Как использовать в Python лямбда-функции

24.06.2019 Editorial Team 2 комментария 6 367 просмотров

Spread the love

В Python и других языках, таких как Java, C# и даже C++, в их синтаксис добавлены лямбда-функции, в то время как языки, такие как LISP или семейство языков ML, Haskell, OCaml и F#, используют лямбда-выражения.

Python-лямбды – это маленькие анонимные функции, подчиняющиеся более строгому, но более лаконичному синтаксису, чем обычные функции Python.

К концу этой статьи вы узнаете:

- Как появились лямбды в Python
- Как лямбды сравниваются с обычными объектами функций
- Как написать лямбда-функцию
- Какие функции в стандартной библиотеке Python используют лямбда-выражения
- Когда использовать или избегать лямбда-функций

Примечания : Вы увидите несколько примеров кода с использованием лямбды, которые явно игнорируют лучшие практики стиля Python. Это предназначено только для иллюстрации концепций лямбда-исчисления или для демонстрации возможностей лямбд.

Эти сомнительные примеры будут противопоставляться лучшим подходам или альтернативам по мере прохождения статьи.

Все примеры, включенные в это руководство, были протестированы в Python 3.7.

Лямбда-исчисление

Лямбда-выражения в Python и других языках программирования имеют свои корни в лямбда-исчислении, модели вычислений, изобретенной Алонзо Черчем (Alonzo Church). Далее мы расскажем, когда появилось лямбда-исчисление и почему эта фундаментальная концепция появилась в экосистеме Python.

История

Алонзо Черч формализовал лямбда-исчисление, как язык, основанный на чистой абстракции, в 1930-х годах. Лямбда-функции также называют лямбда-абстракциями, прямой ссылкой на абстракционную модель первоначального творения Алонзо Черч.

В лямбда-исчисление можно закодировать любое вычисление. Оно является полным по Тьюрингу, но вопреки концепции машины Тьюринга оно является чистым и не сохраняет никакого состояния.

Функциональные языки берут свое начало в математической логике и лямбда-исчислении, в то время как императивные языки программирования охватывают основанную на состоянии модель вычислений, изобретенную Аланом Тьюрингом. Две модели вычислений, лямбда-исчисление и машины Тьюринга, могут быть переведены друг в друга. Эта эквивалентность известна как гипотеза Чёрча-Тьюринга.

Функциональные языки напрямую наследуют философию лямбда-исчисления, применяя декларативный подход программирования, которое придает особое значение абстракции, преобразование данных, композицию и чистоту (без состояния и без побочных эффектов). Примерами функциональных языков являются Haskell, Lisp или Erlang.

Напротив, машина Тьюринга привела к императивному программированию, используемому в таких языках, как Fortran, C или Python.

Императивный стиль состоит из программирования с утверждениями, шаг за шагом управляющего ходом программы с подробными инструкциями. Этот подход способствует мутации и требует управления состояние.

Разделение в обоих подходах относительное, поскольку некоторые функциональные языки включают императивные функции, такие как OCaml, в то время как функциональные функции проникают в императивное семейство языков, в частности, с введением лямбда-функций в Java или Python.

Python по своей сути не является функциональным языком, но на раннем этапе он принял некоторые функциональные концепции. В январе 1994 года к языку были добавлены **map()** , **filter()** , **reduce()** и лямбда-оператор.

Первый пример

Вот несколько примеров, чтобы продемонстрировать функциональный стиль.

Функция тождества (identity function), функция, которая возвращает свой аргумент, выражается стандартным определением функции Python с использованием ключевого слова def следующим образом:

```
>>> def identity(x):
...   return x
```

identity() принимает аргумент **x** и возвращает его при вызове.

Если вы воспользуетесь лямбда-конструкцией, ваш код будет следующим:

```
>>> lambda x: x
```

В приведенном выше примере выражение состоит из:

- **Ключевое слово** : lambda
- **Связанная переменная** : x
- **Тело** : x

Примечание . В контексте этой статьи **связанная переменная** является аргументом лямбда-функции.

Напротив, **свободная переменная** не связана и может указываться в теле выражения. Свободная переменная может быть константой или переменной, определенной в прилагаемой области действия функции.

Напишем немного более сложный пример, функцию, которая добавляет 1 к аргументу, следующим образом:

```
>>> lambda x: x + 1
```

Применим указанную выше функцию к аргументу, заключив функцию и ее аргумент в круглые скобки:

```
>>> (lambda x: x + 1)(2)
3
```

Сокращение – это стратегия лямбда-исчисления для вычисления значения выражения. Оно состоит из замены аргумента x на 2:

```
(lambda x: x + 1)(2) = lambda 2: 2 + 1
                     = 2 + 1
                     = 3
```

Поскольку лямбда-функция является выражением, оно может быть именована. Поэтому вы можете написать предыдущий код следующим образом:

```
>>> add_one = lambda x: x + 1
>>> add_one(2)
3
```

Вышеупомянутая лямбда-функция эквивалентна написанию этого:

```
def add_one(x):
    return x + 1
```

Все эти функции принимают один аргумент. Возможно, вы заметили, что в определении лямбды аргументы не имеют круглых скобок вокруг них. Функции с несколькими аргументами (функции, которые принимают более одного аргумента) выражаются в лямбда-выражениях Python, перечисляя аргументы и разделяя их запятой (,), но не заключая их в круглые скобки:

```
>>> full_name = lambda first, last: f'Full name: {first.title()} {la
>>> full_name('guido', 'van rossum')
'Full name: Guido Van Rossum'
```

Лямбда-функция **full_name** , принимает два аргумента и возвращает строку, интерполирующую два параметра: первый и последний. Как и ожидалось, определение лямбды перечисляет аргументы без скобок, тогда как вызов функции выполняется точно так же, как и обычная функция Python, с круглыми скобками вокруг аргументов.

Анонимные функции

Следующие термины могут использоваться взаимозаменяемо в зависимости от языка программирования:

- Анонимные функции
- Лямбда-функции

- Лямбда-выражения
- Лямбда-абстракции
- Лямбда-форма
- Функциональные литералы

В оставшейся части этой статьи после этого раздела вы в основном увидите термин **лямбда-функция** .

В буквальном смысле, анонимная функция – это функция без имени. В Python анонимная функция создается с помощью ключевого слова **lambda** . Рассмотрим анонимную функцию с двумя аргументами, определенную с помощью лямбды, но не связанную с переменной.

```
>>> lambda x, y: x + y
```

Вышеприведенная функция определяет лямбда-выражение, которое принимает два аргумента и возвращает их сумму.

Помимо демонстрации того, что Python отлично подходит для этой идеи, это никак нельзя практически использовать. Вы можете вызвать эту функцию в интерпретаторе Python:

```
>>> _(1, 2)
3
```

В приведенном выше примере используется только функция интерактивного транслятора, представленная через символ подчеркивания (_).

Вы не можете написать подобный код в модуле Python. Рассматривайте _ в интерпретаторе как побочный эффект, которым мы воспользовались. В модуле Python вы бы присваивали лямбда-имя или передавали лямбда-функцию. Мы будет использовать эти два подхода позже в этой статье.

Примечание . В интерактивном интерпретаторе подчеркивание (_) привязано к последнему вычисленному выражению.

Для получения более подробной информации об использовании этого специального символа в Python, посмотрите Значение подчеркивания в Python (The Meaning of Underscores in Python).

Другой шаблон, используемый в других языках, таких как JavaScript, – это немедленное выполнение лямбда-функции Python. Это называется **выражением немедленного вызова функции** (IIFE – **Immediately Invoked Function Expression** , произносится «iffy»). Вот пример:

```
>>> (lambda x, y: x + y)(2, 3)
5
```

Вышеприведенная лямбда-функция определяется, а затем сразу вызывается с двумя аргументами (2 и 3). Возвращает значение 5, которое является суммой аргументов.

Несколько примеров в этом руководстве используют этот формат, чтобы выделить анонимный аспект лямбда-функции и избежать сосредоточения внимания на лямбда-выражениях в Python как более коротком способе определения функции.

Лямбда-функции часто используются с функциями более высокого порядка, которые принимают одну или несколько функций в качестве аргументов или возвращают одну или несколько функций.

Лямбда-функция может быть функцией более высокого порядка, принимая функцию (нормальную или лямбда-функцию) в качестве аргумента, как в следующем надуманном примере:

```
>>> high_ord_func = lambda x, func: x + func(x)
>>> high_ord_func(2, lambda x: x * x)
6
>>> high_ord_func(2, lambda x: x + 3)
7
```

Python содержит функции высшего порядка в виде встроенных функций или в стандартной библиотеке. Примеры функций высшего порядка **map()** , **filter()** , **functools.reduce()** , а также такие ключевые функции, как **sort()** , **sorted()** , **min()** и **max()** . Мы продемонстрируем использование лямбда-функции вместе с функциями высшего порядка в разделе «Соответствующее использование лямбда-выражений».

Лямбда и обычные функции

Эта цитата из часто задаваемых вопросов по Python Design and History FAQ, похоже, задает тон в отношении общего ожидания использования лямбда-функций в Python:

> В отличие от лямбда функций в других языках, где они добавляют функциональность, лямбды в Python являются лишь сокращенной записью, если вы слишком ленивы, чтобы определить функцию. (Source)

Тем не менее, не позволяйте этому утверждению удерживать вас от использования lambda. На первый взгляд, вы можете согласиться с тем, что лямбда-функция – это функция с некоторым синтаксическим сахаром, сокращающим код для определения или вызова функции. В следующих разделах освещены общие черты и тонкие различия между обычными функциями Python и лямбда-функциями.

Функции

В этот момент вы можете задаться вопросом, что принципиально отличает лямбда-функцию, привязанную к переменной, от обычной функции с единственной строкой **return** : кажется что почти ничего. Давайте проверим, как Python видит функцию, созданную с помощью одного оператора **return** , по сравнению с функцией, созданной с выражением **lambda** .

Модуль `dis` предоставляет функции для анализа байт-кода Python, сгенерированного компилятором Python:

```
>>> import dis
>>> add = lambda x, y: x + y
>>> type(add)
<class 'function'>
>>> dis.dis(add)
  1           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
>>> add
<function <lambda> at 0x7f30c6ce9ea0>
```

Вы можете видеть, что **dis** () предоставляет читаемую версию байт-кода Python, позволяющую проверять низкоуровневые инструкции, которые интерпретатор Python будет использовать при выполнении программы.

Теперь посмотрим на обычный объект функции:

```
>>> import dis
>>> def add(x, y): return x + y
>>> type(add)
<class 'function'>
>>> dis.dis(add)
  1           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
>>> add
<function add at 0x7f30c6ce9f28>
```

Байт-код, интерпретируемый Python, одинаков для обеих функций. Но вы можете заметить, что наименование отличается: имя добавляется для функции, определенной с помощью **def** , тогда как лямбда-функция Python рассматривается как лямбда-выражение.

Traceback

В предыдущем разделе вы видели, что в контексте лямбда-функции Python не предоставлял имя функции, а только **<lambda>** . Это может быть ограничением, которое следует учитывать при возникновении исключения, и в результате трассировки отображается только:

```
>>> div_zero = lambda x: x / 0
>>> div_zero(2)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 1, in <lambda>
ZeroDivisionError: division by zero
```

Трассировка исключения, возникшего при выполнении лямбда-функции, идентифицирует только функцию, вызывающую исключение, как **<lambda>** .

Вот то же исключение, вызванное в нормальной функции:

```
>>> def div_zero(x): return x / 0
>>> div_zero(2)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 1, in div_zero
```

```
ZeroDivisionError: division by zero
```

Нормальная функция вызывает аналогичную ошибку, но приводит к более точной трассировке, потому что у нее есть имя функции, **div_zero** .

Синтаксис

Как вы видели в предыдущих разделах, лямбда имеет синтаксические отличия от нормальной функции. В частности, лямбда имеет следующие характеристики:

- Она может содержать только выражения и не может включать операторы в свое тело.
- Она пишется как одна строка исполнения.
- Она не поддерживает аннотации типов.
- Она может быть немедленно вызвана (IIFE).

Отсутствие утверждений

Лямбда-функция не может содержать утверждения. В лямбда-функции такие операторы, как **return** , **pass** , **assert** или **raise** , вызовут исключение **SyntaxError** . Вот пример добавления assert к телу лямбды:

```
>>> (lambda x: assert x == 2)(2)
  File "<input>", line 1
    (lambda x: assert x == 2)(2)
                     ^
SyntaxError: invalid syntax
```

Этот надуманный пример демонстрирующий что с помощью **assert** , утверждается что параметр **x** имеет значение 2. Но интерпретатор выдает **SyntaxError** при синтаксическом анализе кода, который включает в себя утверждение **assert** в теле лямбда-выражения.

Одиночное выражение

В отличие от обычной функции, лямбда-функция представляет собой одно выражение. Хотя в теле лямбды вы можете разбить выражение на несколько строк, используя скобки или многострочную строку, оно остается одним выражением:

```
>>> (lambda x:
```

```
... (x % 2 and 'odd' or 'even'))(3)
'odd'
```

Приведенный выше пример возвращает строку « **odd** », если лямбда-аргумент нечетный, и « **even** », когда аргумент четный. Он распространяется на две строки, поскольку содержится в скобках, но остается одним выражением.

Аннотации типов

Если вы начали применять анотации типов, которые теперь доступны в Python, у вас есть еще одна веская причина предпочесть нормальные функции лямбда-функциям Python. В лямбда-функции нет эквивалента для следующего:

```
def full_name(first: str, last: str) -> str:
    return f'{first.title()} {last.title()}'
```

Любая ошибка типа в **full_name** () может быть обнаружена такими инструментами, как `mypy` или `pyre`, тогда как в эквивалентной лямбда-функцией сразу будет ошибка **SyntaxError** во время выполнения:

```
>>> lambda first: str, last: str: first.title() + " " + last.title()
  File "<stdin>", line 1
    lambda first: str, last: str: first.title() + " " + last.title()
SyntaxError: invalid syntax
```

IIFE

Вы уже видели несколько примеров немедленного запуска функции:

```
>>> (lambda x: x * x)(3)
9
```

Вне интерпретатора эта функция, вероятно, не будет используется на практике. Это прямое следствие того, что лямбда-функция вызывается сразу после того, как она определена. Но, это конструкция позволяет передать определение лямбды в функцию более высокого порядка, например **map()** , **filter()** или **functools.reduce()** .

Аргументы

Как и обычный объект функции, определенный с помощью **def** , лямбда поддерживают все различные способы передачи аргументов. Это включает:

- Позиционные аргументы
- Именованные аргументы (иногда называемые ключевыми аргументами)
- Переменный список аргументов (часто называемый varargs)
- Переменный список аргументов ключевых слов
- Аргументы только для ключевых слов

Следующие примеры иллюстрируют опции, доступные для передачи аргументов в лямбда-выражения:

```
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
>>> (lambda *args: sum(args))(1,2,3)
6
>>> (lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3)
6
>>> (lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3)
6
```

Декораторы

В Python декоратор – это реализация шаблона, который позволяет добавить поведение к функции или классу. Обычно это выражается синтаксисом **@decorator** с префиксом функции. Вот пример:

```
def some_decorator(f):
    def wraps(*args):
        print(f"Calling function '{f.__name__}'")
        return f(args)
    return wraps
@some_decorator
def decorated_function(x):
    print(f"With argument '{x}'")
```

В приведенном выше примере **some_decorator()** – это функция, которая добавляет поведение к **decorated_function()** , так что при

вызове **decorated_function(2)** получается следующий результат:

```
Calling function 'decorated_function'
With argument 'Python'
```

decorated_function() печатает только *With argument 'Python'*, но декоратор добавляет дополнительное поведение, которое также печатает *Calling function 'decorated_function'*.

Декоратор может быть применен к лямбде. Хотя невозможно декорировать лямбду с помощью синтаксиса **@decorator** , декоратор – это просто функция, поэтому он может вызывать функцию лямбда:

```
1 # Defining a decorator
2 def trace(f):
3     def wrap(*args, **kwargs):
4         print(f"[TRACE] func: {f.__name__}, args: {args}, kwargs:
5         return f(*args, **kwargs)
6
7     return wrap
8
9 # Applying decorator to a function
10 @trace
11 def add_two(x):
12     return x + 2
13
14 # Calling the decorated function
15 add_two(3)
16
17 # Applying decorator to a lambda
18 print((trace(lambda x: x ** 2))(3))
```

add_two() , декорирована **@trace** в строке 11, вызывается с аргументом 3 в строке 15. В отличие от этого, в строке 18 сразу же включается лямбда-функция и встраивается в вызов метода **trace()** , декоратора. Когда вы выполняете код выше, вы получаете следующее:

```
[TRACE] func: add_two, args: (3,), kwargs: {}
[TRACE] func: <lambda>, args: (3,), kwargs: {}
9
```

Посмотрите, как, как вы уже видели, имя лямбда-функции выглядит как **<lambda>** , тогда как **add_two** четко идентифицировано как обычная функция.

Декорирование лямбды таким способом может быть полезно для целей отладки, возможно, для отладки поведения лямбды, используемой в контексте функции более высокого порядка или ключевой функции. Давайте посмотрим пример с **map()** :

```
list(map(trace(lambda x: x*2), range(3)))
```

Первый аргумент **map()** – это лямбда, которая умножает свой аргумент на 2. Эта лямбда декорирована **trace()** . При выполнении приведенный выше пример выводит следующее:

```
[TRACE] Calling <lambda> with args (0,) and kwargs {}
[TRACE] Calling <lambda> with args (1,) and kwargs {}
[TRACE] Calling <lambda> with args (2,) and kwargs {}
[0, 2, 4]
```

Результат [0, 2, 4] представляет собой список, полученный умножением каждого элемента **range(3)** . **range(3)** является простым списком [0, 1, 2].

Замыкание

Замыкание – это функция, в которой каждая свободная переменная, кроме параметров, используемых в этой функции, привязана к определенному значению, определенному в рамках области видимости этой функции. В сущности, замыкания определяют среду, в которой они работают, и поэтому могут вызываться из любого места. Более простое определение замыкания это когда функции более низшего порядка имеют доступ к переменным функции более высшего порядка.

Понятия лямбды и замыкания не обязательно связаны, хотя лямбда-функции могут быть замыканиями так же, как обычные функции также могут быть замыканиями. Некоторые языки имеют специальные конструкции для замыкания или лямбды (например, Groovy с анонимным блоком кода в качестве объекта Closure) или лямбда-выражения (например, лямбда-выражения Java с ограниченным параметром для замыкания).

Вот пример замыкания, построенное с помощью обычной функции Python:

```
1 def outer_func(x):
2     y = 4
3     def inner_func(z):
4         print(f"x = {x}, y = {y}, z = {z}")
5         return x + y + z
```

```
    return inner_func

for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

outer_func() возвращает **inner_func()** , вложенную функцию, которая вычисляет сумму трех аргументов:

- **x** передается в качестве аргумента **outer_func()** .
- **y** является локальной переменной для **outer_func()** .
- **z** аргумент, передаваемый в **inner_func()** .

Чтобы продемонстрировать поведение **outer_func()** и **inner_func()** , **outer_func()** вызывается три раза в цикле **for** , который выводит следующее:

```
x = 0, y = 4, z = 5
closure(5) = 9
x = 1, y = 4, z = 6
closure(6) = 11
x = 2, y = 4, z = 7
closure(7) = 13
```

В строке 9 кода **inner_func()** , возвращаемый вызовом **outer_func()** , привязывается к имени замыкания. В строке 5 **inner_func()** захватывает **x** и **y** , потому что он имеет доступ к своей области видимости, так что при вызове замыкания он может работать с двумя свободными переменными **x** и **y** .

Точно так же лямбда также может быть замыканием. Вот тот же пример с лямбда-функцией Python:

```
def outer_func(x):
    y = 4
    return lambda z: x + y + z
for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

Когда вы выполняете приведенный выше код, вы получаете следующий вывод:

```
closure(5) = 9
```

```
closure(6) = 11
closure(7) = 13
```

В строке 6 **outer_func()** возвращает лямбду и присваивает ее переменную замыкания. В строке 3 тело лямбда-функции ссылается на **x** и **y** .
Переменная **y** доступна во время определения, тогда как **x** определяется во время выполнения, когда вызывается **outer_func()** .

В этой ситуации и нормальная функция, и лямбда ведут себя одинаково. В следующем разделе вы увидите ситуацию, когда поведение лямбды может быть обманчивым из-за времени его оценки (время определения против времени выполнения).

Время оценки

В некоторых ситуациях, связанных с циклами, поведение лямбда-функции Python как замыкания может быть нелогичным. Это требует понимания, когда свободные переменные связаны в контексте лямбды. Следующие примеры демонстрируют разницу при использовании обычной функции по сравнению с лямбда-выражением Python.

Сначала протестируем сценарий, используя обычную функцию:

```
 1 >>> def wrap(n):
 2 ...     def f():
 3 ...         print(n)
 4 ...     return f
 5 ...
 6 >>> numbers = 'one', 'two', 'three'
 7 >>> funcs = []
 8 >>> for n in numbers:
 9 ...     funcs.append(wrap(n))
10 ...
11 >>> for f in funcs:
12 ...     f()
13 ...
14 one
15 two
16 three
```

В нормальной функции **n** вычисляется во время определения, в строке 9, когда функция добавляется в список: **funcs.append (wrap (n))** .

Теперь, при реализации той же логики с лямбда-функцией, наблюдаем неожиданное поведение:

```
 1 >>> numbers = 'one', 'two', 'three'
 2 >>> funcs = []
 3 >>> for n in numbers:
 4 ...     funcs.append(lambda: print(n))
 5 ...
 6 >>> for f in funcs:
 7 ...     f()
 8 ...
 9 three
10 three
11 three
```

Неожиданный результат возникает из-за того, что свободная переменная **n** , как она реализована, связана во время выполнения лямбда-выражения. Лямбда-функция Python в строке 4 является замыканием, которое захватывает **n** , свободную переменную, ограниченную во время выполнения. Во время выполнения при вызове функции **f** из строки 7 значение **n** равно **three** .

Чтобы решить эту проблему, вы можете назначить свободную переменную во время определения следующим образом:

```
 1 >>> numbers = 'one', 'two', 'three'
 2 >>> funcs = []
 3 >>> for n in numbers:
 4 ...     funcs.append(lambda n=n: print(n))
 5 ...
 6 >>> for f in funcs:
 7 ...     f()
 8 ...
 9 one
10 two
11 three
```

Лямбда ведет себя как нормальная функция в отношении аргументов. Следовательно, лямбда-параметр может быть инициализирован значением по умолчанию: параметр **n** принимает значение **n** по умолчанию для внешнего **n** . Лямбда может бы быть записана как **lambda x=n: print(x)** и вернуть такой же результат.

Лямбда вызывается без аргумента в строке 7 и использует значение по умолчанию **n** , установленное во время определения.

Тестирование Лямбды

Лямбды можно тестировать аналогично обычным функциям. Можно использовать как **unittest** , так и **doctest** .

unittest

Модуль **unittest** обрабатывает лямбда-функции Python аналогично обычным функциям:

```
import unittest
addtwo = lambda x: x + 2
class LambdaTest(unittest.TestCase):
    def test_add_two(self):
        self.assertEqual(addtwo(2), 4)
    def test_add_two_point_two(self):
        self.assertEqual(addtwo(2.2), 4.2)
    def test_add_three(self):
        # Should fail
        self.assertEqual(addtwo(3), 6)
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

LambdaTest определяет тестовый пример с тремя методами тестирования, каждый из которых использует сценарий тестирования для **addtwo()** , реализованной как лямбда-функция. Выполнение Python-файла **lambda_unittest.py** , содержащего **LambdaTest** , приводит к следующему:

```
$ python lambda_unittest.py
test_add_three (__main__.LambdaTest) ... FAIL
test_add_two (__main__.LambdaTest) ... ok
test_add_two_point_two (__main__.LambdaTest) ... ok
======================================================================
FAIL: test_add_three (__main__.LambdaTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "lambda_unittest.py", line 18, in test_add_three
    self.assertEqual(addtwo(3), 6)
AssertionError: 5 != 6
----------------------------------------------------------------------
Ran 3 tests in 0.001s
FAILED (failures=1)
```

Как и ожидалось, у нас есть два успешных тестовых примера и один сбой для **test_add_three** : результат равен 5, но ожидаемый результат равен 6. Этот сбой вызван преднамеренной ошибкой в тестовом примере. Изменение ожидаемого результата с 6 на 5 удовлетворит все тесты для **LambdaTest** .

doctest

Модуль **doctest** извлекает интерактивный код Python из **docstring** для выполнения тестов. Хотя синтаксис лямбда-функций Python не поддерживает типичную **docstring** , можно присвоить строку элементу **__doc__** именованной переменной лямбды:

```
addtwo = lambda x: x + 2
addtwo.__doc__ = """Add 2 to a number.
    >>> addtwo(2)
    4
    >>> addtwo(2.2)
    4.2
    >>> addtwo(3) # Should fail
    6
    """
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Тест **doctest** в комментарии к функции lambda **addtwo()** описывает те же тесты, что и в предыдущем разделе.

Когда вы выполняете тесты с помощью **doctest.testmod()** , вы получаете следующее:

```
$ python lambda_doctest.py
Trying:
    addtwo(2)
Expecting:
    4
ok
Trying:
    addtwo(2.2)
Expecting:
    4.2
ok
Trying:
    addtwo(3) # Should fail
Expecting:
    6
```

```
**********************************************************************
File "lambda_doctest.py", line 16, in __main__.addtwo
Failed example:
    addtwo(3) # Should fail
Expected:
    6
Got:
    5
1 items had no tests:
    __main__
**********************************************************************
1 items had failures:
   1 of   3 in __main__.addtwo
3 tests in 2 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

Неудачные результаты теста от того же сбоя, объясненного в выполнении модульных тестов в предыдущем разделе.

Вы можете добавить **docstring** к лямбда-выражению через присвоение **__doc__** для документирования лямбда-функции. Хотя это возможно, синтаксис docstring все же лучше использовать для нормальных функций, а не для лямбда-функции.

Злоупотребления лямбда-выражениями

Несколько примеров в этой статье, если они написаны в контексте профессионального кода Python, будут квалифицированы как злоупотребления.

Если вы обнаружите, что пытаетесь использовать что-то, что не поддерживает лямбда-выражение, это, вероятно, признак того, что нормальная функция подойдет лучше. Хорошим примером является docstring для лямбда-выражения в предыдущем разделе. Попытка преодолеть тот факт, что лямбда-функция Python не поддерживает операторы, является еще одним красным флагом.

Следующие разделы иллюстрируют несколько примеров использования лямбды, которых следует избегать. Такими примерами могут быть ситуации, когда в контексте лямбда-кода Python код демонстрирует следующий шаблон:

- Он не следует руководству по стилю Python (PEP 8)
- Код выглядит громоздким и трудно читаемым.

Возникновение исключения

Попытка вызвать исключение в лямбда-выражении Python заставит вас задуматься дважды. Есть несколько способов сделать это, но лучше избегать чего-то вроде следующего:

```
>>> def throw(ex): raise ex
>>> (lambda: throw(Exception('Something bad happened')))()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 1, in <lambda>
    File "<stdin>", line 1, in throw
Exception: Something bad happened
```

Поскольку утверждением не является синтаксически правильным в лямбда-теле Python, обходной путь в приведенном выше примере состоит в абстрагировании вызова оператора с помощью специальной функции **throw()** . Следует избегать использования этого типа обходного пути. Если вы сталкиваетесь с этим типом кода, вам следует рассмотреть возможность рефакторинга кода для использования обычной функции.

Загадочный стиль

Как и в любых языках программирования, вы может столкнуться с код на Python, который может быть трудно читать из-за используемого стиля. Лямбда-функции, благодаря их краткости, могут способствовать написанию кода, который трудно читать.

Следующий лямбда-пример содержит несколько неудачных стилей:

```
>>> (lambda _: list(map(lambda _: _ // 2, _)))([1,2,3,4,5,6,7,8,9,10
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

Подчеркивание (**_**) относится к переменной, на которую вам не нужно ссылаться в явном виде. Но в этом примере три **_** относятся к разным переменным. Первоначальным рефакторингом этого лямбда-кода может быть присвоение имен переменным:

```
>>> (lambda some_list: list(map(lambda n: n // 2,
                                some_list)))([1,2,3,4,5,6,7,8,9,10])
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

По общему признанию, это все еще трудно будет читать. Все еще используя лямбду, обычная функция может сделать этот код более читабельным, распределив логику по нескольким строкам и вызовам функций:

```
>>> def div_items(some_list):
      div_by_two = lambda n: n // 2
      return map(div_by_two, some_list)
>>> list(div_items([1,2,3,4,5,6,7,8,9,10]))
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

Это все еще не оптимально, но показывает вам возможный путь для создания кода и, в частности, лямбда-функций Python, более удобочитаемых. В разделе Альтернативы лямбда-выражениям вы научитесь заменять **map()** и лямбда-выражения на списки или выражения-генераторы. Это значительно улучшит читабельность кода.

Классы Python

Вы можете, но не должны писать методы класса как лямбда-функции Python. Следующий пример является совершенно допустимым кодом Python, но демонстрирует нетрадиционный код, основанный на лямбде. Например, вместо реализации **__str__** как обычной функции он использует лямбду. Аналогично, **brand** и **year** – это свойства, также реализованные с помощью лямбда-функций вместо обычных функций или декораторов:

```
class Car:
    """Car with methods as lambda functions."""
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
    brand = property(lambda self: getattr(self, '_brand'),
                     lambda self, value: setattr(self, '_brand', val
    year = property(lambda self: getattr(self, '_year'),
                    lambda self, value: setattr(self, '_year', value
    __str__ = lambda self: f'{self.brand} {self.year}'  # 1: error E
    honk = lambda self: print('Honk!')     # 2: error E731
```

При запуске такого инструмента, как `flake8`, инструмент обеспечения соблюдения стилей, будут отображаться следующие ошибки для **__str__** и honk:

```
E731 do not assign a lambda expression, use a def
```

Хотя flake8 не указывает на проблему использования лямбда-функций в свойствах, их трудно читать и они подвержены ошибкам из-за использования нескольких строк, таких как **_brand** и **_year** .

Ожидается, что правильная реализация **__str__** будет выглядеть следующим образом:

```
def __str__(self):
    return f'{self.brand} {self.year}'
```

brand будет написана следующим образом:

```
@property
def brand(self):
    return self._brand
@brand.setter
def brand(self, value):
    self._brand = value
```

Как правило, в контексте кода, написанного на Python, предпочитайте обычные функции лямбда-выражениям. Тем не менее, есть случаи, в которых используется лямбда-синтаксис, как вы увидите в следующем разделе.

Правильное использование лямбда-выражений

Лямбды в Python, как правило, являются предметом споров. Некоторые аргументы против лямбды в Python:

- Проблемы с читабельностью
- Наложение функционального мышления
- Тяжелый синтаксис с ключевым словом lambda

Несмотря на жаркие дебаты, ставящие под сомнение само существование этой функции в Python, лямбда-функции имеют свойства, которые иногда предоставляют ценность языку Python и разработчикам.

Следующие примеры иллюстрируют сценарии, в которых использование лямбда-функций не только подходит, но и поощряется в коде Python.

Классические функциональные конструкции

Лямбда-функции регулярно используются со встроенными функциями **map()** и **filter()** , а также **functools.reduce()** , представленными в модуле **functools** . Следующие три примера являются соответствующими иллюстрациями использования этих функций с лямбда-выражениями в качестве компаньонов:

```
>>> list(map(lambda x: x.upper(), ['cat', 'dog', 'cow']))
['CAT', 'DOG', 'COW']
>>> list(filter(lambda x: 'o' in x, ['cat', 'dog', 'cow']))
['dog', 'cow']
>>> from functools import reduce
>>> reduce(lambda acc, x: f'{acc} | {x}', ['cat', 'dog', 'cow'])
'cat | dog | cow'
```

Возможно, вам придется встретить код, похожий на приведенные выше примеры, хотя и с более актуальными данными. По этой причине важно распознавать эти конструкции. Тем не менее, эти конструкции имеют эквивалентные альтернативы, которые считаются более Pythonic. В разделе Альтернативы лямбдам вы узнаете, как преобразовывать функции высшего порядка и сопровождающие их лямбды в другие, более идиоматические формы.

Ключевые функции

Ключевые функции в Python – это функции высшего порядка, которые принимают ключ параметра в качестве именованного аргумента. Ключ получает функцию, которая может быть лямбда-выражением. Эта функция напрямую влияет на алгоритм, управляемый самой ключевой функцией. Вот некоторые ключевые функции:

- **`sort()`:** метод списка
- **`sorted()`, `min()`, `max()`:** встроенные функции
- **`nlargest()` and `nsmallest()`:** в модуле алгоритма очереди кучи heapq

Представьте, что вы хотите отсортировать список идентификаторов, представленных в виде строк. Каждый идентификатор представляет собой объединение идентификатора строки и числа. При сортировке этого списка с помощью встроенной

функции **sorted()** по умолчанию используется лексикографический порядок, поскольку элементы в списке являются строками.

Чтобы повлиять на выполнение сортировки, вы можете назначить лямбду именованному ключу аргумента так, чтобы сортировка использовала число, связанное с идентификатором:

```
>>> ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> print(sorted(ids)) # Lexicographic sort
['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> sorted_ids = sorted(ids, key=lambda x: int(x[2:])) # Integer sor
>>> print(sorted_ids)
['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

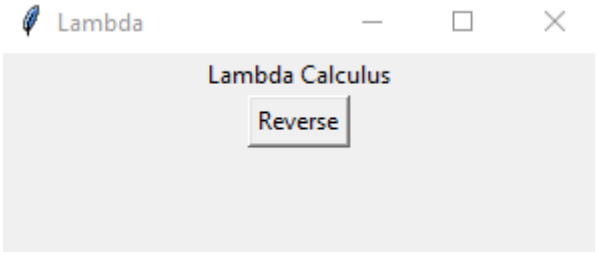
UI Фреймворки

UI фреймворки, такие как Tkinter, wxPython или .NET Windows Forms с IronPython, используют лямбда-функции для отображения действий в ответ на события пользовательского интерфейса.

Простая программа Tkinter, представленная ниже, демонстрирует использование лямбды, назначенной команде кнопки Reverse:

```
import tkinter as tk
import sys
window = tk.Tk()
window.grid_columnconfigure(0, weight=1)
window.title("Lambda")
window.geometry("300x100")
label = tk.Label(window, text="Lambda Calculus")
label.grid(column=0, row=0)
button = tk.Button(
    window,
    text="Reverse",
    command=lambda: label.configure(text=label.cget("text")[::-1]),
)
button.grid(column=0, row=1)
window.mainloop()
```

Нажатие кнопки «*Reverse*» запускает событие, которое запускает лямбда-функцию, изменяя метку с Lambda Calculus на suluclaC adbmaL *:



И wxPython, и IronPython используют одинаковый подход для обработки событий. Обратите внимание, что лямбда-это один из способов обработки событий, но функцию можно использовать для той же цели. В конечном итоге код становится автономным и менее многословным при использовании лямбды, когда объем необходимого кода очень мал.

Интерпритатор Python

Когда вы играете с кодом Python в интерактивном интерпретаторе, лямбда часто являются благословением. Легко создать быструю однострочную функцию для изучения некоторых фрагментов кода, которые никогда не увидят свет вне интерпретатора. Лямбды, написанные в интерпритаторе, ради быстрого запуска, похожи на макулатуру, которую можно выбросить после использования.

timeit

В том же духе, что и эксперименты в интерпретаторе Python, модуль **timeit** предоставляет функции для измерения времени небольших фрагментов кода. В частности, **timeit.timeit()** может вызываться напрямую, передавая некоторый код Python в строку. Вот пример:

```
>>> from timeit import timeit
>>> timeit("factorial(999)", "from math import factorial", number=10
0.0013087529951008037
```

Когда инструкция передается в виде строки, **timeit()** нужен полный контекст. В приведенном выше примере это обеспечивается вторым аргументом, который устанавливает среду, необходимую основной функции для синхронизации. В противном случае возникнет исключение **NameError** .

Другой подход – использовать лямбду:

```
>>> from math import factorial
>>> timeit(lambda: factorial(999), number=10)
0.0012704220062005334
```

Это решение чище, более читабельно и быстрее вводится в интерпретаторе.

Monkey Patching

Для тестирования иногда необходимо полагаться на повторяемые результаты, даже если во время нормального выполнения данного программного обеспечения соответствующие результаты, как ожидается, будут отличаться или даже быть полностью случайными.

Допустим, вы хотите протестировать функцию, которая во время выполнения обрабатывает случайные значения. Но во время выполнения теста вам нужно повторять предсказуемые значения. В следующем примере показано, как лямбда monkey patching может помочь:

```
from contextlib import contextmanager
import secrets
def gen_token():
    """Generate a random token."""
    return f'TOKEN_{secrets.token_hex(8)}'
@contextmanager
def mock_token():
    """Context manager to monkey patch the secrets.token_hex
    function during testing.
    """
    default_token_hex = secrets.token_hex
    secrets.token_hex = lambda _: 'feedfacecafebeef'
    yield
    secrets.token_hex = default_token_hex
def test_gen_key():
    """Test the random token."""
    with mock_token():
        assert gen_token() == f"TOKEN_{'feedfacecafebeef'}"
test_gen_key()
```

Диспетчер контекста помогает изолировать операцию monkey patching функцию из стандартной библиотеки (в этом примере `secrets`). Лямбда назначенная для secrets.token_hex (), заменяет поведение по умолчанию, возвращая статическое значение.

Это позволяет тестировать любую функцию в зависимости от **token_hex()** предсказуемым образом. Перед выходом из диспетчера контекста

поведение **token_hex()** по умолчанию восстанавливается, чтобы устранить любые неожиданные побочные эффекты, которые могут повлиять на другие области тестирования, которые могут зависеть от поведения по умолчанию **token_hex()** .

Среды модульного тестирования, такие как **unittest** и **pytest** , поднимают эту концепцию на более высокий уровень сложности.

С **pytest** , все еще использующим лямбда-функцию, тот же пример становится более элегантным и лаконичным:

```
import secrets
def gen_token():
    return f'TOKEN_{secrets.token_hex(8)}'
def test_gen_key(monkeypatch):
    monkeypatch.setattr('secrets.token_hex', lambda _: 'feedfacecafe
    assert gen_token() == f"TOKEN_{'feedfacecafebeef'}"
```

С помощью **pytest** **secretts.token_hex()** перезаписывается лямбда-выражением, которое будет возвращать детерминированное значение *feedfacecafebeef*, позволяющее подтвердить правильность теста. monkeypatch позволяет вам контролировать область переопределения. В приведенном выше примере при вызове **secretts.token_hex()** в последующих тестах без использования monkey patching будет выполняться обычная реализация этой функции.

Выполнение теста **pytest** дает следующий результат:

```
$ pytest test_token.py -v
============================= test session starts ==================
platform linux -- Python 3.7.2, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
cachedir: .pytest_cache
rootdir: /home/andre/AB/tools/bpython, inifile:
collected 1 item
test_token.py::test_gen_key PASSED
========================== 1 passed in 0.01 seconds ===============
```

Тест проходит, когда мы проверяем, что **gen_token()** был выполнен, и результаты были ожидаемыми в контексте теста.

Альтернативы лямбдам

Хотя существуют веские причины для использования лямбды, есть случаи, когда ее использование не одобряется. Так каковы альтернативы?

Функции более высокого порядка, такие как **map()** , **filter()** и **functools.reduce()** , могут быть преобразованы в более элегантные формы с небольшими изменениями, в частности, со списком или генератором выражений.

Map

Встроенная функция **map()** принимает функцию в качестве первого аргумента и применяет ее к каждому из интерируемых элементов своего второго аргумента. Примерами итерируемых элементов являются строки, списки и кортежи.

map() возвращает итератор, соответствующий преобразованной коллекции. Например, если вы хотите преобразовать список строк в новый список с заглавными буквами, вы можете использовать **map()** следующим образом:

```
>>> list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
['Cat', 'Dog', 'Cow']
```

Вам необходимо вызвать **list()** для преобразования итератора, возвращаемого **map()** , в расширенный список, который можно отобразить в интерпретаторе оболочки Python.

Использование генератора списка исключает необходимость определения и вызова лямбда-функции:

```
>>> [x.capitalize() for x in ['cat', 'dog', 'cow']]
['Cat', 'Dog', 'Cow']
```

Filter

Встроенная функция **filter()** , еще одна классическая функциональная конструкция, может быть преобразована в представление списка. Она принимает предикат в качестве первого аргумента и итеративный список в качестве второго аргумента. Она создает итератор, содержащий все элементы начальной коллекции, удовлетворяющие функции предиката. Вот пример, который фильтрует все четные числа в данном списке целых чисел:

```
>>> even = lambda x: x%2 == 0
>>> list(filter(even, range(11)))
[0, 2, 4, 6, 8, 10]
```

Обратите внимание, что **filter()** возвращает итератор, поэтому необходимо вызывать **list** , который создает список с заданным итератором.

Реализация, использующая конструкцию генератора списка, дает следующее:

```
>>> [x for x in range(11) if x%2 == 0]
[0, 2, 4, 6, 8, 10]
```

Reduce

Начиная с Python 3, **Reduce()** превратился из встроенной функции в функцию модуля **functools** . Что касается **map()** и **filter()** , его первые два аргумента являются соответственно функцией и итерируемым списком. Он также может принимать инициализатор в качестве третьего аргумента, который используется в качестве начального значения результирующего аккумулятора. Для каждого итерируемого элемента **reduce()** применяет функцию и накапливает результат, который возвращается, когда итерация исчерпана.

Чтобы применить **reduce()** к списку пар и вычислить сумму первого элемента каждой пары, вы можете написать так:

```
>>> import functools
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> functools.reduce(lambda acc, pair: acc + pair[0], pairs, 0)
6
```

Более идиоматический подход, использующий выражение генератора в качестве аргумента для **sum()** в следующем примере:

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> sum(x[0] for x in pairs)
6
```

Немного другое и, возможно, более чистое решение устраняет необходимость явного доступа к первому элементу пары и вместо этого использует распаковку:

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> sum(x for x, _ in pairs)
6
```

Использование символа подчеркивания (_) является соглашением Python, указывающим, что вы можете игнорировать второе значение пары.

sum() принимает уникальный аргумент, поэтому выражение генератора не обязательно должно быть в скобках.

Лямбда – это питон или нет?

PEP 8, который является руководством по стилю для кода Python, гласит:

> Всегда используйте оператор **def** вместо оператора присваивания, который связывает лямбду непосредственно с идентификатором. (Источник)

Это правило настоятельно не рекомендует использовать лямбду, привязанную к идентификатору, в основном там, где следует использовать функции. PEP 8 не упоминает другие способы использования лямбды. Как вы видели в предыдущих разделах, лямбды, безусловно, могут найти хорошее применение, хотя они и ограничены.

Возможный способ ответить на этот вопрос заключается в том, что лямбда являются совершенно Pythonic, если нет ничего более доступного Pythonic. Я не буду определять, что означает «Pythonic», оставив вас с определением, которое лучше всего подходит для вашего мышления, а также для вашего личного стиля или стиля кодирования вашей команды.

Заключение

Теперь вы знаете, как использовать лямбды в Python и можете:

- Написать лямбду и использовать анонимные функции
- Мудро выбирать между лямбдами или обычными функциями
- Избегать чрезмерного использования лямбд
- Использовать лямбды с функциями высшего порядка или ключевыми функциями Python

Если у вас есть склонность к математике, вы можете повеселиться, исследуя увлекательный мир лямбда-исчисления (lambda calculus).

Python лямбды подобны соли. Щепотка соли улучшит вкус, но слишком много испортит блюдо.