Компьютерная математика II

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

Тема 10. Проектирование классов

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

май, 2024

10.1 Особенности именования атрибутов и методов класса

Имя атрибута или и методв внутри оператора `class` , которое начинается с двух символов подчеркивания, но не заканчиваеются ими, *автоматически* изменяется добавлениием имени класса в начало имени атрибута или метода. Например, имя `__name` внутри класса `MyClass` *автоматически* заменяется на имя `_MyClass__name` :

In [1]:
```
class MyClass:
    __name = None

i1 = MyClass()
print(i1._MyClass__name)
i1.__name
```

```
None
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[1], line 6
      4 i1 = MyClass()
      5 print(i1._MyClass__name)
----> 6 i1.__name

AttributeError: 'MyClass' object has no attribute '__name'
```

In [ ]:
```
print(dir(i1))
```

Такой механизм именования внутри операторов класса делает имена, начинающиеся с двух символов подчеркивания, но не заканчивающиеся ими, *частными именами*, уникальными и не конфликтующими с аналогичными именами других классов в дереве наследования.

10.2 Классовые методы и статические методы

Классовый метод -- это метод, который вызывается как для объекта класса, так и для экземпляра класса и связывается с объектом КЛАССА, на котором метод был вызван. Классовый метод содержит *автоматически* заполняемый первый аргумент `cls` для связывания с объектом КЛАССА, на котором метод был вызван.

Для определения классового метода используется декоратор `@classmethod` перед оператором `def` .

In [2]:
```
class Class1():
    @classmethod
    def name(cls):
        return f'function is called from {cls.__name__}'

class Class2(Class1):
    ...

Class1.name(), Class2().name()
```

Out[2]:
```
('function is called from Class1', 'function is called from Class2')
```

Статический метод -- это метод, который вызывается как для объекта класса, так и для экземпляра класса и НЕ связывается с объектом, на котором метод был вызван. Статический метод НЕ содержит *автоматически* заполняемого первого аргумента для связывания с объектом, на котором метод был вызван. Статический метод является *обычной функцией*, которая располагается в пространстве имен класса. *Статический метод связан с функциональностью класса, но не зависит от данных класса или экземпляра.*

Для определения статического метода используется декоратор `@staticmethod` перед оператором `def` .

In [3]:
```
class Class1:
    @staticmethod
    def name():
        return 'static method'

Class1.name(), Class1().name()
```

Out[3]:
```
('static method', 'static method')
```

Одним из применений статических методов и классовых методов является определение классов с различными способами создания экземпляров

In [4]:
```
import time

class Date():
    def __init__(self, day, month, year):
        self.day = day; self.month = month; self.year = year

    @staticmethod
    def now():
        t = time.localtime()
        return Date(t.tm_mday, t.tm_mon, t.tm_year)

    def __repr__(self):
        return f'{self.day}.{self.month}.{self.year}'
```

In [5]:
```
date1 = Date(day=10,month=11,year=2012); date2 = Date.now()
date1, date2
```

Out[5]:
```
(10.11.2012, 10.5.2024)
```

Альтернативное определение класса можно реализовать с использованием классового метода

In [6]:
```
class Date():
    def __init__(self, day, month, year):
        self.day = day; self.month = month; self.year = year

    @classmethod
    def now(cls):
        t = time.localtime()
        return cls(t.tm_mday, t.tm_mon, t.tm_year)

    def __repr__(self):
        return f'{self.day}.{self.month}.{self.year}'
```

In [7]:
```
date1 = Date(day=10,month=11,year=2012); date2 = Date.now()
date1, date2
```

Out[7]:
```
(10.11.2012, 10.5.2024)
```

При таком определении метод `now` будет корректно отрабатывать для подклассов класса `Date`

In [8]:
```
class MyDate(Date):
    def __repr__(self):
        return f'{self.day} - {self.month} - {self.year}'

MyDate.now()
```

Out[8]:
```
10 - 5 - 2024
```

10.3 Атрибут-свойство

При определении класса можно задать **атрибут-свойство**, который определяется внутри класса, как метод, но используется, как атрибут.

- Для определения атрибута-свойства с именем `attr_name` используется декоратор `@property` перед методом с аналогичным именем `attr_name(self)` для доступа к значению атрибута-свойства.
- Декоратор `@attr_name.setter` используется перед методом `attr_name(self, value)` для изменения значения атрибута-свойства `attr_name`.
- Декоратор `@attr_name.deleter` используется перед методом `attr_name(self)` для удаления атрибута-свойства `attr_name`.

In [9]:
```
import math

class Class1:
    def __init__(self):
        self._x = None

    @property # с методом можно будет работать, как с атрибутом
    def x_squared(self):
        """squared value of class attribute"""
        return None if self._x is None else self._x**2

    @x_squared.setter
    def x_squared(self, value):
        self._x = math.sqrt(value)

    @x_squared.deleter
    def x_squared(self):
        print('NotImplemented')
```

После создания экземпляра класса `Class1` можно выполнить доступ к атрибуту-свойству класса `x_squared`, связанного с атрибутом класса `_x`, изменить значение атрибута-свойства `x_squared` и удалить атрибут, связанный с атрибутом-свойством класса `x_squared`

In [10]:
```
i = Class1()
print(i.x_squared)
i.x_squared = 25
print(i._x)
del i.x_squared
```

```
None
5.0
NotImplemented
```

Атрибут-свойство `attr_name` определяется с помощью трех одноименных методов `attr_name` класса с различными декораторами (`@property`, `@attr_name.setter`, `@attr_name.deleter`), а вызывается как атрибут экземпляров класса `I.attr_name`.

In [11]:
```
help(Class1)
```

```
Help on class Class1 in module __main__:

class Class1(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  x_squared
 |      squared value of class attribute
```

10.4 Абстрактный класс

Абстрактный класс -- это класс, который содержит **абстрактный метод**. Тело абстрактного метода не определено в дереве наследования: ни внутри самого класса, ни внутри его суперклассов.

Вызов абстрактного метода для экземпляра, созданного на основе абстрактного класса, приведет к ошибке.

Определение *неопределенного метода* ожидается в подклассе абстрактного класса. Вызов абстрактного метода будет обрабатываться корректно для экземпляра подкласса

In [12]:
```
class Super:
    def delegate(self):
        self.action()

    # абстрактный метод с выдачей сообщения об ошибке, если метод не переопределен
    def action(self):
        assert False, "Action must be defined!"

class Provider(Super):
    def action(self): # абстрактный метод определен в подклассе
        print("Action in Provider")

x = Provider()
x.delegate()
```

```
Action in Provider
```

Абстрактные классы и абстрактные методы используются в том случае, когда функциональность абстрактного метода может быть определена только на уровне подклассов. Абстрактный класс указывает, какие методы должны быть реализованы его подклассами.

Например, модуль `numbers` содержит *иерархию абстрактных классов*, представляющих различные числовые типы

In [13]:
```
import numbers

print(dir(numbers))
```

```
['ABCMeta', 'Complex', 'Integral', 'Number', 'Rational', 'Real', '__all__', '__buil
tins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package_
_', '__spec__', 'abstractmethod']
```

In [14]:
```
numbers.Number?
```

```
Init signature: numbers.Number()
Docstring:
All numbers inherit from this class.

If you just want to check if an argument x is a number, without
caring what kind, use isinstance(x, Number).
File:           c:\users\olga\anaconda3\lib\numbers.py
Type:           ABCMeta
Subclasses:     Complex
```

In [15]:
```
i1 = numbers.Complex()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 i1 = numbers.Complex()

TypeError: Can't instantiate abstract class Complex with abstract methods __abs__,
__add__, __complex__, __eq__, __mul__, __neg__, __pos__, __pow__, __radd__, __rmul_
_, __rpow__, __rtruediv__, __truediv__, conjugate, imag, real
```

Модуль `abc` пакета `collections` предоставляет *множество абстрактных классов* для создания собственных коллекций, расширяя возможности встроенных типов данных

In [16]:
```
import collections.abc as abc

print(dir(abc))
```

```
['AsyncGenerator', 'AsyncIterable', 'AsyncIterator', 'Awaitable', 'ByteString', 'Ca
llable', 'Collection', 'Container', 'Coroutine', 'Generator', 'Hashable', 'ItemsVie
w', 'Iterable', 'Iterator', 'KeysView', 'Mapping', 'MappingView', 'MutableMapping',
'MutableSequence', 'MutableSet', 'Reversible', 'Sequence', 'Set', 'Sized', 'ValuesV
iew', '_CallableGenericAlias', '__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__']
```

In [17]:
```
help(abc.Iterable)
```

```
Help on class Iterable in module collections.abc:

class Iterable(builtins.object)
 |  Methods defined here:
 |
 |  __iter__(self)
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__ = GenericAlias(...) from abc.ABCMeta
 |      Represent a PEP 585 generic type
 |
 |      E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).
 |
 |  __subclasshook__(C) from abc.ABCMeta
 |      Abstract classes can override this to customize issubclass().
 |
 |      This is invoked early on by abc.ABCMeta.__subclasscheck__().
 |      It should return True, False or NotImplemented.  If it returns
 |      NotImplemented, the normal algorithm is used.  Otherwise, it
 |      overrides the normal algorithm (and the outcome is cached).
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset({'__iter__'})
```

Описание иерархии классов для представления двоичного дерева поиска можно реализовать с помощью абстрактного класса `AbstractNode`. Класс `AbstractNode` будет использоваться в качестве суперкласса для классов `EmptyNode` и `BinaryNode`

In [18]:
```python
from abc import ABC, abstractmethod

class AbstractNode(ABC): # указываем, что класс абстрактный
    @abstractmethod # помечаем методы, как абстрактные
    def insert(self, value):
        """добавить новый элемент со значением value в дерево"""
    @abstractmethod
    def lcr(self):
        """создать список значений вершин дерева при центрированном обходе дерева""
```

In [19]:
```python
AbstractNode()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 AbstractNode()

TypeError: Can't instantiate abstract class AbstractNode with abstract methods inse
rt, lcr
```

In [20]:
```
class EmptyNode(AbstractNode):
    def __repr__(self):
        return '*'

    def insert(self, value):
        return BinaryNode(self, value, self)

    def lcr(self):
        return []

EmptyNode()
```

Out[20]:
```
*
```

In [21]:
```
help(EmptyNode)
```

```
Help on class EmptyNode in module __main__:

class EmptyNode(AbstractNode)
 |  Method resolution order:
 |      EmptyNode
 |      AbstractNode
 |      abc.ABC
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  insert(self, value)
 |      добавить новый элемент со значением value в дерево
 |
 |  lcr(self)
 |      создать список значений вершин дерева при центрированном обходе дерева
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from AbstractNode:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

10.5 Синглтон: шаблон создания объектов

Иногда при определении класса необходимо гарантировать, чтобы на основе класса мог быть создан только один экземпляр. Такой класс будет называться **синглтоном**.

Возможна следующая реализация: сначала создается вспомогательный класс `Singleton` с единственным атрибутом класса `instance` и единственным методом `__new__` для создания экземпляра класса `Singleton`

In [22]:
```
class Singleton:
    instance = None

    def __new__(cls, *args, **kwarg):
        if cls.instance is None:
            cls.instance = object.__new__(cls)
        return cls.instance
```

Метод `__new__` контролирует, чтобы был создан только один экземпляр класса. Атрибут `instance` хранит ссылку на единственный экземпляр класса.

Далее создается ЛЮБОЙ класс, который наследуется от `Singleton` , чтобы экземпляр этого класса был единственным объектом

In [23]:
```
class Single(Singleton):
    def __init__(self, val1, val2):
        self.attr1 = val1
        self.attr2 = val2
```

Создадим два экземпляра класса `Single`

In [24]:
```
i1 = Single(2, 3); i2 = Single(3, 4)
```

Два экземпляра ссылаются на одну область памяти

In [25]:
```
i1, i2, i1 is i2
```

Out[25]:
```
(<__main__.Single at 0x2b2ecb4d310>, <__main__.Single at 0x2b2ecb4d310>, True)
```

Значения атрибутов для двух экземпляров совпадают и определяются значениями экземпляра, созданного последним

In [26]:
```
i1.attr1, i1.attr2
```

Out[26]:
```
(3, 4)
```

10.6 Композиция

Композиция -- это механизм связывания классов, который отличается от наследования.

Класс реализует композицию, если экземпляр класса содержит ссылки на объекты (**внедренные объекты**) других пользовательских классов.

Экземпляр класса, который содержит ссылки на внедренные объекты других пользовательских классов, называется **объектом-контейнером**.

In [27]:
```
class Whole:
    def __init__(self, data1, data2):
        self.part1 = Part(data1) # внедренный объект
        self.part2 = Part(data2) # внедренный объект
    def __repr__(self):
        return f'''{self.__class__.__name__} contain
        {self.part1} and {self.part2}'''

class Part:
    def __init__(self, data):
        self.data = data
    def __repr__(self):
        return f'{self.__class__.__name__} with data: {self.data}'
```

In [28]:
```
i1 = Whole(1,2) # объект-контейнер
i1
```

Out[28]:
```
Whole contain
        Part with data: 1 and Part with data: 2
```

Делегирование

Делегирование -- это разновидность композиции для создания **класса-оболочки** (Wrapper), когда экземпляр класса-оболочки содержит *единственный внедренный объект* (wrapped). При этом сохраняется бОльшая часть интерфейса внедренного объекта и добавляется дополнительное поведение для внедренного объекта. Поведение внедренного объекта '*оборачивается*' дополнительным поведением, реализованным при определении класса-оболочки.

In [29]:
```
class Wrapper:
    def __init__(self, obj):
        self.wrapped = obj
    def __getattr__(self, attrname):
        print(f'Trace: {attrname}')
        return getattr(self.wrapped, attrname)

x = Wrapper([1,2,3])
x.pop(); x.append(10); x.wrapped
```

Out[29]:
```
Trace: pop
Trace: append
[1, 2, 10]
```

В приведенном примере экземпляр класса-оболочки `Wrapper` *делегирует* выполнение действий внедренному объекту `wrapped`, осуществляя дополнительный вывод `print(f'Trace: {attrname}')` перед выполнением методов объекта `wrapped`.

10.7 Метаклассы