МФТИ

Объектно-Ориентированное программирование на Python

Докладчик: Евграфов Михаил

Инкапсуляция

Публичные и служебные атрибуты

```
class MyClass:
    name: str
    _name: str
    __name: str

    def __init__(self, name: str) -> None:
        self.name = self._name = self.__name = name

>>> my_class = MyClass(name="name")
>>> print(my_class.name)
>>> print(my_class._name)
>>> print(my_class.__name)
name
name
AttributeError: 'MyClass' object has no attribute '__name'
```

Публичные и служебные атрибуты

```
class MyClass:
    name: str
    _name: str
    __name: str

    def __init__(self, name: str) -> None:
        self.name = self._name = self.__name = name

>>> my_class = MyClass(name="name")
>>> print(my_class.name)
>>> print(my_class._name)
>>> print(my_class._MyClass__name)
name
name
name
```

getter и setter

```
class MyClass:
    _name: str

    def __init__(self, name: str) -> None:
        self._name = name

    def get_name(self) -> str:
        return self._name

    def set_name(self, new_name: str) -> str:
        self._name = new_name
```

getter и setter

```
>>> my_class = MyClass(name="name")
>>> print(my_class.get_name())
>>> my_class.set_name("new_name")
>>> print(my_class.get_name())
name
new_name
```

property getter

```
class MyClass:
    _name: str

    def __init__(self, name: str) -> None:
        self._name = name

    def get_name(self) -> str:
        print("get name")
        return self._name

    name = property(get_name)
```

property getter

```
>>> my_class = MyClass(name="name")
>>> print(my_class.name)
>>> my_class.name = "new_name"
>>> print(my_class.name)
get name
name
AttributeError: property ... has no setter
```

property setter

```
class MyClass:
    _name: str

    def __init__(self, name: str) -> None:
        self._name = name

    def get_name(self) -> str:
        print("get name")
        return self._name

    def set_name(self, new_name: str) -> None:
        print("set name")
        self._name = new_name

    name = property(get_name, set_name)
```

property setter

```
>>> my_class = MyClass(name="name")
>>> print(my_class.name)
>>> my_class.name = "new_name"
>>> print(my_class.name)
get name
name
set name
get name
new_name
```

@property

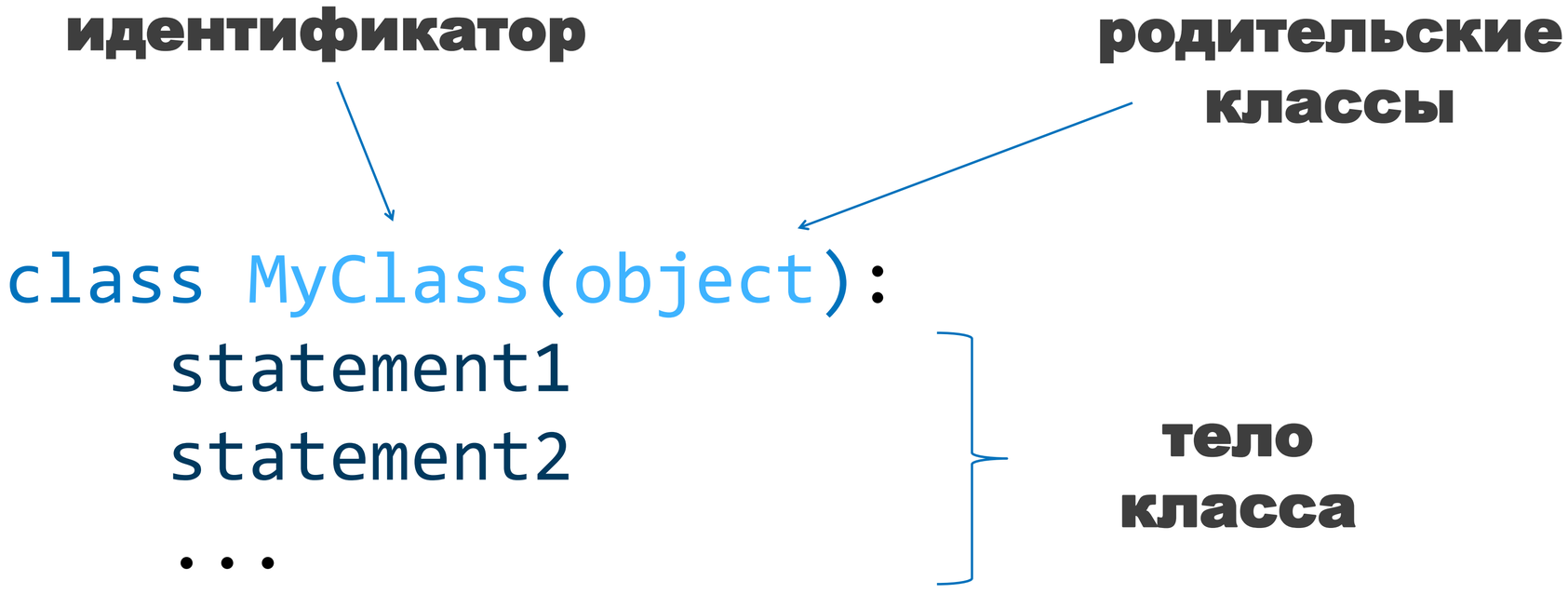
```
class MyClass:
    _name: str

    def __init__(self, name: str) -> None:
        self._name = name

    @property
    def name(self) -> str:
        print("get name")
        return self._name

    @name.setter
    def name(self, new_name: str) -> None:
        print("set name")
        self._name = new_name
```

@property

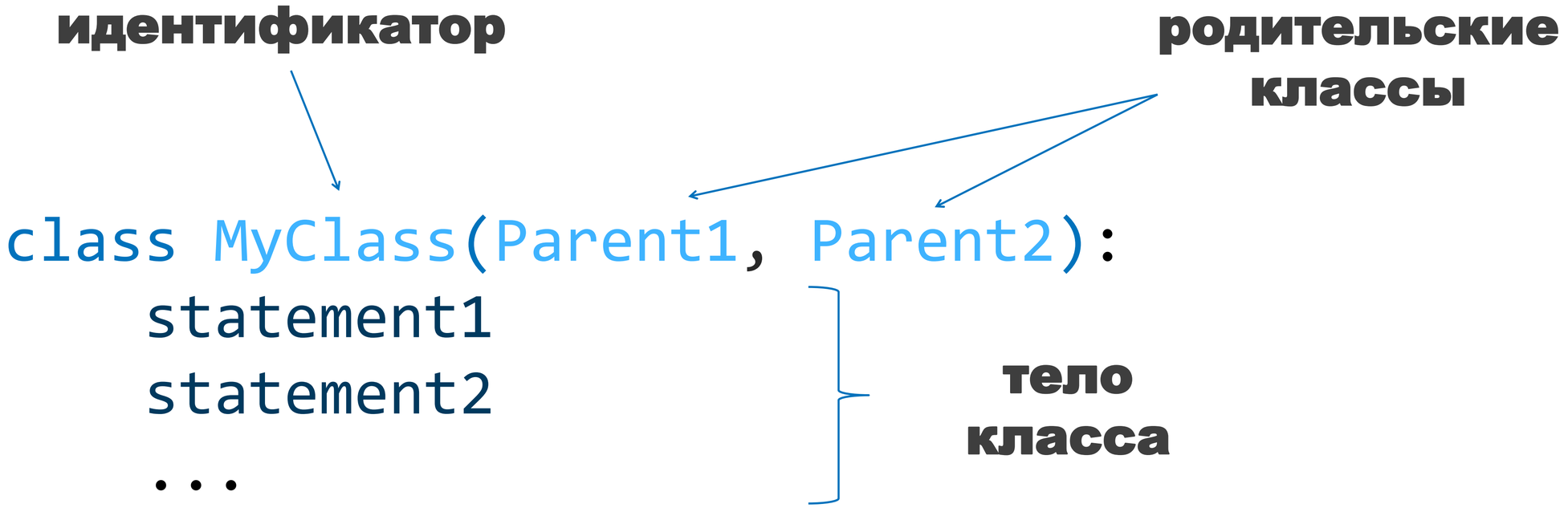
```
>>> my_class = MyClass(name="name")
>>> print(my_class.name)
>>> my_class.name = "new_name"
>>> print(my_class.name)
>>> print(my_class._name)
get name
name
set name
get name
new_name
new_name
```

Наследование

Наследование



Множественное наследование

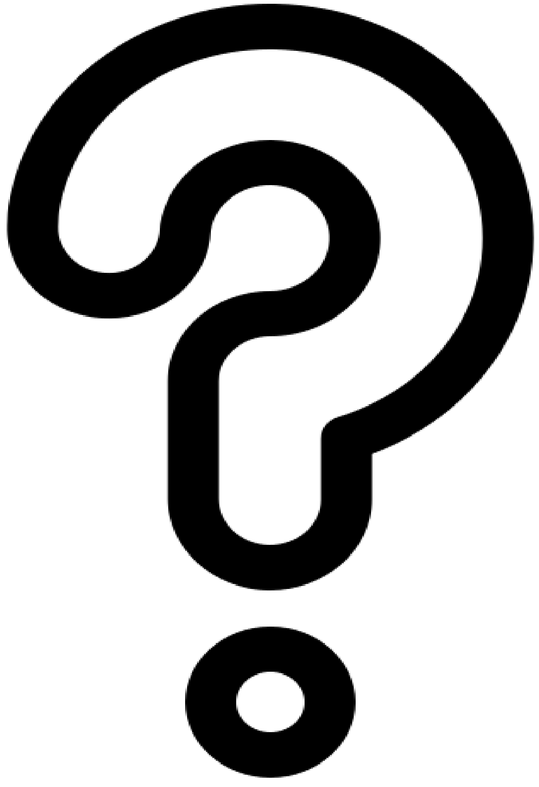


Проблема ромба

```
class Parallelogram:
    def area(self) -> None:
        print("parallelogram area")

class Rectangle(Parallelogram):
    def area(self) -> None:
        print("rectangle area")

class Rhombus(Parallelogram):
    def area(self) -> None:
        print("rhombus area")

class Square(Rectangle, Rhombus):
    pass
```

Проблема ромба

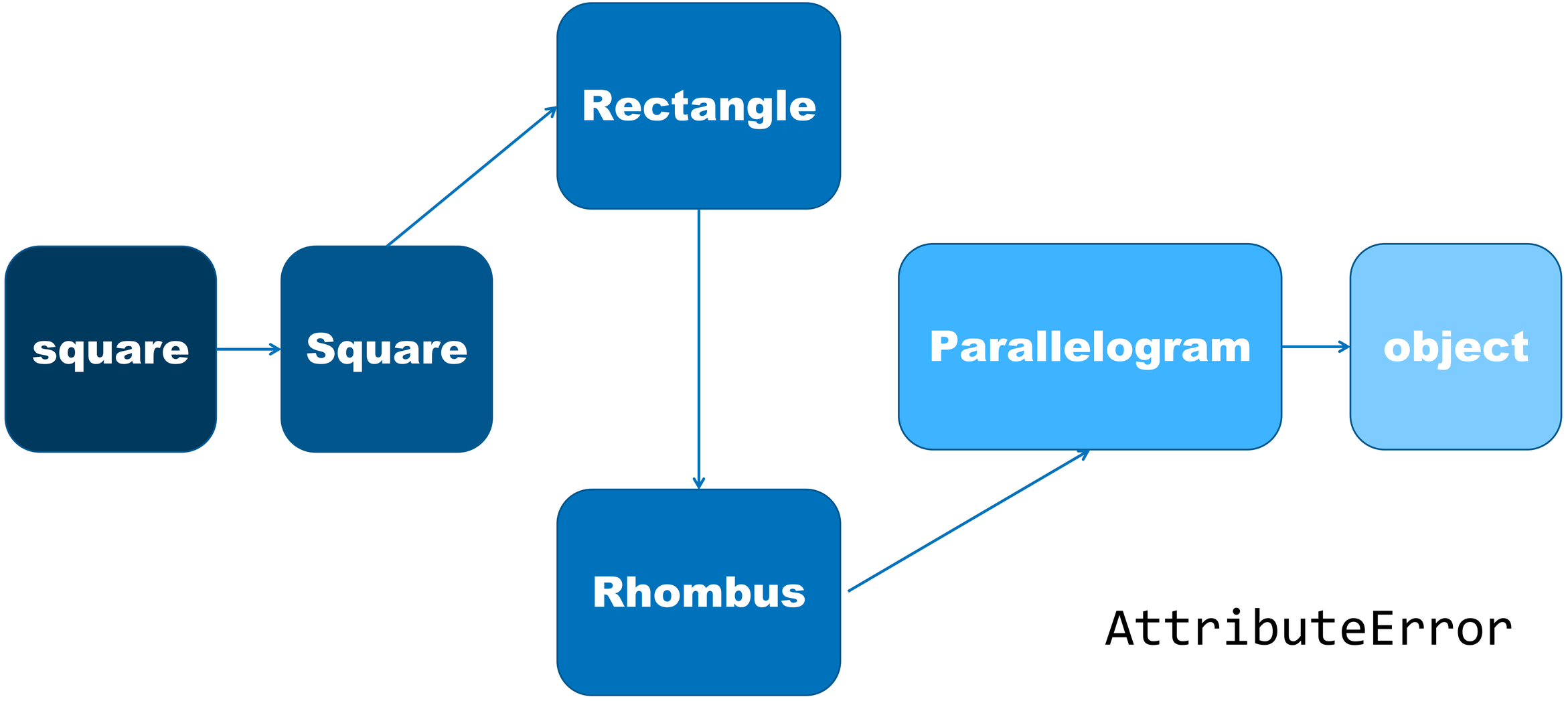
```
>>> square = Square()
>>> square.area()
```



Проблема ромба

```
>>> square = Square()
>>> square.area()
rectangle area
```

MRO




```
AttributeError
```

__mro__

```
>>> print(Square.__mro__)
(
    <class '__main__.Square'>,
    <class '__main__.Rectangle'>,
    <class '__main__.Rhombus'>,
    <class '__main__.Parallelogram'>,
    <class 'object'>
)
```

Зачем нужен super?

```
class Parallelogram:
    def area(self) -> None:
        print("parallelogram area")

class Rectangle(Parallelogram):
    def area(self) -> None:
        print("rectangle area")

class Rhombus(Parallelogram):
    def area(self) -> None:
        print("rhombus area")

class Square(Rectangle, Rhombus):
    def area(self) -> None:
        Rectangle.area(self)
        print("square area")
```

Зачем нужен super?

```
>>> square = Square()
>>> square.area()
rhombus area
square area
```

super

```
class Parallelogram:
    def area(self) -> None:
        print("parallelogram area")

class Rectangle(Parallelogram):
    def area(self) -> None:
        print("rectangle area")

class Rhombus(Parallelogram):
    def area(self) -> None:
        print("rhombus area")

class Square(Rectangle, Rhombus):
    def area(self) -> None:
        super().area()
        print("square area")
```

super

```
>>> square = Square()
>>> square.area()
rhombus area
square area
```

super и __init__

```
class Parent:
    def __init__(self) -> None:
        print("init Parent")

class ChildWrong(Parent):
    def __init__(self) -> None:
        print("init ChildWrong")

class ChildNaive(Parent):
    pass

class ChildGood(Parent):
    def __init__(self) -> None:
        print("init ChildGood")
        super().__init__()
```

super и __init__

```
>>> child_wrong = ChildWrong()
>>> child_naive = ChildNaive()
>>> child_good = ChildGood()
init ChildWrong
init Parent
init ChildGood
init Parent
```

__init__ и множественное наследование

```
class A:
    def __init__(self) -> None:
        print("init A")

class B:
    def __init__(self) -> None:
        print("init B")

class C(A, B):
    def __init__(self) -> None:
        super().__init__()
        print("init C")

>>> c_instance = C()
init A
init C
```

__init__ и множественное наследование

```
class A:
    def __init__(self) -> None:
        print("init A")

class B:
    def __init__(self) -> None:
        print("init B")

class C(A, B):
    def __init__(self) -> None:
        A.__init__(self)
        B.__init__(self)
        print("init C")
```

__init__ и множественное наследование

```
>>> c_instance = C()
init A
init B
init C
```

Полиморфизм

Полиморфизм и протоколы

```
>>> list_ = list(range(5))
>>> tuple_ = tuple(range(4))
>>> set_ = set(range(3))
>>> dict_ = {i: i for i in range(2)}
>>> print(len(list_))
>>> print(len(tuple_))
>>> print(len(set_))
>>> print(len(dict_))
5
4
3
2
```

Полиморфизм и наследование

```
class Parallelogram:
    def area(self) -> None:
        print("parallelogram area")

class Rectangle(Parallelogram):
    def area(self) -> None:
        print("rectangle area")

class Rhombus(Parallelogram):
    def area(self) -> None:
        print("rhombus area")
```

Полиморфизм и наследование

```
>>> polygons = [Parallelogram(), Rectangle(), Rhombus()]
>>> for polygon in polygons:
...     polygon.area()

parallelogram area
rectangle area
rhombus area
```

singledispatch

```
from functools import singledispatch

@singledispatch
def func(arg: int) -> None:
    print(f"given number: {arg}")

@func.register
def _(arg: str) -> None:
    print(f"given string: {arg}")

>>> func("1")
>>> func(1)
given string: 1
given number: 1
```

singledispatchmethod

```
from functools import singledispatchmethod

class Negative:
    @singledispatchmethod
    def negative(self, arg: int, /) -> int:
        print("integer")
        return -arg

    @negative.register
    def _(self, arg: bool, /) -> bool:
        print("bool")
        return not arg
```

singledispatchmethod

```
>>> negative = Negative()
>>> print(negative.negative(1))
>>> print(negative.negative(True))
integer
-1
bool
False
```

Абстракция

Интерфейсы

```
import abc

class Polygon(abc.ABC):
    @abc.abstractmethod
    def get_area(self) -> float:
        pass

class Square(Polygon):
    _side_len: float

    def __init__(self, side_len: float = 1) -> None:
        self._side_len = side_len

>>> square = Square()
TypeError: ...
```

Интерфейсы

```
import abc

class Polygon(abc.ABC):
    @abc.abstractmethod
    def get_area(self) -> float:
        pass

class Square(Polygon):
    def get_area(self) -> float:
        print("square area")

>>> square = Square()
>>> square.get_area()
square area
```

Семинар