53 Python Interview Questions and Answers

Python questions for data scientist and software engineers

Chris I. Follow
Apr 19 · 15 min read ★

Photo by Brooke Cagle on Unsplash

Not so long ago I started a new role as a “Data Scientist” which turned out to be “Python Engineer” in practice.

I would have been more prepared if I’d brushed up on Python’s thread lifecycle instead of recommender systems in advance.

In that spirit, here are my python interview/job preparation questions and answers. Most data scientists write a lot code so this applies to both scientists and engineers.

Whether you're interviewing candidates, preparing to apply to jobs or just brushing up on Python, I think this list will be invaluable.

Questions are unordered. Let's begin.

. . .

1. What is the difference between a list and a tuple?

I've been asked this question in every python / data science interview I've ever had. Know the answer like the back of your hand.

- Lists are mutable. They can be modified after creation.
- Tuples are immutable. Once a tuple is created it cannot by changed
- Lists have order. They are an ordered sequences, typically of the same type of object. Ie: all user names ordered by creation date, `["Seth", "Ema", "Eli"]`
- Tuples have structure. Different data types may exist at each index. Ie: a database record in memory, `(2, "Ema", "2020-04-16") # id, name, created_at`

2. How is string interpolation performed?

Without importing the `Template` class, there are 3 ways to interpolate strings.

```
name = 'Chris'

# 1. f strings
print(f'Hello {name}')

# 2. % operator
print('Hey %s %s' % (name, name))

# 3. format
print(
 "My name is {}".format((name))
)
```

3. What is the difference between "is" and "=="?

Early in my python career I assumed these were the same… hello bugs. So for the record, `is` checks identity and `==` checks equality.

We'll walk through an example. Create some lists and assign them to names. Note that `b` points to the same object as `a` in below.

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

Check equality and note they are all equal.

```
print(a == b)
print(a == c)
#=> True
#=> True
```

But do they have the same identity? Nope.

```
print(a is b)
print(a is c)
#=> True
#=> False
```

We can verify this by printing their object id's.

```
print(id(a))
print(id(b))
print(id(c))
#=> 4369567560
#=> 4369567560
#=> 4369567624
```

`c` has a different `id` than `a` and `b`.

4. What is a decorator?

Another questions I've been asked in every interview. It's deserves a post itself, but you're prepared if you can walk through writing your own example.

A decorator allows adding functionality to an existing function by passing that existing function to a decorator, which executes the existing function as well as additional code.

We'll write a decorator that that logs when another function is called.

Write the decorator function. This takes a function, `func`, as an argument. It also defines a function, `log_function_called`, which calls `func()` and executes some code, `print(f'{func} called.')`. Then it return the function it defined

```
def logging(func):
  def log_function_called():
    print(f'{func} called.')
    func()
  return log_function_called
```

Let's write other functions that we'll eventually add the decorator to (but not yet).

```
def my_name():
  print('chris')

def friends_name():
  print('naruto')

my_name()
friends_name()
#=> chris
#=> naruto
```

Now add the decorator to both.

```
@logging
def my_name():
 print('chris')

@logging
def friends_name():
 print('naruto')

my_name()
friends_name()
#=> <function my_name at 0x10fca5a60> called.
#=> chris
#=> <function friends_name at 0x10fca5f28> called.
#=> naruto
```

See how we can now easily add logging to any function we write just by adding `@logging` above it.

5. Explain the range function

Range generates a list of integers and there are 3 ways to use it.

The function takes 1 to 3 arguments. Note I've wrapped each usage in list comprehension so we can see the values generated.

`range(stop)` : generate integers from 0 to the "stop" integer.

```
[i for i in range(10)]
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range(start, stop)` : generate integers from the "start" to the "stop" integer.

```
[i for i in range(2,10)]
#=> [2, 3, 4, 5, 6, 7, 8, 9]
```

`range(start, stop, step)` : generate integers from "start" to "stop" at intervals of "step".

```
[i for i in range(2,10,2)]
#=> [2, 4, 6, 8]
```

Thanks Searge Boremchuq for suggesting a more pythonic way to do this!

```
list(range(2,10,2))
#=> [2, 4, 6, 8]
```

6. Define a class named car with 2 attributes, "color" and "speed". Then create an instance and return speed.

```
class Car :
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

car = Car('red','100mph')
car.speed
#=> '100mph'
```

7. What is the difference between instance, static and class methods in python?

Instance methods : accept `self` parameter and relate to a specific instance of the class.

Static methods : use `@staticmethod` decorator, are not related to a specific instance, and are self-contained (don't modify class or instance attributes)

Class methods : accept `cls` parameter and can modify the class itself

We're going to illustrate the difference around a fictional `CoffeeShop` class.

```
class CoffeeShop:
    specialty = 'espresso'

    def __init__(self, coffee_price):
        self.coffee_price = coffee_price
    # instance method
    def make_coffee(self):
        print(f'Making {self.specialty} for ${self.coffee_price}')

    # static method
    @staticmethod
    def check_weather():
        print('Its sunny')

    # class method
    @classmethod
    def change_specialty(cls, specialty):
        cls.specialty = specialty
        print(f'Specialty changed to {specialty}')
```

`CoffeeShop` class has an attribute, `specialty`, set to `'espresso'` by default. Each instance of `CoffeeShop` is initialized with an attribute `coffee_price`. It also has 3 methods, an instance method, a static method and a class method.

Let's initialize an instance of the coffee shop with a `coffee_price` of `5`. Then call the instance method `make_coffee`.

```
coffee_shop = CoffeeShop('5')
coffee_shop.make_coffee()
#=> Making espresso for $5
```

Now call the static method. Static methods can't modify class or instance state so they're normally used for utility functions, for example, adding 2 numbers. We used ours to check the weather. `Its sunny`. Great!

```
coffee_shop.check_weather()
#=> Its sunny
```

Now let's use the class method to modify the coffee shop's specialty and then `make_coffee`.

```
coffee_shop.change_specialty('drip coffee')
#=> Specialty changed to drip coffee

coffee_shop.make_coffee()
#=> Making drip coffee for $5
```

Note how `make_coffee` used to make `espresso` but now makes `drip coffee`!

8. What is the difference between "func" and "func()"?

The purpose of this question is to see if you understand that all functions are also objects in python.

```
def func():
    print('Im a function')

func
#=> function __main__.func>

func()
#=> Im a function
```

`func` is the object representing the function which can be assigned to a variable or passed to another function. `func()` with parentheses calls the function and returns what it outputs.

9. Explain how the map function works

`map` returns a map object (an iterator) which can iterate over returned values from applying a function to every element in a sequence. The map object can also be converted to a list if required.

```
def add_three(x):
    return x + 3

li = [1,2,3]

[i for i in map(add_three, li)]
#=> [4, 5, 6]
```

Above, I added 3 to every element in the list.

A reader suggested a more pythonic implementation. Thanks Chrisjan Wust !

```
def add_three(x):
    return x + 3

li = [1,2,3]
list(map(add_three, li))
#=> [4, 5, 6]
```

Also, thanks Michael Graeme Short for the corrections!

10. Explain how the reduce function works

This can be tricky to wrap your head around until you use it a few times.

`reduce` takes a function and a sequence and iterates over that sequence. On each iteration, both the current element and output from the previous element are passed to the function. In the end, a single value is returned.

```
from functools import reduce

def add_three(x,y):
    return x + y

li = [1,2,3,5]

reduce(add_three, li)
#=> 11
```

`11` is returned which is the sum of `1+2+3+5` .

11. Explain how the filter function works

Filter literally does what the name says. It filters elements in a sequence.

Each element is passed to a function which is returned in the outputted sequence if the function returns `True` and discarded if the function returns `False` .

```
def add_three(x):
    if x % 2 == 0:
        return True
    else:
        return False

li = [1,2,3,4,5,6,7,8]

[i for i in filter(add_three, li)]
#=> [2, 4, 6, 8]
```

Note how all elements not divisible by 2 have been removed.

12. Does python call by reference or call by value?

Be prepared to go down a rabbit hole of semantics if you google this question and read the top few pages.

In a nutshell, all names call by reference, but some memory locations hold objects while others hold pointers to yet other memory locations.

```
name = 'object'
```

Let's see how this works with strings. We'll instantiate a name and object, point other names to it. Then delete the first name.

```
x = 'some text'
y = x
x is y #=> True

del x # this deletes the 'a' name but does nothing to the object in
memory

z = y
y is z #=> True
```

What we see is that all these names point to the same object in memory, which wasn't affected by `del x`.

Here's another interesting example with a function.

```
name = 'text'

def add_chars(str1):
    print( id(str1) ) #=> 4353702856
    print( id(name) ) #=> 4353702856

    # new name, same object
    str2 = str1

    # creates a new name (with same name as the first) AND object
    str1 += 's'
    print( id(str1) ) #=> 4387143328

    # still the original object
    print( id(str2) ) #=> 4353702856


add_chars(name)
print(name) #=>text
```

Notice how adding an `s` to the string inside the function created a new name AND a new object. Even though the new name has the same "name" as the existing name.

Thanks Michael P. Reilly for the corrections!

13. How to reverse a list?

Note how `reverse()` is called on the list and mutates it. It doesn't return the mutated list itself.

```
li = ['a','b','c']

print(li)
li.reverse()
print(li)
#=> ['a', 'b', 'c']
#=> ['c', 'b', 'a']
```

14. How does string multiplication work?

Let's see the results of multiplying the string `'cat'` by 3.

```
'cat' * 3
#=> 'catcatcat'
```

The string is concatenated to itself 3 times.

15. How does list multiplication work?

Let's see the result of multiplying a list, `[1,2,3]` by 2.

```
[1,2,3] * 2
#=> [1, 2, 3, 1, 2, 3]
```

A list is outputted containing the contents of [1,2,3] repeated twice.

16. What does "self" refer to in a class?

Self refers to the instance of the class itself. It's how we give methods access to and the ability to update the object they belong to.

Below, passing self to `__init__()` gives us the ability to set the `color` of an instance on initialization.

```
class Shirt:
    def __init__(self, color):
        self.color = color

s = Shirt('yellow')
s.color
#=> 'yellow'
```

17. How can you concatenate lists in python?

Adding 2 lists together concatenates them. Note that arrays do not function the same way.

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

```
a + b
#=> [1, 2, 3, 4, 5]
```

18. What is the difference between a shallow and a deep copy?

We'll discuss this in the context of a mutable object, a list. For immutable objects, shallow vs deep isn't as relevant.

We'll walk through 3 scenarios.

i) Reference the original object. This points a new name, `li2`, to the same place in memory to which `li1` points. So any change we make to `li1` also occurs to `li2`.

```
li1 = [['a'],['b'],['c']]
li2 = li1

li1.append(['d'])
print(li2)
#=> [['a'], ['b'], ['c'], ['d']]
```

ii) Create a shallow copy of the original. We can do this with the `list()` constructor, or the more pythonic `mylist.copy()` (thanks Chrisjan Wust !).

A shallow copy creates a new object, but fills it with references to the original. So adding a new object to the original collection, `li3`, doesn't propagate to `li4`, but modifying one of the objects in `li3` will propagate to `li4`.

```
li3 = [['a'],['b'],['c']]
li4 = list(li3)

li3.append([4])
print(li4)
#=> [['a'], ['b'], ['c']]

li3[0][0] = ['X']
print(li4)
#=> [[['X']], ['b'], ['c']]
```

iii) Create a deep copy. This is done with `copy.deepcopy()` . The 2 objects are now completely independent and changes to either have no affect on the other.

```
import copy

li5 = [['a'],['b'],['c']]
li6 = copy.deepcopy(li5)

li5.append([4])
li5[0][0] = ['X']
print(li6)
#=> [['a'], ['b'], ['c']]
```

19. What is the difference between lists and arrays?

Note: Python's standard library has an array object but here I'm specifically referring to the commonly used Numpy array.

- Lists exist in python's standard library. Arrays are defined by Numpy.
- Lists can be populated with different types of data at each index. Arrays require homogeneous elements.
- Arithmetic on lists adds or removes elements from the list. Arithmetic on arrays functions per linear algebra.
- Arrays also use less memory and come with significantly more functionality.

I wrote another comprehensive post on arrays.

20. How to concatenate two arrays?

Remember, arrays are not lists. Arrays are from Numpy and arithmetic functions like linear algebra.

We need to use Numpy's concatenate function to do it.

```
import numpy as np

a = np.array([1,2,3])
b = np.array([4,5,6])
```

```
np.concatenate((a,b))
#=> array([1, 2, 3, 4, 5, 6])
```

21. What do you like about Python?

Note this is a very subjective question and you'll want to modify your response based on what the role is looking for.

Python is very readable and there is a pythonic way to do just about everything, meaning a preferred way which is clear and concise.

I'd contrast this to Ruby where there are often many ways to do something without a guideline for which is preferred.

22. What is you favorite library in Python?

Also subjective, see question 21.

When working with a lot data, nothing is quite as helpful as pandas which makes manipulating and visualizing data a breeze.

23. Name mutable and immutable objects

Immutable means the state cannot be modified after creation. Examples are: int, float, bool, string and tuple.

Mutable means the state can be modified after creation. Examples are list, dict and set.

24. How would you round a number to 3 decimal places?

Use the `round(value, decimal_places)` function.

```
a = 5.12345
round(a,3)
#=> 5.123
```

25. How do you slice a list?

Slicing notation takes 3 arguments, `list[start:stop:step]` , where step is the interval at which elements are returned.

```
a = [0,1,2,3,4,5,6,7,8,9]

print(a[:2])
#=> [0, 1]

print(a[8:])
#=> [8, 9]

print(a[2:8])
#=> [2, 3, 4, 5, 6, 7]

print(a[2:8:2])
#=> [2, 4, 6]
```

26. What is pickling?

Pickling is the go-to method of serializing and unserializing objects in Python.

In the example below, we serialize and unserialize a list of dictionaries.

```
import pickle

obj = [
    {'id':1, 'name':'Stuffy'},
    {'id':2, 'name': 'Fluffy'}
]

with open('file.p', 'wb') as f:
    pickle.dump(obj, f)

with open('file.p', 'rb') as f:
    loaded_obj = pickle.load(f)

print(loaded_obj)
#=> [{'id': 1, 'name': 'Stuffy'}, {'id': 2, 'name': 'Fluffy'}]
```

27. What is the difference between dictionaries and JSON?

Dict is python datatype, a collection of indexed but unordered keys and values.

JSON is just a string which follows a specified format and is intended for transferring data.

28. What ORMs have you used in Python?

ORMs (object relational mapping) map data models (usually in an app) to database tables and simplifies database transactions.

SQLAlchemy is typically used in the context of Flask, and Django has it's own ORM.

29. How do any() and all() work?

Any takes a sequence and returns true if any element in the sequence is true.

All returns true only if all elements in the sequence are true.

```
a = [False, False, False]
b = [True, False, False]
c = [True, True, True]

print( any(a) )
print( any(b) )
print( any(c) )
#=> False
#=> True
#=> True

print( all(a) )
print( all(b) )
print( all(c) )
#=> False
#=> False
#=> True
```

30. Are dictionaries or lists faster for lookups?

Looking up a value in a list takes O(n) time because the whole list needs to be iterated through until the value is found.

Looking up a key in a dictionary takes O(1) time because it's a hash table.

This can make a huge time difference if there are a lot of values so dictionaries are generally recommended for speed. But they do have other limitations like needing unique keys.

31. What is the difference between a module and a package?

A module is a file (or collection of files) that can be imported together.

```
import sklearn
```

A package is a directory of modules.

```
from sklearn import cross_validation
```

So packages are modules, but not all modules are packages.

32. How to increment and decrement an integer in Python?

Increments and decrements can be done with `+-` and `-=` .

```
value = 5

value += 1
print(value)
#=> 6

value -= 1
value -= 1
print(value)
#=> 4
```

33. How to return the binary of an integer?

Use the `bin()` function.

```
bin(5)
#=> '0b101'
```

34. How to remove duplicate elements from a list?

This can be done by converting the list to a set then back to a list.

```
a = [1,1,1,2,3]
a = list(set(a))
print(a)
#=> [1, 2, 3]
```

Note that sets will not necessarily maintain the order of a list.

35. How to check if a value exists in a list?

Use `in`.

```
'a' in ['a','b','c']
#=> True

'a' in [1,2,3]
#=> False
```

36. What is the difference between append and extend?

`append` adds a value to a list while `extend` adds values in another list to a list.

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

a.append(6)
print(a)
#=> [1, 2, 3, 6]

b.extend([4,5])
print(b)
#=> [1, 2, 3, 4, 5]
```

37. How to take the absolute value of an integer?

This can be done with the `abs()` function.

```
abs(2)
#=> 2

abs(-2)
#=> 2
```

38. How to combine two lists into a list of tuples?

You can use the `zip` function to combine lists into a list of tuples. This isn't restricted to only using 2 lists. It can also be done with 3 or more.

```
a = ['a','b','c']
b = [1,2,3]

[(k,v) for k,v in zip(a,b)]
#=> [('a', 1), ('b', 2), ('c', 3)]
```

39. How can you sort a dictionary by key, alphabetically?

You can't "sort" a dictionary because dictionaries don't have order but you can return a sorted list of tuples which has the keys and values that are in the dictionary.

```
d = {'c':3, 'd':4, 'b':2, 'a':1}

sorted(d.items())
#=> [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

40. How does a class inherit from another class in Python?

In the below example, `Audi`, inherits from `Car`. And with that inheritance comes the instance methods of the parent class.

```
class Car():
    def drive(self):
        print('vroom')

class Audi(Car):
    pass

audi = Audi()
audi.drive()
```

41. How can you remove all whitespace from a string?

The easiest way is to split the string on whitespace and then rejoin without spaces.

```
s = 'A string with     white space'

''.join(s.split())
#=> 'Astringwithwhitespace'
```

2 readers recommended a more pythonic way to handle this following the Python ethos that `Explicit is better than Implicit`. It's also faster because python doesn't create a new list object. Thanks Евгений Крамаров and Chrisjan Wust !

```
s = 'A string with     white space'
s.replace(' ', '')
#=> 'Astringwithwhitespace'
```

42. Why would you use enumerate() when iterating on a sequence?

`enumerate()` allows tracking index when iterating over a sequence. It's more pythonic than defining and incrementing an integer representing the index.

```
li = ['a','b','c','d','e']

for idx,val in enumerate(li):
    print(idx, val)
#=> 0 a
#=> 1 b
#=> 2 c
#=> 3 d
#=> 4 e
```

43. What is the difference between pass, continue and break?

`pass` means do nothing. We typically use it because Python doesn't allow creating a class, function or if-statement without code inside it.

In the example below, an error would be thrown without code inside the `i > 3` so we use `pass`.

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

for i in a:
    if i > 3:
        pass
    print(i)
#=> 1
#=> 2
#=> 3
#=> 4
#=> 5
```

`continue` continues to the next element and halts execution for the current element. So `print(i)` is never reached for values where `i < 3`.

```
for i in a:
    if i < 3:
        continue
    print(i)
#=> 3
#=> 4
#=> 5
```

`break` breaks the loop and the sequence is not longer iterated over. So elements from 3 onward are not printed.

```
for i in a:
    if i == 3:
        break
    print(i)
#=> 1
#=> 2
```

44. Convert the following for loop into a list comprehension.

This `for` loop.

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

```
a2 = []
for i in a:
     a2.append(i + 1)

print(a2)
#=> [2, 3, 4, 5, 6]
```

Becomes.

```
a3 = [i+1 for i in a]

print(a3)
#=> [2, 3, 4, 5, 6]
```

List comprehension is generally accepted as more pythonic where it's still readable.

45. Give an example of the ternary operator.

The ternary operator is a one-line if/else statement.

The syntax looks like `a if condition else b`.

```
x = 5
y = 10

'greater' if x > 6 else 'less'
#=> 'less'

'greater' if y > 6 else 'less'
#=> 'greater'
```

46. Check if a string only contains numbers.

You can use `isnumeric()`.

```
'123a'.isnumeric()
#=> False

'123'.isnumeric()
#=> True
```

47. Check if a string only contains letters.

You can use `isalpha()`.

```
'123a'.isalpha()
#=> False

'a'.isalpha()
#=> True
```

48. Check if a string only contains numbers and letters.

You can use `isalnum()`.

```
'123abc...'.isalnum()
#=> False

'123abc'.isalnum()
#=> True
```

49. Return a list of keys from a dictionary.

This can be done by passing the dictionary to python's `list()` constructor, `list()`.

```
d = {'id':7, 'name':'Shiba', 'color':'brown', 'speed':'very slow'}

list(d)
#=> ['id', 'name', 'color', 'speed']
```

50. How do you upper and lowercase a string?

You can use the `upper()` and `lower()` string methods.

```
small_word = 'potatocake'
big_word = 'FISHCAKE'
```

```
small_word.upper()
#=> 'POTATOCAKE'

big_word.lower()
#=> 'fishcake'
```

51. What is the difference between remove, del and pop?

`remove()` remove the first matching value.

```
li = ['a','b','c','d']

li.remove('b')
li
#=> ['a', 'c', 'd']
```

`del` removes an element by index.

```
li = ['a','b','c','d']

del li[0]
li
#=> ['b', 'c', 'd']
```

`pop()` removes an element by index and returns that element.

```
li = ['a','b','c','d']

li.pop(2)
#=> 'c'

li
#=> ['a', 'b', 'd']
```

52. Give an example of dictionary comprehension.

Below we'll create dictionary with letters of the alphabet as keys, and index in the alphabet as values.

```
# creating a list of letters
import string
list(string.ascii_lowercase)
alphabet = list(string.ascii_lowercase)

# list comprehension
d = {val:idx for idx,val in enumerate(alphabet)}

d
#=> {'a': 0,
#=>  'b': 1,
#=>  'c': 2,
#=> ...
#=>  'x': 23,
#=>  'y': 24,
#=>  'z': 25}
```

53. How is exception handling performed in Python?

Python provides 3 words to handle exceptions, `try` , `except` and `finally` .

The syntax looks like this.

```
try:
    # try to do this
except:
    # if try block fails then do this
finally:
    # always do this
```

In the simplistic example below, the `try` block fails because we cannot add integers with strings. The `except` block sets `val = 10` and then the `finally` block prints `complete` .

```
try:
    val = 1 + 'A'
except:
    val = 10
finally:
    print('complete')

print(val)
#=> complete
#=> 10
```

. . .

Conclusion

You never know what questions will come up in interviews and the best way to prepare is to have a lot of experience writing code.

That said, this list should cover most anything you'll be asked python-wise for a data scientist or junior/intermediate python developer roles.

I hope this was as helpful for you as writing it was for me.

Are there any great questions I missed?

Python Data Science Programming Software Engineering Coding