

Python Tips, Tricks, and Hacks

Перевод статьи "[Python Tips, Tricks, and Hacks](#)". Будет полезна на начальном и среднем этапах изучения Python.

Хотите писать более лаконичный и читаемый код? Вы хотите уместить как можно больше смысла в одно выражение? Считаете, что прочитать о нескольких уловках лучше, чем провести остаток жизни за чтением документации? Вы обратились по адресу. Мы начнем с маленьких уловок, которые вы уже могли встретить, если немного работали с Python. Но я обещаю, что ближе к концу статьи вас ожидает больше безумных вещей. Я старался сделать, чтобы все фрагменты кода запускались без дополнительных изменений. Если хотите, можете скопировать их в оболочку Python и посмотреть, что получится. Обратите внимание, что многие примеры содержат «неправильные» фрагменты, которые закомментированы. Ничто вам не мешает раскомментировать строку и посмотреть, что произойдет.

Небольшое разграничение между true и True в этой статье: когда я говорю, что объект true, это значит, что будучи приведенным к типу boolean, он становится True. Аналогично с false и False.

1 Маленькие уловки

1.1 Четыре типа кавычек

Начнем с того, что вы, возможно, уже знаете. В некоторых языках программирования одинарные и двойные кавычки предназначены для разных вещей. Python позволяет использовать оба варианта (но строка должна начинаться и заканчиваться одним и тем же типом кавычек). В Python также есть еще два типа кавычек: ''' (тройные одинарные) и """ (тройные двойные). Таким образом, можно использовать несколько уровней кавычек, прежде чем придется заботиться об их экранировании. Например, этот код правильный:

```
print """Я бы не хотел никогда услышать, как он говорит: '''Она сказала: "Он сказал: 'Дай мне двести рублей'""""
```

1.2 Правдивость различных объектов

В отличие от некоторых языков программирования, в Python объект считается false, только если он пуст. Это значит, что не нужно проверять длину строки, кортежа или словаря — достаточно проверить его как логическое выражение.

Легко предсказать, что 0 — тоже false, а остальные числа — true.

Например, следующие выражения эквивалентны. В данном случае my_object — строка, но здесь мог оказаться другой тип (с соответствующими изменениями условий блока if).

```
my_object = 'Test' # True example
# my_object = '' # False example

if len(my_object) > 0:
    print 'my_object не пуст'

if len(my_object): # 0 преобразовывается к False
    print 'my_object не пуст'

if my_object != '':
    print 'my_object не пуст'

if my_object: # пустая строка преобразовывается к False
    print 'my_object не пуст'
```

Итак, нет необходимости проверять длину объекта, если вас интересует только, пуст он или нет.

1.3 Проверка на вхождение подстроки

Это маленькая, довольно очевидная подсказка, но я узнал о ней лишь через год изучения Python. Должно быть, вы знаете, что можно проверить, содержится ли нужный элемент в кортеже, списке, словаре, с помощью конструкции 'item in list' или 'item not in list'. Я не мог представить, что это сработает для строк. Я всегда писал что-то вроде этого:

```
string = 'Hi there' # True example
# string = 'Good bye' # False example
if string.find('Hi') != -1:
    print 'Success!'
```

Этот код довольно неуклюжий. Совершенно так же работает 'if substring in string':

```
string = 'Hi there' # True example
# string = 'Good bye' # False example
if 'Hi' in string:
    print 'Success!'
```

Проще и понятней. Может быть, очевидно для 99% людей, но мне хотелось бы узнать об этом раньше, чем я узнал.

1.4 Красивый вывод списка

Обычный формат вывода списка с помощью print не очень удобен. Конечно, становится понятно, что из себя представляет список, но чаще всего пользователь не хочет видеть кавычки вокруг каждого элемента. Есть простое решение, использующее метод join строки:

```
recent_presidents = ['Борис Ельцин', 'Владимир Путин', 'Дмитрий Медведев']
print 'Последними президентами были %s.' % ', '.join(recent_presidents)
# печатает "Последними президентами были Борис Ельцин, Владимир
Путин, Дмитрий Медведев."
```

Метод join преобразовывает список в строку, рассматривая каждый элемент как строку. Разделителем является та строка, для которой был вызван join. Он достаточно умен, чтобы не вставлять разделитель после последнего элемента.

Дополнительный бонус: join работает линейное время. Никогда не создавайте строку складыванием элементов списка в цикле for: это не просто некрасиво, это занимает *квадратичное* время!

1.5 Целочисленное деление и деление с плавающей точкой

Если вы делите целое число на целое, по умолчанию результат обрезается до целого. Например, 5/2 вернет 2.

Есть два способа это исправить. Первый и самый простой способ заключается в том, чтобы преобразовать одно из чисел к типу float. Для констант достаточно добавить «.0» к одному из чисел: 5.0/2 вернет 2.5. Также вы можете использовать конструкцию float(5)/2.

Второй способ дает более чистый код, но вы должны убедиться, что ваша программа не сломается от этого существенного изменения. После вызова 'from __future__ import division' Python всегда будет возвращать в качестве результата деления float. Если вам понадобится целочисленное деление, используйте оператор //: 5//2 всегда возвращает 2.

```
5/2          # Возвращает 2
5.0/2       # Возвращает 2.5
float(5)/2  # Возвращает 2.5
5//2        # Возвращает 2

from __future__ import division
5/2          # Возвращает 2.5
5.0/2       # Возвращает 2.5
float(5)/2  # Возвращает 2.5
5//2        # Возвращает 2
```

В одной из следующих версий Python такое поведение станет дефолтным. Если вы хотите, чтобы ваш код оставался совместимым, используйте оператор `//` для целочисленного деления, даже если вы не используете этот импорт.

1.6 Лямбда-функции

Иногда нужно передать функцию в качестве аргумента или сделать короткую, но сложную операцию несколько раз. Можно определить функцию обычным способом, а можно использовать лямбда-функцию — маленькую функцию, возвращающую результат одного выражения. Следующие два определения полностью идентичны:

```
def add(a,b): return a+b  
  
add2 = lambda a,b: a+b
```

Преимущество лямбда-функции в том, что она является выражением и может быть использована внутри другого выражения. Ниже приведен пример, использующий функцию `map`, которая вызывает функцию для каждого элемента списка и возвращает список результатов. (В следующем пункте я покажу, что `map` практически бесполезен. Но он дает нам возможность привести хороший пример в одну строку.)

```
squares = map(lambda a: a*a, [1,2,3,4,5])  
# теперь squares = [1,4,9,16,25]
```

Без лямбда-функций нам пришлось бы определить функцию отдельно. Мы просто сэкономили одну строку кода и одно имя переменной.

Синтаксис лямбда-функции: *lambda* переменные: выражение
переменные — список аргументов, разделенных запятой. Нельзя использовать ключевые слова. Аргументы не надо заключать в скобки.
выражение — инлайновое выражение Python. Область видимости включает локальные переменные и аргументы. Функция возвращает результат этого выражения.

2 Списки

2.1 Генераторы списков

Если вы использовали Python достаточно долго, вы должны были хотя бы слышать о понятии «list comprehensions». Это способ уместить цикл `for`, блок `if` и присваивание в одну строку. Другими словами, вы можете отображать (`map`) и фильтровать списки одним выражением.

2.1.1 Отображение списка

Начнем с простейшего примера. Допустим, нам надо возвести в квадрат все элементы списка. Свежеиспеченный программист на Python может написать код вроде этого:

```
numbers = [1,2,3,4,5]  
squares = []  
for number in numbers:  
    squares.append(number*number)  
# теперь squares = [1,4,9,16,25]
```

Мы «отобразили» один список на другой. Это также можно сделать с помощью функции `map`:

```
numbers = [1,2,3,4,5]  
squares = map(lambda x: x*x, numbers)  
# теперь squares = [1,4,9,16,25]
```

Этот код определенно короче (одна строка вместо трех), но всё еще некрасив. С первого взгляда сложно сказать, что делает функция `map` (она принимает в качестве аргументов функцию и список и применяет функцию к каждому элементу списка). К тому же мы вынуждены определять функцию, это выглядит довольно беспорядочно. Если бы только существовал более красивый путь... например, генератор списка:

```
numbers = [1,2,3,4,5]
squares = [number*number for number in numbers]
# и снова squares = [1,4,9,16,25]
```

Этот код делает абсолютно то же самое, но он короче, чем первый пример, и понятней, чем второй. Человек без проблем определит, что делает код, для этого даже не обязательно знать Python.

2.1.2 Фильтрация списка

А что, если нас интересует фильтрация списка? Например, требуется удалить элементы, большие или равные 4. (Да, примеры не очень реалистичны. Как бы то ни было...)

Новичок напишет так:

```
numbers = [1,2,3,4,5]
numbers_under_4 = []
for number in numbers:
    if number < 4:
        numbers_under_4.append(number)
# numbers_under_4 = [1,4,9]
```

Очень просто, не так ли? Но код занимает 4 строки, содержит два уровня отступов и при этом делает тривиальную вещь. Можно уменьшить размер кода с помощью функции `filter`:

```
numbers = [1,2,3,4,5]
numbers_under_4 = filter(lambda x: x < 4, numbers)
# numbers_under_4 = [1,2,3]
```

Аналогично функции `map`, о которой мы говорили выше, `filter` сокращает код, но выглядит довольно уродливо. Что, черт возьми, происходит? Как и `map`, `filter` получает функцию и список. Если функция от элемента возвращает `true`, элемент включается в результирующий список. Разумеется, мы можем сделать это через генератор списка:

```
numbers = [1,2,3,4,5]
numbers_under_4 = [number for number in numbers if number < 4]
# numbers_under_4 = [1,2,3]
```

Снова мы получили более короткий, ясный и понятный код.

2.1.3 Одновременное использование `map` и `filter`

Теперь мы можем использовать всю силу генератора списков. Если я вас еще не убедил, что `map` и `filter` тратят слишком много вашего времени, надеюсь, теперь вы со мной согласитесь.

Пусть требуется отобразить и отфильтровать список одновременно. Другими словами, я хочу увидеть квадраты элементов списка, меньших 4. Еще раз, неопит напишет так:

```
numbers = [1,2,3,4,5]
squares = []
for number in numbers:
    if number < 4:
        squares.append(number*number)
# squares = [1,4,9]
```

Увы, код начал растягиваться вправо. Может, получится упростить его? Попробуем использовать `map` и `filter`, но у меня плохое предчувствие...

```
numbers = [1,2,3,4,5]
squares = map(lambda x: x*x, filter(lambda x: x < 4, numbers))
# squares is = [1,4,9]
```

Раньше `map` и `filter` было трудно читать, теперь — невозможно. Очевидно, это не лучшая идея. И снова генератор списков спасает ситуацию:

```
numbers = [1,2,3,4,5]
squares = [number*number for number in numbers if number < 4]
# square = [1,4,9]
```

Получилось немного длиннее, чем предыдущие примеры с генератором списков, но, по моему мнению, вполне читабельно. Определенно лучше, чем цикл for или использование map и filter.

Как вы видите, генератор списков сначала фильтрует, а затем отображает. Если вам обязательно нужно наоборот, получится сложнее. Придется использовать либо вложенные генерации, либо map и filter, либо обычный цикл for, в зависимости от того, что проще. Но это уже выходит за рамки статьи.

Синтаксис генератора списков: [element for variable(s) in list if condition]

list — любой итерируемый элемент

variable(s) — переменная или переменные, которые приравниваются к текущему элементу списка, аналогично циклу for

condition — инлайновое выражение: если оно равно true, элемент добавляется в результат

element — инлайновое выражение, результат которого используется как элемент списка-результата

2.1.4 Выражения-генераторы

Существует обратная сторона генератора списков: весь список должен находиться в памяти. Это не проблема для маленьких списков, как в предыдущих примерах, и даже на несколько порядков больше. Но в конце концов это становится неэффективным.

Выражения-генераторы ([Generator Expressions](#)) появились в Python 2.4. Из всех фишек Python им уделяется, наверно, меньше всего внимания. Отличие их от генераторов списков состоит в том, что они *не загружают* в память список целиком, а создают 'generator object', и в каждый момент загружен только один элемент списка.

Конечно, если вы хотите использовать список для чего-нибудь, это не особо поможет. Но если вы просто передаете его куда-нибудь, где нужен любой итерируемый объект (цикл for, например), стоит использовать функцию генератора.

Выражения-генераторы имеют такой же синтаксис, как генераторы списков, но вместо квадратных скобок используются круглые:

```
numbers = (1,2,3,4,5) # мы стремимся к эффективной работе, поэтому
используем кортеж вместо списка ;)
squares_under_10 = (number*number for number in numbers if number*number <
10)
# squares_under_10 - generator object, из которого можно получить
следующее значение, вызвав метод .next()

for square in squares_under_10:
    print square,
#ВЫВОДИТ '1 4 9'
```

Это более эффективно, чем использование генератора списков.

Итак, имеет смысл использовать выражения-генераторы для списков больших размеров. Если весь список целиком нужен вам для какой-то другой цели, то можно использовать любой из вариантов. Но использовать выражения-генераторы — хорошая привычка, если нет аргументов против этого. Правда, не надейтесь увидеть ускорение работы, если список небольшой.

В качестве финального штриха хочу заметить, что выражения-генераторы достаточно заключить в одни круглые скобки. Например, в случае, если вы вызываете функцию с одним аргументом, можно писать так: some_function(item for item in list).

2.1.5 Заключение

Мне не хочется этого говорить, но мы только прикоснулись к тому, что можно делать с помощью выражений-генераторов и генераторов списков. Здесь можно использовать всю силу for и if, а также оперировать с чем угодно, лишь бы оно было итерируемым объектом.

2 Списки

2.2 Свёртка списка

К сожалению, нельзя написать программу только с помощью генераторов списков. (Я шучу... конечно, можно.) Они могут отображать и фильтровать, но нет простого способа для свёртки списка. Под этим понятием я подразумеваю применение функции к первым двум элементам списка, а затем к получившемуся результату и следующему элементу, и так до конца списка. Можно реализовать это через цикл `for`:

```
1. numbers = [1,2,3,4,5]
2. result = 1
3. for number in numbers:
4.     result *= number
5. # result = 120
```

А можно воспользоваться встроенной функцией `reduce`, принимающей в качестве аргументов функцию от двух параметров и список:

```
1. numbers = [1,2,3,4,5]
2. result = reduce(lambda a,b: a*b, numbers)
3. # result = 120
```

Не так красиво, как генераторы списков, но короче обычного цикла. Стоит запомнить этот способ.

2.3 Прохождение по списку: `range`, `xrange` и `enumerate`

Помните, как в языке C для цикла `for` вы использовали переменную-счетчик вместо элементов списка? Возможно, вы уже знаете, как имитировать это поведение в Python с помощью `range` и `xrange`. Передавая число `value` функции `range`, мы получим список, содержащий элементы от 0 до `value-1` включительно. Другими словами, `range` возвращает индексы списка указанной длины. `xrange` действует похоже, но более эффективно, не загружая весь список в память целиком.

```
1. strings = ['a', 'b', 'c', 'd', 'e']
2. for index in xrange(len(strings)):
3.     print index,
4. # печатает "0 1 2 3 4"
```

Проблема в том, что обычно вам всё равно нужны элементы списка. Что толку от индексов без них? В Python есть потрясающая встроенная функция `enumerate`, которая возвращает итератор для пар индекс → значение:

```
1. strings = ['a', 'b', 'c', 'd', 'e']
2. for index, string in enumerate(strings):
3.     print index, string,
4. # печатает "0 a 1 b 2 c 3 d 4 e"
```

Еще один плюс состоит в том, что `enumerate` выглядит более читаемо, чем `xrange(len())`. Поэтому `range` и `xrange` полезны, наверно, только для создания списка с нуля, а не на основе других данных.

2.4 Проверка всех элементов списка на выполнение условия

Допустим, нам надо проверить, выполняется ли условие хотя бы для одного элемента. До Python 2.5 можно было писать так:

```
1. numbers = [1,10,100,1000,10000]
2. if [number for number in numbers if number < 10]:
3.     print 'At least one element is over 10'
4. # Результат: "At least one element is over 10"
```

Если ни один из элементов не удовлетворяет условию, генератор создаст пустой список, который считается `false`. В противном случае будет создан непустой список, который приводится к `true`. Строго говоря, не нужно проверять все элементы, достаточно остановиться после первого элемента, удовлетворяющего условию. Поэтому предыдущий пример менее эффективен и может быть выбран только в том случае, если вы не можете использовать Python 2.5, но хотите уместить всю логику в одно выражение.

С помощью встроенной функции `any`, введенной в Python 2.5, вы можете решить эту задачу более красиво и эффективно. Функция `any` прервется и вернет `True`, как только найдет элемент, удовлетворяющий условию. Ниже я использую выражение-генератор, которое возвращает `True` или `False` для каждого элемента, и передаю его функции `any`. Генератор вычисляет только необходимые в данный момент значения, а `any` принимает их по очереди.

```
1. numbers = [1,10,100,1000,10000]
2. if any(number < 10 for number in numbers):
3.     print 'Success'
4. # Результат: "Success!"
```

Аналогично, может возникнуть задача проверки, что *все* элементы удовлетворяют условию. Без Python 2.5 придется писать так:

```
1. numbers = [1,2,3,4,5,6,7,8,9]
2. if len(numbers) == len([number for number in numbers if number < 10]):
3.     print 'Success!'
4. # Результат: "Success!"
```

Здесь мы фильтруем список и проверяем, уменьшилась ли его длина. Если нет, то все его элементы удовлетворяют условию. Опять же, без Python 2.5 это единственный способ уместить всю логику в одно выражение.

В Python 2.5 есть более простой путь — встроенная функция `all`. Легко догадаться, что она прекращает проверку после первого элемента, не удовлетворяющего условию. Эта функция работает абсолютно аналогично предыдущей.

```
1. numbers = [1,2,3,4,5,6,7,8,9]
2. if all(number < 10 for number in numbers):
3.     print 'Success!'
4. # Результат: "Success!"
```

2.5 Группировка элементов нескольких списков

Встроенная функция `zip` используется для сжимания нескольких списков в один. Она возвращает массив кортежей, причем *n*-й кортеж содержит *n*-е элементы всех массивов, переданных в качестве аргументов. Это тот случай, когда пример — лучшее объяснение:

```
1. letters = ['a', 'b', 'c']
2. numbers = [1, 2, 3]
3. squares = [1, 4, 9]
4.
5. zipped_list = zip(letters, numbers, squares)
6. # zipped_list = [('a', 1, 1), ('b', 2, 4), ('c', 3, 9)]
```

Эта вещь часто используется как итератор для цикла `for`, извлекающая три значения за одну итерацию ("for letter, number, squares in zipped_list").

2.6 Еще несколько операторов для работы со списками

Ниже перечислены встроенные функции, в качестве аргумента принимающие любой итерируемый объект. `max` и `min` возвращают наибольший и наименьший элемент соответственно. `sum` возвращает сумму всех элементов списка. Опциональный второй параметр задает начальную сумму (по умолчанию 0).

2.7 Продвинутое логические операции с типом `set`.

Я понимаю, что в разделе, посвященном спискам, не положено говорить о множествах (`sets`). Но пока я не использовал их, мне не хватало некоторых логических операций в списках. `Set` отличается от списка тем, что его элементы уникальны и неупорядочены. Над множествами также можно выполнять множество логических операций.

Довольно распространенная задача — убедиться, что элементы списка уникальны. Это просто, достаточно преобразовать его в `set` и проверить, изменилась ли длина:

```
1. numbers = [1,2,3,3,4,1]
2. set(numbers)
3. # возвращает set([1,2,3,4])
4.
5. if len(numbers) == len(set(numbers)):
6.     print 'List is unique!'
7. # не выводит ничего
```

Конечно, вы можете преобразовать сет обратно в список, но не забывайте, что порядок элементов мог не сохраниться. Больше информации об операциях с сетями вы найдете в [документации](#).

3 Словари

3.1 Создание словаря с помощью именованных аргументов

Когда я начал изучать Python, я полностью пропустил альтернативный способ создания словаря. Если передать конструктору dict именованные аргументы, они будут добавлены в возвращаемый словарь. Конечно, на его ключи накладываются те же ограничения, что и на имена переменных. Вот пример:

```
1. dict(a=1, b=2, c=3)
2. # возвращает {'a': 1, 'b': 2, 'c': 3}
```

Этот способ немного очищает код, избавляя вас от летающих вокруг кавычек. Я часто использую его.

3.2 Преобразование словаря в список

Чтобы получить список ключей, достаточно привести словарь к типу list. Но лучше использовать .keys() для получения списка ключей или .iterkeys() для получения итератора. Если вам нужны значения, используйте .values() и .itervalues(). Но помните, что словари не упорядочены, поэтому полученные значения могут быть перемешаны любым мыслимым образом.

Чтобы получить и ключи, и значения в виде списка кортежей, можно использовать .items() или .iteritems(). Возможно, вы часто пользовались этим захватывающим методом:

```
1. dictionary = {'a': 1, 'b': 2, 'c': 3}
2. dict_as_list = dictionary.items()
3. #dict_as_list = [('a', 1), ('b', 2), ('c', 3)]
```

3.3 Преобразование списка в словарь

Обратная операция — превращение списка, содержащего пары ключ-значение, в словарь — делается так же просто:

```
1. dict_as_list = [['a', 1], ['b', 2], ['c', 3]]
2. dictionary = dict(dict_as_list)
3. # dictionary = {'a': 1, 'b': 2, 'c': 3}
```

Вы можете комбинировать методы, добавив именованные аргументы:

```
1. dict_as_list = [['a', 1], ['b', 2], ['c', 3]]
2. dictionary = dict(dict_as_list, d=4, e=5)
3. # dictionary = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

Превращать списка и словари друг в друга довольно удобно. Но следующий совет просто потрясающий.

3.3 "Dictionary Comprehensions"

Хотя в Python нет встроенного генератора словарей, можно сделать нечто похожее ценой небольшого беспорядка в коде. Используем .iteritems() для превращения словаря в список, передадим его выражению-генератору или генератору списков, а затем преобразуем список обратно в словарь.

Допустим, у нас есть словарь пар name:email, а мы хотим получить словарь пар name:is_email_at_a_dot_com (проверить каждый адрес на вхождение подстроки .com):

```
1. emails = {'Dick': 'bob@example.com', 'Jane': 'jane@example.com', 'Stou':  
    'stou@example.net'}  
2.  
3. email_at_dotcom = dict( [name, '.com' in email] for name, email in  
    emails.iteritems() )  
4.  
5. # email_at_dotcom = {'Dick': True, 'Jane': True, 'Stou': False}
```

Конечно, необязательно начинать и заканчивать словарем, можно в некоторых местах использовать и списки.

Это чуть менее читабельно, чем строгие генераторы списков, но я считаю, что это лучше, чем большой цикл for.

4. Выбор значений

4.1. Правильный путь

Начиная с версии 2.5, Python поддерживает синтаксис «value_if_true if test else value_if_false». Таким образом, вы можете выбрать одно из двух значений, не прибегая к странному синтаксису и подробным пояснениям:

```
test = True  
# test = False  
result = 'Test is True' if test else 'Test is False'  
# result = 'Test is True'
```

Увы, это всё еще немного некрасиво. Вы также можете использовать несколько таких конструкций в одной строке:

```
test1 = False  
test2 = True  
result = 'Test1 is True' if test1 else 'Test1 is False, test2 is True' if  
test2 else 'Test1 and Test2 are both False'
```

Сначала выполняется первый if/else, а если test1 = false, выполняется второй if/else. Вы можете делать и более сложные вещи, особенно если воспользуетесь скобками.

Этот способ весьма новый, и я испытываю к нему смешанные чувства. Это правильная, понятная конструкция, она мне нравится... но она всё еще уродлива, особенно при использовании нескольких вложенных конструкций. Конечно, синтаксис всех уловок для выбора значений некрасив. У меня слабость к описанному ниже способу с and/or, сейчас я нахожу его интуитивным, сейчас я понимаю, как он работает. К тому же он ничуть не менее эффективен, чем «правильный» способ.

Хотя инлайновый if/else — новый, более правильный способ, вам всё же стоит ознакомиться со следующими пунктами. Даже если вы планируете использовать Python 2.5, вы встретите эти способы в старом коде. Разумеется, если вам нужна обратная совместимость, будет действительно лучше посмотреть их.

4.2. Уловка and/or

«and» и «or» в Python — сложные создания. Применение and к нескольким выражениям не просто возвращает True или False. Оно возвращает первое false-выражение, либо последнее из выражений, если все они true. Результат ожидаем: если все выражения верны, возвращается последнее, являющееся true; если одно из них false, оно и возвращается и преобразуется к False при проверке логического значения.

Аналогично, операция or возвращает первое true-значение, либо последнее, если ни одно из них не true.

Это вам не поможет, если вы просто проверяете логическое значение выражения. Но можно использовать and и or в других целях. Мой любимый способ — выбор значения в стиле, аналогичном [тернарному оператору](#) языка C "test ? value_if_true : value_if_false":

```
test = True
# test = False
result = test and 'Test is True' or 'Test is False'
# теперь result = 'Test is True'
```

Как это работает? Если `test=True`, оператор `and` пропускает его и возвращает второе (последнее) из данных ему значений: `'Test is True' or 'Test is False'`. Далее, `or` вернет первое `true` выражение, т. е. `'Test is True'`.

Если `test=False`, `and` вернет `test`, останется `test or 'Test is False'`. Т. к. `test=False`, `or` его пропустит и вернет второе выражение, `'Test is False'`.

Внимание, будьте осторожны со средним значением (`"if_true"`). Если оно окажется `false`, выражение с `or` будет всегда пропускать его и возвращать последнее значение (`"if_false"`), независимо от значения `test`.

После использования этого метода правильный способ (п. 4.1) кажется мне менее интуитивным. Если вам не нужна обратная совместимость, попробуйте оба способа и посмотрите, какой вам больше нравится. Если не можете определиться, используйте правильный.

Конечно, если вам нужна совместимость с предыдущими версиями Python, «правильный» способ не будет работать. В этом случае `and/or` — лучший выбор в большинстве ситуаций.

4.3. True/False в качестве индексов

Другой способ выбора из двух значений — использование `True` и `False` как индексов списка с учетом того факта, что `False == 0` и `True == 1`:

```
test = True
# test = False
result = ['Test is False', 'Test is True'][test]
# теперь result = 'Test is True'
```

Этот способ более честный, и `value_if_true` не обязано быть `true`. Однако у него есть существенный недостаток: оба элемента списка вычисляются перед проверкой. Для строк и других простых элементов это не проблема. Но если каждый из них требует больших вычислений или операций ввода-вывода, вычисление обоих выражений недопустимо. Поэтому я предпочитаю обычную конструкцию или `and/or`.

Также заметьте, что этот способ работает только тогда, когда вы уверены, что `test` — булево значение, а не какой-то объект. Иначе придется писать `bool(test)` вместо `test`, чтобы он работал правильно.

5. Функции

5.1. Значения по умолчанию для аргументов вычисляются только один раз

Начнем этот раздел с предупреждения. Эта проблема много раз смущала многих программистов, включая меня, даже после того, как я разобрался в проблеме. Легко ошибиться, используя значения по умолчанию:

```
def function(item, stuff = []):
    stuff.append(item)
    print stuff

function(1)
# ВЫВОДИТ '[1]'
```

```
function(2)
# ВЫВОДИТ '[1,2]' !!!
```

Значения по умолчанию для аргументов вычисляются только один раз, в момент определения функции. Python просто присваивает это значение нужной переменной при каждом вызове функции. При этом он не проверяет, изменилось ли это значение. Поэтому, если вы изменили его, изменение будет в силе при следующих вызовах функции. В предыдущем примере, когда мы добавили значение к списку `stuff`, мы изменили его значение по умолчанию навсегда. Когда мы вызываем функцию снова, ожидая дефолтное значение, мы получаем

измененное.

Решение проблемы: не используйте изменяемые объекты в качестве значений по умолчанию. Вы можете оставить всё как есть, если не изменяете их, но это плохая идея. Вот как следовало написать предыдущий пример:

```
def function(item, stuff = None):
    if stuff is None:
        stuff = []
    stuff.append(item)
    print stuff

function(1)
# ВЫВОДИТ '[1]'

function(2)
# ВЫВОДИТ '[2]', как и ожидалось
```

None неизменяем (в любом случае, мы не пытаемся его изменить), так что мы обезопасили себя от внезапного изменения дефолтного значения.

С другой стороны, умный программист, возможно, превратит это в уловку для использования статических переменных, как в языке C.

5.1.1. Заставляем дефолтные значения вычисляться каждый раз

Если вы не хотите вносить в код функции лишний беспорядок, можно заставить интерпретатор заново вычислять значения аргументов перед каждым вызовом. Следующий [декоратор](#) делает это:

```
from copy import deepcopy

def resetDefaults(f):
    defaults = f.func_defaults
    def resetter(*args, **kwds):
        f.func_defaults = deepcopy(defaults)
        return f(*args, **kwds)
    resetter.__name__ = f.__name__
    return resetter
```

Просто примените этот декоратор к функции, чтобы получить ожидаемые результаты:

```
@resetDefaults # так мы применяем декоратор
def function(item, stuff = []):
    stuff.append(item)
    print stuff

function(1)
# ВЫВОДИТ '[1]'

function(2)
# ВЫВОДИТ '[2]', как и ожидалось
```

5.2. Переменное число аргументов

Python позволяет использовать произвольное число аргументов в функциях. Сначала определяются обязательные аргументы (если они есть), затем нужно указать переменную со звездочкой. Python присвоит ей значение списка остальных (не именованных) аргументов:

```
def do_something(a, b, c, *args):
    print a, b, c, args

do_something(1,2,3,4,5,6,7,8,9)
# ВЫВОДИТ '1, 2, 3, (4, 5, 6, 7, 8, 9)'
```

Зачем это нужно? Например, функция должна принимать несколько элементов и делать с ними одно и то же (например, складывать). Можно заставить пользователя передавать функции список: `sum_all([1,2,3])`. А можно позволить передавать произвольное число аргументов, тогда получится более чистый код: `sum_all(1,2,3)`.

Функция также может иметь переменное число именованных аргументов. После определения всех остальных аргументов укажите переменную с «**» в начале. Python присвоит этой переменной словарь полученных именованных аргументов, кроме обязательных:

```
def do_something_else(a, b, c, *args, **kwargs):
    print a, b, c, args, kwargs

do_something_else(1,2,3,4,5,6,7,8,9, timeout=1.5)
# ВЫВОДИТ '1, 2, 3, (4, 5, 6, 7, 8, 9), {"timeout": 1.5}'
```

Зачем так делать? Я считаю, самая распространенная причина — функция является оберткой другой функции (или функций), и неиспользуемые именованные аргументы могут быть переданы другой функции (см. п. 5.3).

5.2.1. Уточнение

Использование именованных аргументов и произвольного числа обычных аргументов после них, по-видимому, невозможно, потому что именованные аргументы должны быть определены до «*»-параметра. Например, представим функцию:

```
def do_something(a, b, c, actually_print = True, *args):
    if actually_print:
        print a, b, c, args
```

У нас проблема: не получится передать `actually_print` как именованный аргумент, если при этом нужно передать несколько неименованных. Оба следующих варианта вызовут ошибку:

```
do_something(1, 2, 3, 4, 5, actually_print = True)
# actually_print сначала приравнивается к 4 (понятно, почему?), а затем
# переопределяется, вызывая TypeError ('got multiple values for keyword ar
gument')
```

```
do_something(1, 2, 3, actually_print = True, 4, 5, 6)
# Именованные аргументы не могут предшествовать обычным. Происходит Syntax
Error.
```

Единственный способ задать `actually_print` в этой ситуации — передать его как обычный аргумент:

```
do_something(1, 2, 3, True, 4, 5, 6)
# результат: '1, 2, 3, (4, 5, 6)'
```

5.3. Передача списка или словаря в качестве нескольких аргументов

Поскольку можно получить переданные аргументы в виде списка или словаря, нет ничего удивительного в том, что передавать аргументы функции тоже можно из списка или словаря. Синтаксис совершенно такой же, как в предыдущем пункте, нужно поставить перед списком звездочку:

```
args = [5,2]
pow(*args)
# возвращает pow(5,2), т. е. 25
```

А для словаря (что используется чаще) нужно поставить две звездочки:

```

def do_something(actually_do_something=True, print_a_bunch_of_numbers=False):
    if actually_do_something:
        print 'Something has been done'
        #
        if print_a_bunch_of_numbers:
            print range(10)

kwargs = {'actually_do_something': True, 'print_a_bunch_of_numbers': True}
do_something(**kwargs)

# печатает 'Something has been done', затем '[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]'

```

Историческая справка: в Python до версии 2.3 для этих целей использовалась встроенная функция `apply (function, arg_list, keyword_arg_dict)`.

5. Функции

5.4 Декораторы

Декораторы функций довольно просты, но если вы не видели их раньше, вам будет трудно догадаться, как они работают. Это неочевидно, как и большая часть синтаксиса Python. Декоратор — это функция, оборачивающая другую функцию: сначала создается главная функция, затем она передается декоратору. Декоратор возвращает новую функцию, которая используется вместо первоначальной в остальной части программы.

Не будем больше задерживаться на этом. Вот синтаксис:

```

1. def decorator1(func):
2.     return lambda: func() + 1
3.
4. def decorator2(func):
5.     def print_func():
6.         print func()
7.     return print_func
8.
9. @decorator2
10.@decorator1
11.def function():
12.     return 41
13.
14.function()
15.# печатает "42"

```

В этом примере `function` передается в `decorator1`, а он возвращает функцию, которая вызывает `function` и возвращает число, большее ее результата на единицу. Эта функция передается в `decorator2`, который ее вызывает и печатает результат. Так-то.

Следующий пример делает абсолютно то же, но более многословно:

```

1. def decorator1(func):
2.     return lambda: func() + 1
3.
4. def decorator2(func):
5.     def print_func():
6.         print func()
7.     return print_func
8.
9. def function():
10.     return 41
11.
12.function = decorator2(decorator1(function))
13.
14.function()
15.# печатает "42"

```

Обычно декораторы используют для добавления новых возможностей вашим функциям (см. создание методов класса). Еще чаще они не используются совсем. Но чтобы разбираться в коде, вам нужно знать, что это такое.

Чтобы узнать больше о декораторах, обратитесь к статье автора ["Python Decorators Don't Have to be \(that\) Scary"](#), статье [«Python Decorators» на Dr. Dobbs](#) или [разделу «Function Definitions» документации](#).

5.5 Запуск одной из нескольких функций при помощи словаря

Не хватает оператора switch? Вы, возможно, знаете, что в Python нет эквивалента switch для функций (разве что многократное elif). Но вы можете воспроизвести его поведение, создав словарь функций. Например, пусть вы обрабатываете нажатия клавиш и у вас есть следующие функции:

```
1. def key_1_pressed():
2.     print 'Нажата клавиша 1'
3.
4. def key_2_pressed():
5.     print 'Нажата клавиша 2'
6.
7. def key_3_pressed():
8.     print 'Нажата клавиша 3'
9.
10. def unknown_key_pressed():
11.     print 'Нажата неизвестная клавиша'
```

Обычный метод — использование elif:

```
1. keycode = 2
2. if keycode == 1:
3.     key_1_pressed()
4. elif keycode == 2:
5.     key_2_pressed()
6. elif number == 3:
7.     key_3_pressed()
8. else:
9.     unknown_key_pressed()
10. # выводит "Нажата клавиша 2"
```

Но вы также можете положить все функции в словарь, ключами которого будут значения соответствующего keycode. Вы даже можете обработать случай получения неизвестного keycode:

```
1. keycode = 2
2. functions = {1: key_1_pressed, 2: key_2_pressed, 3: key_3_pressed}
3. functions.get(keycode, unknown_key_pressed)()
```

Очевидно, этот код намного понятней предыдущего (особенно при большом числе функций).

6. Классы

6.1 Передача self вручную

Метод — это обычная функция: если он вызывается у экземпляра объекта, этот экземпляр передается в качестве первого аргумента (обычно его называют self). Если по какой-то причине вы вызываете метод не у экземпляра, вы всегда можете первым аргументом передать экземпляр вручную. Например:

```

1. class Class:
2.     def a_method(self):
3.         print 'Привет, метод!'
4.
5. instance = Class()
6.
7. instance.a_method()
8. # печатает 'Привет, метод!', что неудивительно. Вы также можете сделать:
9.
10. Class.a_method(instance)
11. # печатает 'Привет, метод!'

```

Изнутри эти выражения абсолютно одинаковы.

6.2 Проверка на существование метода или свойства

Хотите узнать, есть ли у объекта тот или иной метод или свойство? Вы можете использовать встроенную функцию `hasattr`, которая принимает объект и имя атрибута.

```

1. class Class:
2.     answer = 42
3.
4. hasattr(Class, 'answer')
5. # возвращает True
6. hasattr(Class, 'question')
7. # возвращает False

```

Вы можете также проверить существование атрибута и получить его с помощью встроенной функции `getattr`, которая также принимает объект и имя атрибута. Третий аргумент необязательный и устанавливает значение по умолчанию. Если атрибут не найден и третий аргумент не задан, бросается исключение `AttributeError`.

```

1. class Class:
2.     answer = 42
3.
4. getattr(Class, 'answer')
5. # возвращает 42
6. getattr(Class, 'question', 'Сколько будет 6x9?')
7. # возвращает "Сколько будет 6x9?"
8. getattr(Class, 'question')
9. # бросает исключение AttributeError

```

Не используйте `getattr` и `hasattr` слишком часто. Если вы написали класс так, что необходимо проверять существование свойств, то вы всё сделали неправильно. Свойство должно существовать всегда, а если оно не используется, можно установить его в `None`. Эти функции используются в основном для поддержки полиморфизма, т. е. возможности использовать в вашем коде любых типов объектов.

6.3 Изменение класса после создания

Вы можете добавлять, изменять и удалять свойства и методы класса после его создания, даже после того, как будут созданы экземпляры этого класса. Для этого достаточно использовать запись `Class.attribute`. Изменения применятся ко всем экземплярам класса, вне зависимости от того, когда они были созданы.

```

1. class Class:
2.     def method(self):
3.         print 'Привет, Хабр'
4.
5. instance = Class()
6. instance.method()
7. # печатает "Привет, Хабр"
8.
9. def new_method(self):
10.    print 'Хабр еще торт!'
11.
12. Class.method = new_method
13. instance.method()
14. # печатает "Хабр еще торт!"
15.

```

Фантастика. Но не увлекайтесь изменением существующих методов, это дурной тон. Кроме того, это может нарушить работу методов, использующих изменяемый класс. С другой стороны, добавление методов менее опасно.

6.4 Создание методов класса

Иногда при написании класса возникает потребность в функции, вызываемой от класса, а не его экземпляра. Иногда этот метод создает новые экземпляры, иногда он не зависит от индивидуальных свойств экземпляров. В Python есть два способа это сделать, оба действуют через декораторы. Ваш выбор зависит от того, нужно ли вам знать, какой класс вызвал метод.

Метод класса (class method) принимает вызвавший его класс первым аргументом (аналогично обычные методы получают первым аргументом соответствующий экземпляр). Этому методу известно, запущен он от этого же класса или от подкласса.

Статический метод (static method) не имеет никакой информации о том, откуда он запущен. Это обычная функция, просто в другой области видимости.

Как статические методы, так и методы класса могут быть вызваны непосредственно из класса:

`Class.method()`, либо из экземпляра: `Class().method()`. Во втором случае экземпляр игнорируется, за исключением его класса.

```

1. class Class:
2.     @classmethod
3.     def a_class_method(cls):
4.         print 'Вызван из класса %s' % cls
5.
6.     @staticmethod
7.     def a_static_method():
8.         print 'Понятия не имею, откуда вызван'
9.
10.    def an_instance_method(self):
11.        print 'Вызван из экземпляра %s' % self
12.
13. instance = Class()
14.
15. Class.a_class_method()
16. instance.a_class_method()
17. # оба печатают "Вызван из класса __main__.Class"
18.
19. Class.a_static_method()
20. instance.a_static_method()
21. # оба печатают "Понятия не имею, откуда вызван"
22.
23. Class.an_instance_method()
24. # бросает исключение TypeError
25.
26. instance.an_instance_method()
27. # печатает что-то вроде "Вызван из экземпляра <__main__.Class instance at 0x2e80d0>"

```

Заключение

Нужно больше вдохняющих идей? Для начала неплохо посмотреть страницу [Python Built-in Functions](#). Там довольно много хороших функций, о которых вы, возможно, никогда не слышали.

Если у вас есть полезные уловки или вещи, которые следует знать, автор предлагает добавить их в [свою статью](#).

Счастливого программирования.

С другими статьями автора, а также с источниками информации вы можете ознакомиться [здесь](#).

Переводчик — [Riateche](#).