

Lambda Calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus, a model of computation invented by Alonzo Church. You'll uncover when lambda calculus was introduced and why it's a fundamental concept that ended up in the Python ecosystem.

History

[Alonzo Church](#) formalized [lambda calculus](#), a language based on pure abstraction, in the 1930s. Lambda functions are also referred to as lambda abstractions, a direct reference to the abstraction model of Alonzo Church's original creation.

Lambda calculus can encode any computation. It is [Turing complete](#), but contrary to the concept of a [Turing machine](#), it is pure and does not keep any state.

[Functional](#) languages get their origin in mathematical logic and lambda calculus, while imperative programming languages embrace the state-based model of computation invented by Alan Turing. The two models of computation, lambda calculus and [Turing machines](#), can be translated into each another. This equivalence is known as the [Church-Turing hypothesis](#).

Functional languages directly inherit the lambda calculus philosophy, adopting a declarative approach of programming that emphasizes abstraction, data transformation, composition, and purity (no state and no side effects). Examples of functional languages include [Haskell](#), [Lisp](#), or [Erlang](#).

By contrast, the Turing Machine led to imperative programming found in languages like [Fortran](#), [C](#), or [Python](#).

The imperative style consists of programming with statements, driving the flow of the program step by step with detailed instructions. This approach promotes mutation and requires managing state.

The separation in both families presents some nuances, as some functional languages incorporate imperative features, like [OCaml](#), while functional features have been permeating the imperative family of languages in particular with the introduction of lambda functions in [Java](#), or Python.

Python is not inherently a functional language, but it adopted some functional concepts early on. In January 1994, [map\(\)](#), [filter\(\)](#), [reduce\(\)](#), and the `lambda` operator were added to the language.

First Example

Here are a few examples to give you an appetite for some Python code, functional style.

The [identity function](#), a function that returns its argument, is expressed with a standard Python function definition using the [keyword](#) `def` as follows:

Python

>>>

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

`identity()` takes an argument `x` and returns it upon invocation.

In contrast, if you use a Python lambda construction, you get the following:

Python

>>>

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

In the example above, the expression is composed of:

- **The keyword:** `lambda`
- **A bound variable:** `x`
- **A body:** `x`

Note: In the context of this article, a **bound variable** is an argument to a lambda function.

Improve Your Python

In contrast, a **free variable** is not bound and may be referenced in the body of the expression. A free variable can be a constant or a variable defined in the enclosing [scope](#) of the function.

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

Python

>>>

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

You can apply the function above to an argument by surrounding the function and its argument with parentheses:

Python

>>>

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

[Reduction](#) is a lambda calculus strategy to compute the value of the expression. In the current example, it consists of replacing the bound variable `x` with the argument 2:

Text

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

Because a lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

Python

>>>

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

The above lambda function is equivalent to writing this:

Python

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

These functions all take a single argument. You may have noticed that, in the definition of the lambdas, the arguments don't have parentheses around them. Multi-argument functions (functions that take more than one argument) are expressed in Python lambdas by listing arguments and separating them with a comma (,) but without surrounding them with parentheses:

Python

>>>

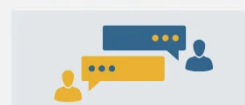
```
>>> full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'
>>> full_name('guido', 'van rossum')
'Full name: Guido Van Rossum'
```

The lambda function assigned to `full_name` takes two arguments and returns a [string](#) interpolating the two parameters `first` and `last`. As expected, the definition of the lambda lists the arguments with no parentheses, whereas calling the function is done exactly like a normal Python function, with parentheses surrounding the arguments.

**A Peer-to-Peer Learning Community for
Python Enthusiasts...Just Like You**

pythonistacafe.com

 PYTHONISTACAFE



 [Remove ads](#)

Anonymous Functions

The following terms may be used interchangeably depending **Improve Your Python**

- Anonymous functions
- Lambda functions
- Lambda expressions
- Lambda abstractions
- Lambda form
- Function literals

For the rest of this article after this section, you'll mostly see the term **lambda function**.

Taken literally, an anonymous function is a function without a name. In Python, an anonymous function is created with the `lambda` keyword. More loosely, it may or not be assigned a name. Consider a two-argument anonymous function defined with `lambda` but not bound to a variable. The `lambda` is not given a name:

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

The function above defines a lambda expression that takes two arguments and returns their sum.

Other than providing you with the feedback that Python is perfectly fine with this form, it doesn't lead to any practical use. You could invoke the function in the Python interpreter:

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

The example above is taking advantage of the interactive interpreter-only feature provided via the underscore (`_`). See the note below for more details.

You could not write similar code in a Python module. Consider the `_` in the interpreter as a side effect that you took advantage of. In a Python module, you would assign a name to the lambda, or you would pass the lambda to a function. You'll use those two approaches later in this article.

Note: In the interactive interpreter, the single underscore (`_`) is bound to the last expression evaluated.

In the example above, the `_` points to the lambda function. For more details about the usage of this special character in Python, check out [The Meaning of Underscores in Python](#).

Another pattern used in other languages like JavaScript is to immediately execute a Python lambda function. This is known as an **Immediately Invoked Function Expression (IIFE)**, pronounce "iffy"). Here's an example:

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

The lambda function above is defined and then immediately called with two arguments (2 and 3). It returns the value 5, which is the sum of the arguments.

Several examples in this tutorial use this format to highlight the anonymous aspect of a lambda function and avoid focusing on `lambda` in Python as a shorter way of defining a function.

Python does not encourage using immediately invoked lambda expressions. It simply results from a lambda expression being callable, unlike the body of a normal function.

Lambda functions are frequently used with [higher-order functions](#), which take one or more functions as arguments or return one or more functions.

A lambda function can be a higher-order function by taking a function (normal or lambda) as an argument like in the following contrived example:

Python

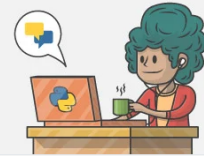
>>>

```
>>> 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 exposes higher-order functions as built-in functions or in the standard library. Examples include `map()`, `filter()`, `functools.reduce()`, as well as key functions like `sort()`, `sorted()`, `min()`, and `max()`. You'll use lambda functions together with Python higher-order functions in [Appropriate Uses of Lambda Expressions](#).

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

pythonistacafe.com



Remove ads

Python Lambda and Regular Functions

This quote from the [Python Design and History FAQ](#) seems to set the tone about the overall expectation regarding the usage of lambda functions in Python:

Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you're too lazy to define a function. ([Source](#))

Nevertheless, don't let this statement deter you from using Python's `lambda`. At first glance, you may accept that a lambda function is a function with some [syntactic sugar](#) shortening the code to define or invoke a function. The following sections highlight the commonalities and subtle differences between normal Python functions and lambda functions.

Functions

At this point, you may wonder what fundamentally distinguishes a lambda function bound to a variable from a regular function with a single return line: under the surface, almost nothing. Let's verify how Python sees a function built with a single [return statement](#) versus a function constructed as an expression (`lambda`).

The [dis](#) module exposes functions to analyze Python bytecode generated by the Python compiler:

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>
```

You can see that `dis()` expose a readable version of the Python bytecode allowing the inspection of the low-level instructions that the Python interpreter will use while executing the program.

Now see it with a regular function object:

Python

>>>

```
>>> 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>
```

The bytecode interpreted by Python is the same for both functions. But you may notice that the naming is different: the function name is `add` for a function defined with `def`, whereas the Python lambda function is seen as `lambda`.

Traceback

You saw in the previous section that, in the context of the lambda function, Python did not provide the name of the function, but only `<lambda>`. This can be a limitation to consider when an exception occurs, and a [traceback](#) shows only `<lambda>`:

Python

>>>

```
>>> 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
```

The [traceback](#) of an exception raised while a lambda function is executed only identifies the function causing the exception as `<lambda>`.

Here's the same exception raised by a normal function:

Python

>>>

```
>>> 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
```

The normal function causes a similar error but results in a more precise traceback because it gives the function name, `div_zero`.

Syntax

As you saw in the previous sections, a lambda form presents syntactic distinctions from a normal function. In particular, a lambda function has the following characteristics:

- It can only contain expressions and can't include statements in its body.
- It is written as a single line of execution.
- It does not support type annotations.
- It can be immediately invoked (IIFE).

No Statements

A lambda function can't contain any statements. In a lambda function, statements like `return`, `pass`, `assert`, or `raise` will raise a [SyntaxError](#) exception. Here's an example of adding `assert` to the body of a lambda:

Python

>>>

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

This contrived example intended to assert that parameter `x` had a value of 2. But, the interpreter identifies a `SyntaxError` while parsing the code that involves the statement `assert` in the body of the `lambda`.

Single Expression

In contrast to a normal function, a Python lambda function is a single expression. Although, in the body of a `lambda`, you can spread the expression over several lines using parentheses or a multiline string, it remains a single expression:

Python

>>>

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

The example above returns the string `'odd'` when the lambda argument is odd, and `'even'` when the argument is even. It spreads across two lines because it is contained in a set of parentheses, but it remains a single expression.

Type Annotations

If you've started adopting type hinting, which is now available in Python, then you have another good reason to prefer normal functions over Python lambda functions. Check out [Python Type Checking \(Guide\)](#) to get learn more about Python type hints and type checking. In a lambda function, there is no equivalent for the following:

Python

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

Any type error with `full_name()` can be caught by tools like [mypy](#) or [pyre](#), whereas a `SyntaxError` with the equivalent lambda function is raised at runtime:

Python

>>>

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

Like trying to include a statement in a lambda, adding type annotation immediately results in a `SyntaxError` at runtime.

IIFE

You've already seen several examples of [immediately invoked function execution](#):

Python

>>>

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

Outside of the Python interpreter, this feature is probably not used in practice. It's a direct consequence of a lambda function being callable as it is defined. For example, this allows you to pass the definition of a Python lambda expression to a higher-order function like `map()`, `filter()`, or `functools.reduce()`, or to a key function.



[Remove ads](#)

Arguments

Like a normal function object defined with `def`, Python lambda expressions support all the different ways of passing arguments. This includes:

- Positional arguments
- Named arguments (sometimes called keyword arguments)
- Variable list of arguments (often referred to as **varargs**)
- Variable list of keyword arguments
- Keyword-only arguments

The following examples illustrate options open to you in order to pass arguments to lambda expressions:

Python

>>>

```
>>> (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
```

Decorators

In Python, a [decorator](#) is the implementation of a pattern that allows adding a behavior to a function or a class. It is usually expressed with the `@decorator` syntax prefixing a function. Here's a contrived example:

Python

```
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}'")
```

In the example above, `some_decorator()` is a function that adds a behavior to `decorated_function()`, so that invoking `decorated_function("Python")` results in the following output:

Shell

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

`decorated_function()` only prints `With argument 'Python'`, but the decorator adds an extra behavior that also prints `Calling function 'decorated_function'`.

A decorator can be applied to a lambda. Although it's not possible to decorate a lambda with the `@decorator` syntax, a decorator is just a function, so it can call the lambda function:

Python

```
1 # Defining a decorator
2 def trace(f):
3     def wrap(*args, **kwargs):
4         print(f"[TRACE] func: {f.__name__}, args: {args}, kwargs: {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()`, decorated with `@trace` on line 11, is invoked with argument 3 on line 15. By contrast, on line 18, a lambda function is immediately invoked and embedded in a call to `trace()`, the decorator. When you execute the code above you obtain the following:

Shell

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

See how, as you've already seen, the name of the lambda function appears as `<lambda>`, whereas `add_two` is clearly identified for the normal function.

Decorating the lambda function this way could be useful for debugging purposes, possibly to debug the behavior of a lambda function used in the context of a higher-order function or a key function. Let's see an example with `map()`:

Python

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

The first argument of `map()` is a lambda that multiplies its argument by 2. This lambda is decorated with `trace()`. When executed, the example above outputs the following:

Shell

```
[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]
```


The result `[0, 2, 4]` is a [list](#) obtained from multiplying each element of `range(3)`. For now, consider `range(3)` equivalent to the list `[0, 1, 2]`.

You will be exposed to `map()` in more details in [Map](#).

A lambda can also be a decorator, but it's not recommended. If you find yourself needing to do this, consult [PEP 8, Programming Recommendations](#).

For more on Python decorators, check out [Primer on Python Decorators](#).



 [Remove ads](#)

Closure

A [closure](#) is a function where every free variable, everything except parameters, used in that function is bound to a specific value defined in the enclosing scope of that function. In effect, closures define the environment in which they run, and so can be called from anywhere.

The concepts of lambdas and closures are not necessarily related, although lambda functions can be closures in the same way that normal functions can also be closures. Some languages have special constructs for closure or lambda (for example, Groovy with an anonymous block of code as Closure object), or a lambda expression (for example, Java Lambda expression with a limited option for closure).

Here's a closure constructed with a normal Python function:

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
6     return inner_func
7
8 for i in range(3):
9     closure = outer_func(i)
10    print(f"closure({i+5}) = {closure(i+5)}")
```

`outer_func()` returns `inner_func()`, a [nested function](#) that computes the sum of three arguments:

- `x` is passed as an argument to `outer_func()`.
- `y` is a variable local to `outer_func()`.
- `z` is an argument passed to `inner_func()`.

To test the behavior of `outer_func()` and `inner_func()`, `outer_func()` is invoked three times in a [for loop](#) that prints the following:

Shell

```
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
```

On line 9 of the code, `inner_func()` returned by the invocation of `outer_func()` is bound to the name `closure`. On line 5, `inner_func()` captures `x` and `y` because it has access to its embedding environment, such that upon invocation of the closure, it is able to operate on the two free variables `x` and `y`.

Similarly, a `lambda` can also be a closure. Here's the same example with a Python `lambda` function:

Python

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

When you execute the code above, you obtain the following output:

Shell

```
closure(5) = 9
closure(6) = 11
closure(7) = 13
```

On line 6, `outer_func()` returns a lambda and assigns it to the variable `closure`. On line 3, the body of the lambda function references `x` and `y`. The variable `y` is available at definition time, whereas `x` is defined at runtime when `outer_func()` is invoked.

In this situation, both the normal function and the lambda behave similarly. In the next section, you'll see a situation where the behavior of a lambda can be deceptive due to its evaluation time (definition time vs runtime).

Evaluation Time

In some situations involving [loops](#), the behavior of a Python lambda function as a closure may be counterintuitive. It requires understanding when free variables are bound in the context of a lambda. The following examples demonstrate the difference when using a regular function vs using a Python lambda.

Test the scenario first using a regular function:

Python

>>>

```
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
```

In a normal function, `n` is evaluated at definition time, on line 9, when the function is added to the list: `funcs.append(wrap(n))`.

Now, with the implementation of the same logic with a lambda function, observe the unexpected behavior:

Python

>>>

```
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
```

The unexpected result occurs because the free variable `n`, as implemented, is bound at the execution time of the lambda expression. The Python lambda function on line 4 is a closure that captures `n`, a free variable bound at runtime. At runtime, while invoking the function `f` on line 7, the value of `n` is three.

To overcome this issue, you can assign the free variable at definition time as follows:

Python

>>>

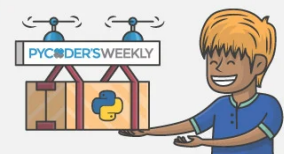
```
1 >>> numbers = 'one', 'two', 'three'
2 >>> funcs = []
3 >>> for n in numbers:
4 ...     funcs.append(lambda n=n: print(n))
5 ...
6 >>> for f in funcs:
7 ...     f()
8 ...
9 one
10 two
11 three
```

A Python lambda function behaves like a normal function in regard to arguments. Therefore, a lambda parameter can be initialized with a default value: the parameter `n` takes the outer `n` as a default value. The Python lambda function could have been written as `lambda x=n: print(x)` and have the same result.

The Python lambda function is invoked without any argument on line 7, and it uses the default value `n` set at definition time.

Your Weekly Dose of All Things Python!

pycoders.com



[Remove ads](#)

Testing Lambdas

Python lambdas can be tested similarly to regular functions. It's possible to use both `unittest` and `doctest`.

unittest

The `unittest` module handles Python lambda functions similarly to regular functions:

Python

```
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 defines a test case with three test methods, each of them exercising a test scenario for `addtwo()` implemented as a lambda function. The execution of the Python file `lambda_unittest.py` that contains LambdaTest produces the following:

Shell

```
$ 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)
```

As expected, we have two successful test cases and one failure for `test_add_three`: the result is 5, but the expected result was 6. This failure is due to an intentional mistake in the test case. Changing the expected result from 6 to 5 will satisfy all the tests for LambdaTest.

doctest

The doctest module extracts interactive Python code from `docstring` to execute tests. Although the syntax of Python lambda functions does not support a typical `docstring`, it is possible to assign a string to the `__doc__` element of a named lambda:

Python

```
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)
```

The doctest in the doc comment of `lambda addtwo()` describes the same test cases as in the previous section.

When you execute the tests via `doctest.testmod()`, you get the following:

Shell

```
$ 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.
```

The failed test results from the same failure explained in the execution of the unit tests in the previous section.

You can add a `docstring` to a Python lambda via an assignment to `__doc__` to document a lambda function. Although possible, the Python syntax better accommodates `docstring` for normal functions than lambda functions.

For a comprehensive overview of unit testing in Python, you may want to refer to [Getting Started With Testing in Python](#).

Lambda Expression Abuses

Several examples in this article, if written in the context of professional Python code, would qualify as abuses.

If you find yourself trying to overcome something that a lambda expression does not support, this is probably a sign that a normal function would be better suited. The `docstring` for a lambda expression in the previous section is a good example. Attempting to overcome the fact that a Python lambda function does not support statements is another red flag.

The next sections illustrate a few examples of lambda usages that should be avoided. Those examples might be situations where, in the context of Python lambda, the code exhibits the following pattern:

- It doesn't follow the Python style guide (PEP 8)
- It's cumbersome and difficult to read.
- It's unnecessarily clever at the cost of difficult readability.



Raising an Exception

Trying to raise an exception in a Python lambda should make you think twice. There are some clever ways to do so, but even something like the following is better to avoid:

```
Python >>>
>>> 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
```

Because a statement is not syntactically correct in a Python lambda body, the workaround in the example above consists of abstracting the statement call with a dedicated function `throw()`. Using this type of workaround should be avoided. If you encounter this type of code, you should consider refactoring the code to use a regular function.

Cryptic Style

As in any programming languages, you will find Python code that can be difficult to read because of the style used. Lambda functions, due to their conciseness, can be conducive to writing code that is difficult to read.

The following lambda example contains several bad style choices:

```
Python >>>
>>> (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]
```

The underscore (`_`) refers to a variable that you don't need to refer to explicitly. But in this example, three `_` refer to different variables. An initial upgrade to this lambda code could be to name the variables:

```
Python >>>
>>> (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]
```

Admittedly, it's still difficult to read. By still taking advantage of a lambda, a regular function would go a long way to render this code more readable, spreading the logic over a few lines and function calls:

```
Python >>>
>>> 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]
```

This is still not optimal but shows you a possible path to make code, and Python lambda functions in particular, more readable. In [Alternatives to Lambdas](#), you'll learn to replace `map()` and `lambda` with list comprehensions or [generator expressions](#). This will drastically improve the readability of the code.

Python Classes

You can but should not write class methods as Python lambda functions. The following example is perfectly legal Python code but exhibits unconventional Python code relying on lambda. For example, instead of implementing `__str__` as a regular function, it uses a lambda. Similarly, `brand` and `year` are [properties](#) also implemented with lambda functions, instead of regular functions or decorators:

Python

```
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', value))

    year = property(lambda self: getattr(self, '_year'),
                    lambda self, value: setattr(self, '_year', value))

    __str__ = lambda self: f'{self.brand} {self.year}' # 1: error E731

    honk = lambda self: print('Honk!') # 2: error E731
```

Running a tool like [flake8](#), a style guide enforcement tool, will display the following errors for `__str__` and `honk`:

Shell

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

Although `flake8` doesn't point out an issue for the usage of the Python lambda functions in the properties, they are difficult to read and prone to error because of the usage of multiple strings like `'_brand'` and `'_year'`.

Proper implementation of `__str__` would be expected to be as follows:

Python

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

`brand` would be written as follows:

Python

```
@property
def brand(self):
    return self._brand

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

As a general rule, in the context of code written in Python, prefer regular functions over lambda expressions. Nonetheless, there are cases that benefit from lambda syntax, as you will see in the next section.



[Remove ads](#)

Appropriate Uses of Lambda Expressions

Lambdas in Python tend to be the subject of controversies. Some of the arguments against lambdas in Python are:

- Issues with readability
- The imposition of a functional way of thinking
- Heavy syntax with the `lambda` keyword

Despite the heated debates questioning the mere existence of

Improve Your Python

sometimes provide value to the Python language and to developers.

The following examples illustrate scenarios where the use of lambda functions is not only suitable but encouraged in Python code.

Classic Functional Constructs

Lambda functions are regularly used with the built-in functions [map\(\)](#) and [filter\(\)](#), as well as [functools.reduce\(\)](#), exposed in the module [functools](#). The following three examples are respective illustrations of using those functions with lambda expressions as companions:

Python

>>>

```
>>> 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'
```

You may have to read code resembling the examples above, albeit with more relevant data. For that reason, it's important to recognize those constructs. Nevertheless, those constructs have equivalent alternatives that are considered more Pythonic. In [Alternatives to Lambdas](#), you'll learn how to convert higher-order functions and their accompanying lambdas into other more idiomatic forms.

Key Functions

Key functions in Python are higher-order functions that take a parameter `key` as a named argument. `key` receives a function that can be a lambda. This function directly influences the algorithm driven by the key function itself. Here are some key functions:

- `sort()`: list method
- `sorted()`, `min()`, `max()`: built-in functions
- `nlargest()` and `nsmallest()`: in the Heap queue algorithm module [heapq](#)

Imagine that you want to sort a list of IDs represented as strings. Each ID is the [concatenation](#) of the string `id` and a number. Sorting this list with the built-in function `sorted()`, by default, uses a lexicographic order as the elements in the list are strings.

To influence the sorting execution, you can assign a lambda to the named argument `key`, such that the sorting will use the number associated with the ID:

Python

>>>

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

UI Frameworks

UI frameworks like [Tkinter](#), [wxPython](#), or .NET Windows Forms with [IronPython](#) take advantage of lambda functions for mapping actions in response to UI events.

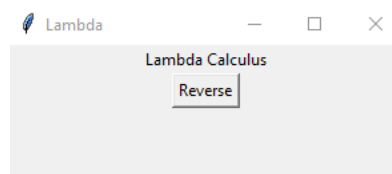
The naive Tkinter program below demonstrates the usage of a lambda assigned to the command of the *Reverse* button:

Python

```
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()
```

Clicking the button *Reverse* fires an event that triggers the lambda function, changing the label from *Lambda Calculus* to *culaculadmaL*:



Both wxPython and IronPython on the .NET platform share a similar approach for handling events. Note that lambda is one way to handle firing events, but a function may be used for the same purpose. It ends up being self-contained and less verbose to use a lambda when the amount of code needed is very short.

To explore wxPython, check out [How to Build a Python GUI Application With wxPython](#).



Remove ads

Python Interpreter

When you're playing with Python code in the interactive interpreter, Python lambda functions are often a blessing. It's easy to craft a quick one-liner function to explore some snippets of code that will never see the light of day outside of the interpreter. The lambdas written in the interpreter, for the sake of speedy discovery, are like scrap paper that you can throw away after use.

timeit

In the same spirit as the experimentation in the Python interpreter, the module `timeit` provides functions to time small code fragments. `timeit.timeit()` in particular can be called directly, passing some Python code in a string. Here's an example:

Python

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

When the statement is passed as a string, `timeit()` needs the full context. In the example above, this is provided by the second argument that sets up the environment needed by the main function to be timed. Not doing so would raise a `NameError` exception.

Another approach is to use a lambda:

Improve Your Python

Python

>>>

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

This solution is cleaner, more readable, and quicker to type in the interpreter. Although the execution time was slightly less for the `lambda` version, executing the functions again may show a slight advantage for the `string` version. The execution time of the setup is excluded from the overall execution time and shouldn't have any impact on the result.

Monkey Patching

For testing, it's sometimes necessary to rely on repeatable results, even if during the normal execution of a given software, the corresponding results are expected to differ, or even be totally random.

Let's say you want to test a function that, at runtime, handles [random values](#). But, during the testing execution, you need to assert against predictable values in a repeatable manner. The following example shows how, with a `lambda` function, monkey patching can help you:

Python

```
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()
```

A context manager helps with insulating the operation of monkey patching a function from the standard library ([secrets](#), in this example). The `lambda` function assigned to `secrets.token_hex()` substitutes the default behavior by returning a static value.

This allows testing any function depending on `token_hex()` in a predictable fashion. Prior to exiting from the context manager, the default behavior of `token_hex()` is reestablished to eliminate any unexpected side effects that would affect other areas of the testing that may depend on the default behavior of `token_hex()`.

Unit test frameworks like `unittest` and `pytest` take this concept to a higher level of sophistication.

With [pytest](#), still using a `lambda` function, the same example becomes more elegant and concise :

Python

```
import secrets

def gen_token():
    return f'TOKEN_{secrets.token_hex(8)}'

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

Improve Your Python

With the [pytest monkeypatch fixture](#), `secrets.token_hex()` is overwritten with a lambda that will return a deterministic value, `feedfacecafebeef`, allowing to validate the test. The `pytest monkeypatch` fixture allows you to control the scope of the override. In the example above, invoking `secrets.token_hex()` in subsequent tests, without using monkey patching, would execute the normal implementation of this function.

Executing the `pytest` test gives the following result:

Shell

```
$ 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 [100%]

===== 1 passed in 0.01 seconds =====
```

The test passes as we validated that the `gen_token()` was exercised, and the results were the expected ones in the context of the test.



[Remove ads](#)

Alternatives to Lambdas

While there are great reasons to use `lambda`, there are instances where its use is frowned upon. So what are the alternatives?

Higher-order functions like `map()`, `filter()`, and `functools.reduce()` can be converted to more elegant forms with slight twists of creativity, in particular with list comprehensions or generator expressions.

To learn more about list comprehensions, check out [When to Use a List Comprehension in Python](#). To learn more about generator expressions, check out [How to Use Generators and yield in Python](#).

Map

The built-in function `map()` takes a function as a first argument and applies it to each of the elements of its second argument, an **iterable**. Examples of iterables are strings, lists, and tuples. For more information on iterables and iterators, check out [Iterables and Iterators](#).

`map()` returns an iterator corresponding to the transformed collection. As an example, if you wanted to transform a list of strings to a new list with each string capitalized, you could use `map()`, as follows:

Python

>>>

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

You need to invoke `list()` to convert the iterator returned by `map()` into an expanded list that can be displayed in the Python shell interpreter.

Using a list comprehension eliminates the need for defining and invoking the `lambda` function:

Python

>>>

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

Improve Your Python

Filter

The built-in function `filter()`, another classic functional construct, can be converted into a list comprehension. It takes a [predicate](#) as a first argument and an iterable as a second argument. It builds an iterator containing all the elements of the initial collection that satisfies the predicate function. Here's an example that filters all the even numbers in a given list of integers:

Python

>>>

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

Note that `filter()` returns an iterator, hence the need to invoke the built-in type `list` that constructs a list given an iterator.

The implementation leveraging the list comprehension construct gives the following:

Python

>>>

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

Reduce

Since Python 3, [reduce\(\)](#) has gone from a built-in function to a `functools` module function. As `map()` and `filter()`, its first two arguments are respectively a function and an iterable. It may also take an initializer as a third argument that is used as the initial value of the resulting accumulator. For each element of the iterable, `reduce()` applies the function and accumulates the result that is returned when the iterable is exhausted.

To apply `reduce()` to a list of pairs and calculate the sum of the first item of each pair, you could write this:

Python

>>>

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

A more idiomatic approach using a [generator expression](#), as an argument to `sum()` in the example, is the following:

Python

>>>

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

A slightly different and possibly cleaner solution removes the need to explicitly access the first element of the pair and instead use unpacking:

Python

>>>

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

The use of underscore (`_`) is a Python convention indicating that you can ignore the second value of the pair.

`sum()` takes a unique argument, so the generator expression does not need to be in parentheses.

Are Lambdas Pythonic or Not?

[PEP 8](#), which is the style guide for Python code, reads:

Always use a `def` statement instead of an assignment statement that binds a lambda expression directly to an identifier.
([Source](#))

This strongly discourages using lambda bound to an identifier, mainly where functions should be used and have more benefits. PEP 8 does not mention other usages of lambda. As you have seen in the previous sections, lambda functions may certainly have good uses, although they are limited.

A possible way to answer the question is that lambda functions are perfectly Pythonic if there is nothing more Pythonic available. I'm staying away from defining what "Pythonic" means, leaving you with the definition that best suits your mindset, as well as your personal or your team's coding style.

Beyond the narrow scope of Python lambda, [How to Write Beautiful Python Code With PEP 8](#) is a great resource that you may want to check out regarding code style in Python.


Conclusion

You now know how to use Python lambda functions and can:

- Write Python lambdas and use anonymous functions
- Choose wisely between lambdas or normal Python functions
- Avoid excessive use of lambdas
- Use lambdas with higher-order functions or Python key functions

If you have a penchant for mathematics, you may have some fun exploring the fascinating world of [lambda calculus](#).

Python lambdas are like salt. A pinch in your spam, ham, and eggs will enhance the flavors, but too much will spoil the dish.


 **Take the Quiz:** Test your knowledge with our interactive "Python Lambda Functions" quiz. Upon completion you will receive a score so you can track your learning progress over time:

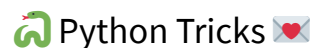
Take the Quiz »

Note: The Python programming language, named after Monty Python, prefers to use [spam](#), ham, and eggs as metasyntactic variables, instead of the traditional foo, bar, and baz.

Mark as Completed



 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Use Python Lambda Functions](#)



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python