**Functional Programming in Python: When and How to Use It**

**Functional programming** is a programming paradigm in which the primary method of computation is evaluation of functions. In this tutorial, you’ll explore functional programming in Python.

Functional programming typically plays a fairly small role in Python code. But it’s good to be familiar with it. At a minimum, you’ll probably encounter it from time to time when reading code written by others. You may even find situations where it’s advantageous to use Python’s functional programming capabilities in your own code.

**In this tutorial, you’ll learn:**

* What the **functional programming** paradigm entails
* What it means to say that **functions** are **first-class citizens** in Python
* How to define **anonymous functions** with the **lambda** keyword
* How to implement functional code using **map()**, **filter()**, and **reduce()**

**What Is Functional Programming?**

A **pure function** is a function whose output value follows solely from its input values, without any observable side effects. In **functional programming**, a program consists entirely of evaluation of pure functions. Computation proceeds by nested or composed function calls, without changes to state or mutable data.

The functional paradigm is popular because it offers several advantages over other programming paradigms. Functional code is:

* **High level:** You’re describing the result you want rather than explicitly specifying the steps required to get there. Single statements tend to be concise but pack a lot of punch.
* **Transparent:** The behavior of a pure function depends only on its inputs and outputs, without intermediary values. That eliminates the possibility of side effects, which facilitates debugging.
* **Parallelizable:** Routines that don’t cause side effects can more easily run in parallel with one another.

Many programming languages support some degree of functional programming. In some languages, virtually all code follows the functional paradigm. Haskell is one such example. Python, by contrast, does support functional programming but contains features of other programming models as well.

While it’s true that an in-depth description of functional programming is somewhat complex, the goal here isn’t to present a rigorous definition but to show you what you can do by way of functional programming in Python.

Remove ads

**How Well Does Python Support Functional Programming?**

To support functional programming, it’s useful if a function in a given programming language has two abilities:

1. To take another function as an argument
2. To return another function to its caller

Python plays nicely in both these respects. As you’ve learned previously in this series, everything in a Python program is an object. All objects in Python have more or less equal stature, and functions are no exception.

In Python, functions are **first-class citizens**. That means functions have the same characteristics as values like strings and numbers. Anything you would expect to be able to do with a string or number you can do with a function as well.

For example, you can assign a function to a variable. You can then use that variable the same as you would use the function itself:

>>>

1>>> def func():

2... print("I am function func()!")

3...

4

5>>> func()

6I am function func()!

7

8>>> another\_name = func

9>>> another\_name()

10I am function func()!

The assignment another\_name = func on line 8 creates a new reference to func() named another\_name. You can then call the function by either name, func or another\_name, as shown on lines 5 and 9.

You can display a function to the console with print(), include it as an element in a composite data object like a list, or even use it as a dictionary key:

>>>

>>> def func():

... print("I am function func()!")

...

>>> print("cat", func, 42)

cat <function func at 0x7f81b4d29bf8> 42

>>> objects = ["cat", func, 42]

>>> objects[1]

<function func at 0x7f81b4d29bf8>

>>> objects[1]()

I am function func()!

>>> d = {"cat": 1, func: 2, 42: 3}

>>> d[func]

2

In this example, func() appears in all the same contexts as the values "cat" and 42, and the interpreter handles it just fine.

**Note:** What you can or can’t do with any object in Python depends to some extent on context. There are some operations, for example, that work for certain object types but not for others.

You can add two integer objects or concatenate two string objects with the plus operator (+). But the plus operator isn’t defined for function objects.

For present purposes, what matters is that functions in Python satisfy the two criteria beneficial for functional programming listed above. You can pass a function to another function as an argument:

>>>

1>>> def inner():

2... print("I am function inner()!")

3...

4

5>>> def outer(function):

6... function()

7...

8

9>>> outer(inner)

10I am function inner()!

Here’s what’s happening in the above example:

* The call on line 9 passes inner() as an argument to outer().
* Within outer(), Python binds inner() to the function parameter function.
* outer() can then call inner() directly via function.

This is known as **function composition**.

**Technical note:** Python provides a shortcut notation called a **decorator** to facilitate wrapping one function inside another. For more information, check out the Primer on Python Decorators.

When you pass a function to another function, the passed-in function sometimes is referred to as a **callback** because a *call back* to the inner function can modify the outer function’s behavior.

A good example of this is the Python function sorted(). Ordinarily, if you pass a list of string values to sorted(), then it sorts them in lexical order:

>>>

>>> animals = ["ferret", "vole", "dog", "gecko"]

>>> sorted(animals)

['dog', 'ferret', 'gecko', 'vole']

However, sorted() takes an optional key argument that specifies a callback function that can serve as the sorting key. So, for example, you can sort by string length instead:

>>>

>>> animals = ["ferret", "vole", "dog", "gecko"]

>>> sorted(animals, key=len)

['dog', 'vole', 'gecko', 'ferret']

sorted() can also take an optional argument that specifies sorting in reverse order. But you could manage the same thing by defining your own callback function that reverses the sense of len():

>>>

>>> animals = ["ferret", "vole", "dog", "gecko"]

>>> sorted(animals, key=len, reverse=True)

['ferret', 'gecko', 'vole', 'dog']

>>> def reverse\_len(s):

... return -len(s)

...

>>> sorted(animals, key=reverse\_len)

['ferret', 'gecko', 'vole', 'dog']

You can check out How to Use sorted() and sort() in Python for more information on sorting data in Python.

Just as you can pass a function to another function as an argument, a function can also specify another function as its return value:

>>>

1>>> def outer():

2... def inner():

3... print("I am function inner()!")

4...

5... # Function outer() returns function inner()

6... return inner

7...

8

9>>> function = outer()

10>>> function

11<function outer.<locals>.inner at 0x7f18bc85faf0>

12>>> function()

13I am function inner()!

14

15>>> outer()()

16I am function inner()!

Here’s what’s going on in this example:

* **Lines 2 to 3:** outer() defines a local function inner().
* **Line 6:** outer() passes inner() back as its return value.
* **Line 9:** The return value from outer() is assigned to variable function.

Following this, you can call inner() indirectly through function, as shown on line 12. You can also call it indirectly using the return value from outer() without intermediate assignment, as on line 15.

As you can see, Python has the pieces in place to support functional programming nicely. Before you jump into functional code, though, there’s one more concept that will be helpful for you to explore: the lambda expression.

Remove ads

**Defining an Anonymous Function With lambda**

Functional programming is all about calling functions and passing them around, so it naturally involves defining a lot of functions. You can always define a function in the usual way, using the def keyword as you have seen in previous tutorials in this series.

Sometimes, though, it’s convenient to be able to define an **anonymous function** on the fly, without having to give it a name. In Python, you can do this with a **lambda** expression.

**Technical note:** The term *lambda* comes from lambda calculus, a formal system of mathematical logic for expressing computation based on function abstraction and application.

The syntax of a lambda expression is as follows:

lambda <parameter\_list>: <expression>

The following table summarizes the parts of a lambda expression:

| **Component** | **Meaning** |
| --- | --- |
| lambda | The keyword that introduces a lambda expression |
| <parameter\_list> | An optional comma-separated list of parameter names |
| : | Punctuation that separates <parameter\_list> from <expression> |
| <expression> | An expression usually involving the names in <parameter\_list> |

The value of a lambda expression is a callable function, just like a function defined with the def keyword. It takes arguments, as specified by <parameter\_list>, and returns a value, as indicated by <expression>.

Here’s a quick first example:

>>>

1>>> lambda s: s[::-1]

2<function <lambda> at 0x7fef8b452e18>

3

4>>> callable(lambda s: s[::-1])

5True

The statement on line 1 is just the lambda expression by itself. On line 2, Python displays the value of the expression, which you can see is a function.

The built-in Python function callable() returns True if the argument passed to it appears to be callable and False otherwise. Lines 4 and 5 show that the value returned by the lambda expression is in fact callable, as a function should be.

In this case, the parameter list consists of the single parameter s. The subsequent expression s[::-1] is slicing syntax that returns the characters in s in reverse order. So this lambda expression defines a temporary, nameless function that takes a string argument and returns the argument string with the characters reversed.

The object created by a lambda expression is a first-class citizen, just like a standard function or any other object in Python. You can assign it to a variable and then call the function using that name:

>>>

>>> reverse = lambda s: s[::-1]

>>> reverse("I am a string")

'gnirts a ma I'

This is functionally—no pun intended—equivalent to defining reverse() with the def keyword:

>>>

1>>> def reverse(s):

2... return s[::-1]

3...

4>>> reverse("I am a string")

5'gnirts a ma I'

6

7>>> reverse = lambda s: s[::-1]

8>>> reverse("I am a string")

9'gnirts a ma I'

The calls on lines 4 and 8 above behave identically.

However, it’s not necessary to assign a variable to a lambda expression before calling it. You can also call the function defined by a lambda expression directly:

>>>

>>> (lambda s: s[::-1])("I am a string")

'gnirts a ma I'

Here’s another example:

>>>

>>> (lambda x1, x2, x3: (x1 + x2 + x3) / 3)(9, 6, 6)

7.0

>>> (lambda x1, x2, x3: (x1 + x2 + x3) / 3)(1.4, 1.1, 0.5)

1.0

In this case, the parameters are x1, x2, and x3, and the expression is x1 + x2 + x3 / 3. This is an anonymous lambda function to calculate the average of three numbers.

As another example, recall above when you defined a reverse\_len() to serve as a callback function to sorted():

>>>

>>> animals = ["ferret", "vole", "dog", "gecko"]

>>> def reverse\_len(s):

... return -len(s)

...

>>> sorted(animals, key=reverse\_len)

['ferret', 'gecko', 'vole', 'dog']

You could use a lambda function here as well:

>>>

>>> animals = ["ferret", "vole", "dog", "gecko"]

>>> sorted(animals, key=lambda s: -len(s))

['ferret', 'gecko', 'vole', 'dog']

A lambda expression will typically have a parameter list, but it’s not required. You can define a lambda function without parameters. The return value is then not dependent on any input parameters:

>>>

>>> forty\_two\_producer = lambda: 42

>>> forty\_two\_producer()

42

Note that you can only define fairly rudimentary functions with lambda. The return value from a lambda expression can only be one single expression. A lambda expression can’t contain statements like assignment or return, nor can it contain control structures such as for, while, if, else, or def.

You learned in the previous tutorial on defining a Python function that a function defined with def can effectively return multiple values. If a return statement in a function contains several comma-separated values, then Python packs them and returns them as a tuple:

>>>

>>> def func(x):

... return x, x \*\* 2, x \*\* 3

...

>>> func(3)

(3, 9, 27)

This implicit tuple packing doesn’t work with an anonymous lambda function:

>>>

>>> (lambda x: x, x \*\* 2, x \*\* 3)(3)

<stdin>:1: SyntaxWarning: 'tuple' object is not callable; perhaps you missed a comma?

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'x' is not defined

But you can return a tuple from a lambda function. You just have to denote the tuple explicitly with parentheses. You can also return a list or a dictionary from a lambda function:

>>>

>>> (lambda x: (x, x \*\* 2, x \*\* 3))(3)

(3, 9, 27)

>>> (lambda x: [x, x \*\* 2, x \*\* 3])(3)

[3, 9, 27]

>>> (lambda x: {1: x, 2: x \*\* 2, 3: x \*\* 3})(3)

{1: 3, 2: 9, 3: 27}

A lambda expression has its own local namespace, so the parameter names don’t conflict with identical names in the global namespace. A lambda expression can access variables in the global namespace, but it can’t modify them.

There’s one final oddity to be aware of. If you find a need to include a lambda expression in a formatted string literal (f-string), then you’ll need to enclose it in explicit parentheses:

>>>

>>> print(f"--- {lambda s: s[::-1]} ---")

File "<stdin>", line 1

(lambda s)

^

SyntaxError: f-string: invalid syntax

>>> print(f"--- {(lambda s: s[::-1])} ---")

--- <function <lambda> at 0x7f97b775fa60> ---

>>> print(f"--- {(lambda s: s[::-1])('I am a string')} ---")

--- gnirts a ma I ---

Now you know how to define an anonymous function with lambda. For further reading on lambda functions, check out How to Use Python Lambda Functions.

Next, it’s time to delve into functional programming in Python. You’ll see how lambda functions are particularly convenient when writing functional code.

Python offers two built-in functions, map() and filter(), that fit the functional programming paradigm. A third, reduce(), is no longer part of the core language but is still available from a module called functools. Each of these three functions takes another function as one of its arguments.

Remove ads

**Applying a Function to an Iterable With map()**

The first function on the docket is **map()**, which is a Python built-in function. With map(), you can apply a function to each element in an iterable in turn, and map() will return an iterator that yields the results. This can allow for some very concise code because a map() statement can often take the place of an explicit loop.

Calling map() With a Single Iterable

The syntax for calling map() on a single iterable looks like this:

map(<f>, <iterable>)

map(<f>, <iterable>) returns in iterator that yields the results of applying function <f> to each element of <iterable>.

Here’s an example. Suppose you’ve defined reverse(), a function that takes a string argument and returns its reverse, using your old friend the [::-1] string slicing mechanism:

>>>

>>> def reverse(s):

... return s[::-1]

...

>>> reverse("I am a string")

'gnirts a ma I'

If you have a list of strings, then you can use map() to apply reverse() to each element of the list:

>>>

>>> animals = ["cat", "dog", "hedgehog", "gecko"]

>>> iterator = map(reverse, animals)

>>> iterator

<map object at 0x7fd3558cbef0>

But remember, map() doesn’t return a list. It returns an iterator called a **map object**. To obtain the values from the iterator, you need to either iterate over it or use list():

>>>

>>> iterator = map(reverse, animals)

>>> for i in iterator:

... print(i)

...

tac

god

gohegdeh

okceg

>>> iterator = map(reverse, animals)

>>> list(iterator)

['tac', 'god', 'gohegdeh', 'okceg']

Iterating over iterator yields the items from the original list animals, with each string reversed by reverse().

In this example, reverse() is a pretty short function, one you might well not need outside of this use with map(). Rather than cluttering up the code with a throwaway function, you could use an anonymous lambda function instead:

>>>

>>> animals = ["cat", "dog", "hedgehog", "gecko"]

>>> iterator = map(lambda s: s[::-1], animals)

>>> list(iterator)

['tac', 'god', 'gohegdeh', 'okceg']

>>> # Combining it all into one line:

>>> list(map(lambda s: s[::-1], ["cat", "dog", "hedgehog", "gecko"]))

['tac', 'god', 'gohegdeh', 'okceg']

If the iterable contains items that aren’t suitable for the specified function, then Python raises an exception:

>>>

>>> list(map(lambda s: s[::-1], ["cat", "dog", 3.14159, "gecko"]))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 1, in <lambda>

TypeError: 'float' object is not subscriptable

In this case, the lambda function expects a string argument, which it tries to slice. The second element in the list, 3.14159, is a float object, which isn’t sliceable. So a TypeError occurs.

Here’s a somewhat more real-world example: In the tutorial section on built-in string methods, you encountered str.join(), which concatenates strings from an iterable, separated by the specified string:

>>>

>>> "+".join(["cat", "dog", "hedgehog", "gecko"])

'cat+dog+hedgehog+gecko'

This works fine if the objects in the list are strings. If they aren’t, then str.join() raises a TypeError exception:

>>>

>>> "+".join([1, 2, 3, 4, 5])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: sequence item 0: expected str instance, int found

One way to remedy this is with a loop. Using a for loop, you can create a new list that contains string representations of the numbers in the original list. Then you can pass the new list to .join():

>>>

>>> strings = []

>>> for i in [1, 2, 3, 4, 5]:

... strings.append(str(i))

...

>>> strings

['1', '2', '3', '4', '5']

>>> "+".join(strings)

'1+2+3+4+5'

However, because map() applies a function to each object of a list in turn, it can often eliminate the need for an explicit loop. In this case, you can use map() to apply str() to the list objects before joining them:

>>>

>>> "+".join(map(str, [1, 2, 3, 4, 5]))

'1+2+3+4+5'

map(str, [1, 2, 3, 4, 5]) returns an iterator that yields the list of string objects ["1", "2", "3", "4", "5"], and you can then successfully pass that list to .join().

Although map() accomplishes the desired effect in the above example, it would be more Pythonic to use a list comprehension to replace the explicit loop in a case like this.

Remove ads

Calling map() With Multiple Iterables

There’s another form of map() that takes more than one iterable argument:

map(<f>, <iterable₁>, <iterable₂>, ..., <iterableₙ>)

map(<f>, <iterable1>, <iterable2>, ..., <iterablen>) applies <f> to the elements in each <iterablei> in parallel and returns an iterator that yields the results.

The number of <iterablei> arguments specified to map() must match the number of arguments that <f> expects. <f> acts on the first item of each <iterablei>, and that result becomes the first item that the return iterator yields. Then <f> acts on the second item in each <iterablei>, and that becomes the second yielded item, and so on.

An example should help clarify:

>>>

>>> def f(a, b, c):

... return a + b + c

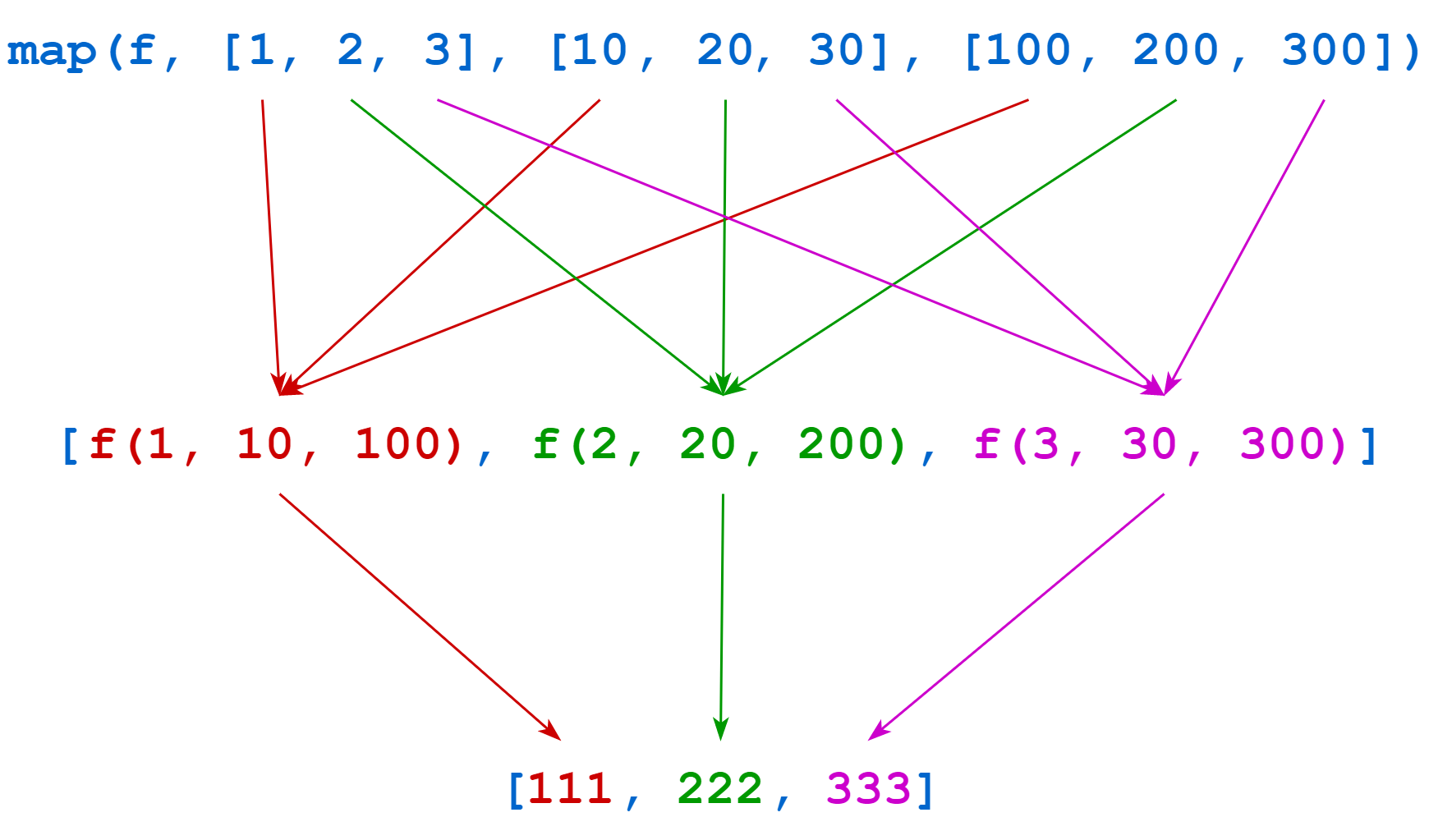
...

>>> list(map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300]))

[111, 222, 333]

In this case, f() takes three arguments. Correspondingly, there are three iterable arguments to map(): the lists [1, 2, 3], [10, 20, 30], and [100, 200, 300].

The first item returned is the result of applying f() to the first element in each list: f(1, 10, 100). The second item returned is f(2, 20, 200), and the third is f(3, 30, 300), as shown in the following diagram:



The return value from map() is an iterator that yields the list [111, 222, 333].

Again in this case, since f() is so short, you could readily replace it with a lambda function instead:

>>>

>>> list(

... map(

... (lambda a, b, c: a + b + c),

... [1, 2, 3],

... [10, 20, 30],

... [100, 200, 300]

... )

... )

This example uses extra parentheses around the lambda function and implicit line continuation. Neither is necessary, but they help make the code easier to read.

**Selecting Elements From an Iterable With filter()**

**filter()** allows you to select or filter items from an iterable based on evaluation of the given function. It’s called as follows:

filter(<f>, <iterable>)

filter(<f>, <iterable>) applies function <f> to each element of <iterable> and returns an iterator that yields all items for which <f> is truthy. Conversely, it filters out all items for which <f> is falsy.

In the following example, greater\_than\_100(x) is truthy if x > 100:

>>>

>>> def greater\_than\_100(x):

... return x > 100

...

>>> list(filter(greater\_than\_100, [1, 111, 2, 222, 3, 333]))

[111, 222, 333]

In this case, greater\_than\_100() is truthy for items 111, 222, and 333, so these items remain, whereas 1, 2, and 3 are discarded. As in previous examples, greater\_than\_100() is a short function, and you could replace it with a lambda expression instead:

>>>

>>> list(filter(lambda x: x > 100, [1, 111, 2, 222, 3, 333]))

[111, 222, 333]

The next example features range(). range(n) produces an iterator that yields the integers from 0 to n - 1. The following example uses filter() to select only the even numbers from the list and filter out the odd numbers:

>>>

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> def is\_even(x):

... return x % 2 == 0

...

>>> list(filter(is\_even, range(10)))

[0, 2, 4, 6, 8]

>>> list(filter(lambda x: x % 2 == 0, range(10)))

[0, 2, 4, 6, 8]

Here’s an example using a built-in string method:

>>>

>>> animals = ["cat", "Cat", "CAT", "dog", "Dog", "DOG", "emu", "Emu", "EMU"]

>>> def all\_caps(s):

... return s.isupper()

...

>>> list(filter(all\_caps, animals))

['CAT', 'DOG', 'EMU']

>>> list(filter(lambda s: s.isupper(), animals))

['CAT', 'DOG', 'EMU']

Remember from the previous tutorial on string methods that s.isupper() returns True if all alphabetic characters in s are uppercase and False otherwise.

**Reducing an Iterable to a Single Value With reduce()**

**reduce()** applies a function to the items in an iterable two at a time, progressively combining them to produce a single result.

reduce() was once a built-in function in Python. Guido van Rossum apparently rather disliked reduce() and advocated for its removal from the language entirely. Here’s what he had to say about it:

So now reduce(). This is actually the one I’ve always hated most, because, apart from a few examples involving + or \*, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what’s actually being fed into that function before I understand what the reduce() is supposed to do. So in my mind, the applicability of reduce() is pretty much limited to associative operators, and in all other cases it’s better to write out the accumulation loop explicitly. (Source)

Guido actually advocated for eliminating all three of reduce(), map(), and filter() from Python. One can only guess at his rationale. As it happens, the previously mentioned **list comprehension** covers the functionality provided by all these functions and much more. You can learn more by reading When to Use a List Comprehension in Python.

As you’ve seen, map() and filter() remain built-in functions in Python. reduce() is no longer a built-in function, but it’s available for import from a standard library module, as you’ll see next.

To use reduce(), you need to import it from a module called functools. This is possible in several ways, but the following is the most straightforward:

from functools import reduce

Following this, the interpreter places reduce() into the global namespace and makes it available for use. The examples you’ll see below assume that this is the case.

Calling reduce() With Two Arguments

The most straightforward reduce() call takes one function and one iterable, as shown below:

reduce(<f>, <iterable>)

reduce(<f>, <iterable>) uses <f>, which must be a function that takes exactly two arguments, to progressively combine the elements in <iterable>. To start, reduce() invokes <f> on the first two elements of <iterable>. That result is then combined with the third element, then that result with the fourth, and so on until the list is exhausted. Then reduce() returns the final result.

Guido was right when he said the most straightforward applications of reduce() are those using associative operators. Let’s start with the plus operator (+):

>>>

>>> def f(x, y):

... return x + y

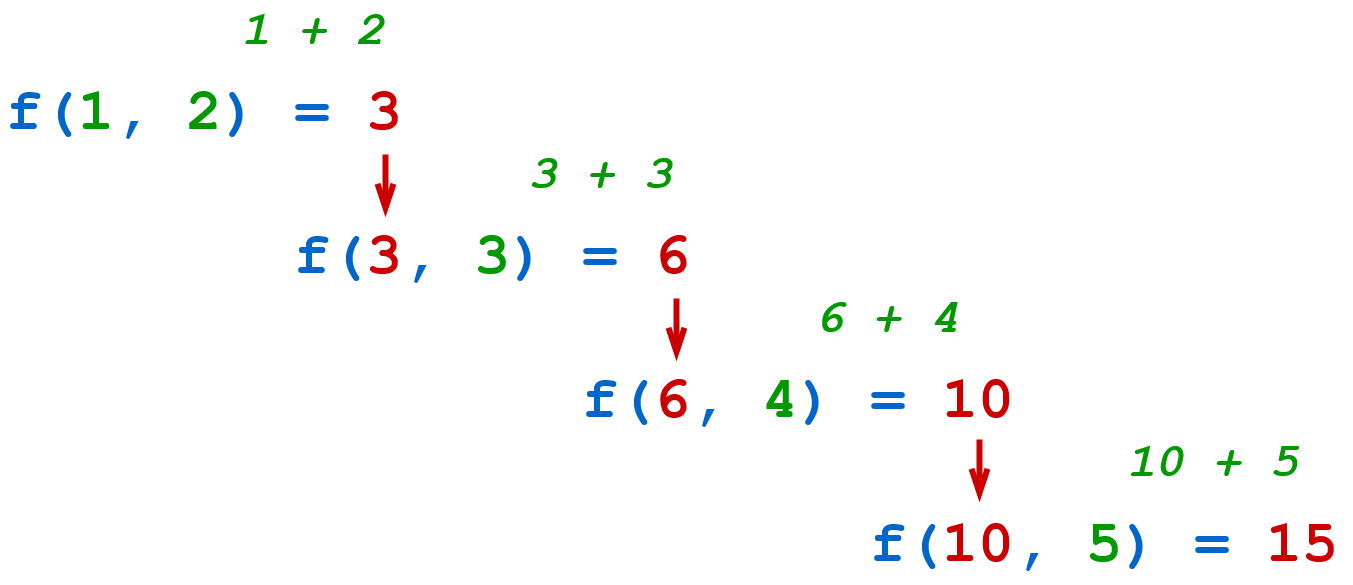
...

>>> from functools import reduce

>>> reduce(f, [1, 2, 3, 4, 5])

15

This call to reduce() produces the result 15 from the list [1, 2, 3, 4, 5] as follows:

reduce(f, [1, 2, 3, 4, 5])

This is a rather roundabout way of summing the numbers in the list! While this works fine, there’s a more direct way. Python’s built-in sum() returns the sum of the numeric values in an iterable:

>>>

>>> sum([1, 2, 3, 4, 5])

15

Remember that the binary plus operator also concatenates strings. So this same example will progressively concatenate the strings in a list as well:

>>>

>>> reduce(f, ["cat", "dog", "hedgehog", "gecko"])

'catdoghedgehoggecko'

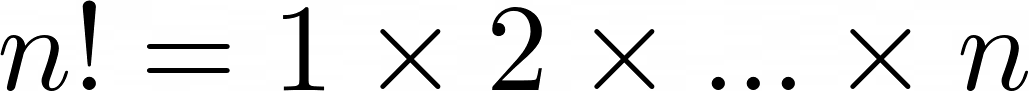
Again, there’s a way to accomplish this that most would consider more typically Pythonic. This is precisely what str.join() does:

>>>

>>> "".join(["cat", "dog", "hedgehog", "gecko"])

'catdoghedgehoggecko'

Now consider an example using the binary multiplication operator (\*). The **factorial** of a positive integer n is defined as follows:



You can implement a factorial function using reduce() and range() as shown below:

>>>

>>> def multiply(x, y):

... return x \* y

...

>>> def factorial(n):

... from functools import reduce

... return reduce(multiply, range(1, n + 1))

...

>>> factorial(4) # 1 \* 2 \* 3 \* 4

24

>>> factorial(6) # 1 \* 2 \* 3 \* 4 \* 5 \* 6

720

Once again, there’s a more straightforward way to do this. You can use factorial() provided by the standard math module:

>>>

>>> from math import factorial

>>> factorial(4)

24

>>> factorial(6)

720

As a final example, suppose you need to find the maximum value in a list. Python provides the built-in function max() to do this, but you could use reduce() as well:

>>>

>>> max([23, 49, 6, 32])

49

>>> def greater(x, y):

... return x if x > y else y

...

>>> from functools import reduce

>>> reduce(greater, [23, 49, 6, 32])

49

Notice that in each example above, the function passed to reduce() is a one-line function. In each case, you could have used a lambda function instead:

>>>

>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])

15

>>> reduce(lambda x, y: x + y, ["foo", "bar", "baz", "quz"])

'foobarbazquz'

>>> def factorial(n):

... from functools import reduce

... return reduce(lambda x, y: x \* y, range(1, n + 1))

...

>>> factorial(4)

24

>>> factorial(6)

720

>>> reduce((lambda x, y: x if x > y else y), [23, 49, 6, 32])

49

This is a convenient way to avoid placing an otherwise unneeded function into the namespace. On the other hand, it may be a little harder for someone reading the code to determine your intent when you use lambda instead of defining a separate function. As is often the case, it’s a balance between readability and convenience.

Remove ads

Calling reduce() With an Initial Value

There’s another way to call reduce() that specifies an initial value for the reduction sequence:

reduce(<f>, <iterable>, <init>)

When present, <init> specifies an initial value for the combination. In the first call to <f>, the arguments are <init> and the first element of <iterable>. That result is then combined with the second element of <iterable>, and so on:

>>>

>>> def f(x, y):

... return x + y

...

>>> from functools import reduce

>>> reduce(f, [1, 2, 3, 4, 5], 100) # (100 + 1 + 2 + 3 + 4 + 5)

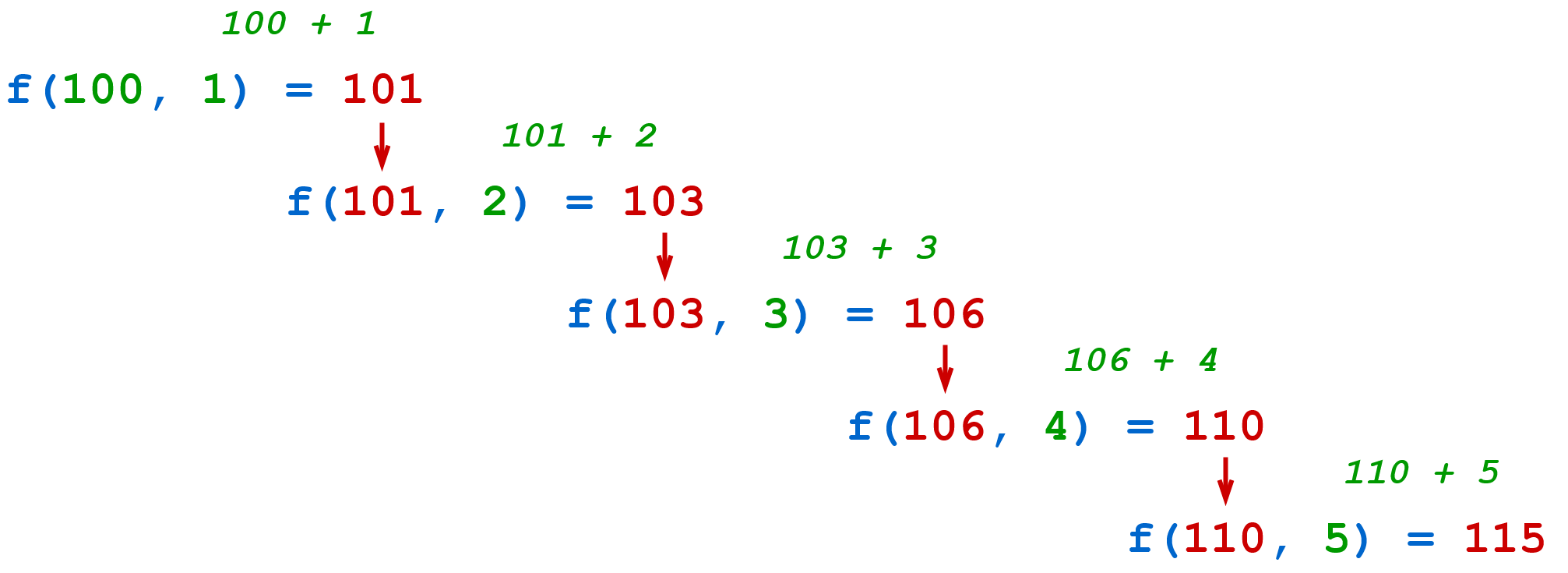
115

>>> # Using lambda:

>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5], 100)

115

Now the sequence of function calls looks like this:



reduce(f, [1, 2, 3, 4, 5], 100)

You could readily achieve the same result without reduce():

>>>

>>> 100 + sum([1, 2, 3, 4, 5])

115

As you’ve seen in the above examples, even in cases where you can accomplish a task using reduce(), it’s often possible to find a more straightforward and Pythonic way to accomplish the same task without it. Maybe it’s not so hard to imagine why reduce() was removed from the core language after all.

That said, reduce() is kind of a remarkable function. The description at the beginning of this section states that reduce() combines elements to produce a *single result*. But that result can be a composite object like a list or a tuple. For that reason, reduce() is a very generalized higher-order function from which many other functions can be implemented.

For example, you can implement map() in terms of reduce():

>>>

>>> numbers = [1, 2, 3, 4, 5]

>>> list(map(str, numbers))

['1', '2', '3', '4', '5']

>>> def custom\_map(function, iterable):

... from functools import reduce

...

... return reduce(

... lambda items, value: items + [function(value)],

... iterable,

... [],

... )

...

>>> list(custom\_map(str, numbers))

['1', '2', '3', '4', '5']

You can implement filter() using reduce() as well:

>>>

>>> numbers = list(range(10))

>>> numbers

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> def is\_even(x):

... return x % 2 == 0

...

>>> list(filter(is\_even, numbers))

[0, 2, 4, 6, 8]

>>> def custom\_filter(function, iterable):

... from functools import reduce

...

... return reduce(

... lambda items, value: items + [value] if function(value) else items,

... iterable,

... []

... )

...

>>> list(custom\_filter(is\_even, numbers))

[0, 2, 4, 6, 8]

In fact, any operation on a sequence of objects can be expressed as a reduction.

**Conclusion**

**Functional programming** is a programming paradigm in which the primary method of computation is evaluation of pure functions. Although Python is not primarily a functional language, it’s good to be familiar with lambda, map(), filter(), and reduce() because they can help you write concise, high-level, parallelizable code. You’ll also see them in code that others have written.

**In this tutorial, you learned:**

* What **functional programming** is
* How **functions** in Python are **first-class citizens**, and how that makes them suitable for functional programming
* How to define a simple **anonymous function** with **lambda**
* How to implement functional code with **map()**, **filter()**, and **reduce()**

With that, you’ve reached the end of this introductory series on the fundamentals of working with Python. Congratulations! You now have a solid foundation for making useful programs in an efficient, Pythonic style.