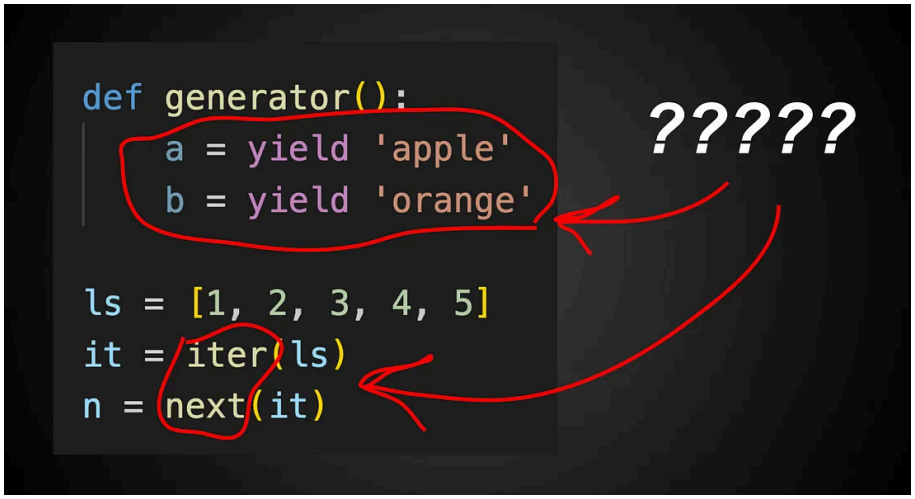
Your subscription payment failed. Update payment method

Member-only story

10 Python Iteration Things I Regret Not Knowing Earlier

Liu Zuo Lin · Following
Published in Level Up Coding · 8 min read · Aug 9, 2024

908 4



1) enumerate() & zip()

Note — *enumerate()* and *zip()* are built-in functions. We don't need to import anything.

The *enumerate()* function allows us to generate both the index and value concurrently while iterating through a list (or tuple). Which is useful if we need to index for whatever reason.

Open in app ↗

Medium Search Write

```
fruits = ['apple', 'orange', 'pear']

for index, fruit in enumerate(fruits):
    print(f'{index=} {fruit=}')

# index=0 fruit='apple'
# index=1 fruit='orange'
# index=2 fruit='pear'
```

The *zip()* function allows us to iterate through 2 or more iterables (eg lists/tuples) concurrently. We simply need to pass our iterables into the *zip()* function.

```
fruits = ['apple', 'orange', 'pear']
prices = [4, 5, 6]
shops = ['A', 'B', 'C']

for f,p,s in zip(fruits, prices, shops):
    print(f, p, s)

# apple 4 A
# orange 5 B
# pear 6 C
```

Do note that the *zip()* function stops generating values if we reach the end of *just one* of the iterables. For instance, in the example below, we are limited by *fruits* which contains only 2 elements.

```
fruits = ['apple', 'orange']
prices = [4, 5, 6, 7, 8, 9]
shops = ['A', 'B', 'C', 'D']

for f,p,s in zip(fruits, prices, shops):
    print(f, p, s)

# apple 4 A
# orange 5 B
```

2) The 'break' keyword

The *break* keyword is used inside a loop — either a for loop or while loop.

When executed, the *break* keyword immediately stops the entire loop.

In the following example, *break* executes in our 2nd iteration where word=='boy'. As such, our for loop stops immediately, and does not go on to 'cat' and 'dog'

```
words = ['apple', 'boy', 'cat', 'dog']
for word in words:
    print(word)

    if word == 'boy':
        break

# apple
# boy
```

The *break* keyword words for while loops too. In the following example, we *break* when num reaches 3, which allows us to get out of the while loop.

```
num = 1
while True:
    print(num, end=' ')

    if num == 3:
        break

    num += 1

# 1 2 3
```

3) The 'continue' keyword

Like *break,* the *continue* keyword is used inside either a for loop or while loop.

When executed, the *continue* keyword skips only the current iteration, and moves on immediately to the next iteration. The *continue* keyword does not stop the entire loop.

In the example below, *continue* is executed if word is either 'boy' or 'cat'. When this is the case, the specific iteration is skipped. As such, only 'apple' and 'dog' are printed

```
words = ['apple', 'boy', 'cat', 'dog']

for word in words:
    if word in ['boy', 'cat']:
        continue

    print(word)

# apple
# dog
```

4) for-else and while-else loops

Yes, while unintuitive, there are such things as *for-else* and *while-else* loops in Python.

In both *for-else* and *while-else* loops, the *else* block runs if the *break* keyword does not execute during the loop.

If *break* doesn't execute during the loop, the *else* block will run:

```
for i in range(3):
    pass
else:
    print('this prints')
```

If *break* executes during the loop, the *else* block will not run:

```
for i in range(3):
    break
else:
    print('not printed')
```

This is useful is cases where we want to check if *break* executes during the loop without having to create an additional variable for this.

Quick Pause

I recently wrote a book — **101 Things I Never Knew About Python**

Check it out here if you wish to support me as a writer!

Link: https://payhip.com/b/vywcf

5) itertools.product

Here's a normal nested for loop (for loop within a for loop)

```
fruits = ['apple', 'orange']
prices = [4, 5]
shops = ['A', 'B']

for fruit in fruits:
    for price in prices:
        for shop in shops:
            print(fruit, price, shop)

# apple 4 A
# apple 4 B
# apple 5 A
# apple 5 B
# orange 4 A
# orange 4 B
# orange 5 A
# orange 5 B
```

If we don't wish to write so many nested for loops, we can use *itertools.product* to help us. In short, *itertools.product* simulates a multi-nested for loop without us having to write the multi-nested for loops.

```
fruits = ['apple', 'orange']
prices = [4, 5]
shops = ['A', 'B']

from itertools import product

for f, p, s in product(fruits, prices, shops):
    print(f, p, s)

# apple 4 A
# apple 4 B
# apple 5 A
# apple 5 B
# orange 4 A
# orange 4 B
# orange 5 A
# orange 5 B
```

itertools.product can handle as many nested for loops as we want it to (though whether your algorithm is an efficient one is another question altogether)

```
for a in A:
    for b in B:
        for c in C:
            for d in D:
                for e in E:
                    print(a, b, c, d, e)

# this is the same as:

from itertools import product

for a,b,c,d,e in product(A,B,C,D,E):
    print(a, b, c, d, e)
```

Note — *itertools* is part of the Python Standard Library, which means we do not need to install anything. We simply need to import it.

6) itertools.pairwise (and other functions)

Let's say we have a bunch of numbers:

```
ls = [1, 2, 3, 4, 5]
```

And in this list, we wish to compare each adjacent pair of numbers — 1 and 2, 2 and 3, 3 and 4, 4 and 5 (for whatever reason)

To do this, we could write a for loop using range(len(ls)-1) and do index manipulation:

```
ls = [1, 2, 3, 4, 5]

for i in range(len(ls)-1):
    l = ls[i]
    r = ls[i+1]
    print(l, r)

# 1 2
# 2 3
# 3 4
# 4 5
```

If we don't wish to deal with this logic (or if we just want to make our code a lot more readable at first glance), we can choose to use *itertools.pairwise* instead.

The *itertools.pairwise* function generates every adjacent pair of elements in our list.

```python
ls = [1, 2, 3, 4, 5]

from itertools import pairwise

for l,r in pairwise(ls):
    print(l, r)

# 1 2
# 2 3
# 3 4
# 4 5
```

Aside from this, the *itertools* module contains many other useful iteration functions.

```python
import itertools

# repeats 1,2,3,1,2,3,... indefinitely
for n in itertools.cycle([1, 2, 3]):
    print(n)

ls = [1, 2, 3, 4, 5]

# lists all permutations of length 2
for perm in itertools.permutations(ls, 2):
    print(perm)

# lists all combinations of length 2
for comb in itertools.combinations(ls, 2):
    print(comb)
```

7) Generator functions

Generator functions are functions that use the *yield* keyword to output stuff.

Here, we have a simple trivial generator function *test()*

```
def test():
    yield 'apple'
    yield 'orange'
    yield 'pear'

print(test())
# <generator object test at 0x100b73740>
```

^ here, if we call *test()* like we would with normal functions, we won't get the *apple, orange and pear* — instead, we print some generator object.

This is because generator functions return a generator object.

And generator objects are meant to be iterated across. The simplest way would be to either 1) use a for loop or 2) wrap our generator object with list()

```
def test():
    yield 'apple'
    yield 'orange'
    yield 'pear'

for fruit in test():
    print(fruit, end=' ')
# apple orange pear

print(list(test()))
# ['apple', 'orange', 'pear']
```

8) Why we use generator functions

We use generator functions if we want to output a value the moment we have it. This is especially useful if we have a large amount of data and don't wish to load all the data into memory (which could lead to a out-of-memory error)

Here's a simple comparison example:

Let's assume the *dostuff()* function does some sort of transformation to *item.*

```
# normal function
def transform(ls):
    out = []
    for item in ls:
        out.append(dostuff(item))
    return out

# generator function
def transform(ls):
    for item in ls:
        yield dostuff(item)
```

In our normal function, we first *finish computing all transformed items,* and store it in out before returning it. So if we have an input list containing 1 million elements, we:

- store all *dostuff(item)* in our output list (stored in memory)
- wait for all 1 million computations to finish before returning it
- which might lead to out-of-memory errors and ineffeciency

In our generator function, we *yield* (output) a value the moment it is ready to be output.

- we don't store all 1 million entries in memory
- we don't wait for all 1 million computations to finish before returning anything

As such, I often use generators in place of normal functions when I want to maximize efficiency and to avoid out-of-memory errors.

9) iter() and next()

Here's a simple list, which we use a for loop to iterate over.

```
ls = ['apple', 'orange', 'pear']

for i in ls:
    print(i)

# apple
# orange
# pear
```

Internally, the for loop actually uses an *iterator object*.

To get the iterator object of our list, we use the *iter()* function.

```
ls = ['apple', 'orange', 'pear']

iterator_obj = iter(ls)

print(iterator_obj)
# <list_iterator object at 0x104db4430>
```

Our iterator object contains a sequence of items that we can access by iterating through it. To do this, we can use the *next()* function to get *only the next item.*

```
ls = ['apple', 'orange', 'pear']

iterator_obj = iter(ls)

print(next(iterator_obj))   # apple
print(next(iterator_obj))   # orange
print(next(iterator_obj))   # pear
```

Note — every time we call *next()* on an iterator object, an internal pointer inside the iterator moves to the next item. So if we call *next()* on the same iterator object, the following item will be returned.

As such, we can simply use *next()* again and again until the iterator object runs out of stuff to return. And when this happens, it raises the StopIteration exception.

```
ls = ['apple', 'orange', 'pear']

iterator_obj = iter(ls)

print(next(iterator_obj))   # apple
print(next(iterator_obj))   # orange
print(next(iterator_obj))   # pear

print(next(iterator_obj))
# StopIteration error
```

We can choose to use this method over for loops and while loops if we want more control over our iteration.

10) Sending data to our generator function

Let's first write a simple generator function.

```
def test():
    a = yield 'apple'
    print('a is', a, end='; ')
    b = yield 'pineapple'
    print('b is', b, end='; ')
    c = yield 'pear'
```

Notice that instead of a simple *yield ABC* statement, now we have *a = yield 'apple'*

The line *a = yield 'apple'* does a couple of things:

- it yields *apple* as per normal
- it then checks if we send anything to the generator using *.send()*
- if we do not send anything, *a* takes the value None
- if we send something, *a* takes the value that we send

Let's first iterate through our generator function without using *.send()*

```
def test():
    a = yield 'apple'
    print('a is', a, end='; ')
    b = yield 'pineapple'
    print('b is', b, end='; ')
    c = yield 'pear'

gen = test()
print(next(gen))    # apple
print(next(gen))    # a is None; pineapple
print(next(gen))    # b is None; pear
```

- here, notice that our function yields *apple, pineapple and pear* as usual
- since we don't use *.send(), a* and *b* are assigned to None

Next, let's actually use *.send()* to send a value to our generator.

When using *.send(),* note that we actually also yield something. As such, we replace the second *next(gen)* with the line *gen.send(7)*

```
def test():
    a = yield 'apple'
    print('a is', a, end='; ')
    b = yield 'pineapple'
    print('b is', b, end='; ')
    c = yield 'pear'

gen = test()
print(next(gen))    # apple
print(gen.send(7))  # a is 7; pineapple
print(next(gen))    # b is None; pear
```

- *a = yield 'apple'* yields *apple* first
- it then checks if we send anything to the generator using *.send()*
- we do send the value 7, so *a* is assigned to the value 7 (hence printing *a is 7*)
- *b = yield 'pineapple'* yields *pineapple* next
- it then checks if we send anything to the generator using *.send()*
- we do not send anything now, so *b* is assigned to None (printing *b is None*)
- *c = yield 'pear'* yields *pear*

Let's take a look at a less trivial example — a simple counting generator:

```
def increment(num=0):
    while True:
        value_from_send = yield num
        if value_from_send is None:
            num += 1
        else:
            num = value_from_send
```

If we don't *.send()* anything to our generator, it just returns 0, 1, 2, 3 ...

```python
def increment(num=0):
    while True:
        value_from_send = yield num
        if value_from_send is None:
            num += 1
        else:
            num = value_from_send

gen = increment()
print(next(gen))    # 0
print(next(gen))    # 1
print(next(gen))    # 2
print(next(gen))    # 3
print(next(gen))    # 4
```

However, if we *.send()* a value to our generator, it starts counting from that value:

```python
def increment(num=0):
    while True:
        value_from_send = yield num
        if value_from_send is None:
            num += 1
        else:
            num = value_from_send

gen = increment()
print(next(gen))    # 0
print(next(gen))    # 1
print(gen.send(10)) # 10
print(next(gen))    # 11
print(next(gen))    # 12
```

As such, if we ever need absolute control over our iteration, we might need to consider using *iter(), next()* and *.send()*

Conclusion

Hopefully this was clear and easy to understand

If You Wish To Support Me As A Creator

- Join my Substack newsletter at https://zlliu.substack.com/ — I send weekly emails relating to Python
- Buy my book — **101 Things I Never Knew About Python** at https://payhip.com/b/vywcf
- *Clap 50 times for this story*
- *Leave a comment telling me your thoughts*
- *Highlight your favourite part of the story*

Thank you! These tiny actions go a long way, and I really appreciate it!

YouTube: https://www.youtube.com/@zlliu246

LinkedIn: https://www.linkedin.com/in/zlliu/

Python Programming Coding Tech Technology

More from the list: "Reading list"
Curated by @reenum

Suhith Illesinghe
Agile is dead
6d ago

Allie Pasc... in UX Collecti...
"Winning" by design: Deceptive UX patterns...
6d ago

Oliver Bennet
Mastering Bash: Essential Commands for Everyda...
Oct 23

View list

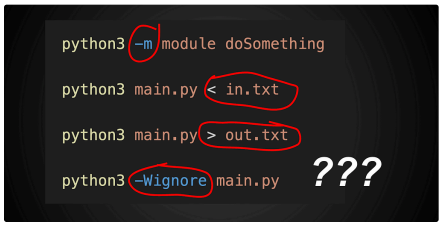
Written by Liu Zuo Lin

Following

96K Followers · Writer for Level Up Coding

My Substack Newsletter: https://zlliu.substack.com | 101 Things I Never Knew
About Python: https://payhip.com/b/vywcf

More from Liu Zuo Lin and Level Up Coding



Liu Zuo Lin in Level Up Coding

13 Python Command Line Things I Regret Not Knowing Earlier

Read free...

Oct 19 667 12

Bryson Meiling in Level Up Coding

Stop making your python projects like it was 15 years ago...

I have a few things I've seen across companies and projects that I've seen...

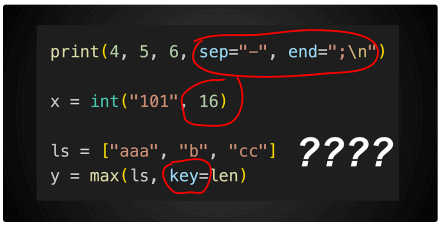
Sep 28 4.3K 45

Hayk Simonyan in Level Up Coding

STOP using Docker Desktop: Faster Alternative Nobody Uses

Ditch Docker Desktop and try this faster, lighter tool that will make your life easier!

Oct 8 2.3K 41



Liu Zuo Lin in Level Up Coding

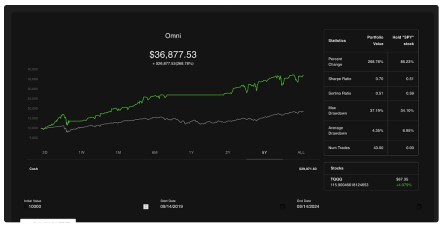
12 Python Built-in Function Things I Regret Not Knowing Earlier

Friend Link Here

Oct 12 873 5

See all from Liu Zuo Lin | See all from Level Up Coding

Recommended from Medium



Austin Starks in DataDrivenInvestor

I used OpenAI's o1 model to develop a trading strategy. It is...

It literally took one try. I was shocked.

Sep 15 5K 132

Nidhi Jain in Code Like A Girl

7 Productivity Hacks I Stole From a Principal Software Engineer

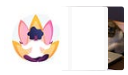
Golden tips and tricks that can make you unstoppable

Oct 15 3.2K 54

Lists

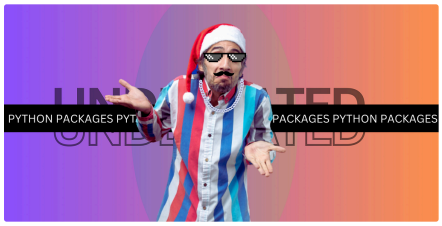
Coding & Development
11 stories · 878 saves

General Coding Knowledge
20 stories · 1688 saves



Stories to Help You Grow as a Software Developer
19 stories · 1447 saves

ChatGPT prompts
50 stories · 2157 saves



Abhay Parashar in The Pythoneers

23 Game-Changing Python Packages You Are Missing Out On

Make Your Life Easy By Exploring These Hidden Gems

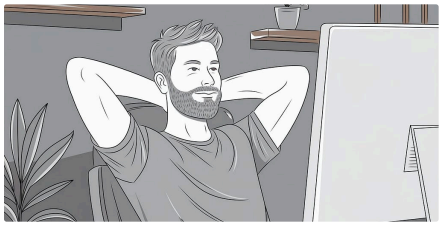
Oct 21 1.3K 9

Bryson Meiling in Level Up Coding

Stop making your python projects like it was 15 years ago...

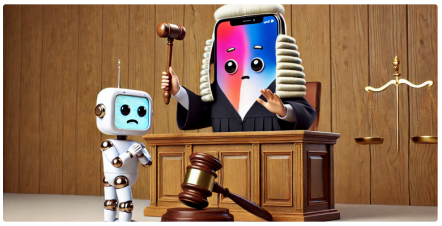
I have a few things I've seen across companies and projects that I've seen...

Sep 28 4.3K 45



Abdur Rahman in Stackademic

20 Python Scripts To Automate Your Daily Tasks



Ignacio de Gregorio

Apple Speaks the Truth About AI. It's Not Good.

A must-have collection for every developer

Oct 6 1.2K 9

Are We Being Lied To?

6d ago 3.7K 119

See more recommendations