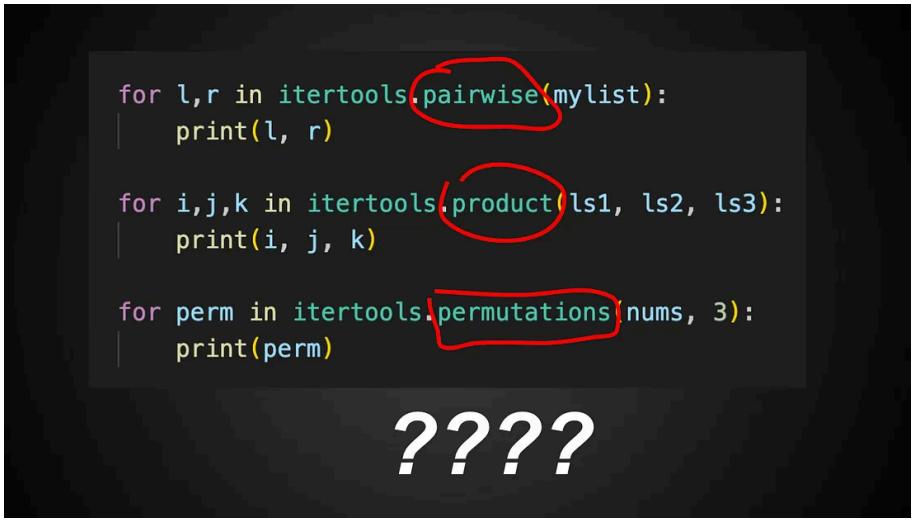
Your subscription payment failed. Update payment method

Member-only story

9 Python Itertools Things I Regret Not Knowing Earlier

Liu Zuo Lin · Following
Published in Level Up Coding · 5 min read · 4 days ago

258 2



Read free: https://zlliu.medium.com/9-python-itertools-things-i-regret-not-knowing-earlier-d504f6121773?sk=04dc80eddd5948fa8ee643844f7a339b

1) itertools.pairwise

itertools.pairwise allows us to generate every adjacent pair of items in some iterable.

Have you ever needed to compare every pair in some list — and wrote this for loop logic to achieve this?

```
fruits = ["apple", "orange", "pear", "pineapple"]

for i in range(len(fruits)-1):
    left = fruits[i]
    right = fruits[i+1]
    print(left, right)

# apple orange
# orange pear
# pear pineapple
```

itertools.pairwise essentially does this for us, and we no longer need to implement this logic ourselves. And our code looks a lot neater and more readable now.

```
fruits = ["apple", "orange", "pear", "pineapple"]

from itertools import pairwise
for left, right in pairwise(fruits):
    print(left, right)

# apple orange
# orange pear
# pear pineapple
```

2) itertools.product

itertools.product allows us to condense a multi-nested for loop into one single loop.

The normal way of doing this using a for loop:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

for a in list1:
    for b in list2:
        for c in list3:
            print(f"{a}{b}{c}", end=" ")
```

^ here, we have 3 nested for loops, which might look a tad messy.

Doing this using *itertools.product:*

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

from itertools import product

for a, b, c in product(list1, list2, list3):
    print(f"{a}{b}{c}", end=" ")
```

^ this does the exact same thing as above, but in a much more elegant and human-readable way.

3) permutations & combinations

itertools.permutations allows us to generate all permutations of some list or iterable:

```
numbers = [1, 2, 3]

from itertools import permutations

for perm in permutations(numbers):
    print(perm, end=" ")

# (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
```

We can also pass in our desired permutation length — for instance, generating all permutations of length 2 (instead of the default 3)

```
numbers = [1, 2, 3]

from itertools import permutations

for perm in permutations(numbers, 2):
    print(perm, end=" ")

# (1, 2) (1, 3) (2, 1) (2, 3) (3, 1) (3, 2)
```

itertools.combinations allows us to generate all combinations of some list or iterable.

For context, a combination is a permutation where order does not matter. Two permutations (1, 3, 2) and (1, 2, 3) are considered different as the order differs.

Conversely, two combinations (1, 3, 2) and (1, 2, 3) are considered the exact same as order does not matter in a combination.

Open in app ↗

Medium Search Write

```
numbers = [1, 2, 3, 4]

from itertools import combinations

for comb in combinations(numbers, 3):
    print(comb, end=" ")

# (1, 2, 3) (1, 2, 4) (1, 3, 4) (2, 3, 4)
```

Quick Pause

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

Here I've compiled 101 cool Python tricks and facts that many (if not most) do not know about. Check it out if you wish to level up your Python knowledge!

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

4) Combinations with replacement

In *itertools.combinations,* each element can only be used once.

In *itertools.combinations_with_replacement,* each element can be reused repeatedly.

Using *itertools.combinations* to generate all combinations of length 2 from [1, 2, 3]:

```
numbers = [1, 2, 3]

from itertools import combinations

for comb in combinations(numbers, 2):
    print(comb, end=" ")

# (1, 2) (1, 3) (2, 3)
```

Using *itertools.combinations_with_replacement* instead — notice now that each individual element can be used more than once.

```
numbers = [1, 2, 3]

from itertools import combinations_with_replacement

for comb in combinations_with_replacement(numbers, 2):
    print(comb, end=" ")

# (1, 1) (1, 2) (1, 3) (2, 2) (2, 3) (3, 3)
```

5) itertools.zip_longest

itertools.zip_longest allows us to iterate through 2 or more iterables (lists, tuples, dicts etc) at the same time, and stopping only after the *longest* iterable is finished.

```
fruits = ["apple", "orange", "pear"]
prices = [1, 2, 3, 4, 5]

from itertools import zip_longest

for fruit, price in zip_longest(fruits, prices):
    print(f"{fruit}={price}", end=" ")

# apple=1 orange=2 pear=3 None=4 None=5
```

- *fruits* has 3 elements, while *prices* has 5
- *itertools.zip_longest* only stops when the longest iterable stops
- in this case, we stop at 5 iterations

- for the missing fruits, None will simply be generated.

Conversely, if we use the default built-in *zip()* function, we stop iterating when the shortest iterable reaches its end.

```
fruits = ["apple", "orange", "pear"]
prices = [1, 2, 3, 4, 5]

for fruit, price in zip(fruits, prices):
    print(f"{fruit}={price}", end=" ")

# apple=1 orange=2 pear=3
```

We can either use *zip()* or *itertools.zip_longest* depending on what our use case is.

6) itertools.cycle

itertools.cycle allows us to repeat a certain sequence of elements forever.

Let's say we want to generate *apple, orange, pear* in this sequence forever. We can do this using a while loop.

```
fruits = ["apple", "orange", "pear"]
index = 0
while True:
    print(fruit, end=" ")
    index += 1
    if index >= len(fruits):
        index = 0

# apple orange pear apple orange pear ...
```

Or instead, we can do it using *itertools.cycle:*

```
from itertools import cycle

fruits = ["apple", "orange", "pear"]
for fruit in cycle(fruits):
    print(fruit, end=" ")

# apple orange pear apple orange pear ...
```

^ this way, our code looks a lot neater and human-readable, and we do not need to manually implement the while loop logic.

7) itertools.groupby

itertools.groupby provides us with functionality to iterate through and at the same time, group together similar elements.

A simple example — same elements are grouped together

```
from itertools import groupby

nums = [1, 1, 1, 1, 2, 2, 3, 3, 3, 1, 1]

for key, group in groupby(nums):
    print(key, list(group))

# 1 [1, 1, 1, 1]
# 2 [2, 2]
# 3 [3, 3, 3]
# 1 [1, 1]
```

Note — 1's that are separate from one another are considered separate groups.

If we pass in the *key* argument, we can customize how *itertools.groupby* groups our elements together.

For instance, let's group numbers together by their *last digit* — to do this, we need to pass a lambda function into the *key* argument.

```python
from itertools import groupby

nums = [1, 11, 41, 42, 5, 55, 65, 75, 23, 23]

for key, group in groupby(nums, key=lambda x:x%10):
    print(key, list(group))

# 1 [1, 11, 41]
# 2 [42]
# 5 [5, 55, 65, 75]
# 3 [23, 23]
```

^ notice that numbers with the same *last digit* are grouped together now.

8) itertools.accumulate

itertools.accumulate provides us with functionality to accumulatively perform some operation on all elements in our iterable.

If we don't provide an operation to *itertools.accumulate,* it defaults to addition.

```python
from itertools import accumulate

nums = [1, 5, 3, 2]

for n in accumulate(nums):
    print(n, end=" ")

# 1 6 9 11
```

Given the list [1, 5, 3, 2]:

- our first element is 1
- our second element is (1 + 5) = 6
- our third element is (1 + 5 + 3) = 9
- our last element is (1 + 5 + 3 + 2) = 11

We can also choose to pass in a custom operation into *itertools.accumulate*

For instance, instead of adding, let's subtract instead:

```
from itertools import accumulate

nums = [1, 5, 3, 2]

for n in accumulate(nums, lambda a,b:a-b):
    print(n, end=" ")

# 1 -4 -7 -9
```

We can also multiple instead of adding:

```
from itertools import accumulate

nums = [1, 5, 3, 2]

for n in accumulate(nums, lambda a,b:a*b):
    print(n, end=" ")

# 1 5 15 30
```

9) itertools.count

itertools.count allows us to create infinite ranges

```
from itertools import count

for n in count(start=1, step=2):
    print(n, end=" ")

# 1 3 5 7 9 11 13 15 ...
```

```python
from itertools import count

for n in count(start=10, step=20):
    print(n, end=" ")

# 10 20 30 40 50 ...
```

^ unlike the built-in *range* function, *itertools.count* can create an infinite range

We can even use negative steps instead:

```python
from itertools import count

for n in count(start=5, step=-5):
    print(n, end=" ")

# 5 0 -5 -10 -15 -20 -25 ...
```

Conclusion

Hopefully this was clear and easy to understand

If You Wish To Support Me As A Creator

- **Buy my book — 101 Things I Never Knew About Python** at https://payhip.com/b/vywcf
- Join my Substack newsletter at https://zlliu.substack.com/ — I send weekly emails newsletters detailing cool Python tricks
- *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!

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

10 I To I

View list

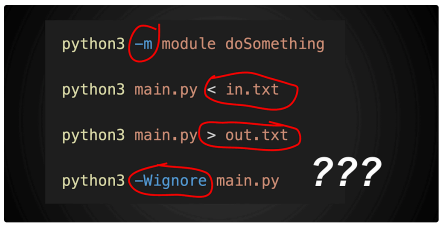
Written by Liu Zuo Lin

96K Followers · Writer for Level Up Coding

Following

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...

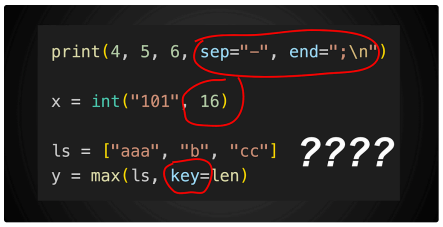
Sep 28 4.3K 45

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

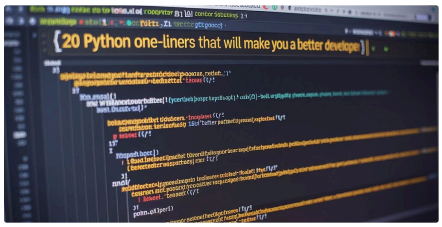
12 Python Built-in Function Things I Regret Not Knowing Earlier

Friend Link Here

Oct 12 873 5

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

Recommended from Medium

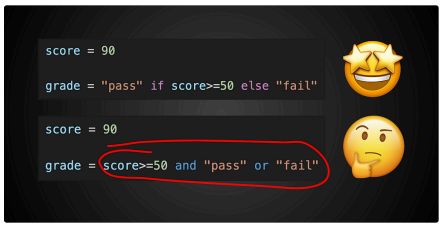


Abdur Rahman in Python in Plain English

20 Python One-Liners That Made Me a Cool Developer 😎

I wish I knew it earlier

5d ago 120 1



Liu Zuo Lin in Level Up Coding

A Whack Alternative To “A if Condition else B” in Python

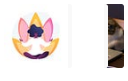
Warning — do not use in production

5d ago 199 4

Lists

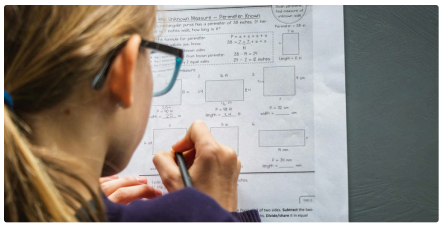
Coding & Development
11 stories · 878 saves

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

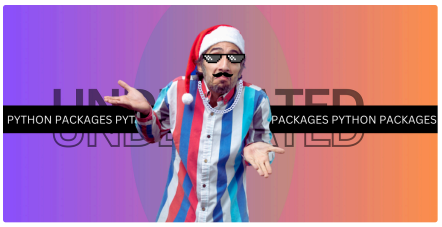


Egor Howell in Towards Data Science

4 Years of Data Science in 8 Minutes

What I have learned in my 4+ year journey of studying data science

5d ago 825 13



Abhay Parashar in The Pythoneers

23 Game-Changing Python Packages You Are Missing Out On

Make Your Life Easy By Exploring These Hidden Gems

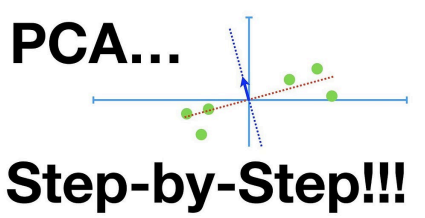
Oct 21 1.3K 9

Karthick Dkk in devsecops-community

The Dangerous Linux Commands You Should Never Run in...

Hey there! If you're managing Linux systems in production, you know things can go from...

Oct 10 641 14



Francesco Franco

Principal Component Analysis with Python (A Deep Dive)

Training a Supervised Machine Learning model—whether it is a traditional one or a...

4d ago 538 9

See more recommendations