eebb

ЯЗЫК PYTHON

Лекция 8. Функции. Часть 2

Пример: смешанные и строго именованные

```
 1 def f(a=20, b=50, *, c=100):
 2     print(a + b + c)
 3
 4 f()
 5 f(1)
 6 f(a=1)
 7 f(b=1)
 8 f(1, 2)
 9 f(1, c=2)
10 f(1, 2, c=3)
```

Пример: только строго именованные

```
 1 def f(*, a=20, b=50, c=100):
 2     print(a + b + c)
 3
 4 f()
 5 f(c=1)
 6 f(a=1)
 7 f(b=1)
 8 f(b=1, a=2)
 9 f(a=1, c=2)
10 f(a=1, b=2, c=3)
```

Пример: проблемы

```
1 def f(a=20, b=50, *, c=100):
2     print(a + b + c)
3
4 f(1, 2, b=3)  # Двойное присваивание b
5 f(a=1, 2)     # Именованные перед позиционными
6 f(1, a=1)     # Двойное присваивание a
7 f(1, 2, 3)    # Слишком много позиционных
```

Локальные и глобальные переменные

- Внутри функции - свой блок кода
- И это сильно отличается от блоков if/while
- Когда программа запускается, Python отслеживает все переменные
- Как они создаются и меняют значения
- Набор имен переменных с их значениями назовем пространством имен
- У программы есть пространство имен

Локальные и глобальные переменные

- То, что происходит внутри if/while/for - влияет на то же пространство имен
- С функцией - все по-другому
- В момент вызова создается новое пространство имен
- А то, которое было тоже используется
- Но по особым правилам

Локальные и глобальные переменные

- Локальный контекст - пространство имен внутри функции
- На момент начала в нем находятся параметры со своими значениями
- Глобальный контекст - пространство имен вне определений функций
- Может меняться между вызовами функции

Разберем на примере

```
 1 a = 1
 2
 3 def f(a, b):
 4     print(a + b)
 5     d = a + 1
 6     print(d)
 7
 8 f(1, 2)
 9 c = 10
10 f(2, 3)
```

Разберем контексты

- На момент определения f в глобальном контексте {a: 1}
- Определение функции добавляет f в глобальный контекст
- Вызов создает локальный контекст {a: 1, b: 2}
- Значение a в нем совпадает с a глобальном
- Но это просто повезло

Разберем контексты

- По завершении функции, локальный контекст прекращает существование
- Потом в глобальный добавляется c
- Новый вызов снова создает локальный контекст {a: 2, b: 3}
- На этот раз в локальном не равно a в глобальном
- И снова по завершении он уничтожается

Взаимодействие контекстов

- Если читаем переменную, сначала смотрим локальный контекст
- Если там переменная есть, берем ее значение
- Если нет, смотрим в глобальный контекст
- Если есть там, берем значение оттуда
- Если и там нет, то бросается исключение

Взаимодействие контекстов

- Если пишем в переменную, то пишем в локальный контекст
- По умолчанию
- А если хотим поменять переменную в глобальном контексте ?
- Или установить ?

Изменение глобального контекста из функции

- Первым делом надо подумать - а надо ли это ?
- Это не самая полезная практика
- Но если все-таки надо
- Есть ключевое слово global

Пример: изменение глобальной переменной

```
 1 a = 1
 2
 3 def f(b, c):
 4     global a
 5     a = b ** c
 6     print(a + b)
 7     d = a + 1
 8     print(c)
 9
10 print(a) # 1
11 f(2, 3)
12 print(a) # 8
```

Локальные функции

- На глобальном уровне функции похожи на переменные
- С точки зрения хранения
- На локальном уровне есть свои переменные
- Логично ожидать своих функций на локальном уровне
- И они есть

Мотивация локальных функций

- Как и для переменных
- Локальность повышает управляемость
- Если что-то нужно только локально, давайте держать локально
- Если функция нужна только для того, чтобы облегчить написание другой, давайте определим ее локально

Мотивация локальных функций

- Еще вариант
- У функции есть "двойник"
- Для реализации удобно ввести дополнительный параметр
- Но не показывать его во внешнем интерфейсе
- Определяем внутреннюю функцию
- И делегируем вызов к ней

Синтаксис локальных функций

- Просто помещаем def в тело функции
- И свое тело функции
- А внутри - можно еще def (хотя, надо подумать, а надо ли)
- И внутри if/while - тоже можно (и тоже подумать)

Пример: локальная функция

```
 1 def fibo(n):
 2     def helper(curr, prev, n):
 3         if n == 0:
 4             return prev
 5         return helper(curr + prev, curr, n - 1)
 6 
 7     return helper(1, 0, n)
 8 
 9 print(fibo(0)) # 0
10 print(fibo(1)) # 1
11 print(fibo(2)) # 1
12 print(fibo(3)) # 2
13 print(fibo(4)) # 3
14 print(fibo(5)) # 5
```

Контексты

- Ситуация с контекстами усложняется
- Добавляется новый уровень
- Или новые уровни (соответственно уровню вложенности)
- Создается по контектсту на каждый уровень вложенности

Поиск переменной при чтении

- Идем от самого внутреннего контекста
- К самому внешнему
- Какую переменную найдем - из такой значение и возьмем
- Не найдем - получим исключение

Разберем на примере

```
 1 a = 1
 2
 3 def f(a, b):
 4     print(a + b)
 5     d = a + 1
 6     def g(c, d):
 7         c1 = a + d + c
 8         print(c1, a, d, c)
 9     g(a, c)
10     d += 1
11     g(a, d)
12
13
14 c = 10
15 f(2, 3)
```

Разберем контекст

- До первого вызова g все уже знаем
- В момент первого вызова
  - Локальный контекст: c = 2, d = 10
  - Глобальный контекст: a = 1, c = 10
  - Между ними еще один контекст: a = 2, b = 3, d = 3

Разберем контекст

- В момент второго вызова
  - Локальный контекст: c = 2, d = 4
  - Глобальный контекст: a = 1, c = 10
  - Между ними еще один контекст: a = 2, b = 3, d = 4

Изменения в "промежуточных" контекстах

- Для изменения глобального контекста есть global
- Для других нелокальных - nonlocal
- Нелокальных может быть несколько
- Поэтому указываем рядом номер контекста
- Нумеруются от ближайшего к глобальному

Находим значения внутри g

- a нет в локальном, но есть в следующем
- d и c есть в локальном
- Из глобального ничего нет
- Если переименовать первый параметр f, то "откроется" глобальная переманная a

Статическая и динамическая вложенность

- Когда одна функция вызывает другую, точка вызова запоминается
- Если f1 вызвала f2, а f2 вызвала f3, мы запоминаем еще одну точку вызова
- И первую тоже помним
- И так вызовы могут вкладываться один в другой

Стек вызовов

- После возвращения, нам адрес возврата не нужен
- Но можем выполнить новый вызов
- Мы добавляем адреса возврата
- И избавляемся - всегда от последнего добавленного
- Это называется стек вызовов

Статическая и динамическая вложенность

- Стек вызовов описывает некоторую вложенность
- Она возникает во время исполнения
- Это динамическая вложенность
- А статическая - это вложенность по месту определения
- Глобальная ли функция, или вложенная
- Или вложенная во вложенную

Статическая и динамическая вложенность

- В любой момент мы находимся в каком-то локальном контексте
- И есть несколько контекстов
- И их набор определяется статической вложенностью
- А при вызове мы меняем этот набор контекстов
- Уже динамически

Разберем на примере

```
 1 v1 = 5
 2 def f(a):
 3     print('f', v1, a)
 4     v2 = v1 + a
 5
 6     def g(b):
 7         print('g', v1, v2, a)
 8         v3 = v1 + v2 + a
 9         f(5)
10
11     if a == 0:
12         g(1)
13
14 print('start')
15 f(0)
```

Порядок вызовов

- Сначала вызываем f(0)
- Потом вызываем g(1)
- Дальше - f(5)
- Потом возвращаемся

Контексты в примере

- В обоих вызовах f у нас два контекста
- А в вызове g - три
- В обоих вызовах f видим v1 и v2
- И не видим v3

Локальные функции и локальные переменные

- Принципиально они схожи
- Но есть практические отличия
- Функции более громоздки
- Поэтому локально вводятся гораздо реже, чем переменные

Функции как параметры

- Здесь - тоже принципиальная схожесть
- Можем передавать обычные значения
- Почему бы не передавать функцию
- Например, функцию для вычисления расчетного листка
- В функцию для вычисления стоимости заказа

Функции как параметры

- Другой пример: поэлементная обработка
- Хотим по списку получить другой список
- Применив преобразование данных поэлементно
- Или - отбросить ненужное, оставить нужное
- Нужное определяется переданной функцией
- Которая применяется к каждому элементу

Рассмотрим пример

```
 1 def change(data, func):
 2     for index, value in enumerate(data):
 3         data[index] = func(data[index])
 4
 5 def square(x):
 6     return x * x
 7
 8 data = [1, 2, 3]
 9 change(data, square)
10 print(data)
```

lambda-функции

- Для передачи параметром часто нужны маленькие простые функции
- И в разных местах разные
- Неудобно специально создавать полное определение функции
- Только чтобы один раз передать ее параметром

lambda-функции

- Выход - lambda-функции
- Однострочно определенные функции
- Ключевое слово lambda
- Потом список параметров без скобок
- Двоеточие, и сразу - возвращаемое значение

Пример на lambda

```
1 def change(data, func):
2     for index, value in enumerate(data):
3         data[index] = func(data[index])
4 
5 data = [1, 2, 3]
6 change(data, lambda x: x * x)
7 print(data)
```

Возвращение функции

- Функцию можно возвращать
- Простой пример
- При расчете стоимости авиабилета может применяться скидка
- Их несколько вариантов
- Каждая рассчитывается по своей формуле

Возвращение функции

- Вариант скидки выбирается
- В зависимости от дня, рейса, количества проданных билетов
- Пишем функцию, возвращающую функцию, вычисляющую скидку по стоимости билета
- В простом варианте это отдельно написанные функции
- И мы выбираем нужную через if

Замыкания

- Можем определить локальную функцию
- И ее вернуть
- Локальная функция можем ссылаться на объемлющий контекст
- По если ее вернуть из функции, то она покидает этот контекст
- Выглядит как проблема

Пример

```
1 def adder(delta):
2     return lambda v: v + delta
3
4 incr = adder(1)
5 add_5 = adder(5)
6 print(incr(10))
7 print(add_5(10))
```

Разбор примера

- Определяем функцию в локальном контексте
- В котором есть delta
- И delta используется внутри функции
- Но мы ее "вынимаем" из контекста
- И можем ее использовать там, где delta не определена

Разбор примера

- Python вместе с функцией сохраняет все ссылки на нелокальные объекты
- То есть отделяясь от изначального контекста
- Функцию "уносит с собой" ссылку на важную для нее часть контекста
- Можно получить забавные эффекты

Пример забавного эффекта

```
 1 def counter(start, step):
 2     state = start
 3 
 4     def incr():
 5         nonlocal state
 6         state += step
 7 
 8     return incr, lambda : state
 9 
10 adder_1, getter_1 = counter(10, 2)
11 adder_2, getter_2 = counter(5, 1)
12 adder_1()
13 adder_1()
14 print(getter_1())
15 adder_2()
```

Переменное количество параметров

- Предположим мы хотим иметь функцию для вычисления среднего арифметического
- Из переменного количества значений
- Можно передавать в функцию список
- Отличный вариант, если данные - уже в списке
- А собирать их в список ради только вызова - не очень красиво

Переменное количество параметров

- Перед именем параметра в определении поставим *
- Если параметр позиционный или смешанный
- И идет в конце списка
- То ему могут соответствовать много параметров в точке вызова
- В том числе - 0 или 1

Пример переменного числа параметров

```
1 def mean(*values):
2     total = 0
3     for v in values:
4         total += v
5     return total / len(values)
6
7 print(mean(1))
8 print(mean(2, 3))
9 print(mean(1, 5, 123))
```