**Question: What is the difference between a tuple and a list in Python?**

**Answer:** A list is a mutable sequence of elements, while a tuple is an immutable sequence of elements. This means that you can modify a list by adding, removing or changing elements, but you cannot do the same with a tuple.

**Question: The following code is supposed to find the sum of the even numbers in a list. What is wrong with the code, and how can it be fixed?**

**Answer:**

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

total = 0

for number in numbers:

  if number % 2 == 0

   total += number

print(total)Code language: Python (python)

The code is missing a colon after the if statement. The fixed code is:

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

total = 0

for number in numbers:

if number % 2 == 0:

  total += number

print(total)Code language: PHP (php)

**Question: What is the difference between a function and a method in Python?**

**Answer:** A function is a standalone block of code that performs a specific task, while a method is a function that is associated with an object and can access and modify its data.

**Question: The following code is supposed to create a dictionary with the keys and values reversed from the original dictionary. What is wrong with the code, and how can it be fixed?**

**Answer:**

original\_dict = {"one": 1, "two": 2, "three": 3}

reversed\_dict = {}

for key, value in original\_dict.items():

reversed\_dict[value] = key

print(reversed\_dict)Code language: Python (python)

The code is correct.

**Question: What is the difference between a while loop and a for loop in Python?**

**Answer:** A for loop is used to iterate over a sequence, such as a list or a string, while a while loop is used to repeat a block of code as long as a certain condition is true.

**Question: The following code is supposed to create a list of the squares of the first five integers. What is wrong with the code, and how can it be fixed?**

**Answer:**

squares = []

for i in range(5)

  squares.append(i \*\* 2)

print(squares)Code language: Python (python)

The code is missing a colon after the for statement. The fixed code is:

squares = []

for i in range(5):

  squares.append(i \*\* 2)

print(squares)Code language: Python (python)

**Question: What is the difference between the “is” operator and the “==” operator in Python?**

**Answer:** The “is” operator checks if two objects are the same object, while the “==” operator checks if two objects have the same value.

**Question: The following code is supposed to remove all the even numbers from a list. What is wrong with the code, and how can it be fixed?**

**Answer:**

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

for number in numbers:

if number % 2 == 0

  numbers.remove(number)

print(numbers)Code language: Python (python)

The code is modifying the list while iterating over it, which can lead to unexpected results. The fixed code is to create a new list instead:

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

odd\_numbers = []

for number in numbers:

if number % 2 != 0:

  odd\_numbers.append(number)

print(odd\_numbers)Code language: Python (python)

**Question: What is the purpose of the pass statement in Python?**

**Answer:** The pass statement is used as a placeholder when a statement is required syntactically, but no code needs to be executed.

**Question: The following code is supposed to create a list of the first five even numbers. What is wrong with the code, and how can it be fixed?**

**Answer:**

even\_numbers = [0, 2, 4, 6, 8]Code language: Python (python)

The code is correct.

**Intermediate Python interview questions**

**Question: What is a generator in Python? How is it different from a regular function?**

**Answer:** A generator is a special type of function in Python that returns an iterator object, which can be iterated over using a for loop. The key difference between a generator and a regular function is that a generator can yield values one at a time, whereas a regular function returns all values at once. This makes generators more memory-efficient, as they do not need to generate all values at once and store them in memory.

**Question: What is the difference between a class method and an instance method in Python?**

**Answer:** A class method is a method that is bound to the class and not the instance of the class. It can be called on the class itself, without the need to create an instance of the class. An instance method, on the other hand, is bound to the instance of the class and can only be called on an instance of the class.

**Question: What is the purpose of the \_\_str\_\_ method in Python classes? How is it different from the \_\_repr\_\_ method?**

**Answer:** The \_\_str\_\_ method is used to define the string representation of an object, and is typically used for displaying the object to end-users. The \_\_repr\_\_ method, on the other hand, is used to define the “official” string representation of an object, and is typically used for debugging and development purposes. The main difference between the two is that \_\_str\_\_ is called when str() is called on an object, while \_\_repr\_\_ is called when repr() is called on an object.

**Question: The following code is supposed to create a class for a basic calculator, but it is not working correctly. Fix the code.**

class Calculator:

    def \_\_init\_\_(self):

        self.result = 0

    def add(self, num1, num2):

        self.result = num1 + num2

    def subtract(self, num1, num2):

        self.result = num1 - num2Code language: Python (python)

**Answer:** The add and subtract methods are not actually returning any values, so they are not useful for external code that wants to use the calculator. To fix this, we can modify the methods to return the result instead of storing it as an attribute of the object:

class Calculator:

    def \_\_init\_\_(self):

        self.result = 0

    def add(self, num1, num2):

        return num1 + num2

    def subtract(self, num1, num2):

        return num1 - num2Code language: Python (python)

**Question: What is a lambda function in Python? How is it different from a regular function?**

**Answer:** A lambda function is a small anonymous function in Python that can have any number of arguments, but can only have one expression. The main difference between a lambda function and a regular function is that a lambda function is defined inline, without a name, and is typically used as a simple, one-line function that is not reusable.

**Question: The following code is supposed to create a generator function that returns a Fibonacci sequence, but it is not working correctly. Fix the code.**

def fibonacci():

    a, b = 0, 1

    while True:

        yield a

        a, b = b, a + bCode language: Python (python)

**Answer:** The code looks correct, but the starting values for a and b should be 0 and 1, respectively. The current code initializes a as 1 and b as 2, which does not generate the correct Fibonacci sequence.

**Question: What is a metaclass in Python? How is it different from a regular class?**

**Answer:** A metaclass is a class that defines the behavior of other classes. It is used to customize the creation of classes and their instances, and can be used to enforce certain constraints or provide additional functionality to classes. The main difference between a metaclass and a regular class is that a metaclass is used to define the behavior of other classes, whereas a regular class is used to define objects that can be instantiated and used in a program.

**Question: The following code is supposed to create a class that represents a bank account, but it is not working correctly. Fix the code.**

class BankAccount:

    def \_\_init\_\_(self, balance):

        self.balance = balance

    def deposit(self, amount):

        self.balance += amount

    def withdraw(self, amount):

        self.balance -= amount

    def check\_balance(self):

        print("Your balance is: ", self.balance)Code language: Python (python)

**Answer:** The code looks correct, but there is no error handling for situations where the user tries to withdraw more money than they have in their account. To fix this, we can add a check in the withdraw method to ensure that the amount being withdrawn does not exceed the current balance:

class BankAccount:

    def \_\_init\_\_(self, balance):

        self.balance = balance

    def deposit(self, amount):

        self.balance += amount

    def withdraw(self, amount):

        if amount > self.balance:

            raise ValueError("Insufficient funds")

        self.balance -= amount

    def check\_balance(self):

        print("Your balance is: ", self.balance)Code language: Python (python)

**Question: The following code is supposed to create a function that takes a list of integers and returns only the even numbers, but it is not working correctly. Fix the code.**

def get\_even\_numbers(numbers):

    even\_numbers = [x for x in numbers if x % 2 == 0]

    return even\_numbersCode language: Python (python)

**Answer:** The code looks correct, but it can be improved by using a generator expression instead of a list comprehension to generate the even numbers on the fly and save memory:

def get\_even\_numbers(numbers):

    even\_numbers = (x for x in numbers if x % 2 == 0)

    return even\_numbersCode language: Python (python)

**Question: What is the difference between a generator function and a normal function in Python? Provide an example of a generator function.**

**Answer:**

* Generator function is a type of function that yields a sequence of values instead of returning a single value.
* Normal function returns a single value and then exits.
* Generator functions use the yield keyword to return a value and pause the execution of the function.
* Normal functions use the return keyword to return a value and immediately exit the function.

Example of a generator function:

def my\_generator(n):

i = 0

while i < n:

yield i

i += 1Code language: Python (python)

This generator function returns a sequence of numbers from 0 to n-1. It does not use the return keyword but instead uses yield to pause and resume the function.

**Senior Python interview questions**

**Question: What is the purpose of lambda functions in Python, and how are they different from regular functions?**

**Answer:** Lambda functions are used to create small, anonymous functions that can be used in place of regular functions. They are different from regular functions in that they do not have a name and can only contain a single expression.

**Question: What is a generator function in Python, and how is it different from a regular function?**

**Answer:** A generator function is a special type of function that can be used to generate a sequence of values on-the-fly. It is different from a regular function in that it uses the “yield” keyword to return values one-at-a-time instead of returning a list of all values at once.

**Question: What is the purpose of the “yield” keyword in Python, and how is it used in generator functions?**

**Answer:** The yield keyword is used in generator functions to return a value and temporarily suspend the function’s execution. When the generator function is called again, it resumes execution from the point it left off and continues generating values one-at-a-time.

**Question: What is meta-programming in Python, and how is it used?**

**Answer:** Meta-programming is writing code that can manipulate code at runtime. It is used to create generic, reusable code, or to modify the behavior of existing code dynamically.

**Question: This code is supposed to create a generator that yields the first n numbers in the Fibonacci sequence. Fix the code.**

def fibonacci(n):

a, b = 0, 1

for i in range(n):

yield b

a, b = b, a+b

print(fibonacci(5))Code language: Python (python)

**Answer:** The code is correct, but the print() statement is printing the generator object instead of the sequence of numbers. To print the sequence of numbers, you can convert the generator to a list:

def fibonacci(n):

a, b = 0, 1

for i in range(n):

yield b

a, b = b, a+b

print(list(fibonacci(5)))Code language: Python (python)

**Question: This code is supposed to return a list of all odd numbers from the input list. Fix the code.**

def get\_odd\_numbers(nums):

odd\_nums = filter(lambda num: num % 2)

return list(odd\_nums)

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(get\_odd\_numbers(nums))Code language: Python (python)

**Answer:** The lambda function inside the filter() function is missing the argument nums in the expression. The corrected code is:

def get\_odd\_numbers(nums):

odd\_nums = filter(lambda num: num % 2, nums)

return list(odd\_nums)

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(get\_odd\_numbers(nums))Code language: Python (python)

**Question: What are decorators in Python, and how are they used?**

**Answer:** Decorators are a way to modify the behavior of a function or class in Python without changing its source code. They are implemented as callable objects that take another function or class as an argument and return a modified version of it.

**Question: This code is supposed to use a lambda function to sort a list of dictionaries by the value of a specific key. Fix the code.**

data = [{'name': 'Alice', 'age': 28}, {'name': 'Bob', 'age': 21}, {'name': 'Charlie', 'age': 35}]

sorted\_data = sorted(data, key=lambda x: x['age'])

print(sorted\_data)Code language: Python (python)

**Answer:**

The code is correct and will sort the list of dictionaries by the age key.

**Question: This code is supposed to create a metaclass that allows classes to be initialized with a dictionary of attributes. Fix the code.**

class DictMeta(type):

def \_\_new\_\_(mcs, name, bases, attrs):

obj = super().\_\_new\_\_(mcs, name, bases, attrs)

obj.\_\_dict\_\_.update(attrs)

return obj

class MyClass(metaclass=DictMeta):

def \_\_init\_\_(self, data):

self.data = data

my\_instance = MyClass({'name': 'Alice', 'age': 28})

print(my\_instance.name, my\_instance.age)Code language: Python (python)

**Answer:**

class DictMeta(type):

def \_\_new\_\_(mcs, name, bases, attrs, \*\*kwargs):

new\_attrs = {}

for key, value in attrs.items():

if isinstance(value, dict):

new\_attrs.update(value)

else:

new\_attrs[key] = value

return super().\_\_new\_\_(mcs, name, bases, new\_attrs)

class MyClass(metaclass=DictMeta):

def \_\_init\_\_(self, name=None, age=None):

self.name = name

self.age = age

my\_instance = MyClass({'name': 'Alice', 'age': 28})

print(my\_instance.name, my\_instance.age)Code language: Python (python)

In the DictMeta class, the \_\_new\_\_ method should call \_\_init\_\_ instead of obj.\_\_dict\_\_.update(attrs) to properly initialize the attributes of the class. In the MyClass class, the \_\_init\_\_ method should update the \_\_dict\_\_ attribute of the instance with the data parameter to properly initialize the instance attributes.

**Question: This code is supposed to use a generator function to yield all prime numbers up to a given limit. Fix the code.**

def primes(limit):

i = 2

while i < limit:

for j in range(2, i):

if i % j == 0:

break

else:

yield i

i += 1

print(list(primes(20)))Code language: Python (python)

**Answer:** The code is correct and will generate a list of prime numbers up to 20.

**Question: This code is supposed to use a context manager to open a file, write some data to it, and then automatically close the file. Fix the code.**

with open('output.txt', 'w') as f:

f.write('hello, world')Code language: Python (python)

**Answer:**

with open('output.txt', 'w') as f:

f.write('hello, worldn')Code language: JavaScript (javascript)

The code is correct, but it should include a newline character at the end of the string to properly format the output in the file.

Common Python Interview Questions

What is Python? What are the benefits of using Python?

What is the difference between list and tuples in Python?

How is memory managed in Python?

What are the tools that help to find bugs or perform static analysis?

What are Python namespaces? Why are they used?

Explain how to delete a file in Python.

What are python modules? Name some commonly used built-in modules in Python.

What is the difference between Python Arrays and lists?

Is Django better than Flask?

Mention the differences between Django, Pyramid, and Flask.

What is the difference between NumPy and SciPy?

How can you copy an object in Python?

How can you convert a string?

Is Python and multi-threading a good idea? List some ways to get some Python code to run in a parallel way.

What is monkey patching in Python and is it ever a good idea?

What would you say are the most common mistakes made using Python?

You should also be able to define and explain a number of terms, including:

PYTHONPATH

Pickling/unpickling

PEP 8

Flask

Local variables

Global variables

Common Technical Python Interview Questions

You will need to show that you have the right technical skills to back up your knowledge of Python.

Compared to other roles in tech and even software development, interviews for a Python Developer tend to be highly technical, and a hiring manager will ask some pretty specific questions to ensure that you have mastered a variety of different Python-related concepts.

Here is an example of a technical frequently asked Python and data science interview question:

Question: How would you sort a dictionary in Python?

Answer: You use the sorted() function to sort a dictionary in Python (dictionaries are unordered data structures that map keys to values).

The method takes in three parameters: object (required), as well as key and reverse (both optional). The sorted() function in Python is not just useful for dictionaries but can be used to sort any iterable objects by a key, such as lists and tuples.

Additional Technical Interview Questions for Python Jobs

Write a program to produce Fibonacci series in Python.

Write a program in Python to check if a number is prime.

Write a program in Python to check if a sequence is a Palindrome.

How would you display a file’s contents in reversed order?

How would you perform unit testing on your Python code?

How do you keep track of different versions of your code?

Using pseudo-code, reverse a String iteratively and recursively.

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 × 99. Find the largest palindrome made from the product of two 3-digit numbers.

How do you debug a Python program?

How would you sort a dictionary in Python?

What are the data types in Python?

Implement slicing in Python.

Write a Python function and explain what’s going on.

Reverse a string in Python.

Check if a Python string contains another string.

Implement breadth-first search in Python.

Implement depth-first search in Python.

Implement wildcards in Python.

Implement merge sort in Python.

Find the height of a binary search tree.

Common Professional Python Interview Questions

You may be asked questions about your Python programming background and work style. The interviewers are assessing how well you would fit in with the company, so remember to show your passion and enthusiasm.

Top Python interview questions in this area include:

Question: Why did you decide to specialize in the Python programming language?

Answer: Some of the reasons you’ll likely want to highlight include Python’s versatility and diversity of applications, its intuitive nature, and ease of use, or the use of Python by top companies including Google, YouTube, Dropbox, Quora, Mozilla, Spotify, and more.

You’ve already proven your technical know-how when it comes to Python, so treat this as an opportunity to discuss why Python specifically interests you.

Additional Common Python Interview Questions

How long have you spent coding primarily in Python?

Do you use Python in any of your personal projects?

Tell me something you don’t like about Python.

What can you offer us that nobody else can?

Tell me a little about yourself.

How would people describe you?

Would you say you work better independently or on a team?

What’s your biggest weakness?

Describe your work style.

What do you do in your spare time?

What motivates you?

Where do you see yourself in five years?

What project are you most proud of?

What can we expect from you in the first three months?

Do you like to participate in the analysis, design, and deployment phases of a project, or do you prefer to concentrate on the pure development of a well-described task? Why?

Common Situational Python Interview Questions

Whether you’re managing a project or interacting with other teams, strong leadership and communication are needed to thrive as a Python Developer. To test your leadership and communication skills, expect interview questions such as:

Question: Pretend I’m not a tech person. Can you explain [Python concept] in simple terms?

Answer: This might be the toughest question you face during your interview – most Python Developers are adept at explaining complex Python concepts, but not in the straightforward way necessary to teach a layperson about concepts and functions in Python like multiple inheritances, string representation, xrange, and range, or import array.

Try use simple analogies and stay away from technical jargon and concepts. Developers will often have to work with diverse teams of product managers, marketers, designers, and other roles, so the ability to communicate complex technical ideas is important for many companies.

Additional Situational Interview Questions for Python Developers

What do you do if there’s a disagreement within your team?

Are you comfortable giving in-depth presentations?

Are you able to explain complex technologies in simple terms?

What is your preferred way of communication with the team?

How do you convince someone to agree with you?

How to deal with a team member who disagrees with you?

Have you ever worked directly with clients or have been in a customer-facing role in the past? If not, would you like to?

What was the last presentation you gave?

What are the qualities of a successful team or project leader?

Common Behavioral Python Interview Questions

Behavioral Python interview questions focus on how you have handled past work situations. These help to reveal your personality and skills.

Behavioral questions also give the interviewers an idea of how you might behave if a similar situation were to occur in the future.

To answer behavioral interview questions, select a specific example and describe the actions you took and the results you achieved.

Examples of behavioral Python interview questions and answers are:

Question: Do you remember a programming project decision you made that was a failure? Why do you think it was a mistake? What did you learn from the experience?

Answer: The focus of your answer here shouldn’t be on the project you were working on or the bad decision you made, but instead, you should detail how you determined that the decision you made wasn’t the right one, how you corrected course after realizing your mistake, and how this lesson changed the way you approached future projects.

By this point in the interview, you should have already had opportunities to showcase your skill and knowledge as a Python programmer. This question is more about showing your ability as a critical thinker and someone who is capable of both acknowledging and learning from your errors.

More Behavioral Python Interview Questions

What was the most interesting project you have participated in? Can you describe it and tell me why you consider it to be so interesting?

Tell me about a time when you handled a challenging situation.

Tell me about a time when you were in conflict with a peer and how the situation was resolved.

Tell me about a time you had to work under pressure.

Give me an example of a time you made a decision that was unpopular. Explain how you handled implementing it.

Share an example of how you were able to motivate a coworker or your team.

Tell me about a goal you achieved and how you achieved it.

Tell me about a goal you failed to achieve.

Tell me about a time you felt you went above and beyond.

Advanced Python Interview Questions

If you’re curious about the skills and expertise top tech companies are looking for in job candidates, here are a few Python interview questions for experienced professionals from Amazon, Google, Facebook, and Microsoft.

Why do we use join() function in Python?

Give an example of shuffle() method?

How do you find bugs or perform static analysis in a Python application?

What are the different ways to create an empty NumPy array in Python?

What are the principal differences between the lambda and def?

What is the output of L[1:] if L = [1,2,3]?

How would you take input from the user in Python?

How do you terminate a line of code in Python?

Merge two sorted linked lists.

Write a function that counts the total of set bits in a 32-bit integer.

Given a string as your input, delete any recurring character and return the new string.

Tell me about a time when you took a risk and failed.

What is the most complex thing you know a lot about? Teach me about it.

What scares you?

Given a list and a number, find two numbers in the list that sums up to the number given.

What steps do you take to keep your skills current?

What motivates you to excel in a role?

Tell me about a project where your initial assumptions ended up being incorrect. How did you move forward after?

What will be the output of the code below? Explain your answer.

def extendList(val, list=[]):

list.append(val)

return list

list1 = extendList(10)

list2 = extendList(123,[])

list3 = extendList('a')

print "list1 = %s" % list1

print "list2 = %s" % list2

print "list3 = %s" % list3

How would you modify the definition of extendList to produce the presumably desired behavior?

Hide answer

The output of the above code will be:

list1 = [10, 'a']

list2 = [123]

list3 = [10, 'a']

Many will mistakenly expect list1 to be equal to [10] and list3 to be equal to ['a'], thinking that the list argument will be set to its default value of [] each time extendList is called.

However, what actually happens is that the new default list is created only once when the function is defined, and that same list is then used subsequently whenever extendList is invoked without a list argument being specified. This is because expressions in default arguments are calculated when the function is defined, not when it’s called.

list1 and list3 are therefore operating on the same default list, whereas list2 is operating on a separate list that it created (by passing its own empty list as the value for the list parameter).

The definition of the extendList function could be modified as follows, though, to always begin a new list when no list argument is specified, which is more likely to have been the desired behavior:

def extendList(val, list=None):

if list is None:

list = []

list.append(val)

return list

With this revised implementation, the output would be:

list1 = [10]

list2 = [123]

list3 = ['a']

2.

What will be the output of the code below? Explain your answer.

def multipliers():

return [lambda x : i \* x for i in range(4)]

print [m(2) for m in multipliers()]

How would you modify the definition of multipliers to produce the presumably desired behavior?

Hide answer

The output of the above code will be [6, 6, 6, 6] (not [0, 2, 4, 6]).

The reason for this is that Python’s closures are late binding. This means that the values of variables used in closures are looked up at the time the inner function is called. So as a result, when any of the functions returned by multipliers() are called, the value of i is looked up in the surrounding scope at that time. By then, regardless of which of the returned functions is called, the for loop has completed and i is left with its final value of 3. Therefore, every returned function multiplies the value it is passed by 3, so since a value of 2 is passed in the above code, they all return a value of 6 (i.e., 3 x 2).

(Incidentally, as pointed out in The Hitchhiker’s Guide to Python, there is a somewhat widespread misconception that this has something to do with lambdas, which is not the case. Functions created with a lambda expression are in no way special and the same behavior is exhibited by functions created using an ordinary def.)

Below are a few examples of ways to circumvent this issue.

One solution would be use a Python generator as follows:

def multipliers():

for i in range(4): yield lambda x : i \* x

Another solution is to create a closure that binds immediately to its arguments by using a default argument. For example:

def multipliers():

return [lambda x, i=i : i \* x for i in range(4)]

Or alternatively, you can use the functools.partial function:

from functools import partial

from operator import mul

def multipliers():

return [partial(mul, i) for i in range(4)]

Finally, the easiest fix may be to simply replace the return value’s [] with ():

def multipliers():

return (lambda x : i \* x for i in range(4))

3.

What will be the output of the code below? Explain your answer.

class Parent(object):

x = 1

class Child1(Parent):

pass

class Child2(Parent):

pass

print Parent.x, Child1.x, Child2.x

Child1.x = 2

print Parent.x, Child1.x, Child2.x

Parent.x = 3

print Parent.x, Child1.x, Child2.x

Hide answer

The output of the above code will be:

1 1 1

1 2 1

3 2 3

What confuses or surprises many about this is that the last line of output is 3 2 3 rather than 3 2 1. Why does changing the value of Parent.x also change the value of Child2.x, but at the same time not change the value of Child1.x?

The key to the answer is that, in Python, class variables are internally handled as dictionaries. If a variable name is not found in the dictionary of the current class, the class hierarchy (i.e., its parent classes) are searched until the referenced variable name is found (if the referenced variable name is not found in the class itself or anywhere in its hierarchy, an AttributeError occurs).

Therefore, setting x = 1 in the Parent class makes the class variable x (with a value of 1) referenceable in that class and any of its children. That’s why the first print statement outputs 1 1 1.

Subsequently, if any of its child classes overrides that value (for example, when we execute the statement Child1.x = 2), then the value is changed in that child only. That’s why the second print statement outputs 1 2 1.

Finally, if the value is then changed in the Parent (for example, when we execute the statement Parent.x = 3), that change is reflected also by any children that have not yet overridden the value (which in this case would be Child2). That’s why the third print statement outputs 3 2 3.

4.

What will be the output of the code below in Python 2? Explain your answer.

def div1(x,y):

print "%s/%s = %s" % (x, y, x/y)

def div2(x,y):

print "%s//%s = %s" % (x, y, x//y)

div1(5,2)

div1(5.,2)

div2(5,2)

div2(5.,2.)

Also, how would the answer differ in Python 3 (assuming, of course, that the above print statements were converted to Python 3 syntax)?

Hide answer

In Python 2, the output of the above code will be:

5/2 = 2

5.0/2 = 2.5

5//2 = 2

5.0//2.0 = 2.0

By default, Python 2 automatically performs integer arithmetic if both operands are integers. As a result, 5/2 yields 2, while 5./2 yields 2.5.

Note that you can override this behavior in Python 2 by adding the following import:

from \_\_future\_\_ import division

Also note that the “double-slash” (//) operator will always perform integer division, regardless of the operand types. That’s why 5.0//2.0 yields 2.0 even in Python 2.

Python 3, however, does not have this behavior; i.e., it does not perform integer arithmetic if both operands are integers. Therefore, in Python 3, the output will be as follows:

5/2 = 2.5

5.0/2 = 2.5

5//2 = 2

5.0//2.0 = 2.0

5.

What will be the output of the code below?

list = ['a', 'b', 'c', 'd', 'e']

print list[10:]

Hide answer

The above code will output [], and will not result in an IndexError.

As one would expect, attempting to access a member of a list using an index that exceeds the number of members (e.g., attempting to access list[10] in the list above) results in an IndexError. However, attempting to access a slice of a list at a starting index that exceeds the number of members in the list will not result in an IndexError and will simply return an empty list.

What makes this a particularly nasty gotcha is that it can lead to bugs that are really hard to track down since no error is raised at runtime.

6.

Consider the following code snippet:

1. list = [ [ ] ] \* 5

2. list # output?

3. list[0].append(10)

4. list # output?

5. list[1].append(20)

6. list # output?

7. list.append(30)

8. list # output?

What will be the ouput of lines 2, 4, 6, and 8? Explain your answer.

Hide answer

The output will be as follows:

[[], [], [], [], []]

[[10], [10], [10], [10], [10]]

[[10, 20], [10, 20], [10, 20], [10, 20], [10, 20]]

[[10, 20], [10, 20], [10, 20], [10, 20], [10, 20], 30]

Here’s why:

The first line of output is presumably intuitive and easy to understand; i.e., list = [ [ ] ] \* 5 simply creates a list of 5 lists.

However, the key thing to understand here is that the statement list = [ [ ] ] \* 5 does NOT create a list containing 5 distinct lists; rather, it creates a a list of 5 references to the same list. With this understanding, we can better understand the rest of the output.

list[0].append(10) appends 10 to the first list. But since all 5 lists refer to the same list, the output is: [[10], [10], [10], [10], [10]].

Similarly, list[1].append(20) appends 20 to the second list. But again, since all 5 lists refer to the same list, the output is now: [[10, 20], [10, 20], [10, 20], [10, 20], [10, 20]].

In contrast, list.append(30) is appending an entirely new element to the “outer” list, which therefore yields the output: [[10, 20], [10, 20], [10, 20], [10, 20], [10, 20], 30].

7.

Given a list of N numbers, use a single list comprehension to produce a new list that only contains those values that are:

(a) even numbers, and

(b) from elements in the original list that had even indices

For example, if list[2] contains a value that is even, that value should be included in the new list, since it is also at an even index (i.e., 2) in the original list. However, if list[3] contains an even number, that number should not be included in the new list since it is at an odd index (i.e., 3) in the original list.

Hide answer

A simple solution to this problem would be as follows

[x for x in list[::2] if x%2 == 0]

For example, given the following list:

# 0 1 2 3 4 5 6 7 8

list = [ 1 , 3 , 5 , 8 , 10 , 13 , 18 , 36 , 78 ]

the list comprehension [x for x in list[::2] if x%2 == 0] will evaluate to:

[10, 18, 78]

The expression works by first taking the numbers that are at the even indices, and then filtering out all the odd numbers.

8.

Given the following subclass of dictionary:

class DefaultDict(dict):

def \_\_missing\_\_(self, key):

return []

Will the code below work? Why or why not?

d = DefaultDict()

d['florp'] = 127

Hide answer

Actually, the code shown will work with the standard dictionary object in python 2 or 3—that is normal behavior. Subclassing dict is unnecessary. However, the subclass still won’t work with the code shown because \_\_missing\_\_ returns a value but does not change the dict itself:

d = DefaultDict()

print d

{}

print d['foo']

[]

print d

{}

So it will “work,” in the sense that it won’t produce any error, but doesn’t do what it seems to be intended to do.

Here is a \_\_missing\_\_-based method that will update the dictionary, as well as return a value:

class DefaultDict(dict):

def \_\_missing\_\_(self, key):

newval = []

self[key] = newval

return newval

But since version 2.5, a defaultdict object has been available in the collections module (in the standard library.)

9.

How would you unit-test the following code?

async def logs(cont, name):

conn = aiohttp.UnixConnector(path="/var/run/docker.sock")

async with aiohttp.ClientSession(connector=conn) as session:

async with session.get(f"http://xx/containers/{cont}/logs?follow=1&stdout=1") as resp:

async for line in resp.content:

print(name, line)

Hide answer

A good answer would suggest a specific async mock library and async test case approach, including an ephemeral event loop that’s guaranteed to terminate (i.e. with a max number of steps before timeout.)

A great answer would point out that synchronisation problems are fundamentally the same in synchronous and asynchronous code, the difference being preemption granularity.

A beautiful answer would take into account that the above code only has one flow (easy) compared to some other code where flows are mixed (e.g. merging two streams into one, sorting, etc). For example, consider following upgrade to the given code:

keep\_running = True

async def logs(cont, name):

conn = aiohttp.UnixConnector(path="/var/run/docker.sock")

async with aiohttp.ClientSession(connector=conn) as session:

async with session.get(f"http://xx/containers/{cont}/logs?follow=1&stdout=1") as resp:

async for line in resp.content:

if not keep\_running:

break

print(name, line)

Here, any of the async statements could have a side-effect of changing the global keep\_running.

10.

How do you list the functions in a module?

Hide answer

Use the dir() method to list the functions in a module.

For example:

import some\_module

print dir(some\_module)

11.

Write a function that prints the least integer that is not present in a given list and cannot be represented by the summation of the sub-elements of the list.

E.g. For a = [1,2,5,7] the least integer not represented by the list or a slice of the list is 4, and if a = [1,2,2,5,7] then the least non-representable integer is 18.

Hide answer

import itertools

sum\_list = []

stuff = [1,2, 5, 7]

for L in range(0, len(stuff)+1):

for subset in itertools.combinations(stuff, L):

sum\_list.append(sum(subset))

new\_list = list(set(sum\_list))

new\_list.sort()

for each in range(0,new\_list[-1]+2):

if each not in new\_list:

print(each)

break

### **Iterate With enumerate() Instead of range()**

This scenario might come up more than any other in coding interviews: you have a list of elements, and you need to iterate over the list with access to both the indices and the values.

There’s a classic coding interview question named FizzBuzz that can be solved by iterating over both indices and values. In FizzBuzz, you are given a list of integers. Your task is to do the following:

1. Replace all integers that are evenly divisible by 3 with "fizz"
2. Replace all integers divisible by 5 with "buzz"
3. Replace all integers divisible by both 3 and 5 with "fizzbuzz"

Often, developers will solve this problem with range():

>>> numbers = [45, 22, 14, 65, 97, 72]

>>> for i in range(len(numbers)):

... if numbers[i] % 3 == 0 and numbers[i] % 5 == 0:

... numbers[i] = 'fizzbuzz'

... elif numbers[i] % 3 == 0:

... numbers[i] = 'fizz'

... elif numbers[i] % 5 == 0:

... numbers[i] = 'buzz'

...

>>> numbers

['fizzbuzz', 22, 14, 'buzz', 97, 'fizz']

Range allows you to access the elements of numbers by index and is a useful tool [for some situations](https://realpython.com/python-range/). But in this case, where you want to get each element’s index and value at the same time, a more elegant solution uses [enumerate()](https://realpython.com/python-enumerate/):

>>> numbers = [45, 22, 14, 65, 97, 72]

>>> for i, num in enumerate(numbers):

... if num % 3 == 0 and num % 5 == 0:

... numbers[i] = 'fizzbuzz'

... elif num % 3 == 0:

... numbers[i] = 'fizz'

... elif num % 5 == 0:

... numbers[i] = 'buzz'

...

>>> numbers

['fizzbuzz', 22, 14, 'buzz', 97, 'fizz']

For each element, enumerate() returns a counter and the element value. The counter defaults to 0, which conveniently is also the element’s index. Don’t want to start your count at 0? Just use the optional start parameter to set an offset:

>>> numbers = [45, 22, 14, 65, 97, 72]

>>> for i, num in enumerate(numbers, start=52):

... print(i, num)

...

52 45

53 22

54 14

55 65

56 97

57 72

By using the start parameter, we access all of the same elements, starting with the first index, but now our count starts from the specified integer value.

### **Use List Comprehensions Instead of map() and filter()**

“I think dropping filter() and map() is pretty uncontroversial[.]”

— Guido van Rossum, Python’s creator

He may have been wrong about it being uncontroversial, but Guido had good reasons for wanting to remove [map()](https://realpython.com/python-map-function/) and [filter()](https://realpython.com/python-filter-function/) from Python. One reason is that Python supports [list comprehensions](https://realpython.com/list-comprehension-python/), which are often easier to read and support the same functionality as map() and filter().

Let’s first take a look at how we’d structure a call to map() and the equivalent list comprehension:

>>> numbers = [4, 2, 1, 6, 9, 7]

>>> def square(x):

... return x\*x

...

>>> list(map(square, numbers))

[16, 4, 1, 36, 81, 49]

>>> [square(x) for x in numbers]

[16, 4, 1, 36, 81, 49]

Both approaches, using map() and the list comprehension, return the same values, but the list comprehension is easier to read and understand.

Now we can do the same thing for the filter() and its equivalent list comprehension:

>>> def is\_odd(x):

... return bool(x % 2)

...

>>> list(filter(is\_odd, numbers))

[1, 9, 7]

>>> [x for x in numbers if is\_odd(x)]

[1, 9, 7]

Like we saw with map(), the filter() and list comprehension approaches return the same value, but the list comprehension is easier to follow.

Developers who come from other languages may disagree that list comprehensions are easier to read than map() and filter(), but in my experience beginners are able to pick up list comprehensions much more intuitively.

Either way, you’ll rarely go wrong using list comprehensions in a coding interview, as it will communicate that you know what’s most common in Python.

### **Debug With breakpoint() Instead of print()**

You may have debugged a small problem by adding [print()](https://realpython.com/python-print/) to your code and seeing what was printed out. This approach works well at first but quickly becomes cumbersome. Plus, in a coding interview setting, you hardly want [print() calls](https://realpython.com/courses/python-print/) peppered throughout your code.

Instead, you should be using a [debugger](https://realpython.com/python-debug-idle/). For non-trivial bugs, it’s almost always faster than using print(), and given that debugging is a big part of writing software, it shows that you know how to use tools that will let you develop quickly on the job.

If you’re using Python 3.7, you don’t need to import anything and can just call [breakpoint()](https://realpython.com/python37-new-features/#the-breakpoint-built-in) at the location in your code where you’d like to drop into the debugger:

# Some complicated code with bugs

breakpoint()

Calling breakpoint() will put you into [pdb](https://realpython.com/python-debugging-pdb/), which is the default Python debugger. On Python 3.6 and older, you can do the same by importing pdb explicitly:

import pdb; pdb.set\_trace()

Like breakpoint(), pdb.set\_trace() will put you into the pdb debugger. It’s just not quite as clean and is a tad more to remember.

There are other debuggers available that you may want to try, but pdb is part of the standard library, so it’s always available. Whatever debugger you prefer, it’s worth trying them out to get used to the workflow before you’re in a coding interview setting.

### **Format strings With f-Strings**

Python has a lot of different ways to handle string formatting, and it can be tricky to know what to use. In fact, we tackle formatting in depth in two separate articles: one about [string formatting in general](https://realpython.com/python-string-formatting/) and one specifically [focused on f-strings](https://realpython.com/python-f-strings/). In a coding interview, where you’re (hopefully) using Python 3.6+, the suggested formatting approach is Python’s f-strings.

f-strings support use of the [string formatting mini-language](https://docs.python.org/3/library/string.html#format-specification-mini-language), as well as powerful string interpolation. These features allow you to add variables or even valid Python expressions and have them evaluated at runtime before being added to the [string](https://realpython.com/python-strings/):

>>> def get\_name\_and\_decades(name, age):

... return f"My name is {name} and I'm {age / 10:.5f} decades old."

...

>>> get\_name\_and\_decades("Maria", 31)

My name is Maria and I'm 3.10000 decades old.

The f-string allows you to put Maria into the string and add her age with the desired formatting in one succinct operation.

The one risk to be aware of is that if you’re outputting user-generated values, then that can introduce security risks, in which case [Template Strings](https://realpython.com/python-string-formatting/#4-template-strings-standard-library) may be a safer option.

### **Sort Complex Lists With sorted()**

Plenty of coding interview questions require some kind of sorting, and there are multiple valid ways you can sort items. Unless the interviewer wants you to implement your own [sorting algorithm](https://realpython.com/sorting-algorithms-python/), it’s usually best to use [sorted()](https://realpython.com/python-sort/).

You’ve probably seen the most simple uses of sorting, such as sorting lists of [numbers](https://realpython.com/python-numbers/) or strings in ascending or descending order:

>>> sorted([6,5,3,7,2,4,1])

[1, 2, 3, 4, 5, 6, 7]

>>> sorted(['cat', 'dog', 'cheetah', 'rhino', 'bear'], reverse=True)

['rhino', 'dog', 'cheetah', 'cat', 'bear]

By default, sorted() has sorted the input in ascending order, and the reverse keyword argument causes it to sort in descending order.

It’s worth knowing about the optional keyword argument key that lets you specify a function that will be called on every element prior to sorting. Adding a function allows custom sorting rules, which are especially helpful if you want to sort more complex data types:

>>> animals = [

... {'type': 'penguin', 'name': 'Stephanie', 'age': 8},

... {'type': 'elephant', 'name': 'Devon', 'age': 3},

... {'type': 'puma', 'name': 'Moe', 'age': 5},

... ]

>>> sorted(animals, key=lambda animal: animal['age'])

[

{'type': 'elephant', 'name': 'Devon', 'age': 3},

{'type': 'puma', 'name': 'Moe', 'age': 5},

{'type': 'penguin', 'name': 'Stephanie, 'age': 8},

]

By passing in a [lambda function](https://realpython.com/python-lambda/) that returns each element’s age, you can easily sort a list of dictionaries by a single value of each of those dictionaries. In this case, the dictionary is now sorted in ascending order by age.

## Leverage Data Structures Effectively

Algorithms get a lot of attention in coding interviews, but data structures are arguably even more important. In a coding interviewing context, picking the right data structure can have a major impact on performance.

Beyond theoretical data structures, Python has powerful and convenient functionality built into its standard data structure implementations. These data structures are incredibly useful in coding interviews because they give you lots of functionality by default and let you focus your time on other parts of the problem.

### **Store Unique Values With Sets**

You’ll often need to remove duplicate elements from an existing dataset. New developers will sometimes do so with lists when they should be using sets, which enforce the uniqueness of all elements.

Pretend you have a function named get\_random\_word(). It will always return a random selection from a small set of words:

>>> import random

>>> all\_words = "all the words in the world".split()

>>> def get\_random\_word():

... return random.choice(all\_words)

You’re supposed to call get\_random\_word() repeatedly to get 1000 random words and then return a data structure containing every unique word. Here are two common, suboptimal approaches and one good approach.

**Bad Approach**

get\_unique\_words() stores values in a list then converts the list into a set:

>>> def get\_unique\_words():

... words = []

... for \_ in range(1000):

... words.append(get\_random\_word())

... return set(words)

>>> get\_unique\_words()

{'world', 'all', 'the', 'words'}

This approach isn’t terrible, but it unnecessarily creates a list and then converts it to a set. Interviewers almost always notice (and ask about) this type of design choice.

**Worse Approach**

To avoid converting from a list to a set, you now store values in a list without using any other data structures. You then test for uniqueness by comparing new values with all elements currently in the list:

>>> def get\_unique\_words():

... words = []

... for \_ in range(1000):

... word = get\_random\_word()

... if word not in words:

... words.append(word)

... return words

>>> get\_unique\_words()

['world', 'all', 'the', 'words']

This is worse than the first approach, because you have to compare every new word against every word already in the list. That means that as the number of words grows, the number of lookups grows quadratically. In other words, the time complexity grows on the order of O(N²).

**Good Approach**

Now, you skip using lists altogether and instead use a set from the start:

>>> def get\_unique\_words():

... words = set()

... for \_ in range(1000):

... words.add(get\_random\_word())

... return words

>>> get\_unique\_words()

{'world', 'all', 'the', 'words'}

This may not look much different than the other approaches except for making use of a set from the beginning. If you consider what’s happening within .add(), it even sounds like the second approach: get the word, check if it’s already in the set, and if not, add it to the data structure.

So why is using a set different from the second approach?

It’s different because sets store elements in a manner that allows near-constant-time checks whether a value is in the set or not, unlike lists, which require linear-time lookups. The difference in lookup time means that the [time complexity](https://stackoverflow.com/a/487278/2141768) for adding to a set grows at a rate of O(N), which is much better than the O(N²) from the second approach in most cases.

### **Save Memory With Generators**

List comprehensions are convenient tools but can sometimes lead to unnecessary memory usage.

Imagine you’ve been asked to find the sum of the first 1000 perfect squares, starting with 1. You know about list comprehensions, so you quickly code up a working solution:

>>> sum([i \* i for i in range(1, 1001)])

333833500

Your solution makes a list of every perfect square between 1 and 1,000,000 and sums the values. Your code returns the right answer, but then your interviewer starts increasing the number of perfect squares you need to sum.

At first, your function keeps popping out the right answer, but soon it starts slowing down until eventually the process seems to hang for an eternity. This is the last thing you want happening in a coding interview.

What’s going on here?

It’s making a list of every perfect square you’ve requested and summing them all. A list with 1000 perfect squares may not be large in computer-terms, but 100 million or 1 billion is quite a bit of information and can easily overwhelm your computer’s available memory resources. That’s what’s happening here.

Thankfully, there’s a quick way to solve the memory problem. You just replace the brackets with parentheses:

>>> sum((i \* i for i in range(1, 1001)))

333833500

Swapping out the brackets changes your list comprehension into a [generator expression](https://realpython.com/introduction-to-python-generators/). Generator expressions are perfect for when you know you want to retrieve data from a sequence, but you don’t need to access all of it at the same time.

Instead of creating a list, the [generator expression](https://realpython.com/courses/python-generators/) returns a generator object. That object knows where it is in the current state (for example, i = 49) and only calculates the next value when it’s asked for.

So when sum iterates over the generator object by calling .\_\_next\_\_() repeatedly, the generator checks what i equals, calculates i \* i, increments i internally, and returns the proper value to sum. The design allows generators to be used on massive sequences of data, because only one element exists in memory at a time.

### **Define Default Values in Dictionaries With .get() and .setdefault()**

One of the most common programming tasks involves adding, modifying, or retrieving an item that may or may not be in a dictionary. Python dictionaries have elegant functionality to make these tasks clean and easy, but developers often check explicitly for values when it isn’t necessary.

Imagine you have a dictionary named cowboy, and you want to get that cowboy’s name. One approach is to explicitly check for the key with a conditional:

>>> cowboy = {'age': 32, 'horse': 'mustang', 'hat\_size': 'large'}

>>> if 'name' in cowboy:

... name = cowboy['name']

... else:

... name = 'The Man with No Name'

...

>>> name

'The Man with No Name'

This approach first checks if the name key exists in the dictionary, and if so, it returns the corresponding value. Otherwise, it returns a default value.

While explicitly checking for keys does work, it can easily be replaced with one line if you use .get():

>>> name = cowboy.get('name', 'The Man with No Name')

get() performs the same operations that were performed in the first approach, but now they’re handled automatically. If the key exists, then the proper value will be returned. Otherwise, the default value will get returned.

But what if you want to update the dictionary with a default value while still accessing the name key? .get() doesn’t really help you here, so you’re left with explicitly checking for the value again:

>>> if 'name' not in cowboy:

... cowboy['name'] = 'The Man with No Name'

...

>>> name = cowboy['name']

Checking for the value and setting a default is a valid approach and is easy to read, but again Python offers a more elegant method with .setdefault():

>>> name = cowboy.setdefault('name', 'The Man with No Name')

.setdefault() accomplishes exactly the same thing as the snippet above. It checks if name exists in cowboy, and if so it returns that value. Otherwise, it sets cowboy['name'] to The Man with No Name and returns the new value.

## Take Advantage of Python’s Standard Library

By default, Python comes with a lot of functionality that’s just an import statement away. It’s powerful on its own, but knowing how to leverage the standard library can supercharge your coding interview skills.

It’s hard to pick the most useful pieces from all of the available modules, so this section will focus on just a small subset of its utility functions. Hopefully, these will prove useful to you in coding interviews and also whet your appetite to learn more about the advanced functionality of these and other modules.

### **Handle Missing Dictionary Keys With collections.defaultdict()**

.get() and .setdefault() work well when you’re setting a default for a single key, but it’s common to want a default value for all possible unset keys, especially when programming in a coding interview context.

Pretend you have a group of students, and you need to keep track of their grades on homework assignments. The input value is a list of tuples with the format (student\_name, grade), but you want to easily look up all the grades for a single student without iterating over the list.

One way to store the grade data uses a dictionary that maps student names to lists of grades:

>>> student\_grades = {}

>>> grades = [

... ('elliot', 91),

... ('neelam', 98),

... ('bianca', 81),

... ('elliot', 88),

... ]

>>> for name, grade in grades:

... if name not in student\_grades:

... student\_grades[name] = []

... student\_grades[name].append(grade)

...

>>> student\_grades

{'elliot': [91, 88], 'neelam': [98], 'bianca': [81]}

In this approach, you iterate over the students and check if their names are already properties in the dictionary. If not, you add them to the dictionary with an empty list as the default value. You then [append](https://realpython.com/python-append/) their actual grades to that student’s list of grades.

But there’s an even cleaner approach that uses a defaultdict, which extends standard dict functionality to allow you to set a default value that will be operated upon if the key doesn’t exist:

>>> from collections import defaultdict

>>> student\_grades = defaultdict(list)

>>> for name, grade in grades:

... student\_grades[name].append(grade)

In this case, you’re creating a defaultdict that uses the list() constructor with no arguments as a default factory method. list() with no arguments returns an empty list, so defaultdict calls list() if the name doesn’t exist and then allows the grade to be appended. If you want to get fancy, you could also use a lambda function as your factory value to return an arbitrary constant.

Leveraging a defaultdict can lead to cleaner application code because you don’t have to worry about default values at the key level. Instead, you can handle them once at the defaultdict level and afterwards act as if the key is always present. For more information on this technique, check out [Using the Python defaultdict Type for Handling Missing Keys](https://realpython.com/python-defaultdict/).

### **Count Hashable Objects With collections.Counter**

You have a long string of words with no punctuation or capital letters and you want to count how many times each word appears.

You could use a dictionary or defaultdict and increment the counts, but collections.Counter provides a cleaner and more convenient way to do exactly that. Counter is a subclass of dict that uses 0 as the default value for any missing element and makes it easier to count occurrences of objects:

>>> from collections import Counter

>>> words = "if there was there was but if \

... there was not there was not".split()

>>> counts = Counter(words)

>>> counts

Counter({'if': 2, 'there': 4, 'was': 4, 'not': 2, 'but': 1})

When you pass in the list of words to Counter, it stores each word along with a count of how many times that word occurred in the list.

Are you curious what the two most common words were? Just use .most\_common():

>>> counts.most\_common(2)

[('there', 4), ('was', 4)]

.most\_common() is a convenience method and simply returns the n most frequent inputs by count.

### **Access Common String Groups With string Constants**

It’s trivia time! Is 'A' > 'a' true or false?

It’s false, because the ASCII code for A is 65, but a is 97, and 65 is not greater than 97.

Why does the answer matter? Because if you want to check if a character is part of the English alphabet, one popular way is to see if it’s between A and z (65 and 122 on the ASCII chart).

Checking the ASCII code works but is clumsy and easy to mess up in coding interviews, especially if you can’t remember whether lowercase or uppercase ASCII characters come first. It’s much easier to use the constants defined as part of the [string module](https://docs.python.org/3/library/string.html).

You can see one in use in is\_upper(), which returns whether all characters in a string are uppercase letters:

>>> import string

>>> def is\_upper(word):

... for letter in word:

... if letter not in string.ascii\_uppercase:

... return False

... return True

...

>>> is\_upper('Thanks Geir')

False

>>> is\_upper('LOL')

True

is\_upper() iterates over the letters in word, and checks if the letters are part of string.ascii\_uppercase. If you print out string.ascii\_uppercase you’ll see that it’s just a lowly string. The value is set to the literal 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.

All string constants are just strings of frequently referenced string values. They include the following:

* string.ascii\_letters
* string.ascii\_uppercase
* string.ascii\_lowercase
* string.digits
* string.hexdigits
* string.octdigits
* string.punctuation
* string.printable
* string.whitespace

These are easier to use and, even more importantly, easier to read.

### **Generate Permutations and Combinations With itertools**

Interviewers love to give real life scenarios to make coding interviews seem less intimidating, so here’s a contrived example: you go to an amusement park and decide to figure out every possible pair of friends that could sit together on a roller coaster.

Unless generating these pairs is the primary purpose of the interview question, it’s likely that generating all the possible pairs is just a tedious step on the way towards a working algorithm. You could calculate them yourself with [nested for loops](https://realpython.com/nested-loops-python/), or you could use the powerful [itertools library](https://realpython.com/python-itertools/).

itertools has multiple tools for generating iterable sequences of input data, but right now we’ll just focus on two common functions: itertools.permutations() and itertools.combinations().

itertools.permutations() builds a list of all permutations, meaning it’s a list of every possible grouping of input values with a length matching the count parameter. The r keyword argument lets us specify how many values go in each grouping:

>>> import itertools

>>> friends = ['Monique', 'Ashish', 'Devon', 'Bernie']

>>> list(itertools.permutations(friends, r=2))

[('Monique', 'Ashish'), ('Monique', 'Devon'), ('Monique', 'Bernie'),

('Ashish', 'Monique'), ('Ashish', 'Devon'), ('Ashish', 'Bernie'),

('Devon', 'Monique'), ('Devon', 'Ashish'), ('Devon', 'Bernie'),

('Bernie', 'Monique'), ('Bernie', 'Ashish'), ('Bernie', 'Devon')]

With permutations, the order of the elements matters, so ('sam', 'devon') represents a different pairing than ('devon', 'sam'), meaning that they would both be included in the list.

itertools.combinations() builds combinations. These are also the possible groupings of the input values, but now the order of the values doesn’t matter. Because ('sam', 'devon') and ('devon', 'sam') represent the same pair, only one of them would be included in the output list:

>>> list(itertools.combinations(friends, r=2))

[('Monique', 'Ashish'), ('Monique', 'Devon'), ('Monique', 'Bernie'),

('Ashish', 'Devon'), ('Ashish', 'Bernie'), ('Devon', 'Bernie')]

Since the order of the values does not matter with combinations, there are fewer combinations than permutations for the same input list. Again, because we set r to 2, each grouping has two names in it.

.combinations() and .permutations() are just small examples of a powerful library, but even these two functions can be quite useful when you’re trying to solve an algorithm problem quickly.

## 1.****What type of language is Python?****

Python is a general-purpose, object-oriented language. It is also an interpreted language.

## 2. ****What are some features of the Python language?****

•      Python is an interpreted language, meaning that Python code does not need to compile before running.

•      Python is a dynamically typed language. This means you don’t need to declare the type of variable when you create it. Python will determine its type dynamically.

•      Python is object-oriented. With Python, you can define classes and use composition and inheritance. Python does not support access specifiers like public and private.

•      Functions in Python are first-class objects. This means that you can assign them to a variable. Python functions can also return other functions or accept them as parameters.

•      Python is a general-purpose language, which is very popular in many industries. Developers use it in automation, web applications, machine learning, big data, and more.

•      It is easy and quick to write code in Python, but Python code generally runs slower than compiled languages.

## 3. ****What is PEP8?****

[PEP8](https://pep8.org/) is the latest set of Python coding standards. It is a document that provides guidelines and best practices for how to write Python code. Its primary focus is to improve the readability and consistency of Python code.

## 4. ****What is the difference between a list and a tuple in Python?****

Both lists and tuples can store an ordered array of objects, however, a tuple is immutable. This means that once a tuple is created with objects in it, the objects can not be swapped out (mutating). A list still allows for objects to be reassigned within the list.

## 5. ****How does Python manage memory?****

Python manages memory in the Python private heap space. Pythons memory manager and garbage collector control the private heap space. There are multiple levels of scope that can be used in conjunction with namespace, including Built-In, Global, Enclosed, and Local.

## 6. ****What is a Python namespace?****

Namespaces in Python ensure that Python variables, functions, and other names don’t clash.

## 7. ****What is the purpose of the PYTHONPATH variable?****

PYTHONPATH is an environmental variable that will tell the operating system where to find Python libraries. This will ensure that your Operating System calls the correct installation of Python on the computer.

## 8. ****What is the purpose of the PYTHONSTARTUP variable?****

This variable contains the path of an initialization file that houses Python source code. It executes each time the Python interpreter starts.

## 9. ****What is the purpose of the PYTHONCASEOK variable?****

This is a variable specifically for running Python on the Windows operating system. It tells the Python interpreter to use case-insensitive match when trying to find imported libraries.

## 10. ****What is the purpose of the PYTHONHOME variable?****

This is an alternate path to search for Python modules.

## 11. ****What are Python modules?****

You can consider a Python module as a code library. A module is a collection of Python classes, functions, and variables. Modules are usually separated from each other when they supply different functionality.

## 12. ****What are some common modules in the Python standard library?****

You will use some of these modules that are included in the[Python standard library](https://docs.python.org/3/library/) often when programming in Python:

•      Email: Used to parse, handle, and generate email messages.

•      String: An index of types of strings, such as all capital or lowercase letters.

•      Sqlite3: Used to deal with the SQLite database.

•      XML: Provides XML support.

•      Logging: Creates logging classes to log system details.

•      Traceback: Allows you to extract and print stack trace details.

## 13. ****What is scope in Python?****

All objects in Python have a scope. Scope is the block of code where the variable is accessible. Here are some scope designations in Python:

•      **Local scope:** Local objects available in the current function.

•      **Global scope:** Objects available through the code execution since their inception.

•      **Module-level scope:** Global objects of the current module accessible in the program.

•      **Outermost scope:** Built-in names callable in the program.

## 14. ****How would you get a user’s home directory (~) in Python?****

You can do this with the os module.

import os

print(os.path.expanduser('~'))

This will return the path to the current user’s directory.

## 15. ****What built-in types does Python support?****

•      Number

•      String

•      Tuple

•      List

•      Dictionary

•      Set

## 16. ****What is a Python decorator?****

A Python decorator is a function that extends the behavior of another Python function without explicitly modifying it.

## 17. ****What is pickling/unpickling in Python?****

Pickling is when a Python object converts into a string representation by a pickle module. It is then placed into a file with the dump() function. Unpickling refers to the reverse process, in which the stored string is retrieved and turned back into an object.

## 18. ****What is a negative index in Python?****

While positive indices begin with position 0 and follow with 1, 2, etc., negative indices end with -1. -2, etc.; -3 is the position before that, and so on. Negative indexes can access elements in a Python list from the end of the list rather than from the beginning.

## 19. ****How are global and local variables defined in Python?****

In general, variables defined outside of functions are global. Variables defined inside a function can also be made global by using the command global x, for example, to create a global variable called x.

## 20. ****Describe a few ways to generate a random number in Python.****

Python comes with a variety of techniques to generate random numbers using its Random library: https://docs.python.org/3/library/random.html

•      Random(): This command returns a floating-point number between 0 and 1.

•      Uniform(x, y): This command returns a floating-point number between the values given for x and y.

•      Randint(x, y): This command returns a random integer between the values given for x and y.

## 21. ****What is a Python dictionary?****

A Python dictionary is one of the standard data structures in Python. It is an unordered collection of elements, acting as a hash-table. These elements are stored within key-value pairs. Dictionaries are indexed by keys.

# Python dictionary

dict={'first':'Bob', 'last':'Smith'}

## 22. ****Is Python case sensitive?****

Yes, Python is a case-sensitive language.

## 23. ****What is the output of the following code?****

print(var) if var='Hello Everyone!'

The output would be:

Hello Everyone!

## 24. ****What is the output of the following code?****

print(var[0]) if var='Hello Everyone!'

The output would be:

H

## 25. ****What is the output of the following code?****

print(var[2:5]) if var='Hello Everyone!'

The output would be:

llo

## 26. ****How would you get all the keys in a Python dictionary?****

You would use the dictionary keys function.

dict={'first':'Bob', 'last':'Smith'}

all\_keys=dict.keys()

## 27. ****How would you get all the values in a Python dictionary?****

You would use the dictionary values function.

dict={'first':'Bob', 'last':'Smith'}

all\_values=dict.values()

## 28. ****How do you convert a string to an integer in python?****

The Python int function will convert a string to an integer. The second parameter is for the base of the number, so 10 for decimal numbers.

x = '100'

int(x, 10)

## 29. ****How do you convert a string to a long in python?****

The Python long function will convert a string to a long. The second parameter is for the base of the number, so 10 for decimal numbers.

x = '100'

long(x, 10)

## 30. ****How do you convert a string to a floating-point number in python?****

The Python float function will convert a string to a floating-point number.

x = '100.12345'

float(x)

## 31. ****How do you convert an object to a string in python?****

The Python str function will convert an object to a string.

dict={'first':'Bob', 'last':'Smith'}

str(dict)

## 32. ****What does the \*\* operator do in Python?****

The \*\* operator performs exponential calculations in Python.

# x equals 8

x=2\*\*3

## 33. ****What does the // operator do in Python?****

The // operator performs floor division. The result will be the whole number in front of the decimal point.

# x equals 2

x=8//3

## 34. ****What does the Python is operator do?****

When the is operator is used in a Python expression, it evaluates to true if the variables on both sides point to the same object.

## z is true

x=3

y=x

z=x is y

## 35. ****What does the Python not in operator do?****

The Python not in operator evaluates to true if it does not find a variable in the specified sequence and false if it does not.

# The following will return true

4 not in [1,2,3]

## 36. ****What does the Python break statement do?****

The Python break statement stops a loop statement and transfers execution of the code to the statement after the loop.

## 37. ****What does the Python continue statement do?****

The Python continue statement causes a loop to skip the rest of its body and continue at the beginning of the loop.

## 38. ****What does the Python pass statement do?****

The Python pass statement is used when a statement is required, but you do not want a command or code to execute.

## 39. ****What are Python dictionary and list comprehensions?****

Comprehensions in Python are like decorators. They are what is called syntactic sugar that helps create filtered and modified lists and dictionaries from existing lists, dictionaries, or sets. They are handy and quick because you can write them in one line of code.

the\_list = [2, 3, 5, 7, 11]

# list comprehension

squared\_list = [x\*\*2 for x in the\_list]

# output is [4 , 9 , 25 , 49 , 121]

# dict comprehension

squared\_dict = {x:x\*\*2 for x in the\_list}

# output is {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}

## 40. ****What are Python lambdas and why are they used?****

Python lambdas are anonymous functions. They can accept multiple arguments but only have a single expression. They are used where a function is needed temporarily and will not be needed by other code. They are also limited in their scope of capabilities compared to a full function.

multiply = lambda x, y : x \* y

print(multiply(2, 5)) # outputs 10

## 41. ****How do you copy an object in Python?****

While the = operator will copy many things in Python, it will not copy a Python object. It only creates a reference to the object. To create a copy of an object in Python, you need to use the copy module. The copy module offers two ways of copying an object.

•      Shallow copy: Copies an object and re-use references from the old object

•      Deep copy: Copies all the values in an object recursively.

from copy import copy, deepcopy

list\_1 = [1, 2, [3, 5], 4]

## shallow

list\_2 = copy(list\_1)

list\_2[3] = 11

list\_2[2].append(12)

list\_2 # output => [1, 2, [3, 5, 12], 11]

list\_1 # output => [1, 2, [3, 5, 12], 4]

## deep

list\_3 = deepcopy(list\_1)

list\_3[3] = 10

list\_3[2].append(13)

list\_3 # output => [1, 2, [3, 5, 6, 13], 10]

list\_1 # output => [1, 2, [3, 5, 6], 4]

## 42. ****What is self in Python?****

In Python, self represents the object or instance of a class when referring to itself internally. In other languages, this variable is often called this. In Python classes, you must explicitly pass self as the first parameter in methods.

class Car():

def \_\_init\_\_(self, model, color):

self.model = model

self.color = color

def display(self):

print("Model is", self.model )

print("color is", self.color )

## 43. ****What is \_\_init\_\_ in Python?****

In Python, the \_\_init\_\_ method is used as a constructor for classes. Whenever a new instance of a class is created, \_\_init\_\_ is called first. Constructors are usually used to set up class attributes.

class Car():

# \_\_init\_\_ sets up the make, model, and color of the car object

def \_\_init\_\_(self, make, model, color):

self.make = make

self.model = model

self.color = color

## 44. ****What is a generator in Python?****

Generators in Python are like functions that can return more than once. This is called yielding. They are used to return an iterable collection of items. They are defined with def just like Python functions.

def squares():

x = range(1, 4) # 1 to 4

for n in x:

yield n\*\*2

for y in squares():

print(y) # prints 1 4 9 16

## 45. ****What does the Python help() function do?****

The Python help() function will return the documentation of Python modules, classes, functions, etc.

## 46. ****What does the Python dir() function do?****

The Python dir() function will return a list of methods and attributes from the object you call it with.

## 47. ****What is the difference between .py and .pyc files?****

Python files that end in the py extension are text-based source files. These are the files you create when writing Python code. Python files with the pyc extension are compiled bytecode files. When a Python application runs, the interpreter looks for existing pyc files. If those do not exist, it then compiles the py source files to bytecode and runs the resulting pyc files.

## 48. ****What is a docstring in Python?****

A Python docstring is the first comment in the code object. The docstring for the code object is available on that code object’s \_\_doc\_\_ attribute and through the help function.

def the\_function():

"""The is a function docstring"""

## 49. ****Write Python code to print the complete contents of a file with error handling for missing files.****

try:

with open('filename','r') as file:

print(file.read())

except IOError:

print(“no such file exists”)

## 50. ****Write Python code to print the sum of all numbers from 25 to 75, inclusive.****

print(sum(range(25,76)))

## 51. ****Write Python code to print the length of each line in a particular file, not counting whitespace at the ends.****

with open('filename.txt', 'r') as file:

print(len(file.readline().rstrip()))

## 52. ****Write Python code to remove the whitespace from the following string – ‘abc def geh ijk’.****

s= ‘abc def geh ijk’.replace(‘ ‘,’’)

## 53. ****Write Python code to remove duplicate items from a list.****

res = []

[res.append(x) for x in test\_list if x not in res]

1. **Why Python is called as Interpreted language?**  
     
   Many books state that Python is an interpreted language. A programming language follows any one of the two approaches to implement the code. The approaches are compilation and interpretation. Some languages follow both concepts.  
     
   During compilation, the entire source code is converted into machine code. Machine code is understandable by the CPU. The program will be executed only if the entire code is compiled successfully.  
     
   In interpretation, the code is executed line by line. Each line of source code in Python is translated into machine code during the execution. There is a Virtual Machine available to interpret the python code. The compilation and interpretation runtime structure is given in the above diagram.  
     
   **2. What is dynamically typed language?**  
     
   It is one of the behaviors of high level programming language to check the type of data stored in a variable during the execution. In programming languages such as C, we must declare the type of data before using it.  
     
   Python do not requires and declaration of data type. The python interpreter will know the data type when we assign a value to a variable. This is why python is known as dynamically typed language.  
     
   **3. What is PEP8 in Python?**  
     
   Python known for its readability. In most cases, we can understand the python code better than any other programming languages. Writing beautiful and readable code is an art. PEP8 is an official style guide given by the community to improve the readability to the top.  
     
   **4. How Python's memory management system works?**  
     
   A python program will contain many objects and data types in it. Allocating memory for each object is must. In python heap spaces are used to store data chunks. CPython contains a memory manager that manages the heap space in memory.  
   Managing memory is an important thing in a program. The memory allocation determines the performance of the program. Python's memory manager handles sharing of information, data caching and garbage collection, etc. This memory manager contains automatic garbage collection.  
     
   **5. What is mutability? How it differs in each data types?**  
     
   The ability of a python object to change itself is called mutability. All objects in python are either mutable or immutable. Once created the immutable objects cannot be changed. But, the mutable objects can be changed.  
   Let us try to check the mutability of two different type of objects in Python.  
   List - Mutable object  
     
   **6. What is Namespace in Python?**  
     
   Name is an identifier given to an object in Python. Every object in python is associated with a particular name. For example, when we assign the value 9 to a variable name number , the object that holds the number is associated with the name number.  
   Namespace is a collection of names associated with every object in Python. The namespace contains all the isolated names and created when we executing the program. As the namespace contains the useful keywords and methods, we can access the , methods easily. The namespace stack controls the scope of variables in python.  
    **7. What is null execution in System?**  
     
   In python, the keyword pass is used to create a statement that do nothing. If someone need to create a function with null operation then we can use the pass statement.  
     
   We can create null function to check whether it works or not. If you are going to develop the implementation steps in a function in future, we can use pass statement in it.  
     
   Don't get confused with comments. Comments and pass statements are different things. The interpreter ignores the comments in a program whereas pass statement do null operation.  
     
   **8. What is string slicing?**  
     
   String slicing is the process of making subsets from strings. We can pass a range in the following syntax to slice the string.  
   Syntax: string[start:end:step]  
   Consider the string "Hello World". We can slice the string using the range values. To print 'Hello' from the string. Our range should be 0 to 5. The slicing creates sub string from start index to end-1 th index.  
     
   **9. What are the ways to concatenate two strings in Python? Which is better?**  
     
   We can use join() or + for concatenating two or more strings. The following codes will explain how to use those methods in Python.  
     
   **10. What is decorator in Python?**  
     
   Decorator is tool that allows us to modify the behavior of a function or a class. The decorator objects are called just before the function that you want modify.  
     
   **11. What is generator in Python?**  
     
   Generator is a method to create iterable objects in python. Creating generators are same as creating functions. Instead of return statement we have to use yield in generator function. It returns iterating object each time it is called.  
     
   **12. What is lambda expression in Python?**  
     
   Lambda is an anonymous function that helps us to create functions in single line of code. This is an important thing in functional programming. The lambda function should contains only one expressions in it.  
     
   **13. How to make a list from comma separated inputs?**  
     
   To make a list from a comma separated values or string we can use split method. Split method splits the comma separated values by taking comma as dilimeter.  
     
   **14. What is the difference between normal division and floor division?**  
     
   The arithmetic operators contains two different types of division operators. One is normal division \ another one is \\ . The normal division returns the float value as result. The floor division returns the whole number as result.  
     
   **15. What is the difference between 'is' and '==' in Python?**  
     
   We use is and == to comparing to objects. But both works in different manner. The == compares the values of two objects whereas the is keyword checks whether two objects are same or not.  
   Using == in Python  
     
   **16. What are the difference between List and Tuple in Python**?  
     
   List is enclosed by brackets whereas the tuples are created using parenthesis.  
   The effectiveness of tuple is higher than the list. So it works faster.  
   The list is mutable object whereas the tuples are immutable objects.  
     
    **17. What is negative rounding in Python?**  
     
   We know that the round keyword in python is used to to rounding of decimal places. It can be used to round off whole numbers sing negative rounding. Let the value of the variable X be 1345. To make is 1300 we have use negative rounding.  
     
   **18. How do you prevent the following code from exception?**  
     
   The scope of the variable a is inside the function. To access the variable across the program we have declare it as global variable.  
     
   **19. What is the unique way to swap two variables in Python?**  
     
   Python has unique way to swap two variables. The swapping can be done in one line of code. Let the values of the variables a and b be 5 and 6 respectively. The swapping program will be the following.  
     
   **20. What are Iterators in Python?**  
     
   The objects in python that implements the iterator protocols are called iterators. iter is the keyword used to create the iterable objects. It iterates throough various iterable objects such as List, dictionary.  
     
   **21. What is the difference between shallow copy and deep copy in Python?**  
     
   In python, when we assign a value of a variable to another variable using assignment operator we are just creating a new variable associated with the old object. The copy module in python contains two different type of copying method called copy and deepcopy.  
   **Shallow Copy**  
   The shallow copy works same as the assignment operator. It just create a new variable that is associated with the old object.  
     
   **Deep Copy**  
   The deep copy method creates independent copy of an object. If we edit the element in a copied object, that does not affect the other object.  
     
   **22. How to generate random numbers in Python?**  
     
   There is no in-bulit function in python to generate random numbers We can use random module for this purpose. We can create random number from a range or list. randint is the simple method used to create a random number between two numbers.  
     
   **23. How to disable the escape sequences in a string?**  
     
   The escape sequences in a string can be disabled using regular expression in Python. The raw output of string can be printed using formatting. The following code will show you the process of printing raw string.  
     
   **24. What is negative indexing?**  
     
   Python allows us to access the string and list objects using negative indexes. The negative index of 0th index is -len(iterable). If a string contains five characters then the first character can be accessed with the help of 0 and -5. The last character of the string points to -1.  
     
   **26. How to create multi line string in Python?**  
     
   We can use the escape character \n to split the strings in the output console. But, to write a string in multiple lines inside the code we need to use triple quotes.  
     
   **27. How to create private data in Python?**  
     
   Not like other object oriented programming languages such as java an C++ , Python does not have any keywords to define public and protected data. Normally, the objects created in python can be accessed by public members.  
   Protected members can be accessed only by the class and its sub classes. The following code examples shows that how the private and public members works in python.  
     
   **28. What is pickling and unpickling?**  
     
   Pickle is a module in Python that converts a python object and converts the object into string representation and the string is further converted into a file using dump method.  
   Unpickling is the exact opposite to the process of pickling. The file created is converted into the old version of python object. The following code contains both pickling and unpickling methods.  
     
   **29. What is frozenset in Python?**  
     
   The immutable and hashable version of a normal set is called frozenset. It possess all the properties of normal sets in python. The updating the set is not possible in frozenset.  
     
   **30. What are the functional programming concepts available in Python?**  
     
   Splitting a program into multiple parts is called functional programming. It is one of the widely used programming paradigm. The functional programming concepts that are available as in built features in Python are zip , map , filter and reduce .

### 1. What is Python and what are its key features?

**Answer:**

Python is a high-level, interpreted programming language known for its readability and simplicity. It emphasizes code readability through its clean syntax, which allows developers to express concepts in fewer lines of code compared to many other languages.

A diagram of a padlock

Description automatically generated

Key features include:

* **Interpreted Language:** Python code is executed line-by-line by the Python interpreter, enabling quick testing and debugging.
* **Dynamically Typed:** Variables do not require explicit declaration of data types; Python infers the type at runtime.
* **High-Level Language:** Abstracts complex details like memory management, making it easier to write and maintain code.
* **Extensive Standard Library:** Offers a rich set of modules and packages that support tasks ranging from file I/O to web development.
* **Cross-Platform Compatibility:** Runs seamlessly on multiple operating systems such as Windows, macOS, and Linux.
* **Support for Multiple Paradigms:** Includes procedural, object-oriented, and functional programming styles.

This combination of features makes Python suitable for a wide variety of applications, from scripting and automation to data analysis and machine learning.

### 2. How do you declare variables, and what are the basic data types in Python?

**Answer:** In Python, variables are created by simply assigning a value to a name without requiring an explicit type declaration. Python’s dynamic typing means that the interpreter determines the variable’s type automatically.

**Example:**

|  |
| --- |
| python x = 10         # Integer name = "Alice" # String pi = 3.14      # Float is\_active = True  # Boolean |

**Basic Data Types:**

* **int:** Integer numbers, e.g., 5, -12, 100
* **float:** Floating-point numbers (decimals), e.g., 3.14, -0.001
* **str:**Strings or text, enclosed in quotes, e.g., "Hello"
* **bool:** Boolean values, either True or False
* **NoneType:**Represents the absence of a value, None

These types form the foundation of data manipulation in Python and are automatically managed during runtime, allowing flexible and readable code.

### 3. What are lists, tuples, and dictionaries? How do they differ?

**Answer: Lists**, **tuples**, and **dictionaries**are fundamental data structures in Python used to store collections of items, but they differ in their properties and use cases.

**List:**

An ordered, mutable collection of items. Lists allow modification after creation (adding, removing, or changing elements). They are defined with square brackets [].

**Example:**

|  |
| --- |
| python fruits = ["apple", "banana", "cherry"] fruits.append("orange")  # Lists can be changed |

**Tuple:**

An ordered, immutable collection. Once created, tuples cannot be modified. Defined using parentheses () or without any delimiters in some contexts.

**Example:**

|  |
| --- |
| python coordinates = (10.0, 20.0) |

**Dictionary:**

An unordered collection of key-value pairs. Dictionaries are mutable and use curly braces **{}.** Keys must be immutable types (strings, numbers, tuples), and values can be any type.

**Example:**

|  |
| --- |
| person = {"name": "Alice", "age": 30, "city": "New York"} |

**Key Differences:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Lists** | **Tuples** | **Dictionaries** |
| **Mutability** | Mutable | Immutable | Mutable |
| **Ordering** | Maintains order | Maintains order | Maintains insertion order (Python 3.7+) |
| **Access** | Integer indexes | Integer indexes | Keys |

### 4. Why is indentation important in Python?

**Answer:**Indentation in Python is not merely a matter of style but a critical syntax requirement that defines the structure and flow of the code. Unlike many other languages that use braces {} or keywords to delimit blocks of code, Python uses indentation levels to group statements.

For example, the body of functions, loops, conditionals, and classes must be indented consistently. Incorrect indentation leads to IndentationError or unexpected behavior.

**Example:**

|  |
| --- |
| python if x > 0:     print("Positive")     print("Number is greater than zero")  # Correct indentation else:     print("Non-positive") |

Proper indentation enhances readability and enforces clean code organization, which aligns with Python’s philosophy of simplicity and clarity.

### 5. What are Python’s basic operators, and how are they used?

**Answer:**

Python provides several types of operators to perform operations on variables and values. The main categories include:

Arithmetic Operators: Used for mathematical calculations.

**+ (Addition), - (Subtraction), \* (Multiplication), / (Division), % (Modulus), \*\* (Exponentiation), // (Floor division).**

**Example:**

|  |
| --- |
| python a = 7 b = 3 print(a + b)   # 10 print(a \*\* b)  # 343 print(a // b)  # 2 |

* **Comparison Operators:** Used to compare values, returning Boolean results.

**==, !=, <, >, <=, >=**

* **Logical Operators:** Combine Boolean expressions.

**and, or, not**

* **Assignment Operators:** Assign or modify the value of a variable.

**=, +=, -=, \*=, /=**

* **Membership Operators:** Test for membership in sequences.

**in, not in**

* **Identity Operators:** Compare the memory locations of two objects.

**is, is not**

Each operator serves specific purposes, allowing flexible and expressive manipulation of data.

### 6. What are the different types of loops in Python? How do they work?

**Answer:**

Python primarily supports two types of loops: for loops and while loops.

**For Loop:**

Iterates over a sequence (like a list, tuple, or string) or any iterable object. It executes the loop body once for each element.

**Example:**

|  |
| --- |
| fruits = ['apple', 'banana', 'cherry'] for fruit in fruits:     print(fruit) |

**While Loop:**

Repeats a block of code as long as a given condition remains True. It’s useful when the number of iterations is not known beforehand.

**Example:**

|  |
| --- |
| count = 0 while count < 5:     print(count)     count += 1 |

|  |
| --- |
| count = 0 while count < 5:     print(count)     count += 1 |

Both loops can be controlled using break (exit loop) and continue (skip to next iteration) statements.

### 7. What is the difference between break, continue, and pass statements in Python loops?

**Answer:**

**break:**Terminates the current loop prematurely, exiting immediately.

|  |
| --- |
| for i in range(5):     if i == 3:         break     print(i) # Output: 0 1 2 |

**continue:**Skips the rest of the current iteration and moves to the next iteration of the loop.

|  |
| --- |
| for i in range(5):     if i == 3:         continue     print(i) # Output: 0 1 2 4 |

**pass:** Does nothing; acts as a placeholder for future code or to create empty loops.

|  |
| --- |
| for i in range(5):     pass  # To be implemented later |

### 8. How do for loops work with the range() function? What are some common use cases?

**Answer:**The range() function generates a sequence of numbers and is commonly used in for loops for iteration based on integer counts.

**Basic usage:**

|  |
| --- |
| for i in range(5):     print(i) # Prints numbers 0 to 4 |

**Specifying start, stop, step:**

|  |
| --- |
| for i in range(2, 10, 2):     print(i) # Prints 2, 4, 6, 8 |

Common use cases include iterating a fixed number of times, generating sequences for indexing, and looping over numeric ranges.

### 9. What are list comprehensions? How do they relate to loops?

**Answer:**

List comprehensions provide a concise way to create lists by embedding a for loop and optional conditional logic inside square brackets.

**Example:**

|  |
| --- |
| squares = [x\*\*2 for x in range(5)] print(squares)  # [0, 1, 4, 9, 16] |

**This is equivalent to:**

|  |
| --- |
| squares = [] for x in range(5):     squares.append(x\*\*2) |

List comprehensions improve readability and often result in more efficient code.

### 10. What is a function in Python and how do you define one?

**Answer:**A function in Python is a reusable block of code that performs a specific task. Functions help organize code, improve readability, and avoid repetition.

You define a function using the **def**keyword followed by the function name and parentheses containing optional parameters. The function body is indented.

**Example:**

|  |
| --- |
| def greet(name):     print(f"Hello, {name}!")  greet("Alice")  # Output: Hello, Alice! |

Functions can return values using the return statement. If no return is specified, the function returns None by default.

### 11. What are \*args and \*\*kwargs in Python functions?

**Answer:**

**\*args** and **\*\*kwargs** allow a function to accept an arbitrary number of arguments.

\*args collects extra positional arguments into a tuple.

**Example:**

|  |
| --- |
| def add(\*args):     return sum(args) print(add(1, 2, 3))  # Output: 6 |

\*\*kwargs collects extra keyword arguments into a dictionary.

**Example:**

|  |
| --- |
| def print\_info(\*\*kwargs):     for key, value in kwargs.items():         print(f"{key}: {value}")  print\_info(name="Alice", age=30) # Output: # name: Alice # age: 30 |

They make functions flexible to handle varying input parameters.

### 12. How do you import and use modules in Python?

**Answer:**Modules are Python files (.py) that contain functions, classes, or variables to be reused in other scripts. You import modules to access their contents.

**Basic import:**

|  |
| --- |
| import math print(math.sqrt(16))  # Output: 4.0 |

**Import specific functions or classes:**

|  |
| --- |
| from math import sqrt, pi print(sqrt(25))  # Output: 5.0 print(pi)        # Output: 3.141592653589793 |

**Import with alias:**

|  |
| --- |
| import numpy as np arr = np.array([1, 2, 3]) |

Using modules promotes code modularity and reuse.

### 13. What is the difference between a module and a package in Python?

**Answer:**

**Module:**A single Python file (.py) containing Python code like functions, variables, or classes.

**Package:**A directory containing multiple modules and a special \_\_init\_\_.py file (can be empty) to mark the directory as a package.

**Example:**

|  |
| --- |
| my\_package/     \_\_init\_\_.py     module1.py     module2.py |

Packages help organize related modules into a hierarchical structure, making large projects manageable.

### 14. What is a lambda function and when would you use it?

**Answer:**A lambda function is an anonymous, small, one-line function defined using the lambda keyword. It can take any number of arguments but has only a single expression.

**Example:**

|  |
| --- |
| square = lambda x: x \*\* 2 print(square(5))  # Output: 25 |

Lambdas are often used as arguments to higher-order functions like map(), filter(), and sorted() where a small, unnamed function is needed temporarily.

**Example with map():**

|  |
| --- |
| nums = [1, 2, 3, 4] squares = list(map(lambda x: x\*\*2, nums)) print(squares)  # Output: [1, 4, 9, 16] |

They improve code conciseness and readability for simple functions.

## ****Intermediate Level Python Interview Question****

Building on the foundational knowledge, the intermediate-level questions delve deeper into Python’s core programming concepts. This section focuses on the following topics:

* Object-Oriented Programming (OOP)
* Data Structures & Algorithms
* Python Libraries & Tools
* File Handling & Exceptions

### 15. What are classes and objects in Python?

A comparison of a paper

Description automatically generated with medium confidence

**Answer:**A class is a blueprint or template that defines the structure and behavior (attributes and methods) that its objects will have. An object is an instance of a class — a concrete realization of the class with specific values.

**Example:**

|  |
| --- |
| class Dog:     def \_\_init\_\_(self, name, age):         self.name = name         self.age = age      def bark(self):         print(f"{self.name} says Woof!")  dog1 = Dog("Buddy", 3) dog1.bark()  # Output: Buddy says Woof! |

Here, Dog is the class, and dog1 is an object (instance) of that class.

### 16. What is inheritance in Python? How does it work?

**Answer:**Inheritance allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass), promoting code reuse and hierarchical relationships.

**Example:**

|  |
| --- |
| class Animal:     def speak(self):         print("Animal speaks")  class Dog(Animal):     def speak(self):         print("Dog barks")  dog = Dog() dog.speak()  # Output: Dog barks |

The Dog class inherits from Animal but overrides the speak method, demonstrating method overriding.

### 17. Explain encapsulation in Python.

**Answer:**Encapsulation restricts access to internal object details to protect data integrity. In Python, this is done by prefixing attribute names with underscores:

* **Single underscore \_var:** Conventionally indicates “protected” (should not be accessed outside class/subclass, but not enforced).
* **Double underscore \_\_var**: Triggers name mangling, making the attribute harder to access externally (private).

**Example:**

|  |
| --- |
| class Person:     def \_\_init\_\_(self, name, age):         self.name = name        # Public         self.\_age = age         # Protected (convention)         self.\_\_salary = 50000   # Private (name mangled)  p = Person("Alice", 30) print(p.name)        # Accessible print(p.\_age)        # Accessible but discouraged # print(p.\_\_salary)  # AttributeError print(p.\_Person\_\_salary)  # Access via name mangling |

Encapsulation helps safeguard sensitive data and enforces controlled access via getter/setter methods.

### 18. What is polymorphism? Provide an example in Python.

**Answer:**Polymorphism means “many forms” — the same operation can behave differently in different classes. It allows methods with the same name to act differently based on the object calling them.

**Example using method overriding:**

|  |
| --- |
| class Bird:     def fly(self):         print("Flying in the sky")  class Penguin(Bird):     def fly(self):         print("Penguins can't fly")  def let\_it\_fly(bird):     bird.fly()  b = Bird() p = Penguin()  let\_it\_fly(b)  # Output: Flying in the sky let\_it\_fly(p)  # Output: Penguins can't fly |

Here, fly behaves differently depending on the object type, demonstrating polymorphism.

### 19. What is the purpose of the \_\_init\_\_ method in a Python class?

**Answer:**The \_\_init\_\_ method is the class constructor, automatically invoked when a new object is created. It initializes the object's attributes with given or default values.

**Example:**

|  |
| --- |
| class Car:     def \_\_init\_\_(self, make, model):         self.make = make         self.model = model  car1 = Car("Toyota", "Camry") print(car1.make)  # Output: Toyota |

Without \_\_init\_\_, the object’s attributes would not be set during creation, leading to less flexible and more error-prone code.

### 20. How do you implement a stack and a queue in Python?

**Answer: Stack:** LIFO (Last In, First Out) structure. Can be implemented using Python lists with .append() to push and .pop() to remove elements.

**Example:**

|  |
| --- |
| stack = [] stack.append(1) stack.append(2) print(stack.pop())  # 2 |

**Queue:** FIFO (First In, First Out) structure. Can be implemented with collections.deque for efficient appends and pops.

**Example:**

|  |
| --- |
| from collections import deque queue = deque() queue.append(1) queue.append(2) print(queue.popleft())  # 1 |

Using these built-in structures is efficient and idiomatic in Python.

### 21. What is recursion? Give an example of a recursive function.

**Answer:**Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem until it reaches a base case.

**Example:**Calculating factorial:

|  |
| --- |
| def factorial(n):     if n == 0:         return 1     else:         return n \* factorial(n-1)  print(factorial(5))  # 120 |

The function repeatedly calls itself with decremented values until n is zero, which terminates the recursion.

### 22. How do list comprehensions improve algorithm implementation?

**Answer:**

List comprehensions simplify creating and transforming lists with concise syntax, often replacing verbose loops.

**Example:**

|  |
| --- |
| # Squares of even numbers from 0 to 9 squares = [x\*\*2 for x in range(10) if x % 2 == 0] |

This combines iteration and conditional filtering in one readable line, enhancing clarity and efficiency in many algorithms.

### 23. What is the collections module in Python and what are some of its useful data structures?

**Answer:**The collections module provides specialized container datatypes beyond Python’s built-in types, offering alternatives that provide additional functionality or performance benefits.

**Key data structures include:**

* **Counter:** A dictionary subclass for counting hashable objects. Useful for tallying frequencies.

|  |
| --- |
| from collections import Counter counts = Counter(['apple', 'banana', 'apple', 'orange']) print(counts)  # Counter({'apple': 2, 'banana': 1, 'orange': 1}) |

* **defaultdict:** Like a regular dictionary, but with a default value for missing keys, avoiding KeyError.

|  |
| --- |
| from collections import defaultdict dd = defaultdict(int) dd['apples'] += 1 print(dd['apples'])  # 1 print(dd['oranges']) # 0 (default value) |

* **namedtuple:** Factory function for creating tuple subclasses with named fields, improving code readability.

|  |
| --- |
| from collections import namedtuple Point = namedtuple('Point', ['x', 'y']) p = Point(1, 2) print(p.x, p.y)  # 1 2 |

* **deque:**Double-ended queue supporting fast appends and pops from both ends. Useful for implementing queues and stacks efficiently.

|  |
| --- |
| from collections import deque d = deque() d.append(1) d.appendleft(2) print(d)  # deque([2, 1]) |

These data structures enable writing efficient and expressive code, especially in scenarios requiring frequency counts, default values, or fast queue operations.

### 24. What is itertools and how does it help with iteration in Python?

**Answer: itertools**is a standard Python module that provides a collection of fast, memory-efficient tools for working with iterators. It enables complex iteration patterns such as Cartesian products, permutations, combinations, and infinite sequences.

**Commonly used functions:**

* **count(start=0, step=1):**Infinite iterator starting from start incremented by step.
* **cycle(iterable):**Repeats elements in an iterable indefinitely.
* **chain(\*iterables):** Combines multiple iterables sequentially.
* **product(\*iterables, repeat=1):** Cartesian product of input iterables.
* **permutations(iterable, r):** All possible orderings of length r.
* **combinations(iterable, r):** All unique combinations of length r.

**Example**:

|  |
| --- |
| import itertools  for combo in itertools.combinations('ABC', 2):     print(combo) # Output: ('A', 'B'), ('A', 'C'), ('B', 'C') |

**itertools** helps write concise and efficient code for iteration-heavy problems without creating intermediate data structures, improving performance in large-scale data handling.

### 25. How do you use regular expressions in Python? What is the re module?

**Answer:**The re module provides support for working with regular expressions (regex), a powerful tool for pattern matching and text processing.

**Key functions:**

* **re.match():** Checks for a match only at the beginning of the string.
* **re.search():** Scans through a string looking for the first location where the regex matches.
* **re.findall():** Returns all non-overlapping matches as a list.
* **re.sub():** Replaces occurrences of a pattern with a specified string.

**Example:**

|  |
| --- |
| import re  text = "Contact: email@example.com" match = re.search(r'\S+@\S+\.\S+', text) if match:     print(match.group())  # email@example.com |

Regular expressions are essential for complex string parsing tasks such as validation, extraction, or transformation in data cleaning and natural language processing.

### 26. What are some commonly used Python libraries for data manipulation and scientific computing?

**Answer:**

**NumPy:** Core library for numerical computing, providing powerful n-dimensional array objects and functions for linear algebra, random sampling, and Fourier transforms.

**Example:**

|  |
| --- |
| import numpy as np arr = np.array([1, 2, 3]) print(arr.mean())  # 2.0 |

**Pandas:** Built on NumPy, Pandas offers flexible and efficient data structures (DataFrame, Series) for data analysis and manipulation with tools for handling missing data, reshaping, and grouping.

**Example:**

|  |
| --- |
| import pandas as pd df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]}) print(df.describe()) Matplotlib / Seaborn: Libraries for creating static, animated, and interactive visualizations. |

**Matplotlib / Seaborn:**Libraries for creating static, animated, and interactive visualizations.

These libraries form the foundation for data science and machine learning workflows in Python.

### 27. What are virtual environments and why are they important in Python development?

**Answer:**Virtual environments allow you to create isolated Python environments with their own installed packages, independent from the system-wide Python installation. This isolation prevents conflicts between project dependencies and ensures reproducibility.

Creating and activating a virtual environment:

|  |
| --- |
| python -m venv myenv source myenv/bin/activate  # Linux/macOS myenv\Scripts\activate     # Windows |

**Benefits include:**

* Avoiding package version conflicts across projects.
* Easier dependency management.
* Cleaner development environment.
* Tools like venv, virtualenv, and conda are widely used for this purpose.

### 28. How do you open and read a file in Python?

**Answer:**In Python, files can be opened using the built-in open() function, which returns a file object. You can read the contents using methods like .read(), .readline(), or .readlines().

**Example:**

|  |
| --- |
| with open('example.txt', 'r') as file:     content = file.read()     print(content) |

Using the with statement ensures that the file is properly closed after its suite finishes, even if an error occurs, making it the preferred way for file handling.

### 29. How do you write data to a file in Python?

**Answer:**To write to a file, open it in write ('w') or append ('a') mode. Use the .write() method to add text.

**Example:**

|  |
| --- |
| with open('output.txt', 'w') as file:     file.write('Hello, World!\n') 'w' mode overwrites the file if it exists or creates a new one.  'a' mode appends data to the end without overwriting. |

### 30. What is exception handling in Python? Why is it important?

**Answer:**

Exception handling is a mechanism to catch and handle runtime errors gracefully, preventing program crashes and allowing the program to respond appropriately.

Python uses try-except blocks to handle exceptions:

|  |
| --- |
| try:     result = 10 / 0 except ZeroDivisionError:     print("Cannot divide by zero.") |

This way, the program catches the error and executes alternate code instead of terminating abruptly.

### 31. How do you catch multiple exceptions in a single block?

**Answer:**You can catch multiple exceptions by specifying a tuple of exception types in the except clause:

|  |
| --- |
| try:     value = int(input("Enter a number: "))     result = 10 / value except (ValueError, ZeroDivisionError) as e:     print(f"Error occurred: {e}") |

This catches both invalid inputs and division by zero errors in one block, simplifying error handling.

### 32. What is the purpose of the finally block in exception handling?

**Answer:**The finally block contains code that will execute regardless of whether an exception was raised or caught. It’s typically used for cleanup actions like closing files or releasing resources.

**Example:**

|  |
| --- |
| try:     file = open('data.txt', 'r')     data = file.read() except IOError:     print("File error.") finally:     file.close() |

This ensures the file is closed even if an error occurs during reading.

## ****Advance Level Python Interview Questions****

This section integrates advanced object-oriented programming techniques and design patterns with performance optimization strategies. It also covers concurrency models to handle parallelism effectively, alongside domain-specific frameworks and libraries critical in real-world applications.

### 33. What are Python decorators and how do they work?

**Answer:**Decorators in Python are functions that modify the behavior of other functions or methods without changing their source code. They allow you to wrap another function in order to extend or alter its behavior dynamically.

A decorator takes a function as input and returns a new function with added functionality.

**Example:**

|  |
| --- |
| def decorator\_func(original\_func):     def wrapper():         print("Before the function call")         original\_func()         print("After the function call")     return wrapper  @decorator\_func def say\_hello():     print("Hello!")  say\_hello() |

**Output:**

|  |
| --- |
| pgsql Copy Before the function call  Hello!  After the function call |

Here, the say\_hello function is wrapped by decorator\_func, which adds behavior before and after the original function call. Decorators are widely used for logging, access control, caching, and more.

### 34. Explain metaclasses and their use cases in Python.

**Answer:**Metaclasses are the "classes of classes" — they define how classes themselves behave. In Python, classes are objects, and metaclasses control the creation and behavior of these class objects.

By default, the metaclass is type, but you can define a custom metaclass by inheriting from type and overriding methods like \_\_new\_\_ or \_\_init\_\_.

* Use cases for metaclasses include:
* Enforcing coding standards or API contracts at class creation.
* Automatically registering classes in frameworks (e.g., plugins).
* Modifying class attributes or methods dynamically.

**Example:**

|  |
| --- |
| python Copy class Meta(type):     def \_\_new\_\_(cls, name, bases, dct):         print(f"Creating class {name}")         return super().\_\_new\_\_(cls, name, bases, dct)  class MyClass(metaclass=Meta):     pass  # Output: Creating class MyClass |

Metaclasses are an advanced feature used primarily in framework and library development.

### 35. How do you implement the Singleton design pattern in Python?

**Answer:**The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

Python implementation often uses overriding \_\_new\_\_ method or decorators.

Example using \_\_new\_\_:]

|  |
| --- |
| class Singleton:     \_instance = None      def \_\_new\_\_(cls, \*args, \*\*kwargs):         if not cls.\_instance:             cls.\_instance = super().\_\_new\_\_(cls)         return cls.\_instance  s1 = Singleton() s2 = Singleton() print(s1 is s2)  # True |

This approach ensures that every instantiation returns the same instance.

### 36. What are descriptors and how do they control attribute access?

**Answer:**Descriptors are objects that manage the access to another object’s attributes by implementing any of the descriptor protocol methods: \_\_get\_\_(), \_\_set\_\_(), and \_\_delete\_\_().

They are used to customize behavior when attributes are accessed, assigned, or deleted.

**Example:**

|  |
| --- |
| python Copy class Descriptor:     def \_\_get\_\_(self, instance, owner):         print("Getting value")         return instance.\_value      def \_\_set\_\_(self, instance, value):         print("Setting value")         instance.\_value = value  class MyClass:     attr = Descriptor()  obj = MyClass() obj.attr = 10   # Output: Setting value print(obj.attr) # Output: Getting value \n 10 |

Descriptors are fundamental to how properties, methods, and static methods work internally in Python.

### 37. Explain the difference between class methods, static methods, and instance methods.

**Answer:**

* **Instance methods:**Regular methods that take self as the first argument, representing the instance. They can access and modify object state.
* **Class methods:**Decorated with @classmethod, take cls as the first argument representing the class, not instance. Useful for factory methods or methods that affect the class as a whole.

|  |
| --- |
| class MyClass:     count = 0      def \_\_init\_\_(self):         MyClass.count += 1      @classmethod     def get\_count(cls):         return cls.count |

* **Static methods:**Decorated with @staticmethod, do not take self or cls and behave like regular functions within the class namespace. They do not modify class or instance state.

|  |
| --- |
| class MathUtils:     @staticmethod     def add(x, y):         return x + y |

### 38. How does method resolution order (MRO) work in multiple inheritance?

**Answer:**MRO determines the order in which base classes are searched when calling a method. Python uses the C3 linearization algorithm for MRO to provide a consistent and predictable order.

You can view MRO with the \_\_mro\_\_ attribute or the mro() method.

**Example:**

|  |
| --- |
| class A:     def method(self):         print("A")  class B(A):     def method(self):         print("B")  class C(A):     def method(self):         print("C")  class D(B, C):     pass  print(D.\_\_mro\_\_) d = D() d.method()  # Output: B |

Here, D inherits from B and C, and the MRO ensures the method from B is called before C.

MRO avoids issues like the diamond problem in multiple inheritance, ensuring consistent behavior.

### 39. What is the Global Interpreter Lock (GIL) and how does it affect Python concurrency?

**Answer:**The Global Interpreter Lock (GIL) is a mutex in CPython (the standard Python interpreter) that allows only one thread to execute Python bytecode at a time, even on multi-core processors.

**Implications:**

* Limits true parallelism in CPU-bound multi-threaded programs because only one thread runs Python code at a time.
* Does not affect I/O-bound programs significantly because threads waiting on I/O release the GIL.
* Can be bypassed using multiprocessing which creates separate processes, each with its own Python interpreter and memory space.
* The GIL simplifies memory management but requires careful design to avoid performance bottlenecks in multithreaded Python applications.

### 40. How can you optimize Python code for speed and memory usage?

**Answer:**Several strategies exist to optimize Python code:

* **Use built-in functions and libraries:**Python’s built-in functions like sum(), map(), and libraries like NumPy are implemented in C and are faster than pure Python loops.
* **Avoid unnecessary computations**: Cache results (memoization), reuse variables, and avoid redundant calculations.
* **Use generators:** Replace lists with generators to process data lazily and save memory.
* **Data structures choice:**Use appropriate data structures (e.g., sets for membership tests instead of lists).
* **Algorithm optimization:** Use efficient algorithms and reduce time complexity.
* **Use local variables:**Accessing local variables is faster than globals.
* **Use Just-In-Time (JIT) compilers:** Tools like PyPy can speed up Python code execution.
* **Profile before optimizing:** Focus on optimizing code sections identified as bottlenecks through profiling.

### 41. What is the difference between threading, multiprocessing, and asyncio in Python?

**Answer:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Threading** | **Multiprocessing** | **Asyncio** |
| **Execution Model** | Multiple threads within a single process | Multiple separate processes | Single-threaded asynchronous event loop |
| **Memory Sharing** | Shares the same memory space | Separate memory space per process | Shares memory within the event loop |
| **GIL Impact** | Limited by Global Interpreter Lock (only one thread runs Python bytecode at a time) | Bypasses GIL for true parallelism | Single-threaded, so no GIL contention |
| **Suitable For** | I/O-bound tasks (network requests, file operations) | CPU-bound tasks (heavy computations) | I/O-bound tasks and high-concurrency scenarios without threads |
| **Communication** | Easy data sharing between threads | Complex and slower inter-process communication | Efficient task switching within a single thread |
| **Programming Style** | Traditional multi-threading | Multi-process programming | Asynchronous programming with async/await |

### 42. How do you implement asynchronous programming using async and await?

**Answer:**

Asynchronous programming in Python uses async functions (coroutines) and the await keyword to pause and resume execution when waiting for I/O or other asynchronous operations.

**Example:**

|  |
| --- |
| import asyncio  async def say\_hello():     print("Hello")     await asyncio.sleep(1)  # Non-blocking sleep     print("World")  asyncio.run(say\_hello()) |

Here, async def defines a coroutine, and await suspends its execution until the awaited coroutine completes, allowing other tasks to run in the meantime. This model improves efficiency in I/O-bound programs by avoiding thread overhead.

### 43. What are common issues with thread safety and how do you handle them?

**Answer:**

Thread safety issues arise when multiple threads access shared resources concurrently, leading to race conditions, data corruption, or inconsistent state.

**Common problems include:**

* **Race conditions:**When the outcome depends on the sequence or timing of threads.
* **Deadlocks:** When threads wait indefinitely for resources locked by each other.
* **Data inconsistency:**Due to unsynchronized access to shared variables.

**Handling thread safety:**

* Use thread synchronization primitives like Locks (threading.Lock), RLocks, Semaphores, and Events to control access.
* Employ thread-safe data structures or design immutable data to avoid shared mutable state.
* Minimize shared state where possible.
* Use higher-level abstractions like the queue.Queue which is thread-safe by design.

**Example of using a lock:**

|  |
| --- |
| import threading  lock = threading.Lock() counter = 0  def increment():     global counter     with lock:         temp = counter         temp += 1         counter = temp |

### 44. How do you use the concurrent.futures module?

**Answer:**The concurrent.futures module simplifies launching parallel tasks using threads or processes with a high-level interface.

**Key components:**

|  |
| --- |
| ThreadPoolExecutor: Executes calls asynchronously using a pool of threads (good for I/O-bound). ProcessPoolExecutor: Uses multiple processes (good for CPU-bound). |

**Example with ThreadPoolExecutor:**

|  |
| --- |
| from concurrent.futures import ThreadPoolExecutor  def task(n):     return n \* 2  with ThreadPoolExecutor(max\_workers=3) as executor:     results = executor.map(task, [1, 2, 3, 4])     print(list(results))  # [2, 4, 6, 8] |

executor.submit() schedules individual calls returning Future objects for fine-grained control.

This module abstracts away thread/process management, allowing easy parallel execution of functions.

### 45. How do you perform web scraping using BeautifulSoup or Scrapy?

**Answer:**

* **BeautifulSoup:**A Python library for parsing HTML and XML documents. It creates parse trees that make extracting data from HTML easy. Used mainly for small to medium scraping tasks.

**Example**:

|  |
| --- |
| import requests from bs4 import BeautifulSoup  url = "https://example.com" response = requests.get(url) soup = BeautifulSoup(response.text, 'html.parser')  for link in soup.find\_all('a'):     print(link.get('href')) |

* **Scrapy:**A full-fledged web scraping and crawling framework. It’s designed for larger, more complex projects, providing asynchronous scraping, data pipelines, and built-in support for exporting scraped data.

You define spiders that crawl web pages and extract structured data efficiently.

### 46. What is Flask and how does it compare to Django?

A screen shot of a black background

Description automatically generated

**Answer:**

**Flask:**A lightweight, micro web framework designed for simplicity and flexibility. It provides the essentials for building web applications without enforcing project structure or dependencies, giving developers freedom to choose components.

**Django:**A full-featured, batteries-included framework that follows the “don’t repeat yourself” principle. It offers an ORM, admin interface, authentication, and a large ecosystem out of the box.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Flask** | **Django** |
| **Size** | Lightweight, minimal | Large, comprehensive |
| **Flexibility** | High, choose your tools | Less flexible, conventions enforced |
| **Learning curve** | Easier for beginners | Steeper, due to features |
| **Use cases** | Small to medium projects, APIs | Complex, data-driven websites |

### 47. Explain how to parse JSON data efficiently in Python.

**Answer:**JSON (JavaScript Object Notation) is a common data interchange format. Python’s built-in json module allows parsing JSON strings into native Python data structures (dict, list).

**Example:**

|  |
| --- |
| import json  json\_string = '{"name": "Alice", "age": 30}' data = json.loads(json\_string) print(data['name'])  # Alice |

**For writing JSON:**

|  |
| --- |
| json\_data = json.dumps(data) For large JSON files, parsing line-by-line or using streaming parsers like ijson improves memory efficiency. |

Handling JSON efficiently enables seamless integration with APIs and configuration files.