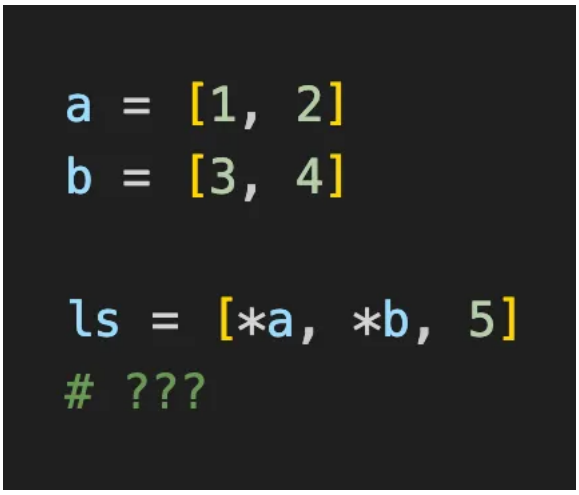
12+ Things I Regret Not Knowing Earlier About Python Lists

Liu Zuo Lin · Following
7 min read · Jul 27, 2024

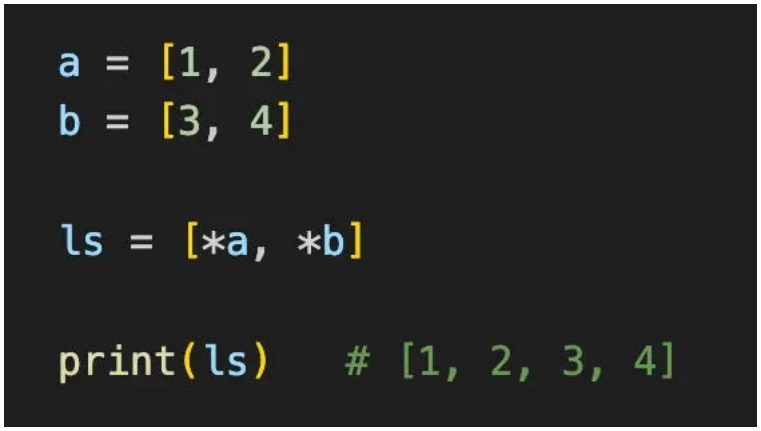


```
a = [1, 2]
b = [3, 4]

ls = [*a, *b, 5]
# ???
```

1) We can use * to combine lists

We can use * to combine lists together.

More specifically, when put in front of a list, the * causes the list to *expand* itself into its container (eg another list)



```
a = [1, 2]
b = [3, 4]

ls = [*a, *b]

print(ls)   # [1, 2, 3, 4]
```

We can combine this with singular elements + choose which lists to expand and which not to.

```
a = [1, 2]
b = [3, 4]
c = [5, 6]

ls = [*a, *b, 100, c]

print(ls)   # [1, 2, 3, 4, 100, [5, 6]]
```

2) We can use * to unpack lists

Let's say we have a list containing some dog info — the first element is its name, the second element is its age, and the rest are its favourite food.

```
dog = ['rocky', 5, 'chicken', 'fish', 'pork']
```

We can use * to *unpack* a section of the list into a variable. For instance, we want the name and age individually, but the favourite food as a list.

```
dog = ['rocky', 5, 'chicken', 'fish', 'pork']

name, age, *fav_food = dog

print(name)     # rocky
print(age)      # 5
print(fav_food) # ['chicken', 'fish', 'pork]
```

Here, *rocky* and 5 are assigned to *name* and *age* as they are the first 2 elements.

However, *fav_food is assigned to *everything else* and catches all unassigned elements as a list.

3) Using * unpack a list as function arguments

Let's say we have a simple function that takes in (a, b, c) and prints them

```
def test(a, b, c):
    print(f'{a=} {b=} {c=}')

test(4, 5, 6)   # a=4 b=5 c=6
```

If we have a list containing 3 elements, we can pass it into this function using the * operator to *unpack* it. The * operator *unpacks* the list's elements into its container.

```
def test(a, b, c):
    print(f'{a=} {b=} {c=}')

numbers = [1, 2, 3]

test(*numbers)  # a=1 b=2 c=3
```

Here, given that numbers = [1, 2, 3], test(*numbers) is the same as test(1, 2, 3)

Note that in this case, test(a, b, c) takes in exactly 3 arguments, so our input list must contain exactly 3 arguments.

4) zip() to iterate through 2 lists concurrently

The built-in zip() function allows us to iterate through 1 or more lists concurrently:

```
fruits = ['apple', 'orange', 'pear']
prices = [4, 5, 6]

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

# apple 4
# orange 5
# pear 6
```

We can iterate through 3 or even more lists concurrently if we wish to:

```
fruits = ['apple', 'orange', 'pear']
prices = [4, 5, 6]
recipes = ['pie', 'juice', 'cake']

for f, p, r in zip(fruits, prices, recipes):
    print(f, p, r)

# apple 4 pie
# orange 5 juice
# pear 6 cake
```

5) enumerate() to generate both index and value during iteration

If we wish to generate both the *index* and *value* of our list element during iteration, we can consider using the built-in enumerate() function.

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

for index, fruit in enumerate(fruits):
    print(index, fruit)

# 0 apple
# 1 orange
# 2 pear
```

Quick Pause

I recently wrote a book **101 Things I Never Knew About Python** detailing Python stuff I didn't learn as early as I could have.

Check it out here: https://payhip.com/b/vywcf

6) .sort() vs sorted()

Both .sort() and sorted() sorts a list, but there is one key difference.

.sort() returns nothing and is done in place — meaning that the original list itself becomes sorted.

```
ls = [1, 4, 2, 3]
new = ls.sort()

print(new)  # None
print(ls)   # [1, 2, 3, 4]
```

sorted() creates and returns a *sorted copy* of our list — meaning that the original list remains as it is and does not get changed. Which can be useful if we want to preserve the original order of things for some reason.

```
ls = [1, 4, 2, 3]
new = sorted(ls)

print(new)  # [1, 2, 3, 4]
print(ls)   # [1, 4, 2, 3]
```

7) .sort() with custom condition

This works for both .sort() and sorted(), but let's just use .sort() as our example. By default, .sort() sorts our numbers in ascending order .

```
ls = [38, 19, 22]
ls.sort()

print(ls)   # [19, 22, 38]
```

If we wish to sort by a custom condition, we can pass in our custom function into the *key* keyword argument. Let's say we want to sort by the *last digit* to get [22, 38, 19]

```
ls = [38, 19, 22]

def condition(num):
    return num % 10

ls.sort(key=condition)

print(ls)   # [22, 38, 19]
```

We can shorten this using an equivalent lambda function if we want to

```
ls = [38, 19, 22]

ls.sort(key=lambda n:n%10)

print(ls)   # [22, 38, 19]
```


This is one basic we which we might go about doing this:

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

new = []
for fruit in fruits:
    new.append(fruit.upper())

print(new)  # ['APPLE', 'ORANGE', 'PEAR']
```

However, we can use list comprehension to do this more elegantly in one line of code. Here, the .upper() transformation is applied *inside* the list comprehension itself.

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

new = [f.upper() for f in fruits]

print(new)  # ['APPLE', 'ORANGE', 'PEAR']
```

We can add a conditional statement to filter fruits that we wish to keep. For instance, let's keep only fruits that have a length of 5 or more:

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

new = [f.upper() for f in fruits if len(f)>=5]

print(new)  # ['APPLE', 'ORANGE']
```

9) Tuples vs Lists

Tuples are essentially immutable lists. Which means that we *cannot* mutate our tuple (add/remove stuff) *after* we create our tuple.

We define a tuple using normal brackets instead of square brackets:

```
ls = [1, 2, 3]
ls.append(4)    # no prob

tup = (1, 2, 3)
tup.append(4)   # error
```

Disadvantages of using tuples over lists:

- because tuples are immutable, we cannot add new elements to our tuple
- because tuples are immutable, we cannot remove elements from our tuple

Advantages of using tuples over lists:

- tuples are immutable, which means they are hashable
- tuples are hashable, which means they can be used as dictionary keys (lists can't)
- tuples are hashable, which means they can be added into a set (lists can't)

10) The .insert(index, element) method

We are probably familiar with the *.append()* method of a list, which simply adds a new element to the *back* of a list.

But did you know that we also have the *.insert()* method, which puts a new element at any index we desire?

Inserting 'AAA' into index 0 — 'AAA' will be at index 0 in our list, and everything else will be pushed back by 1 index.

```
ls = ['apple', 'orange', 'pear']
ls.insert(0, 'AAA')

print(ls)
# ['AAA', 'apple', 'orange', 'pear']
```

Inserting 'AAA' into index 1 — 'AAA' will be at index 1 in our list, and everything else will be pushed back by 1 index.

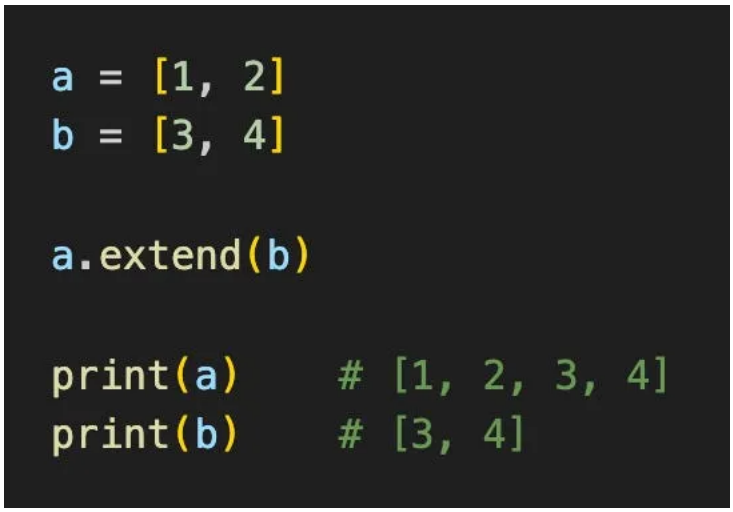
```
ls = ['apple', 'orange', 'pear']
ls.insert(1, 'AAA')

print(ls)
# ['apple', 'AAA', 'orange', 'pear']
```

Note — all elements *after* our inserted index will be pushed back by 1 index, which means that this operation takes O(n) time, or has a linear time complexity. Which just means that this might be slow if we have many many elements in our list.

11) The .extend(other_list) method

We can use the *.extend()* method to add all elements in one list to the back of another.

Here in *a.extend(b),* we add everything from b into a. And b remains unchanged:



```
a = [1, 2]
b = [3, 4]

a.extend(b)

print(a)    # [1, 2, 3, 4]
print(b)    # [3, 4]
```

Technically, we can do *a = a + b ,* but this method creates an entirely new list, then assigns it to a. Which makes this method less memory efficient.

12) list[start:end] = [1, 2, 3]

You probably know that we can replace elements in lists using *list[index] = new_element.*

But did you know that we can replace a *range of elements* this way?

```
ls = ['apple', 'boy', 'cat', 'dog', 'eve']

ls[0:3] = [1, 2, 3]

print(ls)   # [1, 2, 3, 'dog', 'eve']
```

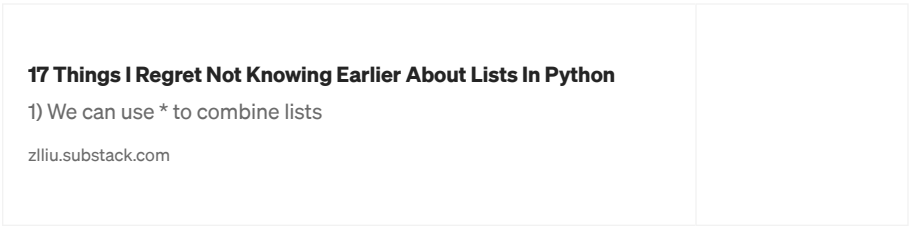
- here, *ls[0:3]* means *['apple', 'boy', 'cat']*
- *ls[0:3] = [1, 2, 3]* means Python will replace *['apple', 'boy', 'cat']* with *[1, 2, 3]*

Note — the lengths of the ranges do not need to match. We can replace a range of length 3 with only 1 element, just like below:

```
ls = ['apple', 'boy', 'cat', 'dog', 'eve']

ls[1:4] = [100]

print(ls)   # ['apple', 100, 'eve']
```

17 Things I Regret Not Knowing Earlier About Lists in Python

Do find the full post on substack and support me if you wish to!



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

The I've

View list

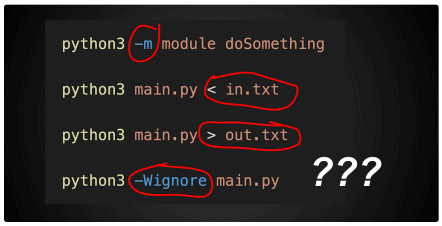
Written by Liu Zuo Lin

96K Followers

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

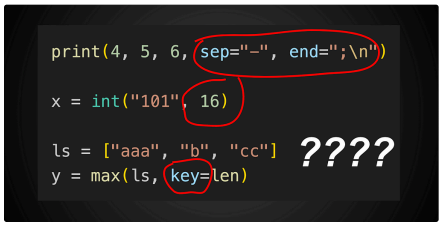


Liu Zuo Lin in Level Up Coding

13 Python Command Line Things I Regret Not Knowing Earlier

Read free...

Oct 19 667 12

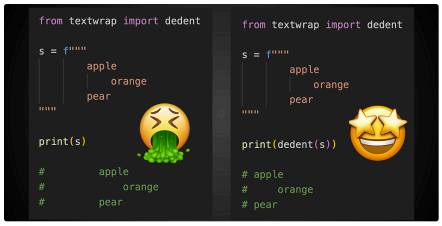


Liu Zuo Lin in Level Up Coding

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

Friend Link Here

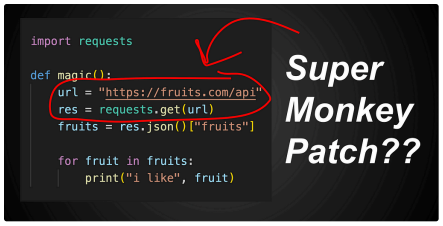
Oct 12 873 5



Liu Zuo Lin in Level Up Coding

Why Python's dedent() Is So Useful When Dealing With Strings

Friend link...



Liu Zuo Lin in Level Up Coding

Baboon Patching in Python

More than a monkey patch

Oct 10 640 4 Oct 2 455 7

See all from Liu Zuo Lin

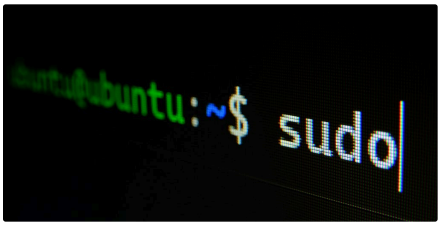
Recommended from Medium

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



CyCoderX in Level Up Coding

5 Linux Command Tricks That Will Change Your Life as a Programmer

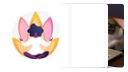
Boost your productivity with these powerful Linux commands for developers!

Oct 10 3.4K 44

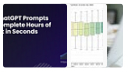
Lists

Coding & Development
11 stories · 878 saves

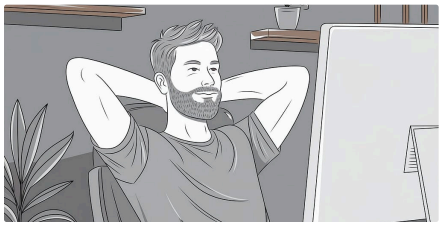
General Coding Knowledge
20 stories · 1688 saves



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

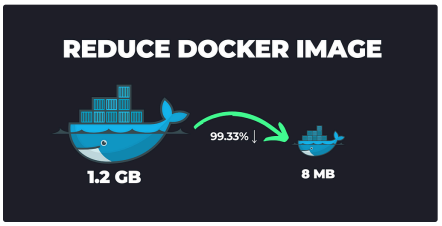


ChatGPT prompts
50 stories · 2157 saves



Abdur Rahman in Stackademic

20 Python Scripts To Automate Your Daily Tasks



Dipanshu in AWS in Plain English

Docker pros are shrinking images by 99%: The hidden techniques yo...

A must-have collection for every developer

Oct 6 1.2K 9

Unlock the secrets to lightning-fast deployments and slashed costs—before yo...

Sep 18 2.4K 11

Paolo Molignini, PhD in Puzzle Sphere

Can you solve this famous interview question?

100 passengers, 100 seats—but the first one sits randomly! What's the chance the last...

Oct 5 1.3K 31

Code geass in Python in Plain English

8 Python Performance Tips I Discovered After Years of Coding ...

Hey Everyone!! I wanted to share these Python Performance tips, that i feel that...

Oct 11 656 8

See more recommendations