GeekBrains

Основы Python

Урок 3. Функции. Словари

На этом уроке:

1. Рассмотрим синтаксис создания функций в Python.
2. Поговорим о словарях.
3. Проясним разницу между позиционными и именованными аргументами.
4. Изучим возможности модуля random.
5. Рассмотрим полезные встроенные функции filter(), map(), zip().

Оглавление

Дополнительные материалы

Используемая литература

Что такое функция в Python

Опыт выполнения домашних заданий предыдущих уроков наверняка подсказывает вам, что нужен какой-то способ для повторного использования уже написанного кода. Это позволило бы приблизиться к выполнению одной из самых важных заповедей в разработке — DRY (не повторяйся). Один из способов был изобретен очень давно — обособление фрагментов кода в функции. Мы уже говорили о них. Функции, реализованные в рамках конкретных классов, называются методами — про них тоже упоминали. Давайте напишем свою функцию для парсинга `url`-адреса:

```
def url_parse(url):
    _t_protocol, _, domain, *resource_address = url.strip('/').split('/')
    t_protocol = _t_protocol[:-1]
    return t_protocol, domain, resource_address
```

Что мы сделали? Взяли код из предыдущего урока и «обернули» его функцией. Описание функции начинается с ключевого слова `def`. Дальше идет пробел и начинается имя функции в стиле snake_case. Имя должно иметь смысл и быть не слишком длинным. Обычно в организациях есть внутренние правила именования функций. В нашем случае все логично: парсер `url`-адреса. Сразу после имени функции идет пара круглых скобок (никаких лишних пробелов!). В них можно указать аргументы (исходные данные) функции. У нас для парсинга нужно знать только сам адрес — его и делаем аргументом. Имена аргументов тоже нужно задавать со смыслом. Желательно, чтобы они не встречались нигде в основном коде. Часто методы объектов (они ведь тоже функции) не имеют аргументов вообще — это нормально. Помните метод списков `.pop()`? После скобок идет двоеточие и начинается тело функции, которое смещено на один индент (4 пробела) относительно предыдущего уровня.

Важно: до и после функции должно быть по две пустых строки! Для методов класса — по одной пустой строке.

По сути тело функции — это тот самый код, который мы хотим повторять многократно в нашей программе. Создание функций не такая уж и сложная задача. Сначала просто пишем код, «чтобы работало». Потом смотрим, есть ли повторяющиеся фрагменты. Если есть, анализируем, какие данные нужны для работы конкретного фрагмента и какой результат получается в итоге. Исходные данные становятся аргументами. Результат — тем, что функция возвращает. В нашем случае это кортеж, записанный после ключевого слова `return` (вернуть). Придумываем функции имя, и готово. Теперь нужно функцию вызвать (call) в тех участках кода, где были те самые повторяющиеся фрагменты.

Например, нам нужно распарсить три адреса:

```
url = 'https://geekbrains.ru/teacher/lessons/7961'
_t_protocol, _, domain, *resource_address = url.strip('/').split('/')
t_protocol = _t_protocol[:-1]
print(t_protocol, domain, resource_address)

url = 'https://www.citilink.ru/catalog/mobile/notebooks/1202859/'
_t_protocol, _, domain, *resource_address = url.strip('/').split('/')
t_protocol = _t_protocol[:-1]
print(t_protocol, domain, resource_address)

url = 'https://www.dns-shop.ru/catalog/17a89aab16404e77/videokarty/'
_t_protocol, _, domain, *resource_address = url.strip('/').split('/')
t_protocol = _t_protocol[:-1]
print(t_protocol, domain, resource_address)
```

Такое решение вполне нормально для начинающего разработчика. Но как быть, если адресов будет 1000? Так не пойдёт. Теперь решение с использованием написанной выше функции:

```
url_1 = 'https://geekbrains.ru/teacher/lessons/7961'
url_1_parsed = url_parse(url_1)
print(url_1_parsed)

url_2 = 'https://www.citilink.ru/catalog/mobile/notebooks/1202859/'
print(url_parse(url_2))

print(url_parse('https://www.dns-shop.ru/catalog/17a89aab16404e77/videokarty/'))
```

Мы осознанно показали три разных варианта работы с функцией. В первом случае результат вызова функции помещаем в отдельную переменную `url_1_parsed`. Обычно так поступаем, если с этим результатом надо что-то еще делать в коде: передать другой функции, использовать в нескольких участках кода и т.д. Как всё происходит? `Python`, как обычно, выполняет программу строка за строкой. Как только он доберётся до первого вызова функции (вторая строка примера), он прерывает выполнение основного кода и запускает тело функции с исходными данными, которые мы передали в точке вызова, — значение переменной `url_1`. **Пока** будем думать, что тело функции ничего не знает об основном коде (внешнем мире), единственное, что известно — это значения аргументов (в нашем случае аргумент один). Важно понимать этот факт, когда вы определяете, что будет аргументами конкретной функции. После того как тело функции выполнится, если это необходимо, результаты передаются в **точку вызова** (в основной код). Бывают функции, которые ничего не возвращают — в них просто не пишем оператор `return`. В `Python` это нормально — на самом деле любая функция возвращает `None` по умолчанию. Вспомним те же методы `.append()` или `.sort()` для списков. Вернемся к примеру: что вернула функция после вызова "`url_parse(url_1)`"? Правильно — кортеж. Этот кортеж присваивается в переменную `url_1_parsed`, с которой можем в дальнейшем выполнять какие-либо манипуляции. Мы, например, просто вывели ее значение в консоль. Вам

сейчас очень важно понять эту цепочку: вызов функции с конкретными значениями аргументов, выполнение тела функции, возврат результатов в точку вызова.

Для второго адреса не стали использовать ненужную по сути в данном случае промежуточную переменную и сразу вывели в консоль результат работы функции — то, что она вернула в точку вызова.

Для третьего адреса сделали ещё проще: не стали даже переменную для хранения самого адреса создавать. Передали его напрямую.

Как вы уже догадались, в зависимости от ситуации вам может быть полезен один из показанных примеров.

*Примечание: вы заметили, что мы добавили вызов метода `.strip()` при парсинге адреса? На прошлом уроке его не было. Пришло время развивать навык чтения «доков» — документации. Теперь вы этим будете заниматься постоянно. Если читать лень, просто уберите его и сравните результаты. Заметили разницу? Возможно, чуть позже вы её точно заметите.

Важно помнить, что в `Python` всё является объектом, даже функции:

```
...
print(type(url_parse), id(url_parse))
# <class 'function'> 30593640
new_url_parse = url_parse
print(type(new_url_parse), id(new_url_parse))
# <class 'function'> 30593640
print(new_url_parse('https://cloud.mail.ru/home/'))
# ('https', 'cloud.mail.ru', ['home'])
```

Здесь мы создали псевдоним (alias) для нашей функции. То есть вызов “`new_url_parse()`” полностью эквивалентен вызову “`url_parse()`”. Данный пример является чисто синтетическим, в реальном коде так делать смысла нет.

*Это загадочное слово `callback`: передаём функцию как объект

Очень часто бывают ситуации, когда нам нужно передать функцию как объект. В этой ситуации на сленге говорят «передаём callback». Пусть есть список вещественных чисел, записанных как строки (например, получили их в результате ввода с клавиатуры). Нужно вычислить их сумму. Первое, что приходит в голову, — использовать цикл `for in` (надеемся, вы не собирались использовать цикл `while`):

```
nums = ['1578.4', '892.4', '354.1', '871.5']
summ = 0
for num in nums:
   summ += float(num)
print(summ)  # 3696.4
```

Если бы список состоял именно из чисел, решение было бы гораздо проще:

```
nums = [1578.4, 892.4, 354.1, 871.5]
print(sum(nums))  # 3696.4
```

В `Python` можно сделать первое решение таким же простым:

```
nums = ['1578.4', '892.4', '354.1', '871.5']
print(sum(map(float, nums)))  # 3696.4
```

Пример сложный, но попробуем разобраться. Начнем с известного: функция `sum()` суммирует числа. Что является аргументом функции `sum()`? Результат выполнения другой функции — map(). Эта очень крутая функция часто используется в `Python`-программах. Не будем пока погружаться в тему итераторов, представим пока работу этой функции по аналогии с солнечными очками. Они пропускают через себя свет, но при этом преобразуют его — делают темнее, возможно, поляризуют. Так и `map()` пропускает через себя элементы последовательности (второй аргумент) и при этом преобразует их в соответствии с тем `callback`, который мы передали первым аргументом. Что передали в этом примере? Объект функции `float()`, которая преобразует строку в вещественное число. **Очень важно** понять разницу между `float` и `float()`. Первое — `callback`, второе — результат вызова (`call`). Попробуйте дописать в этом примере скобки после `float` и посмотрите, что будет. Даже разработчик уровня `junior` должен понимать эту разницу. Поэтому возвращайтесь к примеру день за днём, пока не поймёте.

Закрепим еще одним «хардовым» примером:

```
...
urls = ['https://geekbrains.ru/teacher/lessons/7961',
        'https://www.citilink.ru/catalog/mobile/notebooks/1202859/',
        'https://www.dns-shop.ru/catalog/17a89aab16404e77/videokarty/']
urls_parsed = map(url_parse, urls)
print(*urls_parsed, sep='\n')
# ('https', 'geekbrains.ru', ['teacher', 'lessons', '7961'])
# ('https', 'www.citilink.ru', ['catalog', 'mobile', 'notebooks', '1202859'])
# ('https', 'www.dns-shop.ru', ['catalog', '17a89aab16404e77', 'videokarty'])
```

Здесь вы увидели квинтэссенцию эффективности идеи функции и `callback`. Это и есть `DRY`. При помощи `map()` мы применили `callback` `url_parse` для всех элементов списка `urls`. Получили последовательность распарсенных адресов (по секрету уточним, что это не список, а итератор — подробности на следующих уроках). Дальше мы при помощи оператора `*` распаковали эту последовательность для вывода в консоль (про распаковку сегодня ещё поговорим). Аргумент `sep='\n'` написали, чтобы разделить элементы при выводе управляющим символом перевода строки `\n`. Ведь `sep` — это сокращение от `separator`.

Документирование кода функций

Рано или поздно вы будете использовать библиотеки в своём коде. Как уменьшить время, которое уходит на знакомство с чужим кодом? Читать документацию — вариант, но что если поместить документацию прямо в код?! В Python-разработке это приветствуется. Даже есть понятие docstring.

Одна строка

Краткий вариант — написать пояснение в одну строку:

```
def url_parse(url):
    """Parses url address in tuple of items"""
    _t_protocol, _, domain, *resource_address = url.strip('/').split('/')
    t_protocol = _t_protocol[:-1]
    return t_protocol, domain, resource_address
```

Важно: можно использовать только тройные двойные (не одинарные!) кавычки.

Теперь можно получить эту справку разными способами: удерживая `<Ctrl>` и подводя курсор к части кода, где упоминается функция, или при помощи функции help():

```
print(help(url_parse))
# Help on function url_parse in module __main__:
#
# url_parse(url)
#     Parses url address in tuple of items
#
# None
```

Обратите внимание, что мы опять используем `callback`.

Несколько строк

Многострочные комментарии позволяют подробнее описать код:

```
def url_parse(url):
    """
    Parses url address in tuple of items

    :param url: URL address
    :return: tuple of url address parts
    """

    _t_protocol, _, domain, *resource_address = url.strip('/').split('/')
    t_protocol = _t_protocol[:-1]
    return t_protocol, domain, resource_address
```

Помните, что хороший код — это код, понятный с минимальным количеством комментариев. Поэтому пишите пояснения только когда это реально необходимо. Очевидные вещи не стоит комментировать.

Области видимости переменных

При знакомстве с функциями сегодня была оговорка: «Будем думать, что тело функции ничего не знает об основном коде (внешнем мире)». Пришло время уточнить этот момент. На самом деле из тела функции видны внешние переменные:

```
def say_hello():
    print(name)


name = 'Иван'
say_hello()  # Иван
```

Этот пример **очень-очень-очень плохой**. Никогда так не пишите. Здесь нарушается принцип «явное лучше, чем неявное». Как в тело функции попало значение переменной `name`? Непонятно. Особенно ярко это можно увидеть, если вставить 300 строк кода между телом функции и объявлением переменной `name` в коде. А если переменная `name` так и не будет объявлена? Всё сломается:

```
def say_hello():
    print(name)


say_hello()  # ...NameError: name 'name' is not defined
```

Чтобы понять тонкости, подумаем над кодом:

```
def say_hello():
    name = 'Петр'
    print(name)
```

```
name = 'Иван'
say_hello()  # Петр
```

В чем разница с первым примером? Здесь переменная `name` объявлена дважды: в теле функции и в основном коде. Какое значение взял Python при выводе в консоль? Логика простая: то, которое ближе. Представим, что вам нужно купить гаджет (пусть это будет iPhone). Где вы будете искать сначала? В магазинах вашего города (в теле функции). Если нашли — берёте. Если нет — будете искать за пределами вашего города (в основном коде). Если нашли — берете. Если нет — покупка не удалась. Именно так Python ищет переменные. Для обобщения придумали области видимости:

- **локальная** — самая близкая (в нашем случае это тело функции);
- **глобальная** — то, что по идее видят все функции и не только (в нашем случае — основной код).

Для закрепления пример:

```
def say_hello_wrapper():
    name = 'Петр'

    def say_hello():
        print(name)

    say_hello()


name = 'Иван'
say_hello_wrapper()  # Петр
```

В какой последовательности `Python` искал переменную `name` для вывода в функции `say_hello()`? Сначала в самой функции `say_hello()`, не нашел. Потом на уровень выше — в функции `say_hello_wrapper()`. Нашёл — всё. Что будет, если мы уберём объявление name в функции `say_hello_wrapper()`? `Python` будет искать дальше и найдёт в глобальной области.

Надеемся, что механизм стал вам понятен.

Теперь вопрос посложнее. Можно ли изменить значение глобальной переменной из локальной области? Например, вот так:

```
def say_hello():
    name = 'Петр'
    print(name)
```

```
name = 'Иван'
say_hello()  # Петр
print(name)  # Иван
```

Получается, что нет. Здесь действует правило полупрозрачного зеркала: вы из локальной области видите глобальную, а из глобальной области посмотреть в локальную невозможно. Подтверждение:

```
def say_hello():
   name = 'Петр'
   print(name)


say_hello()  # Петр
print(name)  # ...NameError: name 'name' is not defined
```

Но в Python есть «запрещенный» приём, позволяющий «пробросить» часть локальной области в глобальную. Очень-очень-очень старайтесь избегать его, чтобы избежать проблем с поддержкой кода в будущем:

```
def say_hello():
   global name
   name = 'Петр'
   print(name)


name = 'Иван'
print(name)  # Иван
say_hello()  # Петр
print(name)  # Петр
```

При помощи оператора `global` мы смогли поменять статус переменной `name` в теле функции с «локального» на «глобальный». Это привело к неожиданному поведению: глобальная переменная `name` «почему-то» взяла и поменяла значение с «Иван» на «Петр». Вы знаете почему, а остальные разработчики в команде могут часами искать ошибку в коде.

Необходимость использовать `global`, как правило, означает серьезный просчет в архитектуре приложения, либо это сигнал перейти на ООП. Не используйте `global` при выполнении домашних заданий.

Есть ещё один оператор, связанный с областями видимости, — `nonlocal`. Он весьма полезен и будет рассмотрен на одном из следующих уроков.

Вам нужны ассоциативные массивы — в Python есть словари

***Замечание:** если вы работали в другом стеке, то наверняка слышали слова «отображения», `map`, «именованные массивы». Всё это синонимы словаря `dict` в `Python`.

Предположим, что вы разрабатываете модуль для работы с учётными записями пользователей. Данные пока получаете в виде списков:

```
user_1 = ['Иванов', 'Иван', 'Иванович', 25]
user_2 = ['Петров', 'Петр', 'Петрович', 28]
```

Как, например, мы можем реализовать функцию вывода информации о пользователе:

```
...
def user_info(user):
   print(f'{user[0]} {user[1]}, {user[3]} лет')


user_info(user_1)  # Иванов Иван, 25 лет
user_info(user_2)  # Петров Петр, 28 лет
```

Удобно ли работать через индексы в виде чисел? Нет. А если в будущем появятся новые поля: адрес, профессия и т.д.? Станет ещё сложнее, появятся ошибки. В подобных ситуациях правильнее использовать словарь (dict):

```
user_3 = {
   'first_name': 'Иван',
   'last_name': 'Иванов',
   'patronymic': 'Иванович',
   'age': 25
}
```

Словарь — это набор пар “<ключ>: <значение>”. Какие ключи вы здесь видите? Верно: “first_name”, “last_name”, “patronymic”, “age”. Значения ключей пишем через символ двоеточия. Снаружи словарь оборачиваем в фигурные скобки. Получать значения из словаря можно точно так же, как и из списка — ключ в квадратных скобках:

```
...
print(user_3['first_name'])  # Иванов
print(user_3['age'])  # 25
```

Хранить данные о пользователях таким способом эффективнее: при появлении новых полей просто будем добавлять пары “<ключ>: <значение>” в словарь. Нам теперь не важен порядок элементов:

```
def user_info_adv(user):
   print(f"{user['last_name']} {user['first_name']}, {user['age']} лет")


user_info_adv(user_3)  # Иванов Иван, 25 лет
```

С первого взгляда понятно, как работает эта функция. Внешние кавычки в строке заменили на двойные, иначе все сломается: **всегда необходимо чередовать кавычки**.

В качестве эксперимента сымитируем список при помощи словаря:

```
user_4 = {
   0: 'Иванов',
   1: 'Иван',
   2: 'Иванович',
   3: 25
}
print(user_4[0])  # Иванов
```

Получается, внешне списки и словари чем-то похожи: синтаксис получения элемента по индексу одинаков — через квадратные скобки. Но есть и разница: в списках ключ (индекс) формируется автоматически и мы не можем его задать вручную. В словарях же как раз необходимо при добавлении элемента явно указывать значение ключа, под которым будет храниться элемент.

Очень важно понимать, что **ключом словаря** может быть **только неизменяемый** (immutable) объект!

*Ещё один момент: словарь — по сути хеш-таблица. Это делает его незаменимым в тех алгоритмах, где нужно хранить данные со сложностью поиска или вставки `O(1)`.

Словари: `.get()` и `.setdefault()`

При разработке программ очень часто бывают ситуации, когда мы пытаемся получить из словаря значение по ключу, которого там нет:

```
user_5 = {'first_name': 'Иван', 'last_name': 'Иванов', 'age': 25}
print(user_5['address'])  # ...KeyError: 'address'
```

Получили ошибку ключа. Попробуем теперь воспользоваться классическим геттером `.get()`:

```
...
print(user_5.get('address'))  # None
print(user_5.get('address', 'адрес не задан'))  # адрес не задан
```

Очевидно ли вам, как работает этот метод? Если ключ в словаре не найден, он возвращает значение None. Ошибки 100% не будет. Можно даже задать значение по умолчанию: в нашем примере — `'адрес не задан'`. Главная особенность метода `.get()` — он не меняет ничего в самом словаре, чего не скажешь о методе `.setdefault()` (кстати, в названии уже есть намёк — слова `set` и `default`):

```
user_6 = {'first_name': 'Сергей', 'last_name': 'Сергеев', 'age': 25}
print(user_6.setdefault('address', 'Россия'))  # Россия
print(user_6)
# {'first_name': 'Сергей', 'last_name': 'Сергеев', 'age': 25, 'address':
'Россия'}
```

Если ключа в словаре нет, этот метод его создаёт и присваивает значение, которое передали вторым аргументом. То есть работает, по сути, аналогично методу `.get()`, но при этом прописывает новое значение в словаре, если его не было. **Очень важно** понять эту разницу! А если ключ в словаре был — тогда мы просто получим его значение, как если бы использовали те же самые квадратные скобки. Никаких изменений в самом словаре не произойдёт:

```
user_7 = {'first_name': 'Александр', 'last_name': 'Александров', 'age': 30}
print(user_7.setdefault('age', 18))  # 30
print(user_7)
# {'first_name': 'Александр', 'last_name': 'Александров', 'age': 30}
```

Метод `.setdefault()` достаточно часто используется в разработке на Python.

Словари: `.update()` и `.popitem()`

Как добавлять данные в словарь? Можно через синтаксис квадратных скобок:

```
user_7 = {'first_name': 'Дмитрий'}
user_7['last_name'] = 'Петров'
```

```
print(user_7)  # {'first_name': 'Дмитрий', 'last_name': 'Петров'}
```

Мы делали что-то похожее со списками. Разница лишь в том, что в списке мы можем заменить значение только существующей ячейки, а в словаре можем создать новый ключ. Если необходимо добавить несколько значений из другого словаря, эффективнее использовать метод `.update()`:

```
address = {'street': 'Невский проспект', 'house': 15}
user_8 = {'first_name': 'Роман', 'last_name': 'Григорьев'}
user_8.update(address)
print(user_8)
# {'first_name': 'Роман', 'last_name': 'Григорьев', 'street': 'Невский
проспект', 'house': 15}
```

По смыслу этот метод очень похож на метод `.extend()` для списков.

А помните, мы работали с методом списков `.pop()`? Для словарей есть аналог - `.popitem()`:

```
user_9 = {'first_name': 'Роман', 'last_name': 'Григорьев', 'street': 'Ленина'}
some_pair = user_9.popitem()
print(some_pair)  # ('street', 'Ленина')
print(user_9)  # {'first_name': 'Роман', 'last_name': 'Григорьев'}
```

В нашем случае метод вырезал последнюю пару "`<ключ>: <значение>`". Только в Python 3.7 реализация метода даёт гарантию, что будет вырезана именно последняя (LIFO) пара. В предыдущих версиях это могла быть произвольная пара. Реально этот метод редко используется в коде.

Метод `.pop()` для словарей вырезает значение по ключу:

```
user_10 = {'first_name': 'Егор', 'last_name': 'Родионов'}
last_name = user_10.pop('last_name')
print(last_name)  # Родионов
print(user_10)  #{'first_name': 'Егор'}
```

Подумайте: что будет, если ключа в словаре нет? Верно — ошибка. Можно ли её избежать? Да — задать значение по умолчанию. Если понимать общие принципы, такое поведение становится очевидным.

Словари: цикл `for in`

Рано или поздно вам понадобится обойти элементы словаря. Логично и правильно использовать для этого цикл `for in`:

```
dns = {
   'mail.ru': '94.100.180.201',
   'geekbrains.ru': '178.248.232.209',
   'amazon.com': '205.251.242.103'
}
for item in dns:
   print(item)
# mail.ru
# geekbrains.ru
# amazon.com
```

По умолчанию Python «видит» ключи словарей. Ещё один пример:

```
print('mail.ru' in dns)  # True
print('google.com' in dns)  # False
```

Здесь мы использовали оператор in, проверяющий факт вхождения элемента в какую-либо последовательность. В нашем случае узнали, есть ли конкретная строка среди ключей словаря.

***Очень важно**, что сложность этой операции для словарей `O(1)`, а для списка или кортежа она была бы `O(n)`. Это один из самых часто встречающихся вопросов на собеседованиях.

Чаще всего мы в цикле итерируем не ключи, а пары "`<ключ>: <значение>`". Для доступа к ним используется метод `.items()`:

```
for key, val in dns.items():
   print(f'{key}={val}')
# mail.ru=94.100.180.201
# geekbrains.ru=178.248.232.209
# amazon.com=205.251.242.103
```

В чём особенность данного цикла? Используется две переменных: `key` и `val`. На самом деле это уже знакомое нам параллельное присваивание:

```
for item in dns.items():
   print(item, end=' -> ')
   key, val = item
   print(f'{key}={val}')
# ('mail.ru', '94.100.180.201') -> mail.ru=94.100.180.201
# ('geekbrains.ru', '178.248.232.209') -> geekbrains.ru=178.248.232.209
# ('amazon.com', '205.251.242.103') -> amazon.com=205.251.242.103
```

Метод `.items()` возвращает последовательность кортежей вида `(<ключ>, <значение>)`. Можно работать и с этими кортежами, но гораздо удобнее присвоить элементы в отдельные переменные, как сделано в нашем примере.

Можно проитерировать только значения словаря:

```
for val in dns.values():
   print(val)
# 94.100.180.201
# 178.248.232.209
# 205.251.242.103
```

Также есть метод `.keys()`:

```
for key in dns.keys():
   print(key)
# mail.ru
# geekbrains.ru
# amazon.com
```

Он дает точно такой же результат, как и самый первый пример в данном разделе. Очень часто используют именно тот способ без вызова метода .keys().

Продвинутая работа с аргументами функций

Позиционные аргументы и `*args`

Если вы хотя бы раз заглядывали в исходники (`<Ctrl>` + клик мышкой по имени функции), то видели там `*args` и `**kwargs` в аргументах. Пришла пора открыть завесу таинственности над словом **«распаковка»** в `Python`:

```
def my_sum(args):
   return sum(args)


print(my_sum([1, 5, 89]))  # 95
print(my_sum(1, 5, 89))
# ...TypeError: my_sum() takes 1 positional argument but 3 were given
```

Если передаём аргументы как список (по факту один аргумент), всё хорошо. Если передаём аргументы через запятую (несколько аргументов) — ошибка. Перепишем функцию:

```
def my_sum_adv(*args):
    print(type(args))  # <class 'tuple'>
    return sum(args)


print(my_sum_adv(1, 5, 89))  # 95
```

Теперь она работает для любого количества аргументов. Раз мы написали `*` перед именем переменной в списке аргументов, Python теперь создает из переданных аргументов кортеж, который, как мы знаем, может содержать любое количество элементов. Если же аргументы при вызове функции уже будут в виде списка или кортежа, их необходимо предварительно распаковать:

```
...
print(my_sum_adv(*[1, 5, 89]))  # 95
```

Обязательно поработайте с распаковкой. Очень часто начинающие разработчики не уделяют внимания этой манипуляции, а она встречается в `Python` повсюду.

Примечание: синтаксис `*args` используется, когда функция должна работать с неизвестным количеством позиционных аргументов. Позиционными называют аргументы, роль которых зависит от их порядкового номера при вызове функции.

Необязательные аргументы

Предположим, у нас есть функция, вычисляющая площадь прямоугольника:

```
def box_area(w, h):
    return w * h


print(box_area(5, 10))  # 50
print(box_area(2, 3))  # 6
```

В какой-то момент в проекте появилась необходимость работать с разными единицами измерения и функцию доработали — добавили аргумент масштаба `scale`:

```
def box_area(w, h, scale):
    return w * h * scale ** 2
```

Всё несложно, НО если в проекте модуль с функцией уже используется в десятке скриптов, которые пишут другие программисты, всё сломается. Можно задать другое имя этой функции, например, `box_area_scaled()`, но это не очень хорошее решение: в будущем вы получите много-много функций на все случаи и возникнет путаница. В `Python` мы можем просто задать значение по умолчанию:

```
def box_area(w, h, scale=1.0):
    return w * h * scale ** 2
```

Теперь старый код работает как раньше, но можно передавать дополнительный аргумент:

```
print(box_area(5, 10))  # 50.0
print(box_area(5, 10, 0.01))  # 0.005
```

Нюанс: попробуйте в функции задать целое число `scale=1`. Что поменялось в IDE PyCharm? Теперь аргумент `0.01` подсвечивается — система «ждёт» целочисленных значений.

Если у вас уже начало формироваться мышление разработчика, вы должны задать себе вопрос: можно ли написать “`def box_area(w=1, h, scale=1.0)`”? Ответ: нет. Сначала идут обязательные аргументы (без значения по умолчанию), и только ПОСЛЕ них — необязательные.

Именованные аргументы и `**kwargs`

Идём дальше и пишем функцию для вывода на экран информации о прямоугольнике:

```
def box_show(w, h, unit='м', lang='ru'):
    if lang == 'ru':
        print(f'ширина {w} {unit}, высота {h} {unit}')
    else:
        print(f'width {w} {unit}, height {h} {unit}')


box_show(5, 10) # ширина 5 м, высота 10 м
box_show(5, 10, 'см')  # ширина 5 см, высота 10 см
```

В будущем при вызове этой функции в каком-нибудь модуле вы можете запутаться, что передавать первым аргументом — ширину или высоту. Или может понадобиться задать другой язык вывода, не изменяя единиц измерения:

```
box_show(10, 5)  # ширина 10 м, высота 5 м
box_show(5, 10, 'м', 'en')  # width 5 м, height 10 м
```

Второй пример явно нарушает принцип «простое лучше, чем сложное».

В Python можно передавать аргументы в виде пары "<ключ>=<значение>" — их называют именованными:

```
box_show(h=10, w=5)  # ширина 5 м, высота 10 м
box_show(5, 10, lang='en')  # width 5 м, height 10 м
```

Обратите внимание, что знак = пробелами в этой ситуации не обособляется. Попробуйте написать: "`box_show(h=10, 5)`" — работает? Нет. Тут правило: сначала позиционные аргументы и **только** после них именованные.

Если в будущем мы планируем добавлять другие именованные аргументы в функцию, дописываем в сигнатуру (список аргументов) `**kwargs`. Вот и ещё одна загадка раскрыта.

```
def box_show(w, h, unit='м', lang='ru', **kwargs):
    print(f'kwargs: {kwargs}')
    if lang == 'ru':
        print(f'ширина {w} {unit}, высота {h} {unit}')
    else:
        print(f'width {w} {unit}, height {h} {unit}')


box_show(5, 10, lang='en', color='red')
# kwargs: {'color': 'red'}
# width 5 м, height 10 м
```

В словарь `kwargs` попадают те аргументы, которые не прописаны явно. В данном случае — `color`.

Если вы посмотрите сигнатуры встроенных функций (`sum()`, `sorted()` и др), то заметите там сочетание `(*args, **kwargs)`. Кто готов поразмышлять, зачем так делают? В такую функцию можно передать любое количество позиционных и именованных аргументов.

```
def show_user(name, *args, **kwargs):
    print(f'Пользователь {name}')
    print(f'args: {args}')
    print(f'kwargs: {kwargs}')


show_user('Иван', 'Иванов', age=25)
# Пользователь Иван
# args: ('Иванов',)
# kwargs: {'age': 25}
```

```
user_full_name = ('Иван', 'Иванов')
user_add_info = {'height': 180, 'age': 25}

show_user(*user_full_name, **user_add_info)
# Пользователь Иван
# args: ('Иванов',)
# kwargs: {'height': 180, 'age': 25}
```

Изучите внимательно этот пример. Заметили, как мы распаковываем словарь? Используем `**` перед именем переменной. Получается логично:

- список, кортеж — одно измерение, одна звездочка;
- словарь — два измерения, две звездочки.

А можно ли вместо `args` и `kwargs` использовать другие имена? Просто возьмите и попробуйте. Должно получиться. Но между разработчиками есть негласная договоренность всегда их так именовать — так проще читать незнакомый код.

Модуль `random`

В алгоритмах часто необходимо выбрать случайное значение из последовательности или просто получить случайное число. Если задача не связана с криптографией, можно и нужно использовать модуль `random`.

Предположим, требуется выбрать случайный товар для подарка. Можем решить эту задачу классическим способом через индексы:

```
from random import randrange

products = ['яблоко', 'груша', 'банан', 'авокадо']
random_idx = randrange(len(products))
bonus = products[random_idx]
print(bonus)  # банан
```

В модуле `random` есть две функции, генерирующие случайное целое число: `randrange` и `randint`. Разницу легко запомнить: `randrange` ведёт себя подобно функции `range()` — граница справа не включается в интервал (ещё можно задать шаг). Функция `randint()` генерирует случайное целое число, включая правую границу.

Можно ли решение задачи сделать проще, чтобы выполнить принцип «простое лучше, чем сложное»? Да, можно.

```
from random import choice

products = ['яблоко', 'груша', 'банан', 'авокадо']
bonus = choice(products)
print(bonus)  # банан
```

Вот теперь всё хорошо написано. Между этими решениями — пропасть. Вы уже должны это чувствовать.

Также рекомендуем почитать про функции `choices`, `sample` и `shuffle`.

Очень полезные встроенные функции: `filter()`, `map()`, `zip()`

Фильтруем через `filter()`

Предположим, у нас есть список товаров и необходимо выбрать некоторые из них по заданному критерию — например, на букву `'i'`. Можно решить эту задачу через обычный цикл, но мы же кодим на Python, а в нём «простое лучше, чем сложное»:

```
def accepted(el):
    return el.lower().startswith('i')


products = ['iPad', 'Samsung Galaxy', 'iPhone', 'iRiver']
products_sample = filter(accepted, products)
print(type(products_sample))  # <class 'filter'>
print(*products_sample)  # iPad iPhone iRiver
```

Функция `filter()` принимает первым аргументом `callback`, который должен возвращать `True` или `False` для своего аргумента. Второй аргумент — любая последовательность (`iterable`). Результат функции — итератор (о них подробнее поговорим позже), возвращающий те элементы исходной последовательности, для которых `callback` вернул `True` (берём). С итератором можно работать как обычно в цикле `for in`, но при попытке вывести его через `print()` ничего интересного не увидим — так и должно быть. Если хотим получить очередное значение итератора «голыми руками», используем функцию `next()`. На самом деле это делают редко. Можно преобразовать итератор в список:

`list(products_sample)`. В нашем примере просто распаковали итератор при помощи `*`. Попробуйте дублировать последнюю строку примера — что получилось? Правильно: ничего.

!!!Важно: после вывода всех значений итератор, который вернула функция `filter()`, **истощён**, повторно использовать его не получится — только **создавать** заново!

Если отбросить подробности, то у нас есть классический фильтр по заданному условию. Как правило, он работает существенно быстрее цикла и экономит память.

В качестве `callback` можно использовать и методы классов:

```
nums = ['0', 'Samsung Galaxy', '15.5', '18,1', 'iPhone', '748', 'HelloWord']
int_nums = list(filter(str.isalpha, nums))
print(int_nums)  # ['iPhone', 'HelloWord']
```

Взяли элементы списка, состоящие только из букв (метод `str.isalpha`). Пробел — не буква, поэтому `'Samsung Galaxy'` нет на выходе.

*Высший пилотаж: `lambda`-функции

Вам бы хотелось записать пример с товарами ещё лаконичнее? Тогда придется познать lambda-функции, или, как ещё их называют, анонимные (безымянные) функции:

```
products = ['iPad', 'Samsung Galaxy', 'iPhone', 'iRiver']
products_sample = filter(lambda el: el.lower().startswith('i'), products)
print(*products_sample)  # iPad iPhone iRiver
```

Мы по сути записали функцию `accepted()` в одну строку. Для этого использовали ключевое слово `lambda`, убрали скобки вокруг аргумента и тривиальный `return`. Имя аргумента может быть любым — вместо `el` можно записать `x`, можно использовать несколько аргументов.

Подобный код **очень часто** используется в `Python` — нужно приложить усилия и разобраться в нём.

Преобразуем через `map()`

Решим теперь другую задачу: есть список с ценами на товары (тип данных — `str`), получить из него кортеж с ценами (тип данных — `float`). Классический подход — использовать цикл, но можно проще:

```
prices = ['1578.4', '892.4', '354.1', '871.5']
prices_float = tuple(map(float, prices))
```

```
print(type(prices[0]), type(prices_float[0]), prices_float)
# <class 'str'> <class 'float'> (1578.4, 892.4, 354.1, 871.5)
```

Сигнатура функции `map()` аналогична `filter()`: первый аргумент — `callback`, второй — `iterable`. Работает она следующим образом: берёт первый элемент из `iterable` и выполняет `callback` с ним как с аргументом, результат выдаёт нам. И так с каждым следующим. В итоге опять получаем итератор. В нашем примере мы преобразовали его в кортеж.

Если вкратце: `map()` применяет `callback` к каждому элементу `iterable` и возвращает результат в виде итератора. Как правило, работает быстрее цикла и экономит память. А ещё выглядит лаконично. Вывод: надо как можно чаще использовать её в коде.

А что, если надо было ещё применить скидку к товарам? Используем `lambda`-функцию:

```
DISCOUNT = 7
prices = ['1578.4', '892.4', '354.1', '871.5']
prices_discount = map(
   lambda x: round(float(x) * (100 - DISCOUNT) / 100, 2),
   prices
)
print(*prices_discount)
# 1467.91 829.93 329.31 810.5
```

Здесь не стали преобразовывать в кортеж и округлили цены до второго знака.

А если данные с ошибками? Используем `filter()` и `map()` вместе:

```
prices = ['15,78.4', '892.4', '354.1 rub', '871.5', '47,1']
prices_float = map(float,
                   filter(lambda x: x.replace('.', '', 1).isdigit(), prices))
print(*prices_float)
# 892.4 871.5
```

Для метода `.replace()` задали количество замен 1, чтобы из настоящих вещественных получились только цифры — их фильтр пропустит.

Склеиваем через `zip()`

Пусть есть три списка: имена пользователей, их логины и роли. Нужно вывести информацию о каждом пользователе. Можно использовать цикл со счётчиком:

```
user_names = ['Иван', 'Петр', 'Ольга', 'Сергей']
user_logins = ['ivan', 'petr', 'olga', 'sergey']
user_roles = ['user', 'staff', 'admin', 'user']
for i in range(len(user_names)):
   print(f'{user_names[i]}, {user_logins[i]}, {user_roles[i]}')
# Иван, ivan, user
# Петр, petr, staff
# Ольга, olga, admin
# Сергей, sergey, user
```

Так в Python делать не надо. Вообще старайтесь как можно реже работать через индексы. Перепишем через `zip()`:

```
for name, login, role in zip(user_names, user_logins, user_roles):
   print(f'{name}, {login}, {role}')
```

Какой код легче читать? Какой код будет быстрее работать? Ответы очевидны. Вам наверняка хочется задать вопрос: а если списки разной длины? Вариант со счетчиком 100% сломается, а вариант с `zip()` просто выведет столько строк, сколько элементов в самом коротком списке.

`zip()` берёт сначала все первые элементы, потом — все вторые и так далее. Возвращает он нам, разумеется, итератор. Поэтому просто через `print()` посмотреть результат не получится: либо преобразуем в требуемый тип (список, кортеж, словарь и т.д.), либо распаковываем через `*`.

Если вы научитесь использовать `zip()` в коде, это будет очень круто.

Практическое задание

1. Написать функцию `num_translate()`, переводящую числительные от 0 до 10 с английского на русский язык. Например:

```
>>> num_translate("one")
"один"
>>> num_translate("eight")
"восемь"
```

   Если перевод сделать невозможно, вернуть None. Подумайте, как и где лучше хранить информацию, необходимую для перевода: какой тип данных выбрать, в теле функции или снаружи.

2. *(вместо задачи 1) Доработать предыдущую функцию в `num_translate_adv()`: реализовать корректную работу с числительными, начинающимися с заглавной буквы — результат тоже должен быть с заглавной. Например:

```
>>> num_translate_adv("One")
"Один"
>>> num_translate_adv("two")
"два"
```

3. Написать функцию `thesaurus()`, принимающую в качестве аргументов имена сотрудников и возвращающую словарь, в котором ключи — первые буквы имён, а значения — списки, содержащие имена, начинающиеся с соответствующей буквы. Например:

```
>>> thesaurus("Иван", "Мария", "Петр", "Илья")
{
    "И": ["Иван", "Илья"],
    "М": ["Мария"], "П": ["Петр"]
}
```

Подумайте: полезен ли будет вам оператор распаковки? Как поступить, если потребуется сортировка по ключам? Можно ли использовать словарь в этом случае?

4. *(вместо задачи 3) Написать функцию `thesaurus_adv()`, принимающую в качестве аргументов строки в формате «Имя Фамилия» и возвращающую словарь, в котором ключи — первые буквы фамилий, а значения — словари, реализованные по схеме предыдущего задания и содержащие записи, в которых фамилия начинается с соответствующей буквы. Например:

```
>>> thesaurus_adv("Иван Сергеев", "Инна Серова", "Петр
Алексеев", "Илья Иванов", "Анна Савельева")
{
    "А": {
        "П": ["Петр Алексеев"]
    },
    "С": {
        "И": ["Иван Сергеев", "Инна Серова"],
        "А": ["Анна Савельева"]
    }
}
```

Как поступить, если потребуется сортировка по ключам?

5. Реализовать функцию `get_jokes()`, возвращающую n шуток, сформированных из трех случайных слов, взятых из трёх списков (по одному из каждого):

```
nouns = ["автомобиль", "лес", "огонь", "город", "дом"]
adverbs = ["сегодня", "вчера", "завтра", "позавчера", "ночью"]
adjectives = ["веселый", "яркий", "зеленый", "утопичный",
"мягкий"]
```

Например:

```
>>> get_jokes(2)
["лес завтра зеленый", "город вчера веселый"]
```

Документировать код функции.

Сможете ли вы добавить еще один аргумент — флаг, разрешающий или запрещающий повторы слов в шутках (когда каждое слово можно использовать только в одной шутке)? Сможете ли вы сделать аргументы именованными?

Задачи со * предназначены для продвинутых учеников, которым мало сделать обычное задание.

Дополнительные материалы

1. Лутц Марк. Изучаем Python.
2. Встроенные функции Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://realpython.com/python-lambda/.
2. https://docs.python.org/3/.