Syntax

A lambda form presents syntactic distinctions from a normal function. In particular, a lambda function has the following characteristics:

(a) It can only contain expressions and can't include statements in its body.

(b) It is written as a single line of execution.

(c) It does not support type annotations.

(d) It can be immediately invoked (IIFE).

(a) 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`.

(b) 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:

```
>>>
```

```
>>> (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.

(c) 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()}'
```

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.

(d) 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.

(e) 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
```

(f) What is lambda good for? Why do we need lambda?

Actually, we don't absolutely need lambda; we could get along without it. But there are certain situations where it makes writing code a bit easier, and the written code a bit cleaner. What kind of situations? ... Situations in which

(a) the function is fairly simple, and
(b) it is going to be used only once.

Normally, functions are created for one of two purposes:

(a) to reduce code duplication, or

(b) to modularize code.

> If your application contains duplicate chunks of code in various places, then you can put one copy of that code into a function, give the function a name, and then — using that function name — call it from various places in your code.
> If you have a chunk of code that performs one well-defined operation — but is really long and gnarly and interrupts the otherwise readable flow of your program — then you can pull that long gnarly code out and put it into a function all by itself.

But suppose you need to create a function that is going to be used only once — called from *only one* place in your application. Well, first of all, you don't need to give the function a name. It can be "anonymous". And you can just define it right in the place where you want to use it. That's where lambda is useful.

But, but, but… you say.

> First of all — Why would you want a function that is called only once? That eliminates reason (a) for making a function.
> And the body of a lambda can contain only a single expression. That means that lambdas must be short. So that eliminates reason (b) for making a function.

What possible reason could I have for wanting to create a short, anonymous function?

Well, consider this snippet of code that uses lambda to define the behavior of buttons in a Tkinter GUI interface. (This example is from Mike's tutorial.)