированном виде. Однако можно "заказывать" справку по отдельным объектам:

```

>>> help(planimetry.Circle.length)
Help on function length in module geometry.panimetry:

length(self)
    Метод для вычисления длины окружности.

```

Выход из справки осуществляется с помощью клавиши `q`.

В Python документированы все встроенные объекты и модули библиотеки.

```

>>> print(str.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

```

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`.
encoding defaults to `sys.getdefaultencoding()`.
errors defaults to 'strict'.

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
>>> print(math.trunc.__doc__)
trunc(x:Real) -> Integral
```

Truncates x to the nearest Integral toward 0. Uses the `__trunc__` magic method.

Таким образом, если вам надо узнать, для чего предназначена та или иная функция, и как она работает, это всегда можно сделать, не выходя из интерпретатора Python. Перечень атрибутов модуля или объекта можно посмотреть с помощью встроенной функции `dir()`.

Следует отметить, документирование модулей принято выполнять на английском языке. Многие проекты имеют открытый код, выкладываются в Веб, их смотрят и дорабатывают программисты из разных стран. Использование одного языка позволяет им понимать друг друга. Поэтому профессиональный программист должен владеть английским хотя бы на начальном уровне. Google Translate – тоже вариант.

Практическая работа

Выполните полное документирование модуля, созданного в практической работе прошлого урока.

Урок 14. Пример объектно-ориентированной программы на Python

В конце курса закрепим пройденный материал, написав еще одну небольшую объектно-ориентированную программу на Python.

В ООП очень важно предварительное проектирование. В общей сложности можно выделить следующие этапы разработки объектно-ориентированной программы:

1. Формулирование задачи.
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут создаваться объекты. В случае необходимости установление между классами наследственных связей.
4. Определение ключевых для данной задачи свойств и методов объектов.
5. Создание классов, определение их полей и методов.
6. Создание объектов.
7. Решение задачи путем организации взаимодействия объектов.

Пусть необходимо разработать виртуальную модель процесса обучения. В программе должны быть объекты-ученики, учитель, кладезь знаний.

Потребуется три класса – "учитель", "ученик", "данные". Учитель и ученик во многом похожи, оба – люди. Значит, их классы могут принадлежать одному надклассу "человек". Однако в контексте данной задачи у учителя и ученика вряд ли найдутся общие атрибуты.

Определим, что должны уметь объекты для решения задачи "увеличить знания":

- Ученик должен уметь брать информацию и превращать ее в свои знания.
- Учитель должен уметь учить группу учеников.
- Данные могут представлять собой список знаний. Элементы будут извлекаться по индексу.

```
class Data:
    def __init__(self, *info):
        self.info = list(info)

    def __getitem__(self, i):
        return self.info[i]

class Teacher:
    def __init__(self):
        self.work = 0

    def teach(self, info, *pupil):
        for i in pupil:
            i.take(info)
            self.work += 1
```

```
class Pupil:
    def __init__(self):
        self.knowledge = []

    def take(self, info):
        self.knowledge.append(info)
```

В класс Teacher также добавлено свойство экземпляров work, чтобы подсчитывать количество проделанной учителем работы.

Теперь посмотрим, как объекты этих классов могут взаимодействовать между собой:

```
>>> from test import *
>>> lesson = Data('class', 'object', 'inheritance', 'polymorphism',
'encapsulation')
>>> marIvanna = Teacher()
>>> vasy = Pupil()
>>> pety = Pupil()
>>> marIvanna.teach(lesson[2], vasy, pety)
>>> marIvanna.teach(lesson[0], pety)
>>> vasy.knowledge
['inheritance']
>>> pety.knowledge
['inheritance', 'class']
```

Практическая работа

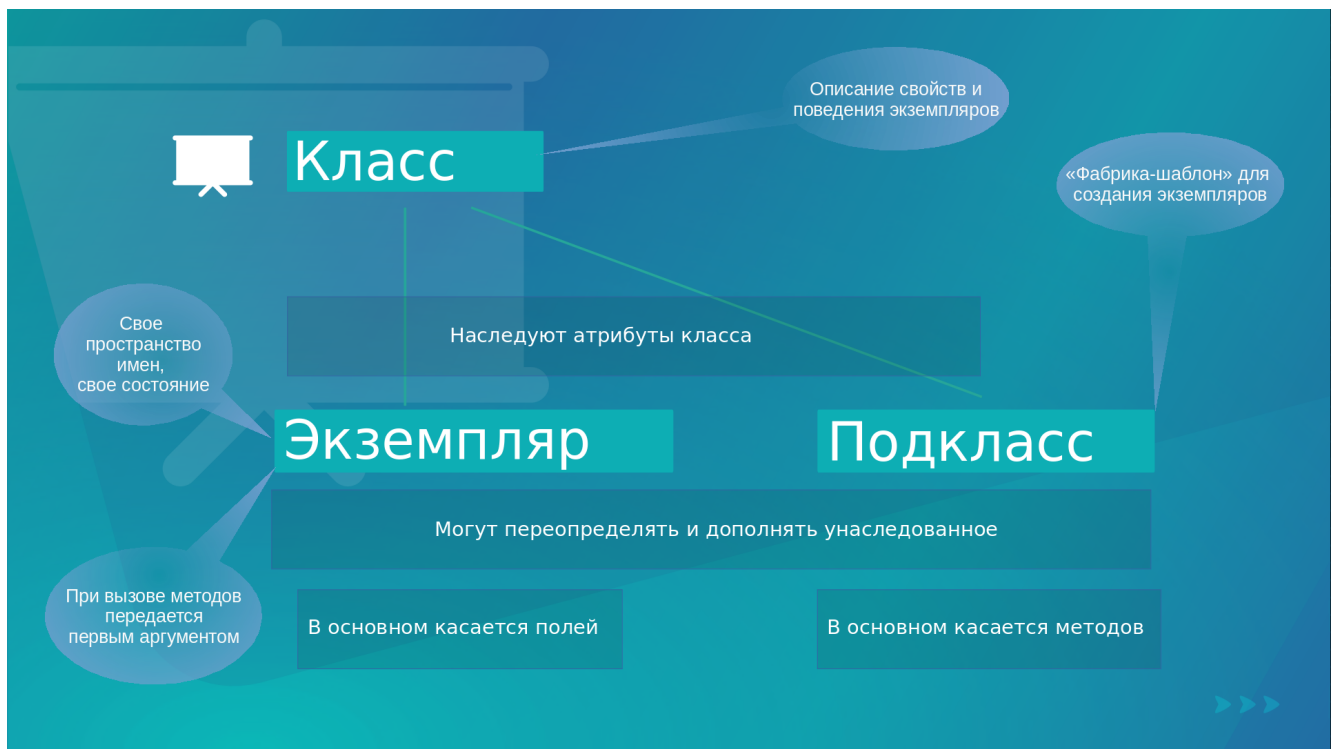
Может ли в этой программе ученик учиться без учителя? Если да, пусть научится чему-нибудь сам.

Добавьте в класс Pupil метод, позволяющий ученику случайно "забывать" какую-нибудь часть своих знаний.

Урок 15. Итоги курса "Объектно-ориентированное программирование на Python"

Общее представление об объектно-ориентированном программировании и его особенностях были рассмотрены в первом уроке. Здесь обобщим изученный в этом курсе материал.

В Python все объекты являются производными классов и наследуют от них атрибуты. При этом каждый объект формирует собственное пространство имен. Python поддерживает такие ключевые особенности объектно-ориентированного программирования как наследование, инкапсуляцию и полиморфизм. Однако инкапсуляцию в понимании сокрытия данных Python поддерживает только в рамках соглашения, но не синтаксиса языка.



В курсе не было уделено внимание множественному наследованию, когда дочерний класс наследуется от нескольких родительских. Такое наследование поддерживается в Python в полной мере и дает возможность в производном классе сочетать атрибуты двух и более классов. При множественном наследовании следует учитывать определенные особенности поиска атрибутов.

Полиморфизм позволяет объектам разных классов иметь схожие интерфейсы. Он реализуется путем объявления в них методов с одинаковыми именами. К проявлению полиморфизма как особенности ООП также можно отнести методы перегрузки операторов.

Кроме наследования, инкапсуляции и полиморфизма существуют другие особенности ООП. Таковой является композиция, или агрегирование, когда класс включает в себя вызовы других классов. В результате при создании объекта от класса-агрегата, создаются объекты других классов, являющиеся составными частями первого.

Классы обычно помещают в модули. Каждый модуль может содержать несколько классов. В свою очередь модули могут объединяться в пакеты. Благодаря пакетам в Python организуются пространства имен.

Преимущества ООП

Особенности объектно-ориентированного программирования наделяют его рядом преимуществ.

Так ООП позволяет использовать один и тот же программный код с разными данными. На основе классов создается множество объектов, у каждого из которых могут быть собственные значения полей. Нет необходимости вводить множество переменных, т. к. объекты получают в свое распоряжение индивидуальные пространства имен. В этом смысле объекты похожи на структуры данных. Объект можно представить как некую упаковку данных, к которой присоединены инструменты для их обработки – методы.

Наследование позволяет не писать новый код, а использовать и настраивать уже существующий за счет добавления и переопределения атрибутов.

Недостатки ООП

ООП позволяет оптимизировать дальнейшую поддержку разрабатываемого приложения, однако предполагает большую роль предварительного анализа предметной области и проектирования. От правильности решений на этом этапе зависит куда больше, чем от непосредственного написания исходного кода.

Следует понимать, что одна и та же задача может быть решена разными объектными моделями, каждая из которых будет иметь свои преимущества и недостатки. Только опытный разработчик может сказать, какую из них будет проще расширять и обслуживать в дальнейшем.

Особенности ООП в Python

По сравнению со многими другими языками в Python объектно-ориентированное программирование обладает рядом особых черт.

Всё является объектом – число, строка, список, функция, экземпляр класса, сам класс, модуль. Так класс – объект, способный порождать другие объекты – экземпляры.

В Python нет просто типов данных. Все типы – это классы.

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 не считается предосудительным непосредственное обращение к свойствам.

И напоследок

Python – это все-таки скриптовый интерпретируемый язык. Хотя на нем пишутся в том числе крупные проекты, часто он используется в веб-разработке, системном администрировании для

создания небольших программ-сценариев. В этом случае обычно достаточно встроенных средств языка, "изобретать" собственные классы излишне.

Однако, поскольку в Python всё – объект и всё пронизано объектно-ориентированной парадигмой, понимание ООП позволит более полно и грамотно использовать возможности языка как инструмента разработки.