# PYTHON

Лекция 9

# ПЛАН ЛЕКЦИИ

- Наследование
- Именованные кортежи, dataclass
- Исключения

# НАСЛЕДОВАНИЕ

- Класс может быть унаследован от другого класса
- Синтаксически - класс-родитель указывается в скобках после имени класса
- Семантически - пространство имен класса-родителя участвует в поиске имен
- Содержательно - определяемый класс содержит уточнение сущности, описанной классом-родителем

# ОТПРАВНАЯ ТОЧКА

```
1 class Point:
2     def __init__(self, x, y):
3         self._x = x
4         self._y = y
5
6     def __str__(self):
7         return f"Point({self._x}, {self._y})"
8
9 # ............
```

# ОТПРАВНАЯ ТОЧКА

```
 1 # ............
 2
 3 class Section:
 4     def __init__(self, frm, to):
 5         self._frm = frm
 6         self._to = to
 7
 8     def __str__(self):
 9         return f"Point({self._x}, {self._y})"
10
11
12 class Polygon:
13     def __init__(self, points):
14         self._points = points
```

# ВАРИАНТЫ РАЗВИТИЯ

- Хотим представить абстрактную сущность "многоугольник"
- Задаем его как набор точек
- Как будто ведем линию из точки в точку
- Не каждый набор точек задает корректный многоугольник

# ВАРИАНТЫ РАЗВИТИЯ

- Можем определить класс Section - отрезок
- В классе Line определить метод intersection
- Чтобы возвращала точку пересечения прямых
- Или None, если прямые не пересекаются

# ВАРИАНТЫ РАЗВИТИЯ

- А класс Section - определить как Line и пару точек
- И в нем свой метод intersection
- Отдельный вопрос - стоит ли считать отрезок подклассом Line
- Я бы не стал

# ВАРИАНТЫ РАЗВИТИЯ

- В случае линии отрезка - можно подумать про абстракцию "линейный объект"
- И в него включить все объекты, которые являются подмножеством точек прямой
- У него будет атрибут "несущая прямая"
- И метод, определяющий, принадлежит ли точка ему

# ВАРИАНТЫ РАЗВИТИЯ

- Варианты линейного объекта: прямая, отрезок, луч
- Точка - тоже, причем на разных несущих прямых
- Пунктиры, разного рода прерывистые линии - тоже

# ВАРИАНТЫ РАЗВИТИЯ

- Научившись проверять пересекаемость отрезков - можем проверить корректность задания многоугольника
- Можем считать периметр многоугольника
- Можем в общем виде посчитать площадь

# ВАРИАНТЫ РАЗВИТИЯ

- Для конкретных многоугольников завести отдельные подклассы
- И в них определить свои методы
- Или переопределить более эффективные варианты
- Например - задавать прямоугольник по координатам противоположных вершин
- И определять площадь по произведению сторон

# ОТПРАВНАЯ ТОЧКА

```
 1 class Polygon:
 2     def __init__(self, points):
 3         self._points = points
 4
 5     def get_perimeter(self):
 6         print('calc perimeter', self)
 7
 8
 9 class Quadrilateral(Polygon):
10     def __init__(self, points):
11         pass
12
13 # ...................
```

# ОТПРАВНАЯ ТОЧКА

```
1 # ..............
2
3 q = Quadrilateral([])
4 q.get_perimeter()
```

# ОБРАЩЕНИЕ В СУПЕРКЛАСС

- Вызывая

  `__init__`

  у прямоугольника, логично вызвать

  `__init__`

  у многоугольника
- Есть универсальная конструкция -

  `super()`

# ОБРАЩЕНИЕ В СУПЕРКЛАСС

- Чисто формально - как будто функция, возвращающая суперкласс как объект (но это не так)
- Полезна всегда, когда хотим не заменить переопределяемый метод, а дополнить

# ПРИМЕР

```
 1 class Polygon:
 2     def __init__(self, points):
 3         print(self)
 4         self._points = points
 5
 6     def get_perimeter(self):
 7         print('calc perimeter', self)
 8
 9
10 # ....................
```

# ПРИМЕР

```
# ..................

class Quadrilateral(Polygon):
    def __init__(self, points):
        super().__init__(points)
        print(self)


q = Quadrilateral([])
q.get_perimeter()
```

# ОБРАЩЕНИЕ В СУПЕРКЛАСС

- В точке вызова

  `super()`

  мы нигде не упоминаем

  `self`

- И это не сломается, если нет

  `__init__`

# ОБРАЩЕНИЕ В СУПЕРКЛАСС

- `super()` возвращает прокси-класс
- В котором есть знание о вызвавшем его классе
- И о self-параметре

# ФОРМАЛЬНЫЕ ПРАВИЛА

- Атрибут объекта ищется сначала в объекте
- Потом в его классе, потом в суперклассе и выше
- (При множественном наследовании сложнее)
- Атрибут класса - аналогично, минуя стадию объекта

# ОТПРАВНАЯ ТОЧКА

```
1 class A:
2     V1 = 1
3     V2 = 2
4
5 class B(A):
6     V2 = 22
7     V3 = 33
8
9 # ....................
```

# ОТПРАВНАЯ ТОЧКА

```
# ......................

class C(B):
    V3 = 333
    V4 = 444

    def __init__(self):
        super().__init__()
        self.V4 = 4444
        self.V5 = 5555

# ........................
```

# ОТПРАВНАЯ ТОЧКА

```
1 # ....................
2
3
4 c = C()
5 print(c.V1, C.V1, B.V1, A.V1)
6 print(c.V2, C.V2, B.V2, A.V2)
7 print(c.V3, C.V3, B.V3)
8 print(c.V4, C.V4)
9 print(c.V5)
```

# АБСТРАКТНЫЕ КЛАССЫ/ МЕТОДЫ

- В чистом языке - нет
- Но если реализация в виде надстройки
- Надо импортировать стандартный модуль

  `abc`

- И использовать класс

  `ABC`

  и декоратор

  `abstractmethod`

# ПРИМЕР

```
1 from abc import ABC, abstractmethod
2
3 class A(ABC):
4
5     @abstractmethod
6     def m(self):
7         ...
8
9 # ..................................
```

# ПРИМЕР

```
1 # ..............................
2
3 class B(A):
4
5     def m(self):
6         print('B.m')
7
8 b = B()
9 b.m()
```

# ФУНКЦИИ ПРО НАСЛЕДОВАНИЕ

- `isinstance` - два аргумента
- `True` если первый аргумент - экземпляр второго

# ФУНКЦИИ ПРО НАСЛЕДОВАНИЕ

- `issubclass`

  - два аргумента

- `True`

  если первый аргумент - подкласс второго

# ПРИМЕР

```
 1 from abc import ABC, abstractmethod
 2
 3 class A(ABC):
 4
 5     @abstractmethod
 6     def m(self):
 7         ...
 8
 9 class B(A):
10
11     def m(self):
12         print('B.m')
13
14 # ....................
```

# ПРИМЕР

```
# ....................
b = B()

print(isinstance(b, B))
print(isinstance(b, A))
print(isinstance(b, object))
print(isinstance(B, B))
print(isinstance(B, object))
print(isinstance(int, object))
print(isinstance(object, object))
```

# РОЛЬ НАСЛЕДОВАНИЯ

- При вызове метода работает duck typing
- Но есть моменты, когда важно наследование
- Например, при создании исключения
- Или при его обработке

# NAMEDTUPLE

- Мотивация: хотим создавать классы для хранения простых объектов
- Аналог структуры с именованными полями
- `setattr/getattr` позволяют работать с динамически задаваемыми именами
- Атрибутом класса можно хранить список имен поддерживаемых полей

# NAMEDTUPLE

- Конструктор может принимать именованные поля
- Если все поля из списка поддерживаемых - вызываем `getattr`
- `namedtuple` - фунция, принимающая имя класса и список имен полей
- Возвращает класс, умеющий конструировать объект с данными полями

# ПРИМЕР

```
1 from collections import namedtuple
2
3 Date = namedtuple('Date', ('year', 'month', 'day'))
4 Person = namedtuple('Person', ('name', 'birthdate', 'id'))
5
6 p = Person('Ivanov Dmitry', Date(1990, 12, 23), 1234567)
7 print(p)
```

# НЮАНСЫ

- Совпадение имени переменной и имени класса - принятая традиция, но не обязательно
- Имя класса будет использовано при создании класса внутри `namedtuple`
- И останется в свойстве `__name__`
- Может запутать отладку

# ПРИМЕР

```
 1 from collections import namedtuple
 2
 3 A = namedtuple('B', ('a', 'b', 'c'))
 4
 5 a = A(1, 2, 3)
 6
 7 print(a)
 8 print(type(a))
 9
10 A.__name__ = 'A'
11
12 print(a)
13 print(type(a))
```

# НЮАНСЫ

- Чисто технически не запрещено в два вызова `namedtuple` передать одно и то же значение
- Это будут два разных класса
- Объекты будут визуально похожими
- И даже равными (равенство определяется конфигурацией полей)
- Но их классы будут разными

# ПРИМЕР

```
 1 from collections import namedtuple
 2
 3 A1 = namedtuple('A', ('a', 'b', 'c'))
 4 A2 = namedtuple('A', ('a', 'b', 'c'))
 5
 6
 7 a11 = A1(1, 2, 3)
 8 a12 = A1(1, 2, 3)
 9 a2 = A2(1, 2, 3)
10
11 print(a11, a12, a2)
12 # ......................
```

# ПРИМЕР

```
1 # ......................
2
3 print(a11 is a12)
4 print(a11 == a12)
5 print(a11 == a2)
6 print(a12 == a2)
7 print()
8 print(isinstance(a2, type(a11)))
9 print(isinstance(a2, type(a12)))
10 print(isinstance(a12, type(a11)))
```

# ДВОЙСТВЕННОСТЬ

- К полям можно обращаться по имени как к атрибуту или по нумерованному индексу
- Конструктор принимает именованные и позиционные поля
- Есть classmethod `_make` - он принимает список значений
- (Все методы начинаются с подчеркиваний, чтобы не конфликтовать с полями)

# ПРИМЕР

```
 1 from collections import namedtuple
 2
 3 Point = namedtuple('Point', ('x', 'y'))
 4
 5
 6 p1 = Point(1, y=2)
 7 print(p1.x, p1[0])
 8
 9 p2 = Point._make([11, 22])
10 print(p2.x, p2[0])
```

# НЕИЗМЕНЯЕМОСТЬ

- Объекты неизменяемы
- Можно хранить в множестве и использовать как ключ в словаре
- Для облегчения создания новых есть метод `_replace`:

```
p = p._replace(x=5)
```

# ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

- В предыдущих примерах порождались классы, требующие всех параметров при инициализации
- Но можно для некоторых задать значения по умолчанию
- Для этого есть параметр `defaults`
- Со списком значений по умолчанию

# ПРИМЕР

```
 1 from collections import namedtuple
 2
 3 C = namedtuple('C', ('a', 'b', 'c', 'd'),
 4                 defaults=(12, 'hello'))
 5
 6 c1 = C(1, 2)
 7 print(c1)
 8 c2 = C(1, 2, d='world')
 9 print(c2)
10 c3 = C(1, 2, 34, 'qwertty')
11 print(c3)
```

# ПОЛЕЗНОЕ

- `_as_dict` - представляет объект в виде словаря
- `_fields` - список имен полей
- `_field_defaults` - значения по умолчаению как словарь

# DATACLASS

- Развитие идеи `namedtuple`
- Через декоратор класса
- И аннотации типов
- Больше гибкости

# ПРИМЕР

```python
from dataclasses import dataclass, field

@dataclass
class Point2D:
    x: float
    y: float

p = Point2D(1.2, 2)
print(p)
```

# НАСТРОЙКИ

- По умолчанию можно менять поля
- Но можно задать параметр в аннотации и запретить это делать
- И в зависимости от этого объекты будут `hashable` или не `hashable`
- Много разного: https://docs.python.org/3/library/dataclasses.html

# ИСКЛЮЧЕНИЯ

- Механизм обработки ошибок
- Чисто технически исключение - Python-объект
- Может быть порожден "изнутри" Python (например, деление на 0)
- Может быть порожден программно - в случае обнаружения фатальной ошибки

# ИСКЛЮЧЕНИЯ

- Факт создания объекта-исключения на исполнение кода не влияет
- Влияем - факт бросания исключения
- Программно бросается с помощью ключевого слова `raise`
- В момент бросания обычное исполнение кода прекращается
- Ищется обработчик исключения

# ИСКЛЮЧЕНИЯ

- Обработчик исключения - конструкция `try/except`
- После `try` - блок кода для исполнения
- После `except` - блок для обработки исключения
- В простейшем варианте - для любых исключений

# ИСКЛЮЧЕНИЯ

- В момент исполнения мы находимся где-то на стеке вызовов
- Каждая точка вызова либо находится непосредственно внутри обработчика, либо нет
- Это статически известный факт про любой точку исходного кода
- А конкретный стек вызовов - это явление динамическое

# ИСКЛЮЧЕНИЯ

- Идем по стеку вызовов и ищем первый вызов внутри подходящего обработчика
- Чей `except` обрабатывает наше исключение
- В текущем примере до любого обработчика
- По ходу движения стековые фреймы сворачиваются (вызовы "завершаются")

# ИСКЛЮЧЕНИЯ

- Дойдя до обработчика - передаем управление в его `except`-блок
- В этом момент исключение считается обработанным
- Возобновляется обычное исполнение кода
- В предельном случае можем сразу найти обработчик, без очистки фреймов
- В противоположном - обработчика не найдем, процесс завершится, напечатается ошибка со стеком вызовов

# ПРИМЕР

```
1 v = int(input())
2
3 try:
4     print(1/v)
5 except:
6     print('got exception')
```

# ПРИМЕР

```
1 v = int(input())
2
3 try:
4     print(1/v)
5 except Exception as exc:
6     print('got exception', exc, type(exc))
```