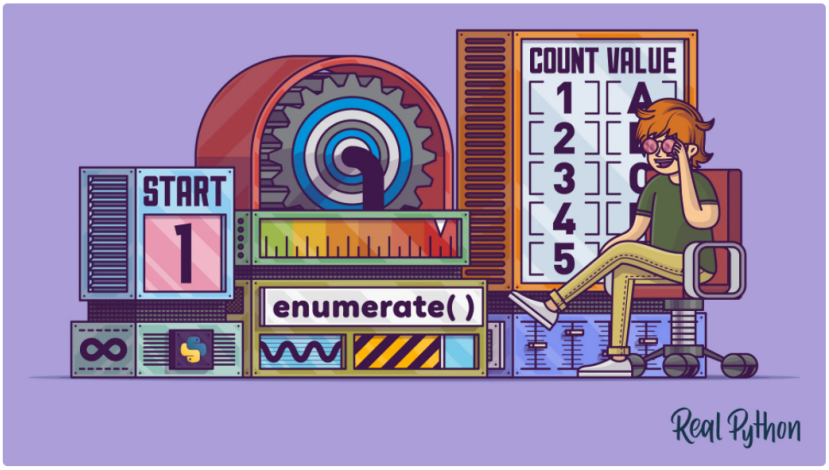

Python enumerate(): Simplify Looping With Counters

by Bryan Weber Nov 18, 2020 10 Comments basics best-practices

Mark as Completed

Table of Contents

In Python, a `for` loop is usually written as a loop over an iterable object. This means you don't need a counting variable to access items in the iterable. Sometimes, though, you do want to have a variable that changes on each loop iteration. Rather than creating and incrementing a variable yourself, you can use Python's `enumerate()` to get a counter and the value from the iterable at the same time!

In this tutorial, you'll see how to:

- Use `enumerate()` to get a counter in a loop
- Apply `enumerate()` to **display item counts**
- Use `enumerate()` with **conditional statements**
- Implement your own **equivalent function** to `enumerate()`
- **Unpack values** returned by `enumerate()`

Let's get started!

Free Download: Get a sample chapter from CPython Internals: Your Guide to the Python 3 Interpreter showing you how to unlock the inner workings of the Python language, compile the Python interpreter from source code, and participate in the development of CPython.

Table of Contents

- Iterating With for Loops in Python
- Using Python's enumerate()
- Practicing With Python enumerate()
- Understanding Python enumerate()
- Unpacking Arguments With enumerate()
- Conclusion

Mark as Completed

Tweet Share Email

Iterating With `for` Loops in Python

A `for` loop in Python uses **collection-based iteration**. This means that Python assigns the next item from an iterable to the loop variable on every iteration, like in this example:

```
Python >>>
>>> values = ["a", "b", "c"]

>>> for value in values:
...     print(value)
...
a
b
c
```

In this example, `values` is a list with three strings, `"a"`, `"b"`, and `"c"`. In Python, lists are one type of iterable object. In the `for` loop, the loop variable is `value`. On each iteration of the loop, `value` is set to the next item from `values`.

Next, you print `value` onto the screen. The advantage of collection-based iteration is that it helps avoid the off-by-one error that is common in other programming languages.

Now imagine that, in addition to the value itself, you want to print the index of the item in the list to the screen on every iteration. One way to approach this task is to create a variable to store the index and update it on each iteration:

```
Python >>>
>>> index = 0

>>> for value in values:
...     print(index, value)
...     index += 1
...
0 a
1 b
2 c
```

In this example, `index` is an integer that keeps track of how far into the list you are. On each iteration of the loop, you print `index` as well as `value`. The last step in the loop is to update the number stored in `index` by one. A common bug occurs when you forget to update `index` on each iteration:

```
Python >>>
>>> index = 0

>>> for value in values:
...     print(index, value)
...
0 a
0 b
0 c
```

In this example, `index` stays at `0` on every iteration because there's no code to update its value at the end of the loop. Particularly for long or complicated loops, this kind of bug is notoriously hard to track down.

Another common way to approach this problem is to use `range()` combined with `len()` to create an index automatically. This way, you don't need to remember to update the index:

```
Python >>>
>>> for index in range(len(values)):
...     value = values[index]
...     print(index, value)
...
0 a
1 b
2 c
```

In this example, `len(values)` returns the length of `values`, which is `3`. Then `range()` creates an iterator running from the default starting value of `0` until it reaches `len(values)` minus one. In this case, `index` becomes your loop variable. In the loop, you set `value` equal to the item in `values` at the current value of `index`. Finally, you print `index` and `value`.

With this example, one common bug that can occur is when you forget to update `value` at the beginning of each iteration. This is similar to the previous bug of forgetting to update the index. This is one reason that this loop isn't considered Pythonic.

This example is also somewhat restricted because `values` has to allow access to its items using integer indices. Iterables that allow this kind of access are called **sequences** in Python.

Technical Detail: According to the Python documentation, an **iterable** is any object that can return its members one at a time. By definition, iterables support the iterator protocol, which specifies how object members are returned when an object is used in an iterator. Python has two commonly used types of iterables:

1. Sequences
2. Generators

Any iterable can be used in a `for` loop, but only sequences can be accessed by integer indices. Trying to access items by index from a generator or an iterator will raise a `TypeError`:

Python >>>

```
>>> enum = enumerate(values)
>>> enum[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'enumerate' object is not subscriptable
```

In this example, you assign the return value of `enumerate()` to `enum`. `enumerate()` is an iterator, so attempting to access its values by index raises a `TypeError`.

Fortunately, Python's `enumerate()` lets you avoid all these problems. It's a **built-in** function, which means that it's been available in every version of Python since it was added in Python 2.3, way back in 2003.

Using Python's `enumerate()`

You can use `enumerate()` in a loop in almost the same way that you use the original iterable object. Instead of putting the iterable directly after `in` in the `for` loop, you put it inside the parentheses of `enumerate()`. You also have to change the loop variable a little bit, as shown in this example:

Python >>>

```
>>> for count, value in enumerate(values):
...     print(count, value)
...
0 a
1 b
2 c
```

When you use `enumerate()`, the function gives you back *two* loop variables:

1. The **count** of the current iteration
2. The **value** of the item at the current iteration

Just like with a normal `for` loop, the loop variables can be named whatever you want them to be named. You use `count` and `value` in this example, but they could be named `i` and `v` or

any other valid Python names.

With `enumerate()`, you don't need to remember to access the item from the iterable, and you don't need to remember to advance the index at the end of the loop. Everything is automatically handled for you by the magic of Python!

> **Technical Details:** The use of the two loop variables, `count` and `value`, separated by a comma is an example of argument unpacking. This powerful Python feature will be discussed further a little later in this article.

Python's `enumerate()` has one additional argument that you can use to control the starting value of the count. By default, the starting value is `0` because Python sequence types are indexed starting with zero. In other words, when you want to retrieve the first element of a list, you use index `0`:

```
Python                                                    >>>

>>> print(values[0])
a
```

You can see in this example that accessing `values` with the index `0` gives the first element, `a`. However, there are many times when you might not want the count from `enumerate()` to start at `0`. For instance, you might want to print a natural counting number as an output for the user. In this case, you can use the `start` argument for `enumerate()` to change the starting count:

```
Python                                                    >>>

>>> for count, value in enumerate(values, start=1):
...     print(count, value)
...
1 a
2 b
3 c
```

In this example, you pass `start=1`, which starts `count` with the value `1` on the first loop iteration. Compare this with the previous examples, in which `start` had the default value of `0`, and see whether you can spot the difference.

Practicing With Python `enumerate()`

You should use `enumerate()` anytime you need to use the count and an item in a loop. Keep in mind that `enumerate()` increments the count by one on every iteration. However, this only slightly limits your flexibility. Since the count is a standard Python integer, you can use it in many ways. In the next few sections, you'll see some uses of `enumerate()`.

Natural Count of Iterable Items

In the previous section, you saw how to use `enumerate()` with `start` to create a natural counting number to print for a user. `enumerate()` is also used like this within the Python codebase. You can see one example in a script that reads reST files and tells the user when there are formatting problems.

> **Note:** reST, also known as reStructured Text, is a standard format for text files that Python uses for documentation. You'll often see reST-formatted strings included as docstrings in Python classes and functions. Scripts that read source code files and tell the user about formatting problems are called **linters** because they look for metaphorical lint in the code.

This example is slightly modified from `rstlint.py`. Don't worry too much about how this function checks for problems. The point is to show a real-world use of `enumerate()`:

```
Python

1 def check_whitespace(lines):
2     """Check for whitespace and line length issues."""
```

```
    for lno, line in enumerate(lines):
        if "\r" in line:
            yield lno+1, "\\r in line"
        if "\t" in line:
            yield lno+1, "OMG TABS!!!1"
        if line[:-1].rstrip(" \t") != line[:-1]:
            yield lno+1, "trailing whitespace"
```

`check_whitespace()` takes one argument, `lines`, which is the lines of the file that should be evaluated. On the third line of `check_whitespace()`, `enumerate()` is used in a loop over `lines`. This returns the line number, abbreviated as `lno`, and the `line`. Since `start` isn't used, `lno` is a zero-based counter of the lines in the file. `check_whitespace()` then makes several checks for out-of-place characters:

1. The carriage return (`\r`)
2. The tab character (`\t`)
3. Any spaces or tabs at the end of the line

When one of these items is present, `check_whitespace()` yields the current line number and a useful message for the user. The count variable `lno` has `1` added to it so that it returns the counting line number rather than a zero-based index. When a user of `rstlint.py` reads the message, they'll know which line to go to and what to fix.

Conditional Statements to Skip Items

Using conditional statements to process items can be a very powerful technique. Sometimes you might need to perform an action on only the very first iteration of a loop, as in this example:

Python >>>

```
>>> users = ["Test User", "Real User 1", "Real User 2"]
>>> for index, user in enumerate(users):
...     if index == 0:
...         print("Extra verbose output for:", user)
...     print(user)
...
Extra verbose output for: Test User
Real User 1
Real User 2
```

In this example, you use a list as a mock database of users. The first user is your testing user, so you want to print extra diagnostic information about that user. Since you've set up your system so that the test user is first, you can use the first index value of the loop to print extra verbose output.

You can also combine mathematical operations with conditions for the count or index. For instance, you might need to return items from an iterable, but only if they have an even index. You can do this by using `enumerate()`:

Python >>>

```
>>> def even_items(iterable):
...     """Return items from ``iterable`` when their index is even."""
...     values = []
...     for index, value in enumerate(iterable, start=1):
...         if not index % 2:
...             values.append(value)
...     return values
...
```

`even_items()` takes one argument, called `iterable`, that should be some type of object that Python can loop over. First, `values` is initialized to be an empty list. Then you create a `for` loop over `iterable` with `enumerate()` and set `start=1`.

Within the `for` loop, you check whether the remainder of dividing `index` by `2` is zero. If it is, then you append the item to `values`. Finally, you return `values`.

You can make the code more Pythonic by using a list comprehension to do the same thing in one line without initializing the empty list:

```
Python >>>

>>> def even_items(iterable):
...     return [v for i, v in enumerate(iterable, start=1) if not i % 2]
...
```

In this example code, `even_items()` uses a list comprehension rather than a `for` loop to extract every item from the list whose index is an even number.

You can verify that `even_items()` works as expected by getting the even-indexed items from a range of integers from `1` to `10`. The result will be `[2, 4, 6, 8, 10]`:

```
Python >>>

>>> seq = list(range(1, 11))

>>> print(seq)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> even_items(seq)
[2, 4, 6, 8, 10]
```

As expected, `even_items()` returns the even-indexed items from `seq`. This isn't the most efficient way to get the even numbers when you're working with integers. However, now that you've verified that `even_items()` works properly, you can get the even-indexed letters of the ASCII alphabet:

```
Python >>>

>>> alphabet = "abcdefghijklmnopqrstuvwxyz"

>>> even_items(alphabet)
['b', 'd', 'f', 'h', 'j', 'l', 'n', 'p', 'r', 't', 'v', 'x', 'z']
```

`alphabet` is a string that has all twenty-six lowercase letters of the ASCII alphabet. Calling `even_items()` and passing `alphabet` returns a list of alternating letters from the alphabet.

Python strings are sequences, which can be used in loops as well as in integer indexing and slicing. So in the case of strings, you can use square brackets to achieve the same functionality as `even_items()` more efficiently:

```
Python >>>

>>> list(alphabet[1::2])
['b', 'd', 'f', 'h', 'j', 'l', 'n', 'p', 'r', 't', 'v', 'x', 'z']
```

Using string slicing here, you give the starting index `1`, which corresponds to the second element. There's no ending index after the first colon, so Python goes to the end of the string. Then you add the second colon followed by a `2` so that Python will take every other element.

However, as you saw earlier, generators and iterators can't be indexed or sliced, so you'll still find `enumerate()` useful. To continue the previous example, you can create a generator function that yields letters of the alphabet on demand:

```
Python >>>

>>> def alphabet():
...     alpha = "abcdefghijklmnopqrstuvwxyz"
...     for a in alpha:
...         yield a

>>> alphabet[1::2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'function' object is not subscriptable
```

```
>>> even_items(alphabet())
['b', 'd', 'f', 'h', 'j', 'l', 'n', 'p', 'r', 't', 'v', 'x', 'z']
```

In this example, you define `alphabet()`, a generator function that yields letters of the alphabet one-by-one when the function is used in a loop. Python functions, whether generators or regular functions, can't be accessed by the square bracket indexing. You try this out on the second line, and it raises a `TypeError`.

You can use generator functions in loops, though, and you do so on the last line by passing `alphabet()` to `even_items()`. You can see that the result is the same as the previous two examples.

Understanding Python `enumerate()`

In the last few sections, you saw examples of when and how to use `enumerate()` to your advantage. Now that you've got a handle on the practical aspects of `enumerate()`, you can learn more about how the function works internally.

To get a better sense of how `enumerate()` works, you can implement your own version with Python. Your version of `enumerate()` has two requirements. It should:

1. Accept an iterable and a starting count value as arguments
2. Send back a tuple with the current count value and the associated item from the iterable

One way to write a function meeting these specifications is given in the Python documentation:

Python >>>

```
>>> def my_enumerate(sequence, start=0):
...     n = start
...     for elem in sequence:
...         yield n, elem
...         n += 1
...
```

`my_enumerate()` takes two arguments, `sequence` and `start`. The default value of `start` is `0`. Inside the function definition, you initialize `n` to be the value of `start` and run a `for` loop over the `sequence`.

For each `elem` in `sequence`, you `yield` control back to the calling location and send back the current values of `n` and `elem`. Finally, you increment `n` to get ready for the next iteration. You can see `my_enumerate()` in action here:

Python >>>

```
>>> seasons = ["Spring", "Summer", "Fall", "Winter"]

>>> my_enumerate(seasons)
<generator object my_enumerate at 0x7f48d7a9ca50>

>>> list(my_enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

>>> list(my_enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

First, you create a list of the four seasons to work with. Next, you show that calling `my_enumerate()` with `seasons` as the `sequence` creates a generator object. This is because you use the `yield` keyword to send values back to the caller.

Finally, you create two lists from `my_enumerate()`, one in which the start value is left as the

default, `0`, and one in which `start` is changed to `1`. In both cases, you end up with a list of tuples in which the first element of each tuple is the count and the second element is the value from `seasons`.

Although you can implement an equivalent function for `enumerate()` in only a few lines of Python code, the actual code for `enumerate()` is written in C. This means that it's super fast and efficient.

Unpacking Arguments With `enumerate()`

When you use `enumerate()` in a `for` loop, you tell Python to use two variables, one for the count and one for the value itself. You're able to do this by using a Python concept called **argument unpacking**.

Argument unpacking is the idea that a tuple can be split into several variables depending on the length of the sequence. For instance, you can unpack a tuple of two elements into two variables:

Python >>>

```
>>> tuple_2 = (10, "a")
>>> first_elem, second_elem = tuple_2
>>> first_elem
10
>>> second_elem
'a'
```

First, you create a tuple with two elements, `10` and `"a"`. Then you unpack that tuple into `first_elem` and `second_elem`, which are each assigned one of the values from the tuple.

When you call `enumerate()` and pass a sequence of values, Python returns an **iterator**. When you ask the iterator for its next value, it yields a tuple with two elements. The first element of the tuple is the count, and the second element is the value from the sequence that you passed:

Python >>>

```
>>> values = ["a", "b"]
>>> enum_instance = enumerate(values)
>>> enum_instance
<enumerate at 0x7fe75d728180>
>>> next(enum_instance)
(0, 'a')
>>> next(enum_instance)
(1, 'b')
>>> next(enum_instance)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

In this example, you create a list called `values` with two elements, `"a"` and `"b"`. Then you pass `values` to `enumerate()` and assign the return value to `enum_instance`. When you print `enum_instance`, you can see that it's an instance of `enumerate()` with a particular memory address.

Then you use Python's built-in `next()` to get the next value from `enum_instance`. The first value that `enum_instance` returns is a tuple with the count `0` and the first element from `values`, which is `"a"`.

Calling `next()` again on `enum_instance` yields another tuple, this time with the count `1` and the second element from `values`, `"b"`. Finally, calling `next()` one more time raises `StopIteration` since there are no more values to be returned from `enum_instance`.

When an iterable is used in a `for` loop, Python automatically calls `next()` at the start of every iteration until `StopIteration` is raised. Python assigns the value it retrieves from the iterable to the loop variable.

If an iterable returns a tuple, then you can use argument unpacking to assign the elements of the tuple to multiple variables. This is what you did earlier in this tutorial by using two loop variables.

loop variables.

Another time you might have seen argument unpacking with a `for` loop is with the built-in `zip()`, which allows you to iterate through two or more sequences at the same time. On each iteration, `zip()` returns a tuple that collects the elements from all the sequences that were passed:

Python >>>

```
>>> first = ["a", "b", "c"]
>>> second = ["d", "e", "f"]
>>> third = ["g", "h", "i"]
>>> for one, two, three in zip(first, second, third):
...     print(one, two, three)
...
a d g
b e h
c f i
```

By using `zip()`, you can iterate through `first`, `second`, and `third` at the same time. In the `for` loop, you assign the element from `first` to `one`, from `second` to `two`, and from `third` to `three`. Then you print the three values.

You can combine `zip()` and `enumerate()` by using **nested** argument unpacking:

Python >>>

```
>>> for count, (one, two, three) in enumerate(zip(first, second, third)):
...     print(count, one, two, three)
...
0 a d g
1 b e h
2 c f i
```

In the `for` loop in this example, you nest `zip()` inside `enumerate()`. This means that each time the `for` loop iterates, `enumerate()` yields a tuple with the first value as the count and the second value as another tuple containing the elements from the arguments to `zip()`. To unpack the nested structure, you need to add parentheses to capture the elements from the nested tuple of elements from `zip()`.

There are other ways to emulate the behavior of `enumerate()` combined with `zip()`. One method uses `itertools.count()`, which returns consecutive integers by default, starting at zero. You can change the previous example to use `itertools.count()`:

Python >>>

```
>>> import itertools
>>> for count, one, two, three in zip(itertools.count(), first, second, third):
...     print(count, one, two, three)
...
0 a d g
1 b e h
2 c f i
```

Using `itertools.count()` in this example allows you to use a single `zip()` call to generate the count as well as the loop variables without nested argument unpacking.

Conclusion

Python's `enumerate()` lets you write Pythonic `for` loops when you need a count and the value from an iterable. The big advantage of `enumerate()` is that it returns a tuple with the counter and value, so you don't have to increment the counter yourself. It also gives you the option to change the starting value for the counter.

In this tutorial, you learned how to:

- Use Python's `enumerate()` in your `for` loops
- Apply `enumerate()` in a few **real-world examples**
- Get values from `enumerate()` using **argument unpacking**
- Implement your own **equivalent function** to `enumerate()`

You also saw `enumerate()` used in some real-world code, including within the CPython code repository. You now have the superpower of simplifying your loops and making your Python code stylish!

Mark as Completed

About **Bryan Weber**

Bryan is a mechanical engineering professor and a core developer of Cantera, the open-source platform for thermodynamics, chemical kinetics, and transport.

» More about Bryan

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Geir Arne

 Jim

 Joanna

 Jacob

Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: basics best-practices

Keep Learning

Related Tutorial Categories: basics best-practices