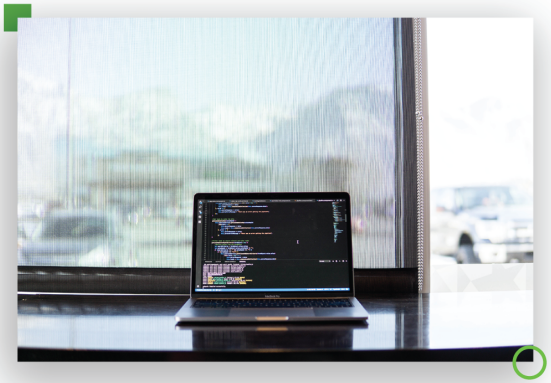

Python List Comprehensions and Generator Expressions

PYTHON

Maria Yakimova
BACKEND ENGINEER / TEAM LEAD

[x for x in range(5)]

(x for x in range(5))

tuple(range(5))

Let's check it

4 Facts About the Lists

First off, a short review on the lists (arrays in other languages).

list is a type of data that can be represented as a collection of elements.

Simple list looks like this – [0, 1, 2, 3, 4, 5]

lists take all possible types of data and combinations of data as their components:

```
 >>> a = 12
>>> b = "this is text"
>>> my_list = [0, b, ['element', 'another element'], (1, 2, 3), a]
>>> print(my_list)
[0, 'this is text', ['element', 'another element'], (1, 2, 3), 12]
```

lists can be indexed. You can get access to any individual element or group of elements using the following syntax:

```
& >>> a = ['red', 'green', 'blue']
>>> print(a[0])
red
```

What is List Comprehension?

Often seen as a part of functional programming in Python, list comprehensions allow you to create lists with a for loop with less code.

Let's look at the following example.

You create a list using a for loop and a range() function.

```
& >>> my_list = []
>>> for x in range(10):
... my_list.append(x * 2)
...
>>> print(my_list)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

And this is how the implementation of the previous example is performed using a list comprehension:

```
>>> comp_list = [x * 2 for x in range(10)]
>>> print(comp_list)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The above example is oversimplified to get the idea of syntax. The same result may be achieved simply using list(range(0, 19, 2)) function. However, you can use a more complex modifier in the first part of comprehension or add a condition that will filter the list. Something like this:

Another available option is to use list comprehension to combine several lists and create a list of lists. At first glance, the syntax seems to be complicated. It may help to think of lists as an outer and inner sequences.

It's time to show the power of list comprehensions when you want to create a list of lists by combining two existing lists.

```
>>> nums = [1, 2, 3, 4, 5]
>>> letters = ['A', 'B', 'C', 'D', 'E']
>>> nums_letters = [[n, l] for n in nums for l in letters]
#the comprehensions list combines two simple lists in a complex list of lists
>>> print(nums_letters)
>>> print(nums_letters)
[[1, 'A'], [1, 'B'], [1, 'C'], [1, 'D'], [1, 'E'], [2, 'A'], [2, 'B'], [2, 'C
>>>
```

Let's try it with text or it's correct to say string object.

```
>>> iter_string = "some text"
>>> comp_list = [x for x in iter_string if x !=" "]
>>> print(comp_list)
['s', 'o', 'm', 'e', 't', 'e', 'x', 't']
```

The comprehensions are not limited to lists. You can create dicts and sets comprehensions as well.

```
>>> print(dict_comp)
{1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10:
```

```
>>> set_comp = {x ** 3 for x in range(10) if x % 2 == 0}
>>> type(set_comp)
<class 'set'>
>>> print(set_comp)
{0, 8, 64, 512, 216}
```

Difference Between Iterable and Iterator

It will be easier to understand the concept of generators if you get the idea of iterables and iterators.

Iterable is a “sequence” of data, you can iterate over using a loop. The easiest visible example of iterable can be a list of integers – [1, 2, 3, 4, 5, 6, 7]. However, it’s possible to iterate over other types of data like strings, dicts, tuples, sets, etc.

Basically, any object that has iter() method can be used as an iterable. You can check it using hasattr()function in the interpreter.

```
>>> hasattr(str, '__iter__')
True
>>> hasattr(bool, '__iter__')
False
```

first iter() method is called on the object to converts it to an iterator object.
next() method is called on the iterator object to get the next element of the sequence.
StopIteration exception is raised when there are no elements left to call.

```
>>> simple_list = [1, 2, 3]
>>> my_iterator = iter(simple_list)
>>> print(my_iterator)
<list_iterator object at 0x7f66b6288630>
>>> next(my_iterator)
1
>>> next(my_iterator)
2
>>> next(my_iterator)
3
>>> next(my_iterator)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Generator Expressions

In Python, generators provide a convenient way to implement the iterator protocol. Generator is an iterable created using a function with a yield statement.

The main feature of generator is evaluating the elements on demand. When you call a normal function with a return statement the function is terminated whenever it encounters a return statement. In a function with a yield statement

```
>>> def my_gen():
... for x in range(5):
... yield x
```

Generator expression allows creating a generator on a fly without a yield keyword. However, it doesn't share the whole power of generator created with a yield function. The syntax and concept is similar to list comprehensions:

```
>>> gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)
>>> for x in gen_exp:
... print(x)
0
4
16
36
64
```

In terms of syntax, the only difference is that you use parentheses instead of square brackets. However, the type of data returned by list comprehensions and generator expressions differs.

```
>>> list_comp = [x ** 2 for x in range(10) if x % 2 == 0]
>>> gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)
>>> print(list_comp)
[0, 4, 16, 36, 64]
>>> print(gen_exp)
<generator object <genexpr> at 0x7f600131c410>
```

can check how much memory is taken by both types using sys.getsizeof() method.

Note: in Python 2 using range() function can't actually reflect the advantage in term of size, as it still keeps the whole list of elements in memory. In Python 3, however, this example is viable as the range() returns a range object.

```
>>> from sys import getsizeof
>>> my_comp = [x * 5 for x in range(1000)]
>>> my_gen = (x * 5 for x in range(1000))
>>> getsizeof(my_comp)
9024
>>> getsizeof(my_gen)
88
```

We can see this difference because while list`creating Python reserves memory for the whole list and calculates it on the spot. In case of generator, we receive only "algorithm"/ "instructions" how to calculate that Python stores. And each time we call for generator, it will only "generate" the next element of the sequence on demand according to "instructions".

On the other hand, generator will be slower, as every time the element of sequence is calculated and yielded, function context/state has to be saved to be picked up next time for generating next value. That "saving and loading function context/state" takes time.

Final Thoughts

offered by language, you're not expected to learn all the language concepts and modules all at once. There are always different ways to solve the same task. Take it as one more tool to get the job done.

TAGS PYTHON

SHARE

yukunix • 3 years ago

Great gradual level of teaching, and also encouraging last paragraph.

3 ^ | v • Reply • Share ›

maguilar • 2 years ago

On

">>> print(comp_list)

[4, 16, 36]

"

There should be a 0 at beginning of resulting list.

1 ^ | v • Reply • Share ›

Mirza • 3 years ago

Great article !

1 ^ | v • Reply • Share ›

Matt Thompson • 3 years ago

Very good article and I particularly like the words of encouragement in the last paragraph.

1 ^ | v • Reply • Share ›

Mahesh Nandgaonkar • a day ago

Very helpful

Thank you....

^ | v • Reply • Share ›

maxfaber • 3 months ago

Very useful. Exactly what I was looking for. Thanks

^ | v • Reply • Share ›

SA • a year ago

dazdazan ➚ SA • a year ago

ss = ['a'+str(x) for x in range(1, 10000)]
print(ss)

Reply • Share ›

Fenouil ➚ SA • a year ago • edited

Like that :

```
[f"{__doc__.__doc__[~(__doc__ is __doc__)<<(__doc__ is __doc__)<<(__doc__ is __doc__)]}{
range(int((-(__doc__ is __doc__)).__doc__[(__doc__ is __doc__)<<(__doc__ is __doc__)<<((
<<(__doc__ is __doc__)<<(__doc__ is __doc__))]+(+(__doc__ is __doc__)).__doc__[-(__doc__
__doc__)<<(__doc__ is __doc__)]*((__doc__ is __doc__)<<(__doc__ is __doc__)<<(__doc__ is
```

or more seriously : `["a{}".format(n) for n in range(10000)]`
to dig in string formating, see :
https://docs.python.org/3/library/string.html#format-examples

Reply • Share ›

SA • a year ago

hi i need to generate a list like this: ['a1', 'a2','a3','a4','a5','a6','a7','a8','a9','a10','a11','a12',...] with can i generate it without typing manually. could you please advise about this? tnx

Reply • Share ›

Ben Chapman ➚ SA • 6 months ago

Try:

my_list = ['a' + str(x) for x in range(1,10001)]

Reply • Share ›

hoohoo • a year ago

Thanks Maria! I was looking for how to write a generator, and your explanation is crystal clear.

Reply • Share ›

蔡文莉 • 3 years ago • edited

I found the generator can only run for loop once. It's not reusable.
>>> a = (x+1 for x in range(5))
>>> for item in a:
... print(item)

```
4
5
>>> for item in a:
... print(item)
...
>>>
```

Reply • Share ›

Frederik Højlund ➚ 蔡文莉 • 7 months ago

Unless you use a list: a = [x+1 for x in range(5)]

Reply • Share ›

YOU MAY ALSO LIKE

Insights, case studies, success stories and professional discoveries

PYTHON, TOOLS VIEWED: 11412

Debugging Python Applications with pdb

Debugging isn't a new trick – most developers actively use it in their work. Of course, everyone has their own approach to debugging, but I've seen too many specialists try to spot bugs using basic things…

PYTHON, TOOLS VIEWED: 96863

Python Asyncio Tutorial. Asynchronous Programming in Python

If for some reason you or your team of Python developers have decided to discover the asynchronous part of Python, welcome to our “Asy…

PYTHON, TOOLS VIEWED: 273871

What is Docker and How to Use it With Python (Tutorial)

Latest articles right in your inbox

Tell about yourself, please

Email address

Your email address

Name

Your name

Position title

Your position title

What are you interested in?

- [ ] Engineering
- [ ] Project Management
- [ ] Business Insights
- [ ] Design and UX/UI

By clicking “SUBSCRIBE” you consent to the processing of your data by Django Stars company for marketing purposes, including sending emails. For details, check our Privacy Policy.

SUBSCRIBE

info@djangostars.com