Lambda Function (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Lambda Function (1)

A **lambda function** is a small anonymous function.

A lambda function can take any number of arguments,
but can have only one expression.

Syntax
lambda arguments : expression

https://www.w3schools.com/python/python_lambda.asp

Lambda Functions (2)

The expression is executed and the result is returned:

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Lambda Functions (3)

Why Use Lambda Functions?

The power of lambda is better shown
when you use them as an anonymous function
inside another function.

Say you have a function definition that takes one argument,
and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

Use that function definition to make a function
that always doubles the number you send in:

```
mydoubler = myfunc(2)

print(mydoubler(11))
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Lambda Functions (4-1)

Or, use the same function definition
to make a function that always triples the number you send in:

```
def myfunc(n):
    return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Lambda Functions (4-2)

Or, use the same function definition
to make both functions, in the same program:

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions
when an anonymous function is required
for a short period of time.

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Lambda Functions (5)

Python and other languages
like Java, C#, and even C++
have had lambda functions added to their syntax,

whereas languages like LISP or
the ML (Meta Language) family of languages,
Haskell, OCaml, and F#,
use lambdas as a core concept.

Python lambdas are little, anonymous functions,
subject to a more *restrictive*
but more *concise* syntax than regular Python functions.

https://realpython.com/python-lambda/

Meta Language (1)

ML (Meta Language) is
a general-purpose functional programming language.

It is known for its use of
the polymorphic Hindley–Milner type system,
which automatically assigns the types of most expressions
without requiring explicit type annotations,
and ensures type safety

– there is a formal proof that
a well-typed ML program does not cause runtime type errors.

https://realpython.com/python-lambda/

Meta Language (2)

ML provides
- pattern matching for function arguments
- garbage collection
- imperative programming
- call-by-value
- currying

It is used heavily in programming language research and is one of the few languages to be completely specified and verified using formal semantics.

Its types and pattern matching make it well-suited and commonly used to operate on other formal languages, such as in compiler writing, automated theorem proving, and formal verification.

https://realpython.com/python-lambda/

Lambda Calculus (1)

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.

https://realpython.com/python-lambda/

First example (1)

Here are a few examples, functional style.

The **identity** function, a function that returns its argument,
is expressed with a standard Python function definition :

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

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

In contrast, if you use a Python lambda construction

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

In the example above, the expression is composed of:

The keyword: lambda
A bound variable: x
A body: x

https://realpython.com/python-lambda/

First example (2)

a bound variable is an argument to a lambda function.

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.

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

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

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

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

https://realpython.com/python-lambda/

First example (3)

Reduction is a lambda calculus strategy
to compute the value of the expression.

In the example, it consists of replacing the bound variable **x** with the argument **2**:

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

Because a lambda function is an expression,
it can be named.

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

The above lambda function is equivalent to writing this:

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

https://realpython.com/python-lambda/

First example (4-1)

These functions all take a single argument.
in the definition of the lambdas,
the arguments don't have parentheses around them.

Multi-argument functions are expressed in Python lambdas
by listing arguments and separating them with a comma (,)
but without surrounding them with parentheses:

https://realpython.com/python-lambda/

First example (4-1)

```
>>> 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.

https://realpython.com/python-lambda/

Anonymous Functions (1)

The following terms may be used interchangeably
depending on the programming language type and culture:

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

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:

https://realpython.com/python-lambda/

Anonymous Functions (2-1)

```
>>> 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:

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

In the interactive interpreter, the single underscore _
is bound to the *last expression evaluated*.

In the example above,
the _ points to the lambda function.

https://realpython.com/python-lambda/

Anonymous Functions (2-2)

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

The example above is taking advantage of the interactive interpreter-only feature provided via the underscore (_).

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 pass the lambda to a function.

https://realpython.com/python-lambda/

Anonymous Functions (3-1)

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”)

```
>>> (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.

https://realpython.com/python-lambda/

Anonymous Functions (3-2)

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.

https://realpython.com/python-lambda/

Anonymous Functions (4-1)

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

```
>>> 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
```

https://realpython.com/python-lambda/

Anonymous Functions (4-1)

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**().

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (1-1)

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.

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (1-2)

differences between normal Python functions and Python lambda functions.

Functions

what fundamentally distinguishes a lambda function bound to a variable
from a regular function with a single return line:
under the surface, almost nothing.

how Python sees a function built with a single return statement
versus a function constructed as an expression (lambda).

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (2)

The **dis** module exposes functions
to analyze Python bytecode
generated by the Python compiler:

```
>>> 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.

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (2)

Now see it with a regular function object:

```
>>> 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>
```

```
>>> 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>
```

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.

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (4-1)

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>:

```
>>> 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
```

https://realpython.com/python-lambda/

Python Lambda and Regular Functions (4-1)

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:

```
>>> 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**.

https://realpython.com/python-lambda/

Syntax

Syntax

a **lambda function** has the following characteristics:

It can only contain expressions and
It cannot 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
    Immediately Invoked Function Expression (IIFE).

https://realpython.com/python-lambda/

No statements

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:

```
>>> (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.

https://realpython.com/python-lambda/

Single expression

Single Expression

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:

```
>>> (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.

True and ‘odd’ or ‘even’

True then evaluate ‘odd’
which is true (non-zero value)
and returns ‘odd’

False and ‘odd’ or ‘even’

False therefore **‘odd’** is not evaluated
And **‘even’** is evaluated
This gives **‘even’**

x % 2 is equal to **1 (True)**
x % 2 is equal to **0 (False)**

https://realpython.com/python-lambda/

Type annotations (1)

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:

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

https://realpython.com/python-lambda/

Type annotations (2)

```
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:

```
>>> 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.

https://realpython.com/python-lambda/

IIFE

IIFE

You've already seen several examples of immediately invoked function execution:

```
>>> (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.

https://realpython.com/python-lambda/

Arguments

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:

```
>>> (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
```

https://realpython.com/python-lambda/

Decorators (1)

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:

```
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:

```
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'.

https://realpython.com/python-lambda/

Decorators (2)

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:

```
# Defining a decorator

def trace(f):

    def wrap(*args, **kwargs):

        print(f"[TRACE] func: {f.__name__}, args: {args}, kwargs: {kwargs}")

        return f(*args, **kwargs)


    return wrap


# Applying decorator to a function

@trace

def add_two(x):

    return x + 2
```

https://realpython.com/python-lambda/

```
# Calling the decorated function

add_two(3)
```



```
# Applying decorator to a lambda
```

Decorators (3)

[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():

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:

[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]

https://realpython.com/python-lambda/

Decorators (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.

https://realpython.com/python-lambda/

Closure (1)

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:

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

https://realpython.com/python-lambda/

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

Closure (2)

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:

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.

https://realpython.com/python-lambda/

Closure (3)

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

```
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)}")
```

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

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

On line 6, outer_func() returns a lambda and assigns it to 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).

https://realpython.com/python-lambda/

Evaluation time (1)

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:

```
>>> def wrap(n):
...     def f():
...         print(n)
...     return f
...
>>> numbers = 'one', 'two', 'three'
>>> funcs = []
>>> for n in numbers:
...     funcs.append(wrap(n))
...
>>> for f in funcs:
...     f()
```

https://realpython.com/python-lambda/

Evaluation time (2)

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

```
>>> numbers = 'one', 'two', 'three'

>>> funcs = []

>>> for n in numbers:

...     funcs.append(lambda: print(n))

...

>>> for f in funcs:

...     f()

...

three

three

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.

https://realpython.com/python-lambda/

Evaluation time (3)

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

```
>>> numbers = 'one', 'two', 'three'

>>> funcs = []

>>> for n in numbers:

...     funcs.append(lambda n=n: print(n))

...

>>> for f in funcs:

...     f()

...

one

two

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.

https://realpython.com/python-lambda/

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

Testing lambdas (1)

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:

```
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)
```

https://realpython.com/python-lambda/

Doctest (1)

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:

```
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.

https://realpython.com/python-lambda/

Doctest (2)

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

```
$ 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.
```

https://realpython.com/python-lambda/

Doctest (3)

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.

https://realpython.com/python-lambda/

Lambda Expression Abuses (1)

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.

https://realpython.com/python-lambda/

Lambda Expression Abuses (2)

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:

```
>>> 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.

https://realpython.com/python-lambda/

Lambda Expression Abuses (3)

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:

```
>>> (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:

```
>>> (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]
```

https://realpython.com/python-lambda/

Lambda Expression Abuses (4)

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:

```
>>> 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.

https://realpython.com/python-lambda/

Lambda Expression Abuses (5)

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:

```
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
```

https://realpython.com/python-lambda/

Lambda Expression Abuses (6)

Running a tool like flake8, a style guide enforcement tool, will display the following errors for __str__ and honk:

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:

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

brand would be written as follows:

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

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

https://realpython.com/python-lambda/

Lambda Expression Abuses (7)

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.

https://realpython.com/python-lambda/

References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun