PYTHON

Лекция 9

ПЛАН ЛЕКЦИИ

- Функции

ЧТО ПРЕДЛАГАЕТ PYTHON

- Функция только с позиционными параметрами - в конце псевдопараметр /
- Функция с позиционными и смешанными параметрами - между ними псевдопараметр /
- Функция только с именованными параметрами - в начале псевдопараметр *
- Функция со смешанными и именованными параметрами - между ними псевдопараметр *

ПРИМЕР

```
1 def sum(a, b, /):
2     return a + b
3
4 print(sum(1, 2))
5 #print(sum(a=1, b=2))
6 #print(sum(a=1, 2))
7 #print(sum(b=1, a=2))
8 #print(sum(b=1, 2))
```

ПРИМЕР

```
1 def is_near(a, b, /, epsilon):
2     return abs(a - b) < epsilon
3
4 print(is_near(1, 1.1, 0.01))
5 print(is_near(1, 1.0001, 0.01))
6 print(is_near(1, 1.0001, epsilon=0.01))
```

ПРИМЕР

```
1 def find_car(*, year, mileage):
2     pass
3
4 #find_car(1995, 50000)
5 find_car(year=1995, mileage=50000)
6 find_car(mileage=50000, year=1995)
```

ПРИМЕР

```
1 def update_car(id, *, year, mileage):
2     pass
3
4 #update_car('qwertyuio', 1995, 50000)
5 update_car('qwertyuio', year=1995, mileage=50000)
6 update_car('qwertyuio', mileage=50000, year=1995)
7 update_car(id='qwertyuio', year=1995, mileage=50000)
8 update_car(id='qwertyuio', mileage=50000, year=1995)
```

ЧТО ЕЩЕ

- Можно использовать оба знака сразу
- Отсутствие знаков - это как будто начинается с / и заканчивается на *
- Можно поставить рядом / и *
- Тогда не будет смешанных параметров

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

- Именованные могут иметь или не иметь значения по умолчанию независимо от остальных
- Позиционные и смешанные - все со значениями по умолчанию должны быть справа
- В этом смысле они - одна группа

В ТОЧКЕ ВЫЗОВА

- Смешанные становятся именованными или позиционными (по отдельности, но в рамках правил)
- Именованные справа, позиционные слева
- Каждый параметр без значения по умолчанию должен быть упомянут - позиционно или именованно
- Не должно быть двойного "присваивания" или отсутствия такового

ПРИМЕР

```
1 def sum(a, b=0, /):
2     return a + b
3
4 print(sum(1, 2))
5 print(sum(1))
```

ПРИМЕР

```
1 def is_near(a, b=0, /, epsilon=0.0001):
2     # epsilon не может быть без значения по умолчанию
3     return abs(a - b) < epsilon
4
5 print(is_near(1))
6 print(is_near(1, 1.1))
7 print(is_near(1, 1.1, 0.01))
8 print(is_near(1, 1.0001, epsilon=0.01))
9 print(is_near(1, epsilon=0.01))
```

ПРИМЕР

```
1 def find_car(*, year=2000, mileage=30000):
2     pass
3
4 find_car(year=1995, mileage=50000)
5 find_car(mileage=50000, year=1995)
6 find_car(mileage=50000)
7 find_car(year=1995)
8 find_car()
```

ПРИМЕР

```
1 def update_car(id='0', *, year, mileage):
2     pass
3
4 update_car('qwertyuio', year=1995, mileage=50000)
5 update_car(mileage=50000, year=1995)
6 update_car(year=1995, mileage=50000, id='qwertyuio')
7 update_car(id='qwertyuio', mileage=50000, year=1995)
```

ПРИМЕР

```
1 def f(a, /, b, c, *, d):
2     print(a, b, c, d)
3
4 f(1, 2, 3, d=5)
5 # f(1, 2, d=5, 3)
6 f(1, 2, c=3, d=5)
7 f(1, 2, d=5, c=3)
```

ПРИМЕР

```
1 def f(a, /, b, c=11, *, d):
2     print(a, b, c, d)
3
4 f(1, 2, 3, d=5)
5 # f(1, 2, d=5, 3)
6 f(1, 2, c=3, d=5)
7 f(1, 2, d=5, c=3)
```

ПРИМЕР

```
1 def f(a, /, b, c=11, *, d):
2     print(a, b, c, d)
3
4 f(1, 2, d=10)
5 f(1, 2, 20, d=10)
6 # f(1, 2, 10, c=10, d=10)
```

!!! ВАЖНЫЙ ФАКТ !!!

- Параметр по умолчанию - это атрибут функции-объекта
- Если его изменить - он изменится
- И у в других вызовах будет измененное значение
- Надо аккуратно со списками, множествами, словарями...

ПЕРЕМЕННОЕ КОЛИЧЕСТВО ПАРАМЕТРОВ

- Иногда мы сразу не знаем, сколько параметров будет
- Как для позиционных, так и для именованных
- Например, в функции типа max
- В одном месте хотим от пяти аргументов, в другом от трех

ПЕРЕМЕННОЕ КОЛИЧЕСТВО ПАРАМЕТРОВ

- Или мы в вызове указываем свойства объекта
- В качестве именованных параметров
- Какие-то могут оказаться известными заранее, а какие-то - нет

ПЕРЕМЕННОЕ КОЛИЧЕСТВО ПАРАМЕТРОВ

- Перед параметром в описании функции может стоять * или **
- * - параметр принимает в себя переменное число позиционных параметров в точке вызова
- ** - переменное число именованных параметров
- * - внутри превращается в кортеж, ** - в словарь

ПРИМЕР

```
 1 def f(*args):
 2     print(type(args))
 3     print(len(args))
 4     for i, v in enumerate(args):
 5         print(i, v)
 6
 7 f()
 8 f(1)
 9 f(1, 2)
10 #f(1, 2, n=3) # так нельзя
```

ПРИМЕР

```
 1 def f(a, *args):
 2     print(a)
 3     print(type(args))
 4     print(len(args))
 5     for i, v in enumerate(args):
 6         print(i, v)
 7
 8 #f() - нельзя
 9 f(1)
10 f(a=1)
11 f(1, 2)
12 #f(a=1, 2) # так нельзя
```

ПРИМЕР

```
 1 def f(a=1, *args):
 2     print(a)
 3     print(type(args))
 4     print(len(args))
 5     for i, v in enumerate(args):
 6         print(i, v)
 7
 8 f()
 9 f(a=22)
10 f(22)
11 f(33, 44)
12 # f(a=33, 44)
```

СТРАННОСТИ

- *-параметр не любит псевдопараметров
- В чем-то это логично
- Местами - не совсем
- Например, нельзя так:

```
def f(*args, /, *, name):
```

ПРИМЕР

```
 1 def f(*args, name):
 2     print(name)
 3     print(type(args))
 4     print(len(args))
 5     for i, v in enumerate(args):
 6         print(i, v)
 7
 8 f(name=1)
 9 f(77, name=1)
10 f(1, 2, name=1)
```

ЛОГИКА PYTHON

- *args сам по себе является границей перед строго именованными параметрами
- * - особый случай
- После параметра со звездочкой начинаются именованные параметры
- Поэтому и *args перед / запрещен

ПРИМЕР

```
1 def f(**kwargs):
2     print(type(kwargs))
3     print(kwargs)
4
5 f()
6 f(name=1)
```

ПРИМЕР

```
1 def f(name, **kwargs):
2     print(type(kwargs))
3     print(kwargs)
4
5 f(123)
6 f(name=1)
7 f(234, name=1)
```

ПРИМЕР

```
1 def f(name=111, **kwargs):
2     print(type(kwargs))
3     print(kwargs)
4
5 f()
6 f(123)
7 f(name=1)
8 f(234, name=1)
```

ЛОГИКА PYTHON

- **kwargs - тоже со звездочкой
- Но с просто звездочкой сочетается
- С / - тоже сочетается, если идет справа
- После **kwargs никакие параметры идти не могут

ПРИМЕР

```
1 def f(name=111, /, **kwargs):
2     print(name)
3     print(type(kwargs))
4     print(kwargs)
5
6 f()
7 f(123)
8 f(1, k=2, name=2)
9 f(234, name=1)
```

ПРИМЕР

```
1 def f(a=1, /, b=2, *, c, **kwargs):
2     print(type(kwargs))
3     print(kwargs)
4
5 f(c=1)
6 f(123, c=3, b=5, d=123)
```

ОСОБЫЙ ПРИМЕР

```
1 def f(*args, **kwargs):
2     print(args)
3     print(kwargs)
4
5 f()
6 f(c=1)
7 f(123, c=3, b=5, d=123)
8 f(123, 234, c=3, b=5, d=123)
```

* И ** В ТОЧКЕ ВЫЗОВА

- Визуально напоминает * и ** в точке определения
- Смысл - другой
- Содержимое последовательности/словаря "вписывается" в текущую точку

ОСОБЫЙ ПРИМЕР

```
1 def sum(a, b):
2     return a + b
3
4 def f(lst):
5     return [sum(*e[:2]) for e in data if len(e) ≥ 2]
```

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

- Функция в Python всегда что-то возвращает
- Либо то, что указано в return
- Если ничего не указано, то None
- Если нет return, то тоже None

ФУНКЦИИ КАК ОБЫЧНЫЕ ЗНАЧЕНИЯ

- Функция может передаваться как параметр
- Функция может создаваться как значение
- Функция может определяться внутри другой функции
- При каждом вызове это будет новая функция

АНОНИМНЫЕ ФУНКЦИИ

- Не хотим создавать короткие одноразовые функции через def
- Есть ключевое слово lambda
- Для интерпретатора они почти не отличаются - кроме атрибута __name__
- Но есть синтаксическое ограничение - lambda состоит из одного выражения

МОДУЛИ

- Мы не можем все писать в одном файле
- Простейшая форма декомпозиции - модули
- Простейший вариант модуля - файл в том же каталоге
- Создадим функцию в отдельном файле

МОДУЛЬ

```python
# fibo.py

def f(n):
    curr, prev = 1, 0
    for _ in range(n):
        curr, prev = curr + prev, curr
    return prev
```

МОДУЛЬ: ИСПОЛЬЗОВАНИЕ

```
1 # main.py
2
3 import fibo
4
5 print(fibo.f(10))
6 print(type(fibo))
7 print(dir(fibo))
```

МОДЕЛЬ МОДУЛЕЙ

- Модуль - тоже объект
- Создается по команде import
- Команда import находит файл fibo.py
- В списке каталогов, определяемых конфигурацией, переменными окружения и т.п.
- В списке есть каталог, где лежит тот, кто инициировал импортирование

МОДЕЛЬ МОДУЛЕЙ

- Код модуля исполняется в контексте импортирования
- Что было на верхнем уровне - исполнится
- Все определения функций исполнятся
- Будут созданы объекты-функции

МОДЕЛЬ МОДУЛЕЙ

- Их имена станут свойствами объекта-модуля
- Аналогично - с глобальными переменными
- И с классами
- И с его собственными импортами

МОДУЛЬ: ИСПОЛЬЗОВАНИЕ

```python
# fibo.py

def f(n):
    curr, prev = 1, 0
    for _ in range(n):
        curr, prev = curr + prev, curr
    return prev

print("I'm fibo module:", id(f))
```

МОДУЛЬ: ИСПОЛЬЗОВАНИЕ

```
1 # main.py
2
3 import fibo
4
5 print("I'm main:", id(fibo.f))
6
7 import fibo
8
9 print("I'm main:", id(fibo.f))
```

МОДЕЛЬ МОДУЛЕЙ

- import - не какая-то особая конструкция
- Может встречаться где угодно
- Может повторяться
- Но при повторении повторного исполнения кода модуля не происходит

ВАРИАНТЫ ИМПОРТА

- Классика:

```
import mname
```

- Создает отдельный объект-модуль с именем mname
- Все символы, определенные в модуле - его атрибуты
- Если не знаете, какой вариант лучше - используйте этот

ВАРИАНТЫ ИМПОРТА

- Классика:

```
import mname as alias
```

- Ищет модуль по имени mname
- Создает отдельный объект-модуль с именем alias
- Все символы, определенные в модуле - его атрибуты

МОДУЛЬ: ПЕРЕИМЕНОВАНИЕ

```python
# fibo.py

def f(n):
    curr, prev = 1, 0
    for _ in range(n):
        curr, prev = curr + prev, curr
    return prev

print("I'm fibo module:", id(f))
```

МОДУЛЬ: ПЕРЕИМЕНОВАНИЕ

```python
# main.py

import fibo as f

print("I'm main:", id(f.f))

import fibo

print("I'm main:", id(fibo.f))
```

ВАРИАНТЫ ИМПОРТА

- Переименование не влияет на контроль за однократной инициализацией
- Удобно, когда возникает конфликт имен
- Или имя модуля длинное
- Или есть идиоматичное сокращение (np для numpy, pd для pandas)