Interview Cake

Python Interview Questions

Get ready for your Python programming interview

These Python interview questions will challenge your algorithmic thinking skills as well as your Python programming skills. The first few questions are more Python-specific, and then we have a bunch of general data structures and algorithms questions in Python (/all-questions/python).

Interview Cake is not just another question database—we walk you through the question step-by-step, giving hints and explanations as you need them, just like a real interviewer.

When should you *not* use a list comprehension?

Answer

Two common cases where you shouldn't use a list comprehension are:

- You don't actually want a list
- The logic is too long

Case 1: You don't actually want a list

List comprehensions build lists, but that's not the only reason we use `for` loops. Sometimes we have functions or methods whose main purpose is their side effect, and they don't return anything meaningful.

```
[team.set_location(HOME) for team in league_teams if team in home_teams_today]
```
Python 2.7

Not only does this look more natural as a conventional `for` loop, it doesn't waste space creating a list that it just ignores.

Python 2.7

```
for team in league_teams:

    if team in home_teams_today:
        team.set_location(HOME)
```

Case 2: The logic is too long

One of the main benefits of list comprehensions is that they make your code shorter *and clearer*. Once you start packing too much into a single statement, it becomes *harder* to follow than a regular for loop.

For example:

Python 2.7

```
active_player_accounts = [player.get_account() for team in league_teams
    if len(team.roster) > 1 for player in team.get_players()
    if not player.is_injured()]
```

While descriptive variable names went a long way into making this piece of code somewhat readable, it's still hard to understand.

Python 2.7

```
active_player_accounts = []

for team in league_teams:

    # teams need to have at least 2 players before they're considered "active"
    if len(team.roster) <= 1:
        continue

    for player in team.get_players():

        # only want active players
        if player.is_injured():
            continue

        account = player.get_account()
        active_player_accounts.append(account)
```

This is clearer because we can:

- Include *comments* to explain the code.
- Use control flow *keywords* like `continue`.
- *Debug* this section of code more easily using `logging` statements or `asserts`.
- Easily see the *complexity* of the code by scanning down the lines, picking out for loops.

Slicing lists from the end

We want to run some analytics on our investments. To start, we're given a list containing the balance at the end of the day for some number of days. The *last* item in the list represents yesterday's closing balance and each previous item refers to the day before.

```
daily_balances = [107.92, 108.67, 109.86, 110.15]
```
Python 2.7

The first step in the process is to grab adjacent items in the list. To get started, we want a function that takes in our list of `daily_balances` and prints pairs of adjacent balances for the last 3 days:

```
show_balances(daily_balances)
```
Python 2.7

should print:

```
"slice starting 3 days ago: [108.67, 109.86]"
"slice starting 2 days ago: [109.86, 110.15]"
```
Python 2.7

We just hired a new intern, Dan, to help us with this but something doesn't seem to be working quite right. Here's his function:

```
def show_balances(daily_balances):

    # do not include -1 because that slice will only have 1 balance, yesterday
    for day in range(-3, -1):
        balance_slice = daily_balances[day : day + 2]

        # use positive number for printing
        print("slice starting %d days ago: %s" % (abs(day), balance_slice))
```

Python 2.7

What's his code is printing, and how can we fix it?

Side note: Although learning the answers to these common Python questions is important, it's *so much more important to be able to quickly solve Python problems you've never seen before*. Interviewers want to see that you can do more than just memorize facts!

If you really want to take your prep to the next level, and learn the *right way of thinking* to quickly solve new problems, check out our free 7-day email course:

Answer

If we run his code:

```
daily_balances = [107.92, 108.67, 109.86, 110.15]
show_balances(daily_balances)
```

Python 2.7

this is what we see:

```
"slice starting 3 days ago: [108.67, 109.86]"
"slice starting 2 days ago: []"
```

Python 2.7

Everything worked fine on the first slice, but the second one is empty. What's going on?

The issue is that list slicing with negative indices can get tricky if we aren't careful. The first time through the loop, we take the slice `daily_balances[-3:-1]` and everything works as expected. However, the second time through the loop, we take the slice `daily_balances[-2:0]`.

Our stop index, 0, isn't the *end* of our list, it's the *first* item! So, we just asked for a slice from the next-to-last item to the very first item, which is definitely not what we meant to do.

> So, why didn't Python return `daily_balances` in reverse order, from the next-to-last item up through the first item? This isn't what we originally wanted, but wouldn't it make more sense than the empty list we got?
>
> Python is happy to slice lists in reverse order, but wants you to be explicit so it knows unambiguously you want reverse slices. If we wanted a reverse slice, we need to use `daily_balances[-2:0:-1]` where the third parameter, -1, is the *step* argument, telling Python to reverse the order of the items it slices.

Essentially, when we asked for the slice `daily_balances[-2:0]` we asked for all the elements starting 2 from the end whose index was less than 0. That's an empty set of numbers, but not an error, so Python returns an empty slice.

So how do we fix this code?

Since `daily_balances` is just a regular list, the fix is simple—use positive indices instead:

Python 2.7

```
def show_balances(daily_balances):
    num_balances = len(daily_balances)

    # still avoid slice that just has yesterday
    for day in range(num_balances - 3, num_balances - 1):
        balance_slice = daily_balances[day : day + 2]

        # need to calculate how many days ago
        days_ago = num_balances - day
        print("slice starting %d days ago: %s" % (abs(days_ago), balance_slice))
```

Count Capital Letters

Write a one-liner that will count the number of capital letters in a file. Your code should work even if the file is too big to fit in memory.

Assume you have an open file handle object, such as:

```
with open(SOME_LARGE_FILE) as fh:
    count =  # your code here
```

Python 2.7

Rest assured—there *is* a clean, readable answer!

Answer

Trying to pull a one-liner out of thin air can be daunting and error-prone. Instead of trying to tackle that head-on, let's work on understanding the framework of our answer, and only afterwards try to convert it into a one-liner. Our first thought might be to keep a running count as we look through the file:

```
count = 0
text = fh.read()

for character in text:

    if character.isupper():
        count += 1
```

Python 2.7

This is a great start—the code isn't very long, but it *is* clear. That makes it easier to iterate over as we build up our solution. There are two main issues with what we have so far if we want to turn it into a one-liner:

- **Variable initialization**: In a one-liner, we won't be able to use something like `count = 0`
- **Memory**: The question says this needs to work even on files that won't fit into memory, so we can't just read the whole file into a string.

Let's try to deal with the memory issue first: we can't use the `read()` method since that reads the whole file at once. There are some other common file methods, `()` (https://docs.python.org/2/library/stdtypes.html#file.readlines) and `()` (https://docs.python.org/2/library/stdtypes.html#file.readline), that might help, so let's look at them.

- `readlines()` is a method that reads a file into a list—each line is a different item in the list. That doesn't really help us—it's still reading the entire file at once, so we still won't have room.
- `readline()` only reads a single line at a time—it seems more promising. This is great from a memory perspective (let's assume each *line* fits in memory, at least for now).

  But, we can't just replace `read()` with `readline()` because that only gives us the first line. We need to call `readline()` over and over until we read the entire file.

  The idea of repeatedly calling a function, such as `readline()`, until we hit some value (the end of the file) is so common, there's a standard library function for it: `iter()`. We'll need the two-argument form of `iter()`, where the first argument is our function to call repeatedly, and the second argument is the value that tells us when to stop (also called the *sentinel*).

  What value do we need as our *sentinel*? Looking at the documentation for `readline()`, it includes the newline character so even blank lines will have at least one character. It returns an empty string *only* when it hits the end of the file, so our *sentinel* is `''`.

  Python 2.7

  ```
  count = 0

  for line in iter(fh.readline, ''):

      for character in line:

          if character.isupper():
              count += 1
  ```

  And this works! But...it's not as clear as it could be. Understanding this code requires knowing about the two-argument `iter()` and that `readline()` returns `''` at the end of the file. Trying to condense all this into a one-liner seems like it might be confusing to follow.

Is there a simpler way to iterate over the lines in the file? If you're using Python 3, there aren't any methods for that on your file handle. If you're using Python 2.7, there *is* something that sounds interesting—`xreadlines()` (https://docs.python.org/2/library/stdtypes.html#file.xreadlines). It iterates over the lines in a file, yielding each one to let us process it before reading the next line. In our code, it might be used like:

Python 2.7

```
count = 0

for line in fh.xreadlines():

    for character in line:

        if character.isupper():
            count += 1
```

It's exactly like our code with `readline()` and `iter()` but even clearer! It's a shame we can't use this in Python3.x though, it seems like it would be great. Let's look at the documentation for this method to see if we can learn what alternatives Python3.x might have:

Python 2.7

```
>> help(fh.readlines)
xreadlines() -> returns self.

For backwards compatibility. File objects now include the performance
optimizations previously implemented in the xreadlines module.
```

Huh? "returns self"—how does that even do anything?

What's happening here is that iterating over the lines of a file is *so common* that they built it right in to the object itself. If we use our file object in an iterator, it starts yielding us lines, just like `xreadlines()`! So we can clean up our code, and make it Python3.x compatible, by just removing `xreadlines()`.

Python 2.7

```
count = 0
for line in fh:

    for character in line:

        if character.isupper():

            count += 1
```

Alright, we've finally solved the issue of efficiently reading the file and iterating over it, but we haven't made any progress on making it a one-liner. As we said in the beginning, we can't initialize variables, so what we need is a function that will just return the count of all capitalized

letters. There isn't a `count()` function in Python (at least, not one that would help us here), but we can rephrase the question just enough to find a function that gets the job done.

Instead of thinking about a "count of capitalized letters", let's think about mapping every letter (every character, even) to a number, since our answer is a number. All we care about are capital letters, and each one adds exactly 1 to our final count. Every other character should be ignored, or add 0 to our final count. We can get this mapping into a single line using Python's inline if-else:

Python 2.7

```
count = 0

for line in fh:

    for character in line:

        count += (1 if character.isupper() else 0)
```

What did this mapping get us? Well, Python didn't have a function to count capital letters, but it *does* have a function to add up a bunch of 1s and 0s: `sum()`.

`sum()` takes any iterable, such as a generator expression, and our latest solution—nested for loops and a single if-else—can easily be rewritten as a generator expression:

Python 2.7

```
count = sum(1 if character.isupper() else 0 for line in fh for character in line)
```

and now we've got a one-liner! It's not *quite* as clear as it could be—seems unnecessary to explicitly sum 0 whenever we have a character that isn't a capital letter. We can filter those out:

Python 2.7

```
count = sum(1 for line in fh for character in line if character.isupper())
```

or we can even take advantage of the fact that Python will coerce `True` to 1 (and `False` to 0):

Python 2.7

```
count = sum(character.isupper() for line in fh for character in line)
```

Best in Subclass

When I was younger, my parents always said I could have pets if I promised to take care of them. Now that I'm an adult, I decided the best way to keep track of them is with some Python classes!

Python 2.7

```
class Pet(object):
    num_pets = 0

    def __init__(self, name):
        self.name = name
        self.num_pets += 1

    def speak(self):
        print("My name's %s and the number of pets is %d" % (self.name, self.num_pets))
```

Since these pets won't sit still long enough to be put into a list, I need to keep track with the class attribute `num_pets`.

That should be enough to get me started. Let's create a few pets:

Python 2.7

```
rover = Pet("rover")
spot = Pet("spot")
```

and see what they have to say:

Python 2.7

```
rover.speak()
spot.speak()
```

Hmm... I'm not getting the output I expect. What did these two lines print, and how do we fix it?

Answer

When we run these lines we get:

Python 2.7

```
"My name's Rover and the number of pets is 1"
"My name's Spot and the number of pets is 1"
```

Something isn't right—it's not counting the number of pets properly. Don't worry Rover, I didn't *replace* you with Spot! What's happening here?

Turns out, there's the difference between *class* and *instance*- attributes. When we created `rover` and added to `num_pets`, we accidentally shadowed↴

> If two variables in different scopes have the same name, only one of them can be used. In most cases, the variable with the innermost scope takes precedence, masking the outer variable.
>
> We say the outer variable is **shadowed** by the inner variable.
>
> Python 2.7
> ```
> def outer():
>     name = "outer"
>
>     def inner():
>         name = "inner"
>         print("function name is %s", % name)
>
>     print("function name is %s", % name)
>     inner()
>     print("function name is %s", % name)
> ```
>
> When we call `outer()`, we'll see:
>
> ```
> function name is outer
> function name is inner
> function name is outer
> ```
>
> Shadowed variables can lead to unexpected behavior—notice how our `name` variable went back to "outer" after we finished running `inner()`. It wasn't overwritten, just temporarily shadowed by `name` inside `inner()`.

`Pet.num_pets` with `rover.num_pets`—and they're two completely different variables now! We can see this even more clearly if we ask our `Pet` class how many pets it knows about:

Python 2.7
```
>> print(Pet.num_pets)
0
```

Our `Pet` class still thinks there are 0 pets, because each new pet adds 1 *and shadows the class attribute `num_pets` with its own instance attribute.*

So how can we fix this? We just need to make sure we refer to, and increment, the *class* attribute:

Python 2.7

```
class Pet(object):
    num_pets = 0
    def __init__(self, name):
        self.name = name

        # change from: self.num_pets += 1
        Pet.num_pets += 1
```

and now, if we run our updated code:

Python 2.7

```
rover = Pet("Rover")
spot = Pet("Spot")
rover.speak()
spot.speak()
```

we get what we wanted:

Python 2.7

```
"My name's Rover and the number of pets is 2"
"My name's Spot and the number of pets is 2"
```

When is a number not itself?

Given some simple variables:

```
big_num_1   = 1000
big_num_2   = 1000
small_num_1 = 1
small_num_2 = 1
```

Python 2.7

What's the output we get from running the following?

```
big_num_1   is big_num_2
small_num_1 is small_num_2
```

Python 2.7

Answer

The Python keyword `is` tests whether two variables *refer to the exact same object*, not just if they are equal. For example:

```
list_1 = [1, 2, 3]
list_2 = [1, 2, 3]
print("list_1 == list_2? %s" % (list_1 == list_2))
print("list_1 is list_2? %s" % (list_1 is list_2))
```

Python 2.7

prints:

```
list_1 == list_2? True
list_1 is list_2? False
```

Python 2.7

Now let's look at our original questions:

```
big_num_1 is big_num_2
```

Python 2.7

returns

```
False
```

Python 2.7

This should make sense—we created two different objects that each hold a number, so while they happen to hold the same value, they aren't referring to the same object in memory.

And for small numbers?

```
small_num_1 is small_num_2
```
Python 2.7

returns

```
True
```
Python 2.7

What's happening here? **Python creates singletons for the most commonly used integers**. One good reason to do this is that small numbers get used so frequently that if Python had to create a brand new object every time it needed a number, and then free the object when it goes out of scope, it would start to actually take a noticeable amount of time.

> The idea behind singletons is that there can only ever be one instance of a particular object, and whenever someone tries to use or create a new one, they get the original.

So, which numbers count as "small numbers"? We can write a quick bit of code to test this out for us:

```
# Python ranges start at 0 and don't include the last number
for num in range(1001):

    # try to create a new object
    num_copy = num * 1

    if num is num_copy:
        print(num, "is a singleton")
    else:
        print(num, "is not a singleton")
```
Python 2.7

prints:

Python 2.7

```
"0 is a singleton"
"1 is a singleton"
"2 is a singleton"
<snip>
"254 is a singleton"
"255 is a singleton"
"256 is a singleton"
"257 is not a singleton"
"258 is not a singleton"
<snip>
```

Python makes singletons for the numbers 0 through 256. But are there other numbers made into singletons? It would make sense that some negative numbers might be worth making only once—it's pretty common to look at, say, the last few characters in a string, or the last few elements in a list. We can run the same code but check negative numbers instead:

Python 2.7

```
# up to but not including 0 - we already checked it
for num in range(-1000, 0):

    # try to create a new object
    num_copy = num * 1

    if num is num_copy:
        print(num, "is a singleton")
    else:
        print(num, "is not a singleton")
```

which prints:

Python 2.7

```
<snip>
"-7 is not a singleton"
"-6 is not a singleton"
"-5 is a singleton"
"-4 is a singleton"
"-3 is a singleton"
"-2 is a singleton"
"-1 is a singleton"
```

This shows that numbers from -5 up to and including 256 have singleton instances, so they *could* be tested against each other with `is`. But we shouldn't do that, since the next version of Python might change the range of singleton numbers. And any code that took advantage of these singletons won't be able to check for equality for numbers outside this range. A simple equality check with numbers is always safest.

Here's a hint

Here at Interview Cake, we've decided to keep all our interview questions inside Python dictionaries. We have a default template to get us started:

Python 2.7

```
question_template = {
    "title": "default title",
    "question": "default question",
    "answer": "default answer",
    "hints": []
}
```

and a function to help us populate new questions:

Python 2.7

```
def make_new_question(title, question, answer, hints=None):
    new_q = question_template.copy()

    # always require title, question, answer
    new_q["title"] = title
    new_q["question"] = question
    new_q["answer"] = answer

    # sometimes there aren't hints, that's fine. Otherwise, add them:
    if hints is not None:
        new_q["hints"].extend(hints)

    return new_q
```

Then we added a few questions (abbreviated for simplicity):

```
question_1 = make_new_question("title1", "question1", "answer1", ["q1 hint1", "q1 hint2"])
question_2 = make_new_question("title2", "question2", "answer2")
question_3 = make_new_question("title3", "question3", "answer3", ["q3 hint1"])
```
Python 2.7

What did we do wrong? How can we fix it?

Answer

Things start going wrong after question 1. Let's look at question 2 and see what we got:

```
{
    "title": "title2",
    "question": "question2",
    "answer": "answer2",
    "hints": ["q1 hint1", "q1 hint2"]
}
```
Python 2.7

Question 2 wasn't supposed to have any hints! And question 3 is even more jumbled:

```
{
    "title": "title3",
    "question": "question3",
    "answer": "answer3",
    "hints": ["q1 hint1", "q1 hint2", "q3 hint1"]
}
```
Python 2.7

It has hints from question 1 *and* its own hints! What's going on here?

The problem all stems from how we used our `question_template`—there's nothing wrong with the template itself, but when we call `question_template.copy()`, we're making a *shallow copy* of our dictionary. A shallow copy basically does this:

Python 2.7

```
def shallow_copy(original_dict):
    new_dict = {}

    for key, value in original_dict.items():
        new_dict[key] = value

    return new_dict
```

All of the items, keys and values, refer to the exact same objects after making a shallow copy. The issue arises with our hints because it's a list, which is mutable. Our copy of `question_template` points to the *same exact object* as the hints in our template! We can see this if we just print out our template:

Python 2.7

```
question_template = {
    "title": "default title",
    "question": "default question",
    "answer": "default answer",
    "hints": ["q1 hint1", "q1 hint2", "q3 hint1"]
}
```

One way around this problem would be to *overwrite* the list of hints every time:

Python 2.7

```
def make_new_question(title, question, answer, hints=None):
    new_q = question_template.copy()
    new_q["title"] = title
    new_q["question"] = question
    new_q["answer"] = answer

    if hints is not None:

        # overwrite the mutable hints default here
        new_q["hints"] = hints

    return new_q
```

This works for our simple dictionary here, since we know the only mutable element is the hints variable. However, this can get more confusing if we had a deeper structure to copy over, such as a list of dicts of lists (of lists of...). That's actually pretty common when working on REST APIs that

return giant nested JSON dictionaries.

A more general solution is to ensure that any mutable objects have true copies created, rather than just passing along a "reference" to the original object. And if that mutable object is a container, then any of *its* mutable elements need to make true copies, and so on, recursively. Rolling this code by hand would be error-prone and tedious—luckily, Python has a standard library function that can help: `deepcopy()`.

We can just drop that in where our shallow copy was:

Python 2.7

```
from copy import deepcopy

def make_new_question(title, question, answer, hints=None):

    # make a deep copy before doing anything else
    new_q = deepcopy(question_template)
    new_q["title"] = title
    new_q["question"] = question
    new_q["answer"] = answer

    if hints is not None:
        new_q["hints"].extend(hints)

    return new_q
```

Now, the list of hints in each new question will be a brand new list, so changes to it won't affect other questions or the template question.

Generating the Matrix

We want to build a matrix of values, like a multiplication table. The output we want in this case should be a list of lists, like:

```
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```
Python 2.7

Trying to keep our code clean and concise, we've come up with a matrix generator:

```
iterator = (i for i in range(1, 4))
matrix = [[x * y for y in iterator] for x in iterator]
```
Python 2.7

But our output isn't what we expected. Can you figure out what we got instead, and how to fix it?

Answer

Instead of 3 lists with 3 elements, if we run the code above we get:

```
[[2, 3]]
```
Python 2.7

The reason we didn't get what we expected is because our iterator is a **generator**. Generators in Python have an interesting property—they create values *lazily*, which allows them to save space. For this same reason though, they only create each value *once*. After an element has been yielded by a generator, there's no way to go back and get that value again.

It's easiest to see what happens when we walk through this code step by step. Inside our list comprehension, we have nested for loops:

- Start with the outer for loop, which is `for x in iterator`. This is the first time we've called the iterator, so we get `x = 1`
- Go inside the inner list comprehension to reach our inner for loop, `for y in iterator`. Since we already called our iterator once before, we get `y = 2`
- Do our computation, `x * y = 2`, which is the first value in our first list
- Keep working through the *inner* for loop, yielding the next value so that `y = 3`
- Do our computation, where our x is still 1 so `x * y = 3`, which is the second value in our first list
- Try to keep working through the *inner* for loop, and that loop ends, since there are no more values in our iterator
- Pop back up to the *outer* for loop, but the iterator is still empty, so we finish our list comprehension

and that's how we end up with only 2 values in our matrix.

The simplest way to fix our code in this case is to not use a generator. We just have to make our iterator into a list:

Python 2.7

```
iterator = [i for i in range(1, 4)]
matrix = [[x * y for y in iterator] for x in iterator]
```

Pro tip: Learn how to avoid the dreaded "whiteboard freeze." In our 7-day email crash course, we'll teach you the strategy for quickly breaking down and solving any coding interview question.