Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» ММФ БГУ

Тема 8. Основы объектно-ориентированного программирования в Python

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

апрель, 2024

Python поддерживает три парадигмы программирования:

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

Основными концепциями 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.

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.

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

In [1]:
```
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 [2]:
```
MyClass, MyClass.var1, MyClass.var2, MyClass.f1()
```

Out[2]:
```
(__main__.MyClass, 1, 'second', 'Method of MyClass')
```

Атрибуты объекта класса можно *изменять* за пределами оператора `class` с использованием оператора присваивания

In [3]:
```
MyClass.var1 = 2; f'{MyClass.var1 = }'
```

Out[3]:
```
'MyClass.var1 = 2'
```

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

In [4]:
```
MyClass.var3 = 'new'; f'{MyClass.var3 = }'
```

Out[4]:
```
"MyClass.var3 = 'new'"
```

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

In [5]:
```
[x for x in dir(MyClass) if not x.startswith("_")]
```

Out[5]:
```
['f1', 'var1', 'var2', 'var3']
```

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

In [6]:
```
MyClass.__doc__
```

Out[6]:
```
'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 [7]:
```
i1 = MyClass(); i2 = MyClass()
i1, i2
```

Out[7]:
```
(<__main__.MyClass at 0x18c10d45090>, <__main__.MyClass at 0x18c10d47e90>)
```

Объекты классов и объекты экземпляров классов являются объектами разных типов

In [8]:
```
type(MyClass), type(i1)
```

Out[8]:
```
(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 [9]:
```
i1.var1, i2.var2
```

Out[9]:
```
(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 [10]:
```
MyClass.var1 = 10; i1.var1, i2.var1
```

Out[10]:
```
(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 [11]:
```
i1.var1 = 5; print(MyClass.var1, i1.var1, i2.var1)
i1.var4 = 4
[x for x in dir(MyClass) if not x.startswith("_")]
```

Out[11]:
```
10 5 10
['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.

8.3 Методы класса

Основным назначением методов класса является обработка экземпляров классов.

Для того, чтобы метод класса мог обрабатывать конкретный экземпляр класса, он/метод должен содержать **специальный первый аргумент**, который по соглашению называется `self`, для *автоматической* передачи методу класса экземпляра класса, на котором был вызван метод.

In [12]:
```
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 [13]:
```
i1 = MyClass(); i1, i1.f1(2,3)
```

Out[13]:
```
(<__main__.MyClass at 0x18c10d33750>,
 (<__main__.MyClass at 0x18c10d33750>, 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 [14]:
```
MyClass.f1, i1.f1
```

Out[14]:
```
(<function __main__.MyClass.f1(self, x, y)>,
 <bound method MyClass.f1 of <__main__.MyClass object at 0x0000018C10D33750>>)
```

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 [15]:
```
bound_method = i1.f1
bound_method.__self__, bound_method.__func__
```

Out[15]:
```
(<__main__.MyClass at 0x18c10d33750>,
 <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**. 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.

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 при построении 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 [16]:
```
class MyClass:
    def __init__(self, who):
        self.name = who

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

Out[16]:
```
('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 [17]:
```
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 [18]:
```
c1, c2 = Complex(1, 2), Complex(3)
c1.r, c2.i, c1.modulus(), c2.modulus()
```

Out[18]:
```
(1, 0, 2.23606797749979, 3.0)
```

8.5 Дерево наследования

Классы могут *наследоваться* друг от друга и образовывать дерево наследования. **Дерево наследования** -- это дерево связанных между собой объектов классов и экземпляров классов. В корне любого дерева наследования расположен объект *корневого класса* `object` .

Код называют объектно-ориентированным, когда объекты формируют дерево наследования.

Объект экземпляра класса *автоматически* связывается в дереве наследования с объектом класса, на основе которого он был создан.

Для указания связи экземпляра класса со своим классом в дереве наследования используется атрибут `__class__` . Атрибут `__class__` для экземпляра класса возвращает объект класса, на основе которого экземпляр был создан. Функция `type` от экземпляра класса также возвращает объект класса, на основе которого экземпляр был создан

In [19]:
```
c1.__class__, type(c1)
```

Out[19]:
```
(__main__.Complex, __main__.Complex)
```

Функция-предикат `isinstance(instance, (class1, class2, ..., classn))` проверяет, связан ли экземпляр класса `instance` с каким-нибудь из классов `class1, class2, ..., classn` в дереве наследования при движении по дереву снизу вверх от экземпляра

In [20]:
```
isinstance(c1, Complex), isinstance(c1, object)
```

Out[20]:
```
(True, True)
```

In [21]:
```
isinstance(c1, int), isinstance(c1, (int,Complex))
```

Out[21]:
```
(False, True)
```

Для рассматриваемого класса в дереве наследования все классы, расположенные выше рассматриваемого класса и связанные с ним, называются **суперклассами**; все классы, расположенные ниже рассматриваемого класса и связанные с ним, называются **подклассами**.

Подклассы предназначены для *специализированного* поведения,отличного от поведения суперклассов, но если подкласс имеет уникальную зависимость, которой нет у суперкласса или другого подкласса, используйт**е композиц**ию.

Связывание класса со своими суперклассами осуществляется *автоматически* при выполнении оператора `class` в соответствии с порядком перечисления наследуемых суперклассов в круглых скобок в строке заголовка оператора `class`

```
class Имя_класса(Имя_суперкласса1,...,Имя_суперклассаN):
    ...
```

Порядок перечисления суперклассов слева направо определяет порядок расположения суперклассов в дереве наследования.

Наличие у класса более одного суперкласса называется **множественным наследованием**.

При наследовании атрибуты и методы суперклассов становятся атрибутами и методами текущего класса. Классы **наследуют** атрибуты и методы своих суперклассов. Суперклассы **наследуют** атрибуты и методы своих суперклассов и т.д. при движении по дереву наследования до корневого класса снизу вверх и слева направо.

Экземпляр класса **наследует** атрибуты и методы ВСЕХ классов, которые связаны с экземпляром класса в дереве наследования.

При ссылке на атрибут/метод объекта класса или экземпляра класса выполняется поиск ПЕРВОГО вхождения атрибута/метода в дереве наследования. Сначала просматривается пространство имен объекта, а затем все классы выше него снизу вверх и слева направо. Поиск останавливается в первом месте, где найден атрибут/метод. Каждая ссылка на атрибут/метод запускает новую процедуру *восходящего поиска в дереве наследования*

In [22]:
```
class Class1: x = 1; y = 2
class Class2: x = 3; y = 4
class Class3(Class1, Class2): x = 5
Class3.x, Class3.y
```

Out[22]:
```
(5, 2)
```

Просмотреть порядок пространств имен для поиска атрибута класса можно с помощью атрибута `__mro__` этого класса

In [23]:
```
Class1.__mro__, Class3.__mro__
```

Out[23]:
```
((__main__.Class1, object),
 (__main__.Class3, __main__.Class1, __main__.Class2, object))
```

Экземпляры могут быть созданы для любого класса в дереве наследования, а не только для классов в нижней части дерева. Класс, на основе которого создается экземпляр, определяет уровень, откуда будет начинаться поиск значений атрибутов и версии методов, которые экземпляр будет задействовать.

In [24]:
```
i1 = Class1(); i2 = Class2(); i3 = Class3()
print(i1.x, i2.x, i3.x)
```

```
1 3 5
```

In [25]:
```
[isinstance(i3, cl) for cl in (object, Class1, Class2, Class3)]
```

Out[25]:
```
[True, True, True, True]
```

In [26]:
```
Class1.__bases__, Class2.__bases__, Class3.__bases__
```

Out[26]:
```
((object,), (object,), (__main__.Class1, __main__.Class2))
```

Функция-предикат `issubclass(subclass, class)` проверяет, связан ли класс `subclass` с классом `class` в дереве наследования при движении по дереву снизу вверх от класса `subclass`

In [27]:
```
issubclass(Class3, Class1), issubclass(Class3, object), issubclass(bool, (int, obje
```

Out[27]:
```
(True, True, True)
```

При определении класса без указания суперклассов объект корневого класса `object` задается в качестве суперкласса *автоматически*.

Следующие определения классов являются эквивалентными

In [28]:
```
class Class1():
    ...

class Class1(object):
    ...
```

Так как при поиске значения атрибута или метода "выигрывает" самая нижняя версия атрибута/метода, обход дерева наследования поддерживает **настройку кода** (customizing) путем переопределения методов суперкласса в подклассах. Методы подклассов замещают код одноименных методов суперклассов и тем самым *настраивают* унаследованное поведение. Метод суперклассов не изменяется.

Пример **настройки кода за счет *замены/специализации* метода** суперкласса одноименным методом класса

In [29]:
```
class Class1:
    def y(x): return "Class1"
class Class2:
    x = 3; y = 4
class Class3(Class1, Class2):
    def y(x): return "Class3"

i3 = Class3()
print(i3.y())
```

```
Class3
```

При движении по дереву наследования сверху вниз коды становятся все более специфичными.

Если методу подклассов нужна гарантия того, что также выполняется код одноименного метода суперкласса, тогда метод подкласса должен явно вызывать метод суперкласса внутри одноименного метода класса в виде `Суперкласс.метод(self,...)` *с явным указанием аргумента* `self` .

Пример **настройки кода за счет *расширения* кода метода** инициализации `__init__` суперкласса `Super` внутри одноименного метода класса `Sub`

In [30]:
```
class Super:
    def __init__(self, x):
        # стандартный код
        ...
        print(f'Super: {x=}')

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x) # super().__init__(self, x) или super(Sub,self).__i
        # специальный код
        ...
        print(f'Sub: {y=}')

i = Sub(1,2)
```

```
Super: x=1
Sub: y=2
```

В качестве суперклассов можно использовать классы для встроенных типов, *наследуя* и **настраивая встроенные типы** для конкретных задач

In [31]:
```
class MyList(list):
    def top(self):
        return self[-1]

my_list = MyList(range(5))
print(my_list, len(my_list), my_list[-1:0:-1], my_list.top(), sep = ", ")
```

```
[0, 1, 2, 3, 4], 5, [4, 3, 2, 1], 4
```

In [32]:
```
class Stack(list):
    def top(self):
        return self[-1]

    def push(self, item):
        list.append(self, item)

    def push1(self, item):
        self.append(item)

    def pop(self):
        if not self:
            return None
        else:
            return list.pop(self)

my_stack = Stack("Stack"); my_stack.push("New Element"); my_stack.push1("New Elemen
my_stack
```

Out[32]:
```
['S', 't', 'a', 'c', 'k', 'New Element', 'New Element']
```

Извлечение элементов из пустого стека с помощью метода `pop` не приведет к ошибке

In [33]:
```
[my_stack.pop() for _ in my_stack*2]
```

Out[33]:
```
['New Element',
 'New Element',
 'k',
 'c',
 'a',
 't',
 'S',
 None,
 None,
 None,
 None,
 None,
 None,
 None]
```

Извлечение элементов из пустого списка с помощью метода `pop` приведет к ошибке `IndexError`

In [34]:
```
list1 = list(range(5))
[list1.pop() for _ in list1*2]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[34], line 2
      1 list1 = list(range(5))
----> 2 [list1.pop() for _ in list1*2]

Cell In[34], line 2, in <listcomp>(.0)
      1 list1 = list(range(5))
----> 2 [list1.pop() for _ in list1*2]

IndexError: pop from empty list
```

8.6 Перегрузка операций и встроенных функций

ВСЕ встроенные операции в Python: бинарные операции (арифметические операции: `+`, `*` и др., операции сравнения: `<`, `==` и др.), операции индексации, операция вызова, операция принадлежности и др., а также встроенные функции `str`, `bool`, `len` и др., могут быть *переопределены* для поддержки вычислений с использованием экземпляров пользовательских классов.

Переопределение встроенных операций и встроенных функций с помощью методов класса называется **перегрузкой операций**.

Перегрузка операций чаще используется разработчиками новых инструментов для других программистов, а не разработчиками прикладных приложений.

Перегрузка операций часто применяется при реализации математических объектов.

Перегрузка операций реализуется в классе Python посредством определения специально именованных методов, которые называются **методами перегрузки операций**. Методы перегрузки операций содержат в начале и в конце своих имен по два символа подчеркивания (*dunder method*, dunder comes from 'double underscore' или *magic methods*), чтобы отличать их от других имен, определенных в классах. Например, `__init__`, `__add__` и др. Имена вида `__dunder_method__` не являются встроенными или зарезервированными. Специальный синтаксис имен методов перегрузки операций связан с предотвращением возможного случайного переопределения имен. В Python определено более 90 методов перегрузки операций.

Методы перегрузки операции вызывается *автоматически*, когда в выражении с соответствующей операцией встречаются экземпляры класса, для которых метод перегрузки операций определен.

Соответствие между операцией и именем метода перегрузки операций устанавливается *автоматически*.

Результатом выполнения операции является возвращаемое значение вызванного метода перегрузки операции.

Например, в выражении с операцией `+` над экземплярами одного и того же класса `int` вызывается метод `__add__`, определенный в классе `int`

In [35]:
```
help(int.__add__)
```

```
Help on wrapper_descriptor:

__add__(self, value, /)
    Return self+value.
```

In [36]:
```
a = 1; b = 2
a + b, a.__add__(b)
```

Out[36]:
```
(3, 3)
```

В следующем примере метод перегрузки операций `__add__` для экземпляров класса `Stack` наследуется от суперкласса `list`

In [37]:
```
my_stack = Stack("Stack")
my_stack + [1,2,3], my_stack.__add__([1,2,3])
```

Out[37]:
```
(['S', 't', 'a', 'c', 'k', 1, 2, 3], ['S', 't', 'a', 'c', 'k', 1, 2, 3])
```

Возвращаемое значение метода перегрузки операции становится результатом выполнения соответствующей операции.

Методы перегрузки операций не являются обязательными. Если метод перегрузки операций отсутствует при поиске в дереве наследования, тогда соответствующее выражение будет приводить к исключению `TypeError`

In [38]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
```

In [39]:
```
c1, c2 = Complex(2,1), Complex(3,2)
c1 + c2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[39], line 2
      1 c1, c2 = Complex(2,1), Complex(3,2)
----> 2 c1 + c2

TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

Определим метод `__add__` перегрузки операции `+` для класса `Complex`

In [40]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def __add__(self, other):
         if isinstance(other, Complex):
            return 'Should be defined'
```

In [41]:
```
c1, c2 = Complex(2,1), Complex(3,2)
c1 + c2
```

Out[41]:
```
'Should be defined'
```

Метод инициализации **init**

Метод инициализации `__init__` считается самым часто используемым методом перегрузки операций. *Метод инициализации* используется для инициализации атрибутов экземпляра класса значениями аргументов, указанными при создании экземпляра класса.

In [42]:
```
class MyClass:
    def __init__(self, who):
        self.name = who

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

Out[42]:
```
('Inna', 'Oleg')
```

Методы **str**, **repr** строкового представления экземпляра класса

Вторыми по частоте использования методами перегрузки операций после метода инициализации `__init__` являются *методы строкового представления* `__str__` и `__repr__` .

Метод строкового представления `__str__` определяет 'неформальное' представление экземпляра класса `instance` . Метод `__str__` вызывается *автоматически* при преобразовании экземпляра класса к строковому объекту с помощью вызова `str(instance)` , а также при вызове функции `print` .

Метод строкового представления `__str__` должен возвращать строковый объект.

Метод строкового представления `__repr__` определяет 'формальное' представление экземпляра класса `instance` , например в виде выражения Python, необходимого для создания экземпляра класса. Метод `__repr__` вызывается *автоматически* при вызове функции `repr(instance)` , для отображения в ячейках вывода, а также при вызове функции `str(instance)` , если метод `__str__` отсутствует.

Метод строкового представления `__repr__` должен возвращать строковый объект.

Пример. Класс `Complex` не содержит методов строкового представления, но наследует их от корневого класса `object` . Методы строкового представления класса `object` возвращают описание объекта экземпляра в виде строки

In [43]:
```
c1, str(c1), c1.__repr__()
```

Out[43]:
```
(<__main__.Complex at 0x18c11a33610>,
 '<__main__.Complex object at 0x0000018C11A33610>',
 '<__main__.Complex object at 0x0000018C11A33610>')
```

Определим методы `__str__` и `__repr__` перегрузки строкового представления для класса `Complex`

In [44]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def __str__(self):
        return f'{self.r}+{self.i}i'
    def __repr__(self):
        return f'Complex({self.r},{self.i})'
c1 = Complex(1,2)
```

In [45]:
```
print(c1, f'{c1!r}', complex(1,2))
str(c1), repr(c1), c1
```

Out[45]:
```
1+2i Complex(1,2) (1+2j)
('1+2i', 'Complex(1,2)', Complex(1,2))
```

Метод **call** вызова экземпляра класса

Метод вызова `__call__` считается третьим по частоте использования методом перегрузки операций после метода инициализации `__init__` и методов строкового представления `__str__` и `__repr__` .

Пример. Для класса `MyClass` определим метод инициализации `__init__` с одним атрибутом `name` , метод строкового представления `__str__` и метод вызова `__call__`

In [46]:
```
class MyClass:
    def __init__(self, who):
        self.name = who
    def __str__(self):
        return f'{self.name}'
    def __call__(self, *pargs, **kargs):
        print(f'Instance of class {type(self).__name__} with name {self} is called
```

In [47]:
```
i1 = MyClass('Inna'); i2 = MyClass('Oleg')
i1(1, 2, 3)
i2(4, y=5, z=6)
```

```
Instance of class MyClass with name Inna is called with arguments: (1, 2, 3), {}
Instance of class MyClass with name Oleg is called with arguments: (4,), {'y': 5, '
z': 6}
```

Методы **add**, **radd**, **iadd** операции сложения +

Для выполнения бинарной операции сложения над экземплярами произвольного класса необходимо определить метод `__add__` этого класса. При этом метод `__add__` будет вызываться для ЛЕВОГО операнда: вызов `a+b` *автоматически* заменяется на `a.__add__(b)` или точнее на `type(a).__add__(a,b)` .

Пусть метод `__add__` опрделеяется с аргументами `self` и `other` . Тогда первый аргумент `self` будет соответствовать левому операнду при выполнении операции сложения, а второй аргумент `other` -- правому операнду. Результатом сложения является НОВЫЙ экземпляр класса `Complex`

Пример. При реализации метода `__add__` для класса `Complex` полагаем, что левый и правый операнды операции сложения являются экземплярами класса `Complex` . Результатом сложения является НОВЫЙ экземпляр класса `Complex` .

In [48]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def __repr__(self):
        return f'Complex({self.r},{self.i})'
    def __add__(self, other):
         if isinstance(other, Complex):
            return Complex(self.r+other.r, self.i+other.i)
```

In [49]:
```
c1, c2 = Complex(2,1), Complex(3,2)
c1 + c2
```

Out[49]:
```
Complex(5,3)
```

Для выполнения бинарной операции сложения, когда экземпляром класса является только правый операнд, а левый операнд является экземпляром ДРУГОГО класса, необходимо определить метод `__radd__`, который будет вызываться для ПРАВОГО операнда

In [50]:
```
def f(self, other):
    if isinstance(other, (int, float)):
        return self + Complex(other)
    if isinstance(other, complex):
        return self + Complex(other.real, other.imag)

Complex.__radd__ = f
```

In [51]:
```
c1 = Complex(3,2)
1 + c1, 1 + 4j + c1
```

Out[51]:
```
(Complex(4,2), Complex(4.0,6.0))
```

Так как операция сложения реализуется двумя методами перегрузки `__add__` и `__radd__`, то вызывается либо метод `__add__` для ЛЕВОГО операнда или, при отсутствии метода `__add__`, вызывается метод `__radd__` для ПРАВОГО операнда. При отсутствии обоих методов возникает исключение `TypeError`.

При отсутствии метода `__iadd__` для перегрузки оператора дополненного присваивания будет вызываться метод `__add__`

In [52]:
```
c1 = Complex(2,1)
c1 += c1; print(c1)
```

```
Complex(4,2)
```

In [53]:
```
c1 = Complex(2,1)
c1 += 2
print(c1)
```

```
None
```

Пример.

In [54]:
```
def f(self, other):
    if isinstance(other, (int, float)):
        self.r += other
        return self
    if isinstance(other, Complex):
        self.r += other.r
        self.i += other.i
        return self

Complex.__iadd__= f
```

In [55]:
```
c1, c2 = Complex(2,1), Complex(3,2)
c1 += c2; print(c1)
c1 += 2; print(c1)
```

```
Complex(5,3)
Complex(7,3)
```

Каждая бинарная операция имеет похожие методы перегрузки, как и у операции сложения `__add__`, `__radd__`, `__iadd__`. Например, для вычислений с использованием операции умножения, можно определять методы `__mul__`, `__rmul__`, `__imul__` перегрузки операции умножения `*`.

К бинарным арифметическим операция относятся: `+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`

Методы операций сравнения

Каждая *операция сравнения* (`<`, `<=`, `==`, `!=`, `>=`, `>`) может быть перегружена для пользовательского класса с помощью методов `__lt__`, `__le__`, `__eq__`, `__ne__`, `__ge__`, `__gt__`, соответственно.

Все операции сравнения определены в корневом классе `object`, но реализации есть только для методов `__eq__` и `__ne__`. Сравнение на равенство объектов внутри метода `__eq__` реализовано с помощью сравнения `is`. Метод `__ne__` реализуется с использованием результата вызова метода `__eq__`

In [56]:
```
[x in dir(object) for x in ('__lt__', '__le__', '__eq__', '__ne__', '__ge__', '__gt
```

Out[56]:
```
[True, True, True, True, True, True]
```

In [57]:
```
c1 = Complex(2,1); c2 = Complex(2.,1.)
c1 == c1, c1 != c2
```

Out[57]:
```
(True, True)
```

In [58]:
```
c1>c2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 1
----> 1 c1>c2

TypeError: '>' not supported between instances of 'Complex' and 'Complex'
```

Пример. Для класса `Complex` определим метод инициализации `__init__` и метод перегрузки операции сравнения на равенство `__eq__`

In [59]:
```
c1 = Complex(2,1); c2 = Complex(2.,1.)
c1 == c1, c1 != c2
```

Out[59]:
```
(True, True)
```

In [60]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def __eq__(self, other):
        return self.r==other.r and self.i==other.i
```

In [61]:
```
c1 = Complex(2,1); c2 = Complex(2.,1.)
c1 == c1, c1 != c2
```

Out[61]:
```
(True, False)
```

В пользовательском классе достаточно определить только три метода перегрузки `__lt__`, `__le__`, `__eq__` для операций сравнения `<`, `<=`, `==`, соответственно. Остальные методы перегрузки `__ne__`, `__ge__`, `__gt__` для операций сравнения `!=`, `>=`, `>`, соответственно, будут наследоваться от класса `object` и выполняться с использованием трех методов операций сравнения, переопределенных (customized) внутри пользовательского класса.

Метод булевого значения **bool**

Метод `__bool__` вызывается *автоматически* при вызове встроенной функции `bool` и должен возвращать булевые значения `True` или `False`.

Булевое значение ЛЮБОГО экземпляра класса равно `True`, если внутри класса не определен метод `__bool__`.

In [62]:
```
bool(c1), bool(Complex(0))
```

Out[62]:
```
(True, True)
```

При использовании экземпляров классов в логических выражениях сначала вызывается метод `__bool__` для экземпляра класса. Если метод `__bool__` отсутствует, то булево значение объекта определяется по результату вызова метода `__len__` . Ненулевая длина объекта означает, что объект истинный, а нулевая -- что ложный. Если методы `__bool__` и `__len__` для класса не определены, тогда любые экземпляры класса считается истинными.

Пример. Переопределим класс `Complex` созданием методов `__init__` , `__abs__` , `__bool__`

In [63]:
```
bool(Complex(1)), bool(Complex())
```

Out[63]:
```
(True, True)
```

In [64]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart
    def __abs__(self):
        return (self.r**2 + self.i**2)**(1/2)
    def __bool__(self):
         return bool(abs(self))
```

In [65]:
```
bool(Complex(1)), bool(Complex())
```

Out[65]:
```
(True, False)
```

Метод `__len__` вызывается *автоматически* при вызове встроенной функции `len` .

Методы **getitem**, **setitem** операции индексации

Для выполнения *операции индексации* для экземпляра произвольного класса необходимо определить метод перегрузки операций `__getitem__` этого класса.

В случае использования экземпляра класса в выражении индексирования вида `<экземпляр>[i]` *автоматически* вызывается метод `__getitem__(self, i)` , унаследованный экземпляром, с передачей экземпляра в первом аргументе и индекса, указанного в квадратных скобках, во втором аргументе метода `__getitem__` . В дополнение к позиционному индексированию метод `__getitem__` поддерживает также индексирование с использованием среза `<экземпляр>[i:j]` и `<экземпляр>[i:j:k]`

Пример. Пользовательский класс `MyClass` с методами `__init__` и `__getitem__`

In [66]:
```
class MyClass:
    def __init__(self, data):
        self.data = data
    def __getitem__(self, index):
        if isinstance(self.data, (str, tuple, list)):
            return index, self.data[index]
        else:
            return self.data
```

In [67]:
```
i1 = MyClass([1,2,3,4]); i2 = MyClass(2)
i1[0], i1[:], i1[-1:0:-1], i2[5]
```

Out[67]:
```
((0, 1),
 (slice(None, None, None), [1, 2, 3, 4]),
 (slice(-1, 0, -1), [4, 3, 2]),
 2)
```

Для выполнения присваивания индексированному элементу экземпляра произвольного класса вида `<экземпляр>[i] = value` или `<экземпляр>[i:j] = value` или `<экземпляр>[i:j:k] = value` необходимо определить метод перегрузки операций `__setitem__` этого класса. Первый аргумент метода `__setitem__` содержит экземпляр класса, второй аргумент -- индекс, указанный в квадратных скобках, третий аргумент -- присваиваемое значение

Пример. Пользовательский класс `MyClass` с методами `__init__`, `__repr__` и `__setitem__`

In [68]:
```
class MyClass:
    def __init__(self, data):
        self.data = data
    def __repr__(self):
        return f'{self.data}'
    def __setitem__(self, index, value):
        if isinstance(self.data, list):
            self.data[index] = value
        else:
            self.data = value
```

In [69]:
```
i1 = MyClass([1,2,3,4]); i2 = MyClass(2)
i1[0:2] = [4, 4]; i2[4] = 12
i1, i2
```

Out[69]:
```
([4, 4, 3, 4], 12)
```

Пользовательский класс Complex

Реализация пользовательского класса `Complex`, представленная на лекции, определен с помощью ДЕВЯТИ методов перегрузки операций: метод инициализации `__init__`, методы строкового представления `__str__`, `__repr__`, методы перегрузки операции сложения `__add__`, `__radd__`, `__iadd__`, метод перегрузки операции сравнения `__eq__`, методы перегрузки вызовов функций `__abs__`, `__bool__`

In [70]:
```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.r, self.i = realpart, imagpart

    def __str__(self):
        return f'{self.r}+{self.i}i'

    def __repr__(self):
        return f'Complex({self.r},{self.i})'

    def __add__(self, other):
         if isinstance(other, Complex):
            return Complex(self.r+other.r, self.i+other.i)

    def __radd__(self, other):
         if isinstance(other, (int, float)):
            return self + Complex(other)
         if isinstance(other, complex):
            return self + Complex(other.real, other.imag)

    def __iadd__(self, other):
         if isinstance(other, (int, float)):
            self.r += other
            return self
         if isinstance(other, Complex):
            self.r += other.r; self.i += other.i
            return self

    def __eq__(self, other):
        return self.r==other.r and self.i==other.i

    def __abs__(self):
        return (self.r**2 + self.i**2)**(1/2)

    def __bool__(self):
        return bool(abs(self))
```