Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 8. Основы объектно-ориентированного программирования в Python

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

апрель, 2025

Python поддерживает три парадигмы программирования:

- *процедурная парадигма*
- *функциональная парадигма*
- **объектно-ориентированная парадигма**

Основными концециями объектно-ориентированной парадигмы являются понятия **объекта** и **класса**.

8.1 Понятия класса и экземпляра класса

Объект — это базовая сущность в Python для представления данных, обладающая определенным состоянием и поведением.

Класс — это способ описания множества *однотипных объектов* с одинаковыми свойствами и одинаковыми методами для совершения действий над объектами этого множества.

Встроенные типы данных (`int` , `float` , `complex` , `bool` , `str` , `tuple` , `list` , `dict` , `set`) являются *встроенными классами* в Python.

Пользовательский класс определяет НОВЫЙ ТИП ДАННЫХ, который не является встроенным типом в Python, и используется для *создания* объектов НОВОГО ТИПА. Объекты, созданные на основе класса, называются **экземплярами класса** (class instance).

Объектно-ориентированная парадигма — это программирование с применением экземпляров пользовательских классов.

С точки зрения программирования, класс — это способ группирования переменных, созданных явными присваиваниями, и функций с целью дальнейшего многократного их использования посредством создания экземпляров класса (объектов нового типа, созданного пользователем).

Классы группируют внутри себя переменные и функции, которые после создания экземпляра этого класса, становятся атрибутами экземпляра.

Атрибуты — это имена/переменные, которые связаны с объектом и используются для получения информации об объекте, а также для совершения действий над этим объектом.

Функциональные атрибуты объекта называются **методами**.

Нефункциональные атрибуты объектов класса и экземпляров класса будем в дальнейшем называть атрибутами.

8.2 Оператор создания класса

Для создания класса используется оператор `class`.

Оператор `class` является составным оператором, тело которого состоит из операторов явного присваивания `=` и операторов определения функций `def`.

Синтаксический шаблон оператора `class`:

```
class NameOfClass(...):
    atribute1 = ...
    ...
    atributeN = ...

    def method1(...):
        ...
    ...
    def methodM(...):
        ...
```

Рекомендовано начинать имя класса с буквы верхнего регистра (CapWords names).

При выполнении оператора `class` осуществляется *выполнение* операторов из тела класса, *создание* объекта класса и *неявное присваивание* объекта класса переменной `NameOfClass`.

In [2]:
```
class MyClass:
    """A simple example class"""
    var1 = 1
    var2 = 'second'
    def f1():
        var3 = 'local'
        return 'Method of MyClass'
```

Переменные и функции, определенные внутри оператора `class` *на верхнем уровне*, становятся атрибутами и методами созданного объекта класса

In [3]:
```
MyClass, MyClass.var1, MyClass.var2, MyClass.f1()
```

Out[3]:
```
(__main__.MyClass, 1, 'second', 'Method of MyClass')
```

Атрибуты объекта класса можно *изменять* за пределами оператора `class` с использованием оператора присваивания, т.е. механизм инкапсуляции на уровне языка не реализован

In [4]:
```
MyClass.var1 = 2; f'{MyClass.var1 = }'
```

Out[4]:
```
'MyClass.var1 = 2'
```

Атрибуты объекта класса можно *создавать* за пределами оператора `class` с использованием оператора присваивания для НОВОГО атрибута объекта класса

In [5]:
```
MyClass.var3 = 'new'; f'{MyClass.var3 = }'
```

Out[5]:
```
"MyClass.var3 = 'new'"
```

Просмотрим все атрибуты и методы класса `MyClass`, которые НЕ начинаются с символа подчеркивания

In [6]:
```
[x for x in dir(MyClass) if not x.startswith("_")]
```

Out[6]:
```
['f1', 'var1', 'var2', 'var3']
```

Строковый литерал строк документации записывается в атрибут `__doc__` объекта класса при выполнении оператора `class`

In [7]:
```
MyClass.__doc__
```

Out[7]:
```
'A simple example class'
```

С объектом класса в коде можно выполнить два действия: использовать/создать атрибуты и методы объекта класса и создать экземпляры класса.

Для создания *экземпляра класса* необходимо обратиться к объекту класса как к функции

In [8]:
```
i1 = MyClass(); i2 = MyClass()
i1, i2
```

Out[8]:
```
(<__main__.MyClass at 0x1d4117ceea0>, <__main__.MyClass at 0x1d40eb98e30>)
```

Объекты классов и объекты экземпляров классов являются объектами разных типов

In [9]:
```
type(MyClass), type(i1)
```

Out[9]:
```
(type, __main__.MyClass)
```

Атрибуты и методы объекта класса становятся атрибутами и методами для КАЖДОГО экземпляра класса. Говорят, что атрибуты/методы класса **наследуются** всеми экземплярами, созданными на основе класса

In [10]:
```
i1.var1, i2.var2
```

Out[10]:
```
(2, 'second')
```

Все экземпляры, созданные на основе класса, *разделяют* пространство имен этого класса. Это означает, что изменение значений атрибутов и методов объекта класса отражается на ВСЕХ экземплярах этого класса

In [11]:
```
MyClass.var1 = 10; i1.var1, i2.var1
```

Out[11]:
```
(10, 10)
```

Присваивание нового значения для атрибута класса через объект экземпляра класса создает новую одноименную переменную в *собственном пространстве имен* конкретного экземпляра класса. Изменение значения атрибута не отражается ни на объекте класса, ни на других экземплярах этого класса

In [12]:
```
i1.var1 = 5; print(MyClass.var1, i1.var1, i2.var1)
i1.var4 = 4
[x for x in dir(MyClass) if not x.startswith("_")]
```

```
10 5 10
```

Out[12]:
```
['f1', 'var1', 'var2', 'var3']
```

С экземпляром класса в коде можно выполнить следующие действия: использовать/создать атрибуты и методы, которые расположены в собственном пространстве имен экземпляра или которые унаследованы от класса. Механизм инкапсуляции на уровне языка Python не реализован.

8.3 Методы класса

Основным назначением методов класса является обработка экземпляров классов.

Для того, чтобы метод класса мог обрабатывать конкретный экземпляр класса, он/метод при определении ДОЛЖЕН содержать **специальный первый аргумент**, который по соглашению называется `self` (при определении метода аргумент `self` указан явно), для *неявной* передачи методу класса экземпляра класса, на котором был вызван метод (при вызове метода аргумент указан неявно).

In [14]:
```
class MyClass:
    def f1(self, x, y):
        return self, x, y
```

Аналогом `self` в Python является `this` в C++ и Java.

При вызове первый аргумент `self` заполняется *автоматически* для ссылки на экземпляр, на котором произведен вызов, для дальнейшей обработки экземпляра класса. Значения аргументов, перечисленные в круглых скобках при вызове метода БЕЗ ЯВНОГО УКАЗАНИЯ ЗНАЧЕНИЯ ДЛЯ SELF, сопоставляются со вторым и последующими аргументам метода класса.

In [15]:
```
i1 = MyClass(); i1, i1.f1(2,3)
```

Out[15]:
```
(<__main__.MyClass at 0x1d411774230>,
 (<__main__.MyClass at 0x1d411774230>, 2, 3))
```

Аргумент `self` метода класса *связывает* метод класса с экземпляром класса, на котором осуществляется вызов этого метода. Так как на основании одного класса может быть создано несколько экземпляров этого класса, каждый экземпляр класса должен обрабатываться одним и тем же методом уникально и независимо от других экземпляров класса.

Вызов `экземпляр.метод(аргументы)` *автоматически* отображается на вызов метода класса `класс.метод(экземпляр, аргументы)`, передавая `экземпляр` в первом аргументе унаследованной функции `метод`.

Объект метода класса для экземпляра класса, является не объектом функции, а **объектом связанного метода**. В объект связанного метода Python *автоматически* помещает экземпляр класса первым аргументом, связывая экземпляр класса и метод класса

In [16]:
```
MyClass.f1, i1.f1
```

Out[16]:
```
(<function __main__.MyClass.f1(self, x, y)>,
 <bound method MyClass.f1 of <__main__.MyClass object at 0x000001D411774230>>)
```

Через атрибут `__self__` объекта связанного метода можно получить доступ к экземпляру класса, на котором вызывается метод. Через атрибут `__func__` объекта связанного метода можно получить доступ к объекту функции класса, который метод реализует.

In [17]:
```
bound_method = i1.f1
bound_method.__self__, bound_method.__func__
```

Out[17]:
```
(<__main__.MyClass at 0x1d411774230>,
 <function __main__.MyClass.f1(self, x, y)>)
```

Если действия над экземплярами класса реализуются при определении класса с помощью методов класса, то это называется **концепцией инкапсуляции** (объединение данных и поведения + ограничение доступа к атрибутам). Код для обработки экземпляров класса пишется только один раз в виде метода класса, метод класса наследуется экземплярами класса при их создании и может использоваться каждым экземпляром класса по необходимости. Взаимодействие в коде с объектом осуществляется как с '*черным ящиком*' через методы объекта, когда внутренние детали реализации методов не важны. При этом атрибуты экземпляров не изменяются в коде напрямую, а только через методы. **Ограничение доступа к атрибутам не реализовано в Python на уровне языка.**

8.4 Метод инициализации экземпляра класса

Для создания экземпляра класса прежде всего нужно создать соответствующий класс, используя оператор `class <Имя_класса>(…)` . При выполнении оператора `class` будет создан объект класса и реализована ссылка переменной `<Имя_класса>` на этот объект класса.

Далее для создания экземпляра класса следует вызвать объект класса как функцию, указывая `<Имя_класса>` и аргументы вызова либо пустые круглые скобки.

Если после создания экземпляра класса необходимо *инициализировать* данные этого объекта, то оператор `class` должен содержать метод `__init__` (не всегда так бывает).

Метод `__init__` инициализирует экземпляр класса: он обязан при вызове метода передать значения аргументов атрибутам объекта.

При этом в операторе `def` метода `__init__` первым аргументом должен быть указан специальный аргумент `self` .

При вызове метода инициализации экземпляра класса `__init__` первый аргумент `self` *автоматически* становится ссылкой на объект созданного экземпляра. Значения аргументов, перечисленные в круглых скобках при вызове объекта класса как функции `<Имя_класса>(<аргумент1>,...,<аргументN>)` , сопоставляются со вторым и последующими аргументам метода класса `__init__(self, <аргумент1>,...,<аргументN>)` .

In [18]:
```
class MyClass:
    def __init__(self, who):
        self.name = who

i1 = MyClass('Inna'); i2 = MyClass('Oleg')
i1.name, i2.name
```

Out[18]:
```
('Inna', 'Oleg')
```

Присваивание атрибутам аргумента `self` *создает* атрибут экземпляра или *изменяет значение* существующего атрибута в экземпляре, но не в классе. Такие присваивания возможны только внутри методов классов.

Значения атрибутов для каждого экземпляра класса уникальные и не зависят от значений аналогичных атрибутов других экземпляров.

Анализируя код метода инициализации `__init__` , можно узнать имена атрибутов для экземпляров класса.

In [19]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def modulus(self):
        return (self.r**2 + self.i**2)**(1/2)
```

Экземпляры класса `Complex` будут иметь два атрибута `r` и `i` и один метод `modulus` , унаследованный от класса `Complex`

In [20]:
```
c1, c2 = Complex(1, 2), Complex(3)
c1.r, c2.i, c1.modulus(), c2.modulus()
```

Out[20]:
```
(1, 0, 2.23606797749979, 3.0)
```