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 10. Проектирование классов

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

май, 2025

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 [4]:
```
class MyClass:
    __name = None

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

```
None
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[4], line 6
      4 i1 = MyClass()
      5 print(i1._MyClass__name)
----> 6 i1.__name

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

In [7]:
```
print(dir(i1))
```

```
['_MyClass__name', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '_
_ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', '__weakref__']
```

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 [12]:
```
class Class1():
    @classmethod
    def name(cls):
        return f'function is called from {cls.__name__}'

class Class2(Class1):
    ...

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

Out[12]:
```
('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 данных экземпляра класса.*

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

In [15]:
```
class Class1:
    @staticmethod
    def name():
        return 'static method'

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

Out[15]:
```
('static method', 'static method')
```

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

In [18]:
```
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 [20]:
```
date1 = Date(day=10,month=11,year=2012); date2 = Date.now()
date1, date2
```

Out[20]:
```
(10.11.2012, 15.5.2025)
```

Альтернативное определение класса можно реализовать с использованием классового метода

In [23]:
```
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 [25]:
```
date1 = Date(day=10,month=11,year=2012); date2 = Date.now()
date1, date2
```

Out[25]:
```
(10.11.2012, 15.5.2025)
```

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

In [28]:
```
class MyDate(Date):
    def __repr__(self):
        return f'{self.day} - {self.month} - {self.year}'

MyDate.now()
```

Out[28]:
```
15 - 5 - 2025
```

10.3 Атрибут-свойство

При определении класса можно задать **атрибут-свойство**, который определяется внутри класса, как метод (`def prop(self): ...`), но используется для экземпляра класса, как атрибут (`i.prop`). При этом для атрибута-свойства можно специальным образом обработать доступ (*геттер*), изменение (*сеттер*) и удаление.

- Для определения атрибута-свойства с именем `prop` используется декоратор `@property` перед методом с аналогичным именем `prop(self)` для доступа к значению атрибута-свойства `i.prop` (реализация *геттера*).
- Декоратор `@prop.setter` используется перед методом `prop(self, value)`, который выполняется, когда в коде записано `i.prop = value` (реализация *сеттера*).
- Декоратор `@prop.deleter` используется перед методом `prop(self)` который выполняется, когда в коде записано `del i.prop`.

In [32]:
```
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 [35]:
```
i = Class1()
print(i.x_squared)
i.x_squared = 25
print(i._x)
del i.x_squared
```

```
None
5.0
NotImplemented
```

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

In [38]:
```
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
 |
 |  __weakref__
 |      list of weak references to the object
 |
 |  x_squared
 |      squared value of class attribute
```

Используя синтаксис атрибутов-свойств, можно *запретить изменение* атрибутов-свойств за счет отсутствия реализации метода сеттера

In [40]:
```python
class Class1:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        return _x

    @x.deleter
    def x(self):
        del _x

i = Class1()
i._x = 5
#i.x = 5
i.__dict__
```

Out[40]:
```
{'_x': 5}
```

10.4 Абстрактный класс

Абстрактный класс -- это класс, который определяется как суперкласс для других классов и содержит абстрактные методы. **Абстрактный метод** -- это метод класса, тело которого не определено в дереве наследования: ни внутри самого класса, ни внутри его суперклассов. Определение абстрактного метода ожидается в подклассе абстрактного класса.

Вызов абстрактного метода для экземпляра, созданного на основе абстрактного класса, приведет к ошибке. Вызов абстрактного метода будет обрабатываться корректно для экземпляра подкласса.

In [46]:
```
class Abstract:
    # абстрактный метод с выдачей сообщения об ошибке, если метод не переопредел
    def action(self):
        raise TypeError("Action must be defined in subclasses")

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

x = Provider().action()
try:
    Abstract().action()
except TypeError as e:
    print(e)
```

```
Action in Provider
Action must be defined in subclasses
```

Альтернативная реализация предыдущего примера с использованием класса `ABC` и функции декоратора `abstractmethod` модуля `abc` из стандарнтной библиотеки:

In [48]:
```
from abc import ABC, abstractmethod

class Abstract(ABC): # указываем, что класс абстрактный
    @abstractmethod # помечаем методы, как абстрактные
    def action(self):
        ...

class Provider(Abstract):
    def action(self):
        print("Action in Provider")

x = Provider().action()
try:
    Abstract().action()
except TypeError as e:
    print(e)
```

```
Action in Provider
Can't instantiate abstract class Abstract without an implementation for abstract
method 'action'
```

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.

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 [51]:
```
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 дерев
```

In [52]:
```
try:
    AbstractNode()
except TypeError as e:
    print(e)
```

```
Can't instantiate abstract class AbstractNode without an implementation for abstr
act methods 'insert', 'lcr'
```

In [53]:
```
class EmptyNode(AbstractNode):
    def __repr__(self):
        return '*'

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

    def lcr(self):
        return []

EmptyNode()
```

Out[53]: *

In [54]:
```
help(EmptyNode.insert)
```

```
Help on function insert in module __main__:

insert(self, value)
    добавить новый элемент со значением value в дерево
```

Модуль `numbers` содержит абстрактные классы, представляющие иерархию различных числовых типов

In [62]:
```
import numbers

print(dir(numbers))
```

```
['ABCMeta', 'Complex', 'Integral', 'Number', 'Rational', 'Real', '__all__', '__bu
iltins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__pack
age__', '__spec__', 'abstractmethod']
```

In [64]:
```
numbers.Complex?
```

```
Init signature: numbers.Complex()
Docstring:
Complex defines the operations that work on the builtin complex type.

In short, those are: a conversion to complex, .real, .imag, +, -,
*, /, **, abs(), .conjugate, ==, and !=.

If it is given heterogeneous arguments, and doesn't have special
knowledge about them, it should fall back to the builtin complex
type as described below.
File:           c:\users\user\anaconda3\lib\numbers.py
Type:           ABCMeta
Subclasses:     Real
```

In [65]:
```
try:
    numbers.Complex()
except TypeError as e:
    print(e)
```

```
Can't instantiate abstract class Complex without an implementation for abstract m
ethods '__abs__', '__add__', '__complex__', '__eq__', '__mul__', '__neg__', '__po
s__', '__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 [67]:
```
import collections.abc as abc

print(dir(abc))
```

```
['AsyncGenerator', 'AsyncIterable', 'AsyncIterator', 'Awaitable', 'Buffer', 'Byte
String', 'Callable', 'Collection', 'Container', 'Coroutine', 'Generator', 'Hashab
le', 'ItemsView', 'Iterable', 'Iterator', 'KeysView', 'Mapping', 'MappingView',
'MutableMapping', 'MutableSequence', 'MutableSet', 'Reversible', 'Sequence', 'Se
t', 'Sized', 'ValuesView', '_CallableGenericAlias', '__all__', '__builtins__', '_
_cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__sp
ec__']
```

10.5 Синглтон: шаблон проектирования

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` .

Метод `__new__` вызывается для *создания* экземпляра класса, его первым аргументом является ссылка на класс `cls` , остальные аргументы соответствуют аргументам при вызове класса, как функции. Метод `__new__` должен возвращать созданный объект. После метода `__new__` вызывается метод `__init__` *инициализации* экземпляра класса.

In [75]:
```
class Singleton:
    instance = None

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

Метод `__new__` в классе `Singleton` контролирует с помощью атрибута класса `instance`, чтобы был создан только один экземпляр класса, возвращая существующий экземпляр вместо создания нового экземпляра. Атрибут `instance` хранит ссылку на единственный экземпляр класса.

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

In [78]:
```
class Single(Singleton):
    def __init__(self, val1, val2):
        self.attr1 = val1
        self.attr2 = val2
```

Создадим два экземпляра класса `Single`

In [80]:
```
i1 = Single(2, 3); i2 = Single(3, 4)
```

Два экземпляра ссылаются на одну область памяти

In [85]:
```
i1 is i2
```

Out[85]:
```
True
```

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

In [88]:
```
i1.__dict__, i2.__dict__
```

Out[88]:
```
({'attr1': 3, 'attr2': 4}, {'attr1': 3, 'attr2': 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 [94]:
```
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 [96]:
```
i1 = Whole(1,2) # объект-контейнер
i1
```

Out[96]:
```
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 [100…
```
class Wrapper:
    def __init__(self, obj):
        self.wrapped = obj
    def __getattr__(self, attrname):
        print(f'Trace: {attrname}')
        return getattr(self.wrapped, attrname)
    def __repr__(self):
        return repr(self.wrapped)

x = Wrapper([1,2,3])
print(f'{x = }'); x.pop(); print(f'{x = }'); x.append(10); print(f'{x = }')
```

```
x = [1, 2, 3]
Trace: pop
x = [1, 2]
Trace: append
x = [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 Класс данных

Классы данных (data classes) появились в Python 3.7. Класс данных определяется с помощью декоратора `@dataclass` из модуля `dataclasses`

In [104…
```python
from dataclasses import dataclass

@dataclass
class ClassData:
    attr1: str # аннотация типа обязательна
    attr2: int
    attr3: tuple = 0,

    def f(self):
        print(self.__dict__)
```

В классе данных метод инициализации `__init__` создается *автоматически*. Порядок следования аргументов при создании экземпляра класса должен совпадать с порядком следования атрибутов при определении класса. Значения атрибутов класса являются значениями по умолчанию при создании экземпляров класса:

In [107…
```python
iData = ClassData('s', 1)
iData.f()
```

```
{'attr1': 's', 'attr2': 1, 'attr3': (0,)}
```

В классе данных метод строкового представления `__repr__` создается *автоматически*:

In [111…
```python
f'{iData = }'
```

Out[111…
```
"iData = ClassData(attr1='s', attr2=1, attr3=(0,))"
```

В классе данных метод сравнения на равенство `__eq__` создается *автоматически*. Метод возвращает `True`, когда одноименные атрибуты экземпляров ссылаются на объекты с одинаковым значением:

In [114…
```python
i1 = ClassData('1', 1, (1,))
i2 = ClassData('1', 1, (1,))
i1 == i2
```

Out[114…
```
True
```

Для создания атрибутов, ссылающихся на изменяемые типы, необходимо использовать функцию `field` из модуля `dataclasses` с аргументом `default_factory`:

In [117…
```python
from dataclasses import field

try:
    @dataclass
    class ClassData:
        attr1: list = field(default_factory=[])
        attr2: list = []
except ValueError as e:
    print(e)

i1 = ClassData()
i1
```

```
mutable default <class 'list'> for field attr2 is not allowed: use default_factor
y
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[117], line 11
      8 except ValueError as e:
      9     print(e)
---> 11 i1 = ClassData()
     12 i1

TypeError: ClassData.__init__() missing 2 required positional arguments: 'attr1'
and 'attr2'
```

Для создания вычисляемого атрибута необходимо задать значение вычислямого атрибута в классе с использованием функции `field`, как `field(init=False)`, чтобы атрибут не нужно было определять при создании экземпляра класса, и определить метод `__post_init__(self)` для вычисления значения атрибута:

In [120…

```
@dataclass
class Vector:
    x: float = 0.
    y: float = 0.
    # атрибут не задается при инициализации
    length: float = field(init=False)

    def __post_init__(self):
        self.length = (self.x**2 + self.y**2)**0.5
vec = Vector(3,4)
vec
```

Out[120…

```
Vector(x=3, y=4, length=5.0)
```

Мы познакомились с назначением аргументов `default_factory` и `init` функции `field`. Использование аргументов `default`, `repr`, `compare` функции `field` также являются распространенным

In [123…

```
?field
```

```
Signature:
field(
    *,
    default=<dataclasses._MISSING_TYPE object at 0x000001AEDEE5EC00>,
    default_factory=<dataclasses._MISSING_TYPE object at 0x000001AEDEE5EC00>,
    init=True,
    repr=True,
    hash=None,
    compare=True,
    metadata=None,
    kw_only=<dataclasses._MISSING_TYPE object at 0x000001AEDEE5EC00>,
)
Docstring:
Return an object to identify dataclass fields.

default is the default value of the field.  default_factory is a
0-argument function called to initialize a field's value.  If init
is true, the field will be a parameter to the class's __init__()
function.  If repr is true, the field will be included in the
object's repr().  If hash is true, the field will be included in the
object's hash().  If compare is true, the field will be used in
comparison functions.  metadata, if specified, must be a mapping
which is stored but not otherwise examined by dataclass.  If kw_only
is true, the field will become a keyword-only parameter to
__init__().

It is an error to specify both default and default_factory.
File:      c:\users\user\anaconda3\lib\dataclasses.py
Type:      function
```

Использование классов данных сокращает код за счет автоматической реализации методов инициализации, строкового представления и сравнения, но не добавляет классу новой функциональности. Все автоматически созданные методы можно переопределять внутри класса.

10.8 Декоратор класса и декоратор метода класса

Декораторы классов, с которыми мы познакомились:

`@dataclass` из модуля `dataclasses`

Декораторы методов, с которыми мы познакомились:

`@classmethod`

`@staticmethod`

`@property`, `@prop.setter`, `@prop.deleter`

`@abstractmethod` из модуля `abc`

10.9 Метакласс (не описан)