PYTHON

Лекция 3

ПЛАН ЛЕКЦИИ

- Типы данных в Python
- Динамическая типизация
- Duck typing

ДАННЫЕ В PYTHON

- Универсальная концепция - объект
- Все - объект
- Целые, вещественные числа, True, False - объекты
- Строки - объекты

ДАННЫЕ В PYTHON

- Функции, модули - объекты
- Классы - тоже объекты
- А у объектов есть состояние и поведение
- Инкапсуляции - в Python почти нет
- Зато полиморфизм в полный рост

ПОВЕДЕНИЕ И СОСТОЯНИЕ

- Поведение объекта определяется его методами
- Метод - это разновидность функции
- Состояние определяется его свойствами
- Свойства объекта можно увидеть с помощью функции dir

ПРИМЕР

```
1 import sys
2
3 for o in ["", 123, 123.44, True, print, dir, sys]:
4     if hasattr(o, '__name__'):
5         print(o.__name__)
6     print(o)
7     print(dir(o))
8     print()
```

ЧТО ВИДИМ

- Разные объекты, у каждого - какие-то свойства
- Многие обрамляются подчеркиваниями
- Это свойства для системных нужд
- Свойства можно прочитать как переменные

РАБОТА С СОСТОЯНИЕМ

- В классическом ООП состояния меняются через методы
- В Python можно запретить изменение состояния в обход методов, но специальными усилиями
- И для стандартных функций/классов эти усилия приложены
- Но никто не мешает изменять состояние объекта "извне"
- В частности, своей функции

ПРОСТОЙ ПРИМЕР

```
1 def f():
2     pass
3
4 f.fld = 12345
5 print(f.fld)
```

МЕТАФОРЫ ООП

- Классическая: класс - чертеж, объекты - детали
- Во время исполнения можем создавать детали по фиксированному набору проектов
- Иногда - можно добавить чертежей, но тоже фиксированный набор (Java)
- В традиционном JavaScript-е - чертежей нет
- Но можем по детали заказать такую же, потом что-то в ней поменять

МЕТАФОРЫ ООП

- Классическая: класс - чертеж, объекты - детали
- Во время исполнения можем создавать детали по фиксированному набору проектов
- Иногда - можно добавить чертежей, но тоже фиксированный набор (Java)
- В традиционном JavaScript-е - чертежей нет
- Но можем по детали заказать такую же, потом что-то в ней поменять

NONE

ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

- True, False
- В логическом контексте другие типы преобразуются в логические
- None превращается в False
- Любой числовой 0 превращается в False

ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

- nan превращается в True
- Любые пустые коллекции превращаются в False
- Все остальное превращается в True
- Но этим можно более тонко управлять

ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

- Можно определить в классе метод __bool__
- Если он есть, то bool вызовет его
- И есть __len__
- Если нет __bool__, но есть __len__, то bool вызовет его
- И результат будет зависет от равенства нулю результата __len__

ДИНАМИЧЕСКОЕ ИЗМЕНЕНИЕ СЕМАНТИКИ ПРЕОБРАЗОВАНИЯ В BOOL

```
 1 def to_bool_1(v):
 2     print("to bool 1")
 3     return True
 4
 5 def to_bool_2(v):
 6     print("to bool 2")
 7     return False
 8
 9 class C:
10     pass
11
12 # ..........
```

ДИНАМИЧЕСКОЕ ИЗМЕНЕНИЕ СЕМАНТИКИ ПРЕОБРАЗОВАНИЯ В BOOL

```
 1 # ......
 2
 3 v = C()
 4 def f():
 5     if v:
 6         print(123)
 7
 8 f()
 9 type(resp).__bool__ = to_bool_1
10 f()
11 type(resp).__bool__ = to_bool_2
12 f()
```

"ПО МАККАРТИ"

- Если ответ понятен, то не считаем дальше
- Хороший код часто это использует
- Пример: if b != and a // b > 0:

СРАВНЕНИЯ

- 8 стандартных неравенств
- "Двойное" неравенство: if 5 < a <= 10:
- Есть тонкое отличие от двух сравнений с and
- Средняя часть в "двойном" случае вычисляется один раз

РАЗБЕРЕМ ПРИМЕР

```
1 def f(v):
2     print(v)
3     return v
4
5 print(f(1) < f(2) < f(3))
6 print(f(2) < f(1) < f(3))
7
8 print(f(1) < f(2) and f(2) < f(3))
9 print(f(2) < f(1) and f(1) < f(3))
```

СРАВНЕНИЯ

- Есть сравнения на семантическое равенство: = и !=
- Есть сравнения на идентичность: is и not is
- Проверка, что две переменные ссылаются на один объект: a is b
- Семантическое равенство определено для стандартных классов

СРАВНЕНИЯ

- Для своих классов оно по умолчанию сводится к идентичности
- Понятие равенства можно переопределить: метод __eq__
- Неравенства - тоже
- Идентичность не переопределяется
- Еще есть логическая операция in / not in

== VS IS

- Даже для встроенных типов возможна неидентичность при семантической эквивалентности
- Пример: 5 is int(4.9 + 0.1) верно, а вот 500 is int(499.9 + 0.1)
- Пример: "123" is str(123) неверно
- А вот 5 is int(5.0) верно
- А 5 is 5.0 неверно

ЧИСЛЕННЫЕ ТИПЫ

- Встроенные - int, float, complex
- Импортируемые из стандартной библиотеки - Rational, Decimal
- Комплексные числа поддержаны в синтаксисе
- 5j, 1 - 2j

ЧИСЛОВЫЕ ОПЕРАЦИИ

- Бинарные +, -, *, /, //, % -
- Возведение в степень: x ** y
- -23 // 3 == -8, -23 % 3 == 1 # классическое определение теории вычетов
- 23 // -3 == -8, 23 % -3 == 1 # инверсия знаков не меняет результата
- -23 // -3 == -7, -23 % -3 == -2 # зеркало позитивного случая

ЧИСЛОВЫЕ ОПЕРАЦИИ

- Семантика арифметики имеет отличия между целыми и вещественными с нулевой дробной частью
- Особенно на больших значениях
- int - "большие целые", float - "как double в C"
- Возведение в нецелую степень подразумевает преобразование int во float
- Даже если результат - математически целый

РАЗБЕРЕМ ПРИМЕР

```
123456789 ** 40 # OK

123456789.o ** 40 # переполнение

123456789 ** 40.0 # переполнение

(123456789 ** 20) ** 0.05 # легкая неточность

(123456789 ** 40) ** 0.025 # переполенение
```

ДЛЯ ЦЕЛЫХ

- Побитовые операции: |, &, ^, <<, >>, ~
- Считаем, что у неотрицательных спереди бесконечно много нулей
- У отрицательных - бесконечно много единиц
- Определяем длину более длинного аргумента

ВЕЩЕСТВЕННАЯ АРИФМЕТИКА

- Есть специальные значения: float('inf'), -float('inf'), float('nan')
- Есть положительный и отрицательный нули
- float('inf') большое положительное число, не влезшее в диапазон
- -float('inf') большое по модулю отрицательное число, не влезшее в диапазон

ВЕЩЕСТВЕННАЯ АРИФМЕТИКА

- Можно выполнять операции между бесконечностью и числом
- Константа, деленная на бесконечность того же знака, даст положительный 0
- При разных знаках - отрицательный 0
- Внутренние битовые представления отличаются и на печати - тоже
- Но по неравенствам и семантическому равенству - они равны

NAN

- nan получается при неопределенностях
- Например при сложении inf и -inf
- Или умножении бесконечности на 0
- Что не совсем стыкуется с идеей "большое положительное число, не влезшее в диапазон"

ТОНКОСТИ

- 0.1 * 3 == 3.0 - неверно
- Слева - 0.30000000000000004
- Храним двоичные дроби
- Непереодическая десятичная может быть периодической двоичной
- И могут накопиться ошибки округления

СТРОКИ

- В Python нет отдельного символьного типа
- Есть строки
- Строка может быть длиной 1
- Функция-конструктор - str

СТРОКОВЫЕ КОНСТАНТЫ

- Могут ограничиваться двойными или одинарными кавычками
- Разницы нет, но лучше придерживаться единого стиля
- Две строковые константы, идущие одна за другой, склеиваются
- Можно без операций (важно, чтобы были константы)

СТРОКОВЫЕ КОНСТАНТЫ

- Можно перенести через \ + перевод строки
- Но это сбивает логику смещений
- Можно использовать операцию +, но это лучше для неконстантных строк

ПРИМЕР

```
 1 def f():
 2     s1 = 'string'
 3     s2 = 'some very \
 4 long string'
 5     s3 = 'some very '
 6     'long string'  # именно с таким смещением
 7     s4 = 'some very long ' + \
 8          s1
 9     s5 = ('some very ' +
10          'long string')
```

СТРОКОВЫЕ КОНСТАНТЫ

- Другой тип длинных констант - "текст"
- Короткие фрагменты, разделенные переводом строки
- И их надо оставить в тексте
- "Строенные" кавычки - двойные или одинарные

ПРИМЕР

```
1 s1 = '''Однажды в студеную зимнюю пору
2 Я из лесу вышел, был сильный мороз
3 '''
```

DOC-СТРОКИ

- Перед классами, функциями, в начале модулей принято ставить строковые константы
- Обычно - многострочные в двойных кавычках
- На ход исполнения не влияют
- Но специальные утилиты их извлекают
- Из них порождается документация

ПРИМЕР

```
1 def fact(n):
2     """
3     This function calculates factorial of n
4     """
5
6     pass
```

СТРОКИ С ПЕРЕМЕННЫМ СОДЕРЖИМЫМ

- Строить конкатенациями - плохой стиль
- Тем более, что конкатенация определена только для пар строк
- Вариант 1 - операция %
- Вариант 2 - метод format
- Вариант 3 - интерполяция ("форматные строки")

ПРИМЕР

```
1 for i in range(10):
2     print("%d ^ 2 = %d" % (i, i * i))
3
4 for i in range(10):
5     print("%4d ^ 2 = %-8d" % (i, i * i))
```

ПОПОДРОБНЕЕ

- Похоже на printf в C
- Считается самым старомодным способом
- Но иногда удобен
- Например, если кортеж уже есть (приходит параметром)

ПРИМЕР

```
1 txt1 = "My name is {fname}, I'm {age}"
2        .format(fname = "John", age = 36)
3 txt2 = "My name is {0}, I'm {1}".format("John",36)
4 txt3 = "My name is {}, I'm {}".format("John",36)
```

РАЗБЕРЕМСЯ

- Можно ссылаться по именам
- Можно повторять одно и тоже значение
- Но иногда получается boilerplate
- Это проблему решают форматные строки
- За счет удобства повторений

ФОРМАТИРОВАННЫЕ СТРОКИ

- Напишем f перед строковой константой
- Неважно, какие кавычки
- Внутри можно использовать {}
- В фигурных скобках писать любое выражение

ПРИМЕР

```python
for i in range(10):
    print(f"{i} ^ 2 = {i * i}")

for i in range(10):
    print(f"{i:%4d} ^ 2 = {i * i:-4d}")
```

ЧТО ВЫБРАТЬ

- Конкатенация - разовое простое сцепление 2-3 строк
- "По умолчанию" - форматные строки
- Если есть повторяемость - format
- Или данные уже в словаре
- % - если простая структура и данные уже в кортеже