PYTHON

Лекция 10

ПЛАН ЛЕКЦИИ

- Модули
- Пространства имен

ИМЯ МОДУЛЯ

- У каждого модуля есть имя
- Хранится в переменной

```
__name__
```

- Зависит от того, как модуль появился
- Был ли он импортирован
- Или с него началось исполнение

МОДУЛЬ

```
 1 # fibo.py
 2
 3 print("My name is", __name__)
 4
 5 # main.py
 6
 7 import fibo
 8
 9 ## Сравним python3 fibo.py
10 ## и python3 main.py
```

ДВОЙНОЕ НАЗНАЧЕНИЕ

- Хорошая практика - писать в конце файла условное предложение
- Сравнивать

```
__name__
```

с

```
'__main__'
```

- И действовать в зависимости от результата сравнения

```
1 if __name__ == '__main__':
2     pass
```

ДВОЙНОЕ НАЗНАЧЕНИЕ

- Вариант 1: вспомогательная логика с изолированной функциональностью
- Например, преобразования строк
- Можно сделать его самодостаточной утилитой
- Например, применять преобразования к каждой строке входного потока

ДВОЙНОЕ НАЗНАЧЕНИЕ

- Если преобразований несколько - можно выбирать через параметр командной строки
- Это полезно и само по себе, и как способ быстро познакомиться с логикой работы модуля
- Пример: модули архивации в Python

ДВОЙНОЕ НАЗНАЧЕНИЕ

- Вариант 2: трудно свести в самодостаточную утилиту
- Можно выводить краткую справку о модуле
- Можно запускать тесты
- По хорошо именованным тестам можно сложить представление о том, что модуль делает

ДВОЙНОЕ НАЗНАЧЕНИЕ

- Вариант 3: основной модуль, точка входа в программу
- Задумайтесь о возможности использовать его как библиотеку
- Изолируйте функциональность в классах/функциях
- Получите код, переиспользуемый в качестве библиотеки
- Может потребовать времени, но повысит качество кода

ВИДИМОСТЬ ПЕРЕМЕННЫХ

- В первом приближении - глобальная и локальная
- Если точнее - локальных может быть много
- По вложенности определений функций
- Начнем без вложенности
- И пока без модульности

ПРИМЕР

```
 1 V1 = 1
 2 V2 = 2
 3
 4 def f():
 5     v = 123
 6     V2 = 234
 7
 8     print(V1, V2, v)
 9     # print(V3)
10
11 f()
```

ЧТЕНИЕ

- У каждой функции есть своя таблица имен
- Она создается при вызове
- Изначально в ней живут параметры
- И добавляются локальные переменные по мере выполнения присваиваний

ЧТЕНИЕ

- При чтении сначала смотрим в локальную таблицу имен
- Если имя не нашли - в смотрим в глобальной
- Если не нашли и там - бросится исключение
- Если вызов не прямой - локальные таблицы промежуточных функций никак не влияют

ПРИМЕР

```
 1 V1 = 1
 2 V2 = 2
 3
 4 def f():
 5     v = 123
 6     V2 = 234
 7
 8     print(V1, V2, v)
 9     # print(V3)
10
11 # ............
```

ПРИМЕР

```
1  # ......................
2
3
4  def f2():
5      V1 = 1111
6      V2 = 2222
7      V3 = 3333
8      f()
9
10 f2()
```

ЗАПИСЬ

- Присваивание идет в локальный контекст
- Если это не изменить особой конструкцией
- Есть ключевое слово

```
global
```

- Употребляется внутри функции

ПРИМЕР

```
 1 V1 = 1
 2 V2 = 2
 3
 4 def f():
 5     global V2, V3
 6     v1 = 123
 7     V2 = 234
 8
 9     print(V1, V2)
10     V3 = 345
11     print(V3)
12
13 f()
14 print(V1, V2, V3)
```

ЗАПИСЬ

- Присваивание в

  `v2`

  и в

  `v3`

  в примере совершаются в глобальной таблице имен
- Чтение - фактически тоже

ЗАПИСЬ

- Возможна тонкая ситуация - если перед

  ```
  'global v'
  ```

  что-то присвоить в

  ```
  v
  ```

- А сразу после - прочитать

GLOBAL

- Во избежание неприятностей Python такое запрещает
- `global` не обязан быть первой конструкцией

GLOBAL

- Но в том, что идет до, не должны упоминаться переменные из

```
global
```

- В этом смысле условные конструкции игнорируются
- Проблема фиксируется в момент определения функции

ЧТО ПРОИЗОЙДЕТ ?

```
 1 def f(n):
 2     print(n)
 3     print(v) # нельзя использовать переменную,
 4              # которая имеет шанс стать локальной
 5     if n > 0: # но это не синтаксическая ошибка
 6         v = 123
 7
 8 v = 123
 9 f(-5)
10 print('v', v)
```

ЧТО ПРОИЗОЙДЕТ ?

```
1 def f(n):
2     if n < 0:
3         print(v)
4
5     if n > -5:
6         v = 123
7
8 v = 123
9 f(-10)
```

ЧТО ПРОИЗОЙДЕТ ?

```
 1 def f(n):
 2
 3     while n < 3:
 4         if n == 2:
 5             print(v)
 6
 7         if n == -5:
 8             v = 123
 9
10         n += 1
11
12 v = 123
13 f(-10)
```

ВЛОЖЕННЫЕ ОПРЕДЕЛЕНИЯ

- Если определения вложены, то в дело вступают новые таблицы символов
- От каждой функции, внутри которой функция определена
- Идем от самого внутреннего уровня к самому внешнему
- Самый внутренний - сама функция, самый внешний - глобальный

ПРИМЕР

```
1 V1 = 1
2 V2 = 2
3
4 def f1():
5     V2 = 11
6     V3 = 22
7
8 # ................
```

ПРИМЕР

```
1  # ..................................
2
3      def f2():
4          V3 = 111
5          V4 = 222
6          print(V1, V2, V3, V4)
7
8      print(V1, V2, V3)
9      f2()
10     print(V1, V2, V3)
11
12 print(V1, V2)
13 f1()
14 print(V1, V2)
```

NONLOCAL

- `nonlocal` - неточный аналог `global`
- Для изменения нелокальных переменных
- Синтаксисически эквивалентна, с точностью до ключевого слова
- Семантически есть важные отличия

NONLOCAL

- Переменные, указанные в `nonlocal` - обязаны существовать
- В отличие от `global`

NONLOCAL

- Они ищутся начиная с ближайшей объемлющей
- Если найдена - поиск прекращается
- Если найдена только на глобальном уровне - это ошибка

ПРИМЕР

```
 1 V1 = 1
 2 V2 = 2
 3
 4 def f1():
 5     V2 = 11
 6     V3 = 22
 7
 8     def f2():
 9         V3 = 111
10         V4 = 222
11
12 # ......................
```

ПРИМЕР

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


        def f3():
            nonlocal V2, V3
            global V1
            V1 = 1111
            V2 = 2222
            V3 = 3333

        f3()
        print(V1, V2, V3, V4)

    print(V1, V2, V3)
    f2()
```

СТАТИКА И ДИНАМИКА

- В случае "функция внутри функции" нет чистой статичности
- Потому что модуль инициализируется один раз
- И функции глобального уровня создаются тоже один раз
- И классы, и глобальные переменные - тоже
- Вложенная функция создается заново при каждом вызове объемлющей

ПРИМЕР

```
 1 Q = []
 2
 3 def f1():
 4     def f2():
 5         def f3():
 6             pass
 7
 8         Q.append(f3)
 9
10     f2()
11     f2()
12
13 f1()
14 print(Q)
```

СТАТИКА И ДИНАМИКА

- Строгой статичности нет
- Но поиск переменных по таблицам определяется синтаксической вложенностью
- Не порядком вызова
- Рекурсия и перемещение функций могут усложнять понимание

ПРИМЕР

```
 1 V1 = 1
 2 V2 = 2
 3
 4 def f1(start):
 5     V2 = 22
 6     V3 = start
 7
 8     def f2():
 9         nonlocal V2, V3
10         print(V2, V3)
11         V2 += 1
12         V3 += 1
13
14 # ..................................
```

ПРИМЕР

```
 1 # ..............................
 2     V2 = 222
 3
 4     return f2
 5
 6 def f3(start):
 7     return f1(start)
 8
 9 g0 = f1(0)
10 g1 = f3(10)
11 g2 = f3(20)
12
13 # ..............................
```

ПРИМЕР

```
 1 # ..............................
 2
 3 print()
 4 for _ in range(5):
 5     g0()
 6 print()
 7 for _ in range(5):
 8     g1()
 9 print()
10 for _ in range(5):
11     g2()
```

СТАТИКА И ДИНАМИКА

- При каждом вызове f1 создается своя f2
- Привязка к объемлющим переменным создается относительно конкретного экземпляра f1
- Но значения не фиксируются
- При их изменении кем-то функция это заметит

ИНТЕРЕСНОЕ РАЗВИТИЕ

- Локальная функция видит объемлющую
- И может вызвать ее рекурсивно
- И передать ей параметры
- В частности из того, что она видит определенным в объемлющей
- Развитие темы - на самостоятельное рассмотрение

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ И МОДУЛИ

- Функция как объект может переехать из модуля в модуль
- Но глобальным контекстом для нее будет тот модуль, в котором она определена
- Классический Python-код не испортит модуль, из которого его вызвали

ПРИМЕР

```
1 # m.py
2 V1 = 5
3
4 def f():
5     global V1
6     V1 = 55
```

ПРИМЕР

```
1 # main.py
2
3 from m import f, V1 as mV1
4
5 V1 = 123
6
7 f()
8
9 #..................
```

ПРИМЕР

```python
# main.py

#.................................
print(V1, mV1)

import m

print(m.V1)

print(id(V1), id(fV1), id(m.V1))
print(id(f), id(m.f))
```

СТРУКТУРА

- Ключевое слово class
- Имя, двоеточие, тело со смещением

```
1 # Простейшее определение
2 class C:
3     pass
```

КЛАСС КАК ОБЪЕКТ

- Класс сам по себе является объектом
- В теле класса можно определять переменные
- И исполнять обычный код
- Этот код исполняется сразу

КОД УРОВНЯ КЛАССА

```
 1 print(1)
 2
 3 class C:
 4     print('C.1')
 5     V = 1234
 6     print('C.2')
 7
 8 def m():
 9     print(222)
10
11 print(2)
12 print(C.V)
```

АТРИБУТЫ КЛАССА

- Переменная, определенная на уровне класса, становится свойством класса как объекта
- Аналог статических полей класса в Java/C++
- Но не во всем
- В момент инициализации переменных класса самого класса еще нет

ПРИМЕР

```
 1 class Config:
 2     __SEC_PER_MIN = 60
 3
 4     TIMEOUT_MIN = 100
 5
 6     TIMEOUT_SEC = TIMEOUT_MIN * __SEC_PER_MIN
 7
 8     # так нельзя - Config еще не существует
 9     #TIMEOUT_SEC = Config.TIMEOUT_MIN *
10                    Config.__SEC_PER_MIN
11
12 print(Config.TIMEOUT_SEC)
```

АЛЬТЕРНАТИВА

```
 1 class Config:
 2
 3     TIMEOUT_MIN = 100
 4
 5     @classmethod # детали попозже
 6     def initialize(cls)
 7         sec_per_min = 60
 8         cls.TIMEOUT = cls.TIMEOUT_MIN * sec_per_min
 9
10 Config.initialize()
11
12 print(Config.TIMEOUT)
```

МЕТОДЫ/ФУНКЦИИ КЛАССА

- Внутри класса можно определять функции
- Это экзотичная практика, но так можно
- И вызывать их через ссылку на класс
- В схеме разрешения ссылок на переменные уровень класса не участвует
- К методам это тоже относится

ПРИМЕР

```
 1 V = 12
 2 class C:
 3     V = 23
 4
 5     def hello(name='world'):
 6         print('hello', name)
 7
 8     def m():
 9         print(V)
10         print(C.V)
11
12 C.hello()
13 C.m()
```

СОЗДАНИЕ ОБЪЕКТА

- Вызываем класс как функцию
- Надо ли передавать параметры - зависит от наличия символа

  `'__init__'`

  в пространстве имен класса
- Если ничего специально не делать - его там нет
- Тогда параметры не нужны

ПРИМЕР

```
1 class C:
2     pass
3
4 c1 = C()
5 c2 = C()
6 print(c1, c2)
```