ООП в Python

Фургайло В.А.

Идея ООП

- Необходимо отобразить реальные или абстрактные предметы/объекты на программный код

- Классы помогают объединить функционал связанный общей идеей и смыслом в одну сущность, причем у этой сущности может быть внутреннее состояние, а также методы, которые позволяют данное состояние модифицировать

Пример:

- человек, с именем, возрастом, массой, с возможностью ходить/бегать;
- RPG-игра с разными персонажами
- обертка вокруг соединения к базе данных; метода выбора данных и тд.

Проблемы процедурного подхода

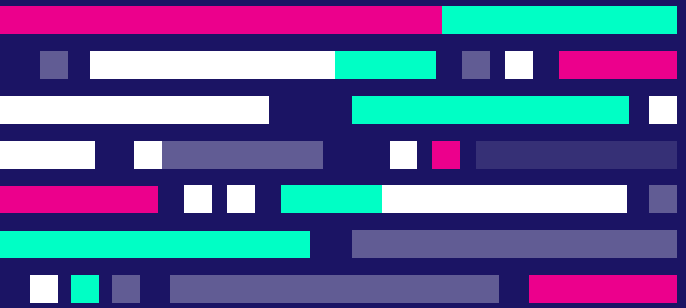
Допустим нам нужен объект `circle = (x, y, R)`

- Если использовать кортеж, нам важно знать порядок переменных. Решение: использовать именованный кортеж `Circle = collections.namedtuple("Circle", "x y radius") # x = circle.x`
- При именованном кортеже, можно указать ошибочное значение (радиус < 0). Однако это можно отслеживать во внутренних функциях, выкидывая исключение
- Можно использовать список `circle = [36. 77. 8]` и обращаться с помощью, например `RADIUS=2, circle[RADIUS]`. Однако это не решает предыдущие проблемы и можно случайно использовать метод `.sort()`
- Поэтому необходимо использовать подход, который решает такие типичные проблемы

Объектно-ориентированная терминология

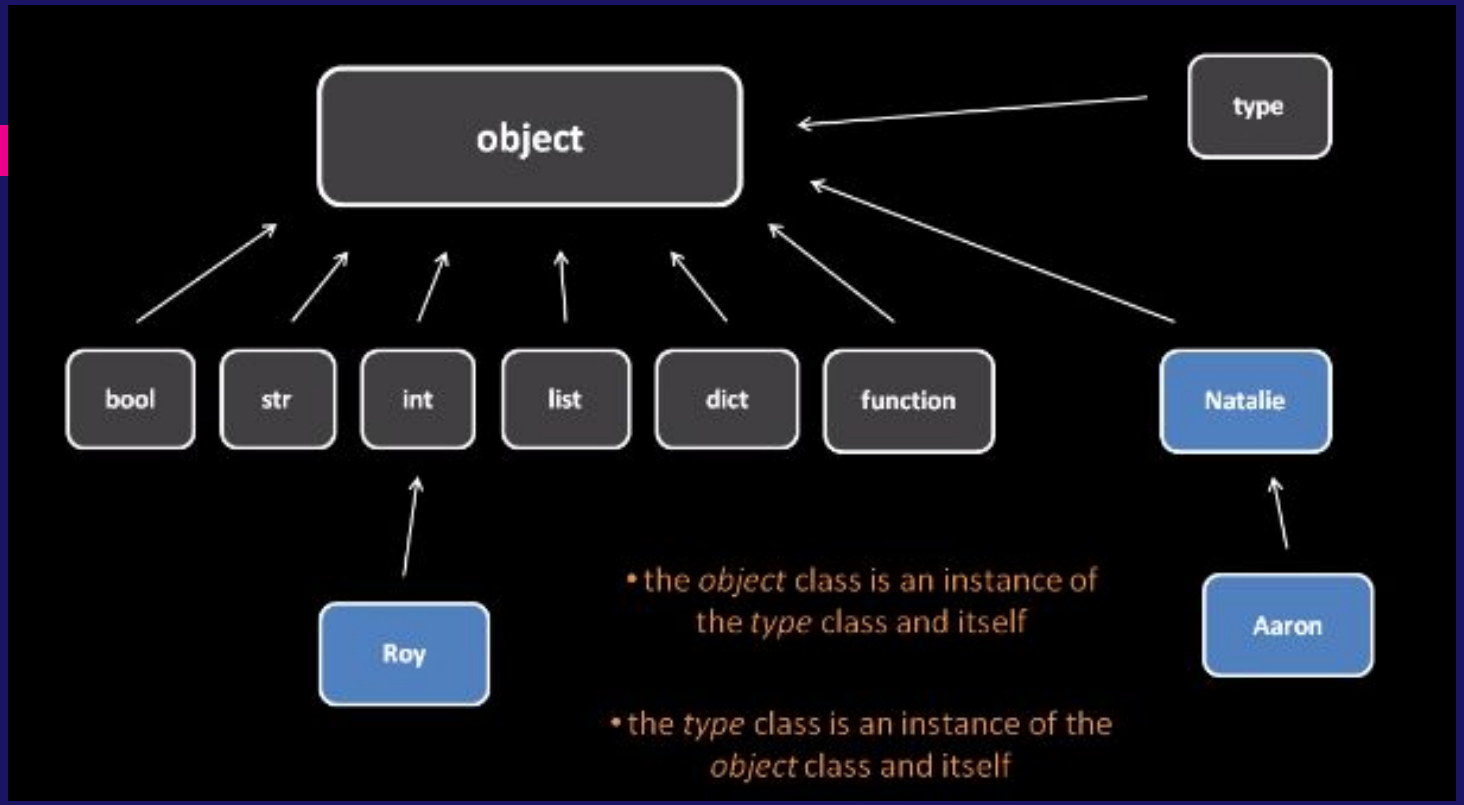
- Решение задачи `circle` - упаковать данные, представляющие окружность и набор функций в единую структуру - создание нового типа данных `Circle`
- ООП = наследование + инкапсуляция + полиморфизм
- В `Python` **класс, тип и тип данных** взаимозаменяемые понятия
- **Объект** - описание сущности, его характеристик и действий. **Экземпляр** класса - конкретный объект
- Многие классы имеют **специальные методы,** например `__add()__` , `__len()__` . НИКОГДА не называется свои функции или переменные такими именами (в начале и в конце __)
- Объекты состоят из **атрибутов** - методов и данных
- Атрибуты данных реализуются как **переменные экземпляра**, то есть уникальные для конкретного объекта

Объектно-ориентированная терминология



- **Свойство** – это элемент данных объекта, доступ к которым используется как доступ к переменной экземпляра, но само обращение неявно делается методами доступа
- **Специализировать класс** (или создать подкласс) - это значит наследовать его от кого-то (со всеми атрибутами) и добавить или заменить новые атрибуты
- В наследовании есть **базовый класс** (родитель, суперкласс) и подкласс (порожденный класс, дочерний класс)
- В `Python` любой класс наследуется от единого базового класса `object`
- Любой метод наследника можно переопределить, это называется динамическое связывание типов или полиморфизм

Дерево наследования Python



Отличие от C++

Python	C++
нижнее подчеркивание перед атрибутом дает доступ “protected”, двойное нижнее - “private”	public, private, protected модификаторы доступа
Только public	Наследование public, private, protected
duck typing для полиморфизма	Перегрузка методов (статический полиморфизм), виртуальные методы (динамический полиморфизм)

Собственные классы

Объявление класса:

```
class className:
    suite

class className(base_classes):
    suite
```

`suite` - тело класса. Вместо `suite` можно написать `pass` - тогда тело будет неопределенно

Атрибуты и методы

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Инициализация переменных(экземпляра) класса. Параметр self - ссылка на сам объект. Здесь указываются и атрибуты данных класса

Здесь определяется атрибуты данных. Вне можно обращаться, как .x .y

Определяется == и !=

Атрибуты и методы

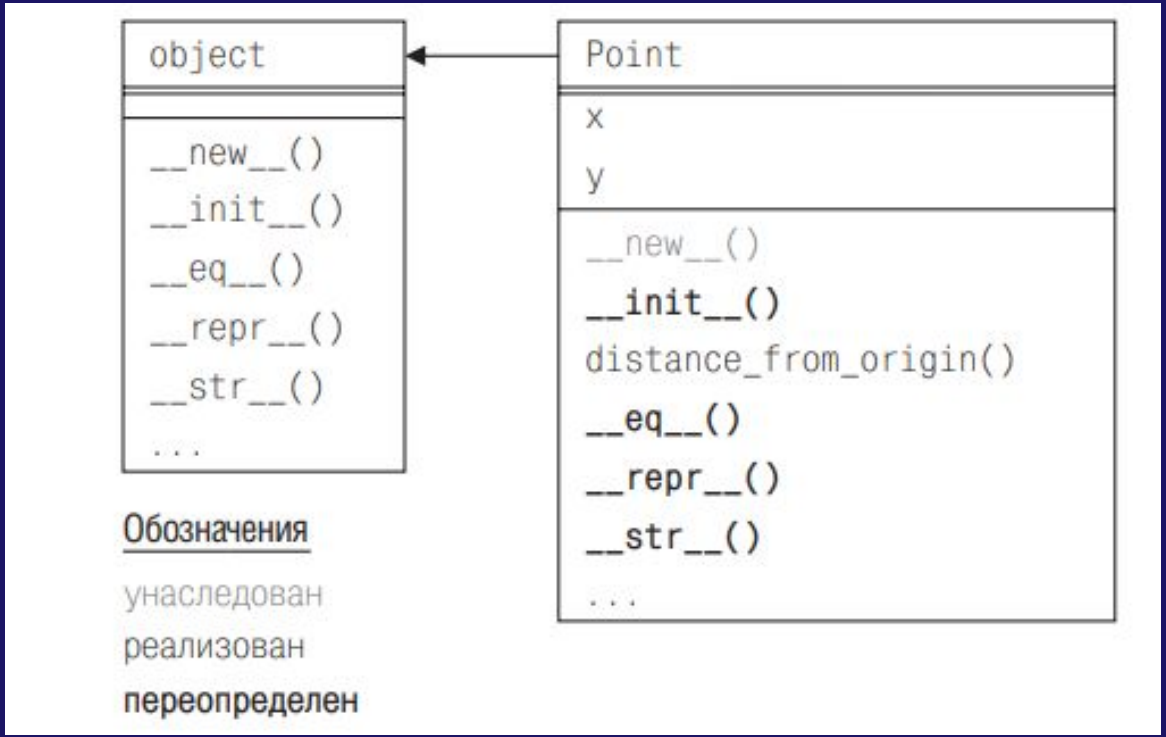
```
import Shape
a = Shape.Point()
repr(a)                          # вернет: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                           # вернет: '(3, 4)'
b.distance_from_origin()         # вернет: 5.0
```

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Атрибуты и методы

Чтобы создать объект
`p = Point(3,4)`
вызывается метод `__new__()`
а потом инициализация `__init()__`
и возвращает ссылку на
экземпляр класса

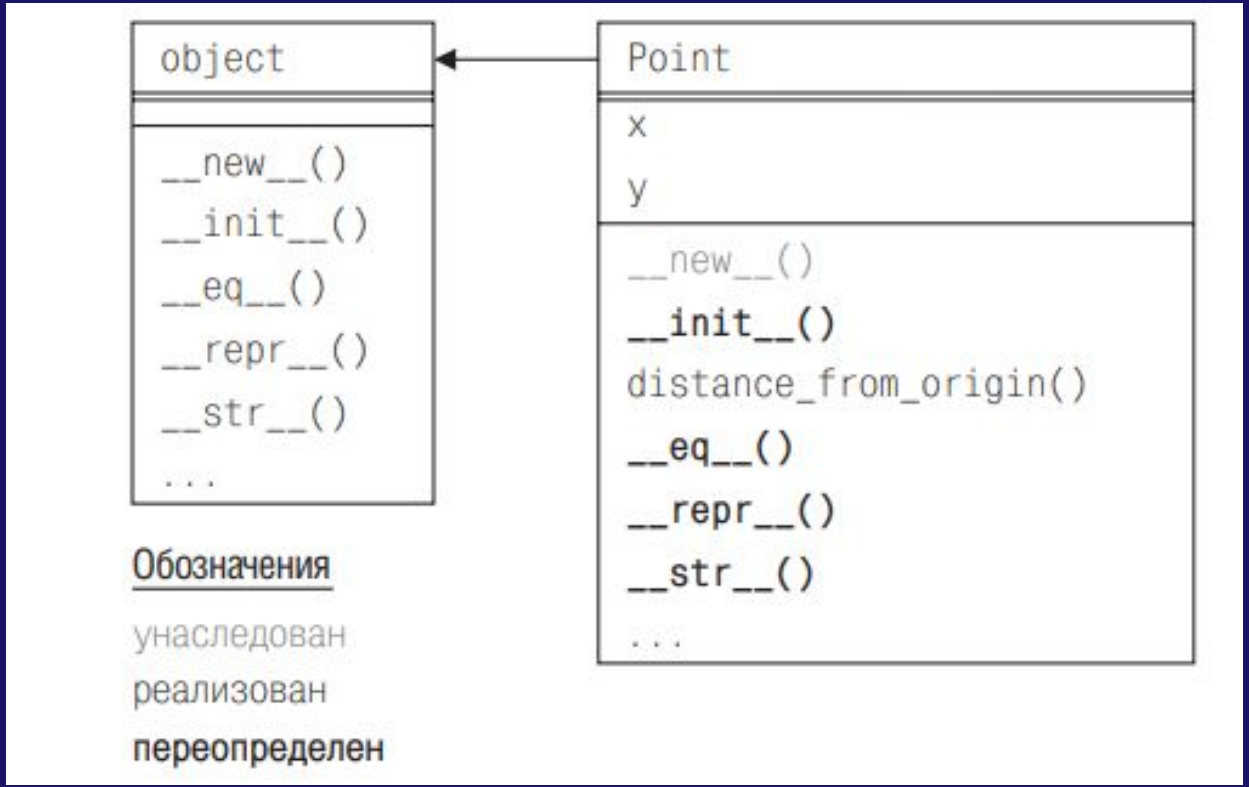
AttributeError - не найден атрибут



Атрибуты и методы

Функция **super()** вызывает метод **базового** класса

```
def __init__(self):
    super().__init__()
```



Атрибуты и методы

Здесь приведены методы сравнения. Однако не получится сравнить int и Point (AttributeError у int) - будет искаться внутри int атрибут x и y

Специальный метод	Пример использования	Описание
`__lt__(self, other)`	`x < y`	Возвращает True, если x меньше, чем y
`__le__(self, other)`	`x <= y`	Возвращает True, если x меньше или равно y
`__eq__(self, other)`	`x == y`	Возвращает True, если x равно y
`__ne__(self, other)`	`x != y`	Возвращает True, если x не равно y
`__ge__(self, other)`	`x >= y`	Возвращает True, если x больше или равно y
`__gt__(self, other)`	`x > y`	Возвращает True, если x больше, чем y

Атрибуты и методы

__repr__() и __str__() возвращают строку. Отличие в том, что str используется для human-readable представления, a repr строку для Python. Ощутимо при приведении типа к str и repr:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> str(now)
'2019-03-29 01:29:23.211924'
>>> repr(now)
'datetime.datetime(2019, 3, 29, 1, 29, 23, 211924)'
```

Атрибуты и методы

Пример `__repr__()` Функция `eval` исполняет команду (необходимо передать имя модуля)

```
p = Shape.Point(3, 9)
repr(p)                                    # вернет: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                    # вернет: 'Point(3, 9)'
```

Наследование и полиморфизм

```
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Наследование и полиморфизм

Наследование реализуется перечислением классов. В данном случае мы наследуемся от `Point`

В `__init__()` инициализируем и базовый класс с помощью `super()`