1. What is the differnce between a module and package?

Trong Python, module và package đều dùng để tổ chức và tái sử dụng code, nhưng chúng khác nhau về mức độ.

Một module chỉ đơn giản là một file Python – tức là một file có phần mở rộng .py, chứa các hàm, class hoặc biến mà mình có thể import và dùng ở nơi khác.

Ví dụ: file math_utils.py là một module.

Còn package là một thư mục (folder) chứa nhiều module hoặc nhiều sub-package, và bên trong nó thường có file __init__.py để Python nhận biết đây là một package.

Nói ngắn gọn:

- Module = một file Python.
- Package = một thư mục chứa nhiều module.

"Module là một file Python đơn lẻ, còn package là một thư mục chứa nhiều module, thường có file __init__.py để đánh dấu đó là một package."

2. Is Python a compiled language or an interpreted language?

Python là ngôn ngữ chủ yếu thông dịch, có nghĩa là code được thực thi từng dòng bởi trình thông dịch tại runtime, nhưng đồng thời Python cũng tạm biên dịch sang bytecode – một dạng trung gian không phải machine code – trước khi trình thông dịch thực thi, giúp chương trình chạy linh hoạt và dễ debug hơn.

"Python chủ yếu là thông dịch, nhưng code được biên dịch tạm sang bytecode trước khi chạy."

3. What are the benefits of using Python language as a tool in the present scenario?

4. What are global, protected and private attributes in Python?

Trong Python, các biến hoặc thuộc tính được phân loại theo phạm vi truy cập: global, protected và private.

- Global variables là biến được định nghĩa ở phạm vi toàn cục, ngoài tất cả các hàm hoặc class. Nếu muốn sử dụng biến global bên trong một hàm, mình cần dùng từ khóa global.
- Protected attributes được đặt với một dấu gạch dưới ở đầu tên (_var). Về mặt kỹ thuật vẫn có thể truy cập từ bên ngoài class, nhưng theo chuẩn lập trình thì không nên truy cập trực tiếp vì chúng được coi là "chỉ dùng nội bộ".
- Private attributes được đặt với hai dấu gạch dưới ở đầu tên (__var). Những thuộc tính này không thể truy cập trực tiếp từ bên ngoài class, và nếu cố truy cập sẽ gây ra lỗi AttributeError.

Tóm lại, cách đặt tên giúp Python phân biệt mức độ truy cập: global thì mọi nơi dùng được, protected thì hạn chế dùng ngoài class, private thì tuyệt đối không dùng ngoài class.

"Global là biến toàn cục, dùng được ở mọi nơi;
Protected bắt đầu bằng _, nên hạn chế truy cập ngoài class;
Private bắt đầu bằng __, không thể truy cập trực tiếp từ ngoài class."

5. Is python case-sensitive?

Python là ngôn ngữ phân biệt chữ hoa và chữ thường, nghĩa là các biến hoặc tên hàm giống nhau về chữ nhưng khác về hoa thường sẽ được coi là các identifier khác nhau, ví dụ var, Var và VAR là ba biến hoàn toàn riêng biệt.

"Python phân biệt chữ hoa và chữ thường, nên các tên giống nhau nhưng khác hoa thường là khác nhau."

6. What is Pandas?

7. How is Exceptional handling done in Python?

Trong Python, việc xử lý ngoại lệ (exception handling) được thực hiện bằng ba từ khóa chính là try, except, và finally.

Phần code có khả năng gây lỗi sẽ được đặt trong try, còn except dùng để bắt và xử lý lỗi nếu nó xảy ra, giúp chương trình không bị dừng đột ngột.

Mình cũng có thể dùng nhiều except khác nhau để xử lý từng loại lỗi riêng, ví dụ như ValueError hoặc ZeroDivisionError.

Còn finally là phần sẽ luôn được chạy, dù có lỗi hay không, thường dùng để dọn dẹp tài nguyên, đóng file, v.v.

Nói ngắn gọn thì cơ chế này giúp chương trình Python ổn định hơn và tránh bị crash khi có lỗi xảy ra.

"Python xử lý ngoại lệ bằng các khối try–except–finally.
Try chứa code có thể gây lỗi, except xử lý lỗi, và finally luôn chạy dù có lỗi hay không — thường để dọn dẹp tài nguyên."

8. Difference between for loop and while loop in Python.

Về cơ bản, for loop và while loop đều dùng để lặp, nhưng khác nhau ở cách xác định điều kiện lặp.

For loop thường được dùng khi mình biết số lần lặp cụ thể hoặc muốn lặp qua một dãy phần tử như list, tuple hay string.

Ví dụ: for i in range(5): – mình biết chắc nó sẽ chạy 5 lần.

Còn while loop thì dùng khi chưa biết trước số lần lặp, mà chỉ muốn lặp đến khi điều kiện còn đúng.

Ví dụ: while x < 10: – nó sẽ tiếp tục chạy cho đến khi điều kiện sai.

Tóm lại, for loop phù hợp khi biết rõ số vòng lặp, còn while loop thì linh hoạt hơn khi điều kiện thay đổi trong quá trình chạy.

"For loop dùng khi biết trước số lần lặp, còn while loop dùng khi chỉ có điều kiện dừng mà không biết trước chạy bao nhiêu lần."

9. Is Indentation Required in Python?

10. What is the use of self in Python?

Trong Python, self được dùng trong các class để tham chiếu đến chính đối tượng (object) hiện tại.
Nói cách khác, nó giúp mình truy cập tới các biến và hàm của chính đối tượng đó.
Ví dụ, khi mình khai báo một phương thức trong class, thì tham số đầu tiên thường là self.
Nhờ có self, mình có thể viết kiểu như self.name hoặc self.age để gán hoặc truy cập thuộc tính của object.
Nếu không có self, Python sẽ không biết biến đó thuộc về đối tượng nào.
Tóm lại, self giúp class làm việc với dữ liệu riêng của từng object.
"self là tham chiếu đến chính đối tượng hiện tại trong class.
Nó dùng để truy cập hoặc thay đổi thuộc tính và phương thức của object đó."

11. How does Python manage memory? Explain the role of reference counting and garbage collection.

Trong Python, bộ nhớ được quản lý tự động thông qua một cơ chế kết hợp reference counting và garbage collection.
Reference counting (đếm tham chiếu) là phương pháp Python theo dõi số lượng biến tham chiếu tới một đối tượng. Mỗi khi có một biến mới trỏ tới đối tượng, con số này tăng lên; khi một biến mất hoặc không còn trỏ tới đối tượng nữa, con số giảm đi. Khi số lượng tham chiếu bằng 0, Python sẽ tự động giải phóng bộ nhớ của đối tượng đó.
Tuy nhiên, reference counting không xử lý được các vòng tham chiếu (cyclic references), ví dụ khi hai đối tượng tham chiếu lẫn nhau. Vì vậy, Python còn có garbage collector, một cơ chế bổ sung, quét các đối tượng không còn được truy cập nhưng vẫn tạo vòng tham chiếu, và giải phóng bộ nhớ của chúng.
Nhờ hai cơ chế này, Python tự động quản lý bộ nhớ, giúp lập trình viên không phải giải quyết việc cấp phát hay giải phóng bộ nhớ thủ công, vừa tiện lợi vừa an toàn.
"Python quản lý bộ nhớ bằng reference counting để giải phóng các đối tượng không còn tham chiếu, và garbage collector để xử lý các vòng tham chiếu, tự động giải phóng bộ nhớ."

12.

13. How is memory management done in Python?

Trong Python, quản lý bộ nhớ được thực hiện thông qua một khu vực gọi là private heap, nơi lưu trữ tất cả các đối tượng và cấu trúc dữ liệu. Lập trình viên không thể truy cập trực tiếp vào heap này, nó được Python interpreter quản lý hoàn toàn.
Python cũng có garbage collector (bộ thu gom rác) để tự động giải phóng bộ nhớ các đối tượng không còn được tham chiếu, giúp tránh rò rỉ bộ nhớ. Ngoài ra, Python sử dụng reference counting để theo dõi số lượng tham chiếu tới mỗi đối tượng; khi một đối tượng không còn tham chiếu nào, nó sẽ được giải phóng.
Nhờ cơ chế này, lập trình viên không cần phải quản lý bộ nhớ thủ công, Python sẽ tự động phân bổ và giải phóng bộ nhớ một cách hiệu quả, vừa an toàn vừa tiện lợi.
"Python quản lý bộ nhớ bằng private heap, reference counting và garbage collector, tự động phân bổ và giải phóng bộ nhớ khi đối tượng không còn dùng nữa."

14. How to delete a file using Python?

15. Which sorting technique is used by sort() and sorted() functions of python?

Trong Python, cả sort() và sorted() đều sử dụng thuật toán TimSort để sắp xếp. TimSort là thuật toán lai, kết hợp giữa Merge Sort và Insertion Sort, được thiết kế để hoạt động hiệu quả trên dữ liệu thực tế, đặc biệt là khi dữ liệu đã phần nào sắp xếp sẵn.
Ưu điểm của TimSort là nó là thuật toán sắp xếp ổn định (stable), nghĩa là nếu hai phần tử bằng nhau, thứ tự ban đầu của chúng sẽ được giữ nguyên. Đồng thời, độ phức tạp thuật toán trong trường hợp xấu nhất là O(n log n), giúp xử lý dữ liệu lớn một cách hiệu quả.
Vì vậy, khi bạn dùng list.sort() hay sorted(list), Python sẽ tự động áp dụng TimSort, giúp sắp xếp nhanh và ổn định.
"Hàm sort() và sorted() trong Python dùng Thuật toán TimSort, một thuật toán ổn định kết hợp Merge Sort và Insertion Sort, hiệu quả với dữ liệu lớn và dữ liệu đã phần nào sắp xếp sẵn."

16. Differentiate between List and Tuple?

Trong Python, List và Tuple đều là kiểu dữ liệu dùng để lưu trữ nhiều giá trị, nhưng chúng khác nhau về khả năng thay đổi, cú pháp và hiệu năng:
List là mutable, nghĩa là bạn có thể thêm, xóa hoặc thay đổi các phần tử sau khi tạo. List dùng dấu ngoặc vuông [ ] để khai báo. Vì tính linh hoạt, List tốn nhiều bộ nhớ hơn và một số thao tác có thể chậm hơn Tuple.
Tuple là immutable, tức là không thể thay đổi các phần tử sau khi tạo. Tuple dùng dấu ngoặc tròn ( ) để khai báo. Nhờ bất biến, Tuple tiết kiệm bộ nhớ và lặp nhanh hơn, phù hợp khi dữ liệu không cần chỉnh sửa.
Nói cách khác, nếu bạn cần dữ liệu có thể thay đổi, dùng List; nếu dữ liệu cố định và cần hiệu năng tốt hơn, dùng Tuple.
"List có thể thay đổi, dùng ngoặc vuông và linh hoạt nhưng chậm hơn; Tuple cố định, dùng ngoặc tròn và nhanh hơn."

17. What is slicing in Python?

Slicing trong Python là kỹ thuật dùng để trích xuất một phần của chuỗi, danh sách hoặc tuple. Bạn có thể chỉ định start, stop và step để lấy ra đúng phần bạn muốn.
Start là chỉ số bắt đầu, phần tử tại chỉ số này sẽ được bao gồm. Nếu bỏ qua, mặc định từ đầu chuỗi hoặc danh sách.
Stop là chỉ số kết thúc, phần tử tại chỉ số này không được bao gồm. Nếu bỏ qua, mặc định đến cuối.
Step là bước nhảy giữa các phần tử, nếu bỏ qua thì mặc định là 1.
Kết quả của slicing là một danh sách hoặc chuỗi mới được tạo từ phần đã chọn. Slicing rất hữu ích khi bạn muốn lấy một đoạn con từ dữ liệu mà không thay đổi dữ liệu gốc.
"Slicing trong Python là kỹ thuật trích xuất một phần chuỗi, danh sách hoặc tuple bằng cách chỉ định start, stop và step, và kết quả là một đối tượng mới."

18. How is multithreading achieved in Python?

Trong Python, đa luồng (multithreading) được thực hiện bằng module threading. Module này cho phép chúng ta chạy nhiều luồng đồng thời trong cùng một tiến trình, nghĩa là mỗi luồng có thể thực hiện một công việc riêng, ví dụ như đọc dữ liệu từ file, gọi API, hay xử lý I/O.

Tuy nhiên, Python có Global Interpreter Lock (GIL), nên tại một thời điểm chỉ có một luồng thực thi bytecode Python, điều này làm giảm hiệu quả đa luồng cho các tác vụ tính toán nặng. Vì vậy, đa luồng trong Python phù hợp nhất với các tác vụ I/O như đọc ghi file hoặc network, còn với các tác vụ CPU-bound nặng, thường người ta sẽ dùng multiprocessing thay vì threading để tận dụng đa nhân CPU.

"Python dùng module threading để đa luồng, hiệu quả với I/O nhưng bị GIL hạn chế cho các tác vụ tính toán nặng."

19. Which is faster, Python list or Numpy Arrays?

Trong Python, NumPy arrays thường nhanh hơn Python lists khi thực hiện các phép tính số học và thao tác trên dữ liệu lớn. Nguyên nhân là NumPy arrays được cài đặt bằng ngôn ngữ C và tối ưu hóa hiệu năng, đồng thời quản lý bộ nhớ hiệu quả hơn.

Trong khi đó, Python lists linh hoạt hơn, có thể chứa nhiều kiểu dữ liệu khác nhau, nhưng do tính năng động và không được tối ưu hóa cho tính toán số học, nên chậm hơn khi làm việc với số lượng lớn dữ liệu số.

Tuy nhiên, nếu bạn cần lưu trữ dữ liệu hỗn hợp hoặc thay đổi kích thước danh sách thường xuyên, Python lists có thể phù hợp hơn. Nói cách khác, NumPy arrays mạnh về hiệu năng và tính toán số học, còn lists mạnh về sự linh hoạt.

"NumPy arrays nhanh hơn Python lists cho tính toán số học lớn vì được tối ưu bằng C, còn lists linh hoạt nhưng chậm hơn."

22. Write a program to produce the Fibonacci series in python.

23. What is the difference between shallow copy and deep copy in Python?

Trong Python, khi sao chép một đối tượng, có hai cách phổ biến: shallow copy và deep copy.

Shallow copy là sao chép bề mặt, tức là chỉ sao chép đối tượng cha, còn các đối tượng con bên trong vẫn dùng chung tham chiếu với bản gốc.

Vì vậy, nếu thay đổi đối tượng con trong bản sao, bản gốc cũng bị ảnh hưởng.

Shallow copy thường nhanh hơn và tốn ít bộ nhớ.

Deep copy là sao chép toàn bộ đối tượng, bao gồm cả các đối tượng con bên trong.

Vì vậy, bản sao hoàn toàn độc lập với bản gốc, thay đổi gì trong bản sao cũng không ảnh hưởng đến bản gốc.

Deep copy chậm hơn và tốn nhiều bộ nhớ hơn vì cần sao chép tất cả các thành phần.

Tóm lại, shallow copy chia sẻ các thành phần bên trong, còn deep copy sao chép toàn bộ đối tượng hoàn chỉnh.

"Shallow copy sao chép bề mặt, các đối tượng con vẫn chia sẻ với bản gốc.
Deep copy sao chép toàn bộ, bản sao hoàn toàn độc lập."

24. What is the process of compilation and linking in Python?

Trong Python, quá trình biên dịch và liên kết khác với các ngôn ngữ như C hay C++.
Compilation (biên dịch): Khi bạn chạy một chương trình Python, mã nguồn .py trước tiên được biên dịch thành bytecode — một dạng trung gian, thấp hơn so với mã máy nhưng không thể đọc trực tiếp. Bytecode này được lưu dưới dạng file .pyc để Python có thể thực thi nhanh hơn trong lần chạy sau.
Linking (liên kết): Python không có quá trình liên kết tách biệt như trong C/C++. Thay vào đó, khi import các module hoặc package, Python sẽ tự động liên kết các phần này vào chương trình tại runtime. Nói cách khác, việc sử dụng module, thư viện được xử lý động, và bytecode của các module sẽ được nạp khi cần.
Tóm lại, Python biên dịch sang bytecode, sau đó liên kết các module và package khi chạy, giúp lập trình viên không phải lo về việc liên kết thủ công.
"Python biên dịch mã nguồn sang bytecode và liên kết các module, package động khi chạy, không cần liên kết thủ công như C/C++."

25. What is break, continue, and pass in Python?

Trong Python, break, continue và pass là ba câu lệnh đặc biệt dùng trong vòng lặp hoặc khối code:
break: ngay lập tức dừng vòng lặp hiện tại và chuyển sang câu lệnh tiếp theo bên ngoài vòng lặp.
continue: bỏ qua phần còn lại của vòng lặp hiện tại và chuyển sang vòng lặp tiếp theo.
pass: là câu lệnh rỗng, không làm gì cả, thường dùng làm placeholder khi Python yêu cầu có một khối code nhưng mình chưa muốn viết gì.
Nói cách khác, break dừng vòng lặp, continue bỏ qua lần lặp hiện tại, còn pass chỉ để giữ chỗ.
"Break dừng vòng lặp, continue bỏ qua lần lặp hiện tại, pass không làm gì, chỉ để giữ chỗ."

26. What is PEP 8?

27. What is an Expression?

Trong Python, expression là một sự kết hợp của giá trị, biến, toán tử hoặc lời gọi hàm mà Python sẽ tính toán và trả về một giá trị mới.
Nói cách khác, expression là một đoạn mã mà khi chạy sẽ sinh ra một kết quả.
Ví dụ: 2 + 3, x * y, hay len("Hello") đều là expressions vì chúng trả về giá trị khi được đánh giá.
Expression là khối xây dựng cơ bản của chương trình Python, bởi mọi phép tính, thao tác hay logic đều dựa trên expression.
"Expression là đoạn mã mà Python tính toán và trả về một giá trị.
*Ví dụ như 2 + 3, x * y, hoặc len("Hello")."*

28. What is a decorator in Python and when would you use it?

Trong Python, decorator là một hàm dùng để thay đổi hoặc mở rộng hành vi của một hàm khác mà không cần sửa đổi trực tiếp code của hàm đó. Nói cách khác, decorator "bọc" hàm gốc và thêm chức năng mới. Cú pháp thường dùng là dấu @ trước tên decorator.
Dùng decorator khi muốn tái sử dụng logic chung, thường như:
Logging: ghi lại thông tin khi hàm được gọi
Authorization: kiểm tra quyền trước khi thực hiện hàm
Caching: lưu kết quả hàm để dùng lại
Decorator giúp code gọn gàng, tái sử dụng cao và dễ bảo trì, đặc biệt khi bạn muốn áp dụng cùng một logic cho nhiều hàm khác nhau.
"Decorator là hàm bọc hàm khác để mở rộng chức năng mà không sửa code gốc, dùng cho logging, authorization hay caching."

29. What is type conversion in Python?

Trong Python, type conversion là quá trình chuyển giá trị từ kiểu dữ liệu này sang kiểu dữ liệu khác.
Điều này thường cần khi mình muốn thực hiện các phép toán hoặc xử lý dữ liệu mà yêu cầu kiểu dữ liệu cụ thể.
Python cung cấp nhiều hàm để chuyển đổi kiểu cơ bản:
int() → chuyển sang số nguyên
float() → chuyển sang số thực
str() → chuyển sang chuỗi
list() → chuyển sang danh sách
tuple() → chuyển sang tuple
set() → chuyển sang set
dict() → chuyển một sequence key-value thành dictionary
Nói ngắn gọn, type conversion giúp dữ liệu ở đúng kiểu cần thiết để xử lý trong chương trình.
"Type conversion là chuyển dữ liệu từ kiểu này sang kiểu khác.
Python có các hàm như int(), float(), str(), list()... để thực hiện việc này."

30. Name some commonly used built-in modules in Python?

Python có rất nhiều module có sẵn (built-in modules) giúp bạn thực hiện các chức năng khác nhau mà không cần cài thêm gì. Một số module phổ biến thường dùng bao gồm:
os: thao tác với hệ điều hành, như đọc/ghi file, quản lý thư mục.
sys: truy cập các biến và hàm của Python runtime, ví dụ như argv, exit.
math: các hàm toán học cơ bản như sin, cos, sqrt.
random: tạo số ngẫu nhiên, chọn phần tử ngẫu nhiên từ danh sách.
datetime: xử lý ngày giờ, tính toán khoảng thời gian.
json: chuyển đổi dữ liệu giữa Python và định dạng JSON.
Những module này cực kỳ hữu ích, tiết kiệm thời gian và giúp code gọn gàng hơn.
"Các module built-in hay dùng trong Python gồm os, sys, math, random, datetime, và json."

31. What is the difference between xrange and range in functions.

Trong Python 2, range và xrange đều dùng để tạo dãy số cho vòng lặp, nhưng khác nhau về cách cấp phát bộ nhớ.
range trả về một list chứa toàn bộ các số trong dãy ngay lập tức. Điều này có thể tốn nhiều bộ nhớ nếu dãy lớn.
xrange trả về một đối tượng generator-like, tức là tạo từng số khi cần, không lưu toàn bộ list trong bộ nhớ.
Vì vậy xrange tiết kiệm bộ nhớ hơn cho các dãy lớn, còn được gọi là lazy evaluation.
Còn trong Python 3, xrange không còn tồn tại nữa, và range trong Python 3 hoạt động giống như xrange của Python 2.
"Trong Python 2, range tạo list đầy đủ, còn xrange tạo số từng phần khi cần, tiết kiệm bộ nhớ. Trong Python 3, chỉ còn range, và nó hoạt động giống xrange."

32. What is the zip function?

zip() là hàm tích hợp sẵn trong Python, dùng để kết hợp nhiều iterable (như list, tuple) lại với nhau theo từng phần tử tương ứng.
"zip() dùng để ghép các phần tử của nhiều danh sách hoặc tuple lại với nhau theo từng cặp tương ứng."

33. What is the Django architecture?

Django sử dụng kiến trúc MTV (Model-Template-View), tương tự MVC nhưng tên gọi khác:
Model: đại diện dữ liệu và business logic (tương ứng với database).
Template: giao diện (HTML, CSS), chịu trách nhiệm hiển thị dữ liệu.
View: xử lý request, gọi Model và trả dữ liệu cho Template.
Ngoài ra, Django còn có URL dispatcher để map URL đến view tương ứng.
Luồng request: User → URL → View → Model → View → Template → Response
"Django dùng kiến trúc MTV: Model quản lý dữ liệu, View xử lý request, Template hiển thị giao diện, cùng URL dispatcher điều hướng request."

34. What is inheritance in Python?

Trong Python, inheritance hay kế thừa là cơ chế cho phép một class con (child class) kế thừa các thuộc tính và phương thức từ một class cha (parent class).
Điều này giúp:
- Tái sử dụng code, không phải viết lại những phương thức đã có
- Mở rộng chức năng của class cha bằng cách thêm hoặc ghi đè phương thức
- Tổ chức các class theo cấp bậc, giúp code dễ quản lý và bảo trì

Python hỗ trợ nhiều loại kế thừa:
- Single inheritance: 1 class con kế thừa từ 1 class cha
- Multiple inheritance: 1 class con kế thừa từ nhiều class cha
- Multilevel inheritance: kế thừa nhiều cấp
- Hierarchical inheritance: nhiều class con kế thừa từ 1 class cha

Nói ngắn gọn: inheritance giúp tái sử dụng code, mở rộng và quản lý cấu trúc class.
"Inheritance là khi class con kế thừa thuộc tính và phương thức của class cha, giúp tái sử dụng code và mở rộng chức năng."

35. What are *args and ** kwargs in Python?

Trong Python, *args và **kwargs là cách truyền tham số linh hoạt cho hàm.
*args cho phép hàm nhận nhiều tham số vị trí (positional arguments) mà không cần biết trước số lượng.
Ví dụ, mình có thể truyền 2, 3, 5 hay nhiều tham số, hàm vẫn xử lý được.
**kwargs cho phép hàm nhận nhiều tham số theo dạng key-value (keyword arguments), tức là tên biến + giá trị, cũng không cần biết trước số lượng.
Nói ngắn gọn:
- *args dùng cho tham số dạng vị trí.
- **kwargs dùng cho tham số dạng từ khóa.
Cách này giúp viết hàm linh hoạt và tái sử dụng dễ dàng trong nhiều tình huống.
*"*args là để nhận nhiều tham số vị trí, còn **kwargs nhận nhiều tham số theo dạng key-value.*
Cả hai giúp hàm linh hoạt hơn, không cần biết trước số lượng tham số."

36. Do runtime errors exist in Python? Explain with an example.

Trong Python, runtime errors (lỗi thời gian chạy) hoàn toàn tồn tại. Đây là những lỗi xảy ra khi chương trình đang thực thi, nghĩa là mã nguồn có thể chạy bình thường nhưng khi gặp một tình huống cụ thể sẽ gây lỗi.
Ngoài ra, Python còn có duck typing, nghĩa là chương trình sẽ giả sử một đối tượng có khả năng thực hiện hành động nào đó; nếu đối tượng đó không hỗ trợ, lỗi runtime sẽ xảy ra.
Những lỗi này khác với lỗi cú pháp, vì lỗi cú pháp sẽ bị phát hiện ngay khi biên dịch, còn runtime errors chỉ xuất hiện khi chạy chương trình.
"Python có lỗi runtime. Chúng xảy ra khi chương trình đang chạy, ví dụ chia cho 0 (ZeroDivisionError) hoặc cộng chuỗi với số (TypeError)."

37. What are "docstrings" in Python?

Trong Python, docstrings (documentation strings) là các chuỗi đa dòng được dùng để chú thích, giải thích chức năng của một hàm, lớp hoặc module.
Docstrings được đặt bên trong ba dấu nháy kép """ hoặc ba dấu nháy đơn ''' ngay sau khai báo hàm, lớp hoặc module.
Mục đích là để người đọc hiểu chức năng, cách sử dụng mà không cần xem chi tiết code.
Python còn cung cấp thuộc tính .__doc__ để truy xuất docstring của hàm hoặc lớp.
Điều quan trọng là docstrings không phải là comment, vì chúng được lưu trữ trong runtime và có thể truy xuất, trong khi comment chỉ để đọc cho con người.
"Docstrings là chuỗi đa dòng dùng để document hàm, lớp hoặc module, giúp giải thích chức năng và có thể truy xuất bằng .__doc__."

38. How can you capitalize the first letter of a string in Python?

39. What are the generators in Python?

Trong Python, generator là một loại hàm đặc biệt giúp trả về một tập hợp các giá trị lần lượt từng phần một, thay vì trả về toàn bộ dữ liệu cùng lúc như list.
Các generator sử dụng từ khóa yield thay vì return để tạo ra giá trị. Mỗi lần gọi generator, nó sẽ tiếp tục chạy từ vị trí yield trước đó, giúp tiết kiệm bộ nhớ khi làm việc với dữ liệu lớn.
Generator thường được dùng khi cần xử lý luồng dữ liệu lớn hoặc vô hạn mà không muốn tải toàn bộ vào bộ nhớ.
Nói ngắn gọn, generator là một cách để tạo iterator hiệu quả, vừa tiết kiệm bộ nhớ, vừa dễ quản lý dữ liệu tuần tự.
"Generator là hàm trả về giá trị từng phần một bằng yield, giúp tiết kiệm bộ nhớ và tạo iterator hiệu quả cho dữ liệu lớn."

41. What is GIL?

GIL là viết tắt của Global Interpreter Lock, tức là một khóa toàn cục của trình thông dịch Python.
Nó được dùng để đồng bộ hóa việc truy cập tới các đối tượng Python, đảm bảo rằng chỉ có một thread Python được thực thi bytecode tại một thời điểm trong mỗi tiến trình.
Điều này giúp Python tránh lỗi khi nhiều thread cùng truy cập dữ liệu chung, nhưng đồng thời cũng giới hạn hiệu suất của multi-threading cho các tác vụ nặng CPU.
Tuy nhiên, GIL không ảnh hưởng nhiều tới các tác vụ I/O-bound, như đọc ghi file hoặc xử lý mạng, vì ở đó các thread thường chờ I/O.
Nói ngắn gọn, GIL giúp Python an toàn về bộ nhớ khi chạy nhiều thread, nhưng cũng là lý do Python không thực sự tận dụng được đa lõi cho các tác vụ tính toán nặng.
"GIL (Global Interpreter Lock) là khóa của Python, chỉ cho một thread thực thi bytecode tại một thời điểm. Nó bảo vệ dữ liệu nhưng giới hạn hiệu suất multi-threading CPU-bound."

43. What are the strengths of Django, Flask, and FastAPI, and what use cases are they best suited for?

Trong Python, ba framework web phổ biến là Django, Flask và FastAPI, mỗi cái có ưu điểm riêng và phù hợp với các loại dự án khác nhau:
Django: Là framework cao cấp, đầy đủ tính năng (full-stack). Có rất nhiều công cụ tích hợp sẵn như ORM, authentication, admin panel. Phù hợp với dự án lớn, phức tạp, cần phát triển nhanh mà vẫn tuân theo chuẩn, ví dụ hệ thống thương mại điện tử, mạng xã hội, nền tảng quản lý.
Flask: Là micro-framework, rất nhẹ và linh hoạt. Cho phép bạn tùy chỉnh mọi thứ và chỉ thêm những thành phần bạn cần. Phù hợp với dự án nhỏ đến trung bình, API nhỏ, prototyping hoặc ứng dụng đơn giản.
FastAPI: Framework mới, nổi bật với hiệu năng cao và hỗ trợ asynchronous. Hỗ trợ tự động tạo API docs dựa trên type hints. Phù hợp với ứng dụng web hiện đại, API nhanh, dự án có yêu cầu xử lý đồng thời nhiều request, ví dụ hệ thống microservices hoặc API backend cho mobile app.
Nói ngắn gọn: Django = lớn, đầy đủ, phát triển nhanh theo chuẩn. Flask = nhẹ, linh hoạt, cho dự án nhỏ/nhanh. FastAPI = nhanh, hiện đại, API và xử lý bất đồng bộ."

"Django mạnh về dự án lớn, full-featured; Flask nhẹ, linh hoạt cho dự án nhỏ; FastAPI nhanh, hiện đại, tối ưu cho API và async."

44. What is PIP?

Trong Python, PIP là viết tắt của Python Installer Package.
Nói đơn giản, PIP là công cụ dòng lệnh dùng để cài đặt các thư viện, module Python từ Internet.
Với PIP, bạn có thể:
Tìm và cài các gói (package) Python mà bạn cần
Quản lý các phiên bản của gói
Gỡ bỏ các gói không cần thiết
Ví dụ: nếu bạn muốn cài pandas, chỉ cần chạy lệnh pip install pandas và Python sẽ tự tải và cài đặt thư viện đó.
Nói ngắn gọn: PIP giúp cài đặt, quản lý và cập nhật các thư viện Python một cách dễ dàng.
"PIP là công cụ dòng lệnh để cài đặt và quản lý các thư viện Python từ Internet."

45. How can you ensure that your Python code is compatible with both Python 2 and Python 3?

Để đảm bảo code Python chạy được trên cả Python 2 và Python 3, bạn có thể áp dụng một số chiến lược và công cụ sau:
1. Sử dụng module __future__: Cho phép bạn import các tính năng của Python 3 vào Python 2
2. Dùng thư viện hỗ trợ tương thích: six hoặc future, giúp chuyển đổi các khác biệt giữa Python 2 và 3, như xử lý string, bytes, dict, v.v.
3. Viết code tuân theo best practices:
- Dùng print() thay vì print statement.
- Dùng range() thay vì xrange() trong Python 2.
- Sử dụng unicode literals cho chuỗi: u"hello".
- Cẩn thận với integer division, nên dùng // hoặc from __future__ import division.
4. Sử dụng công cụ chuyển đổi tự động: 2to3 giúp chuyển đổi code Python 2 sang Python 3.
Nhìn chung, bằng cách kết hợp import từ __future__, thư viện hỗ trợ và tuân theo best practices, bạn có thể viết code tương thích với cả hai phiên bản mà không gặp lỗi runtime."
"Sử dụng __future__, thư viện như six, tuân theo best practices và công cụ 2to3 để code chạy được trên cả Python 2 và 3."

46. What is the difference between %, /, // ?

Trong Python, ba toán tử này đều liên quan đến phép chia, nhưng có ý nghĩa khác nhau:
– Modulus operator (%): trả về phần dư của phép chia.
– Division operator (/): trả về kết quả phép chia thực (float), ngay cả khi chia hai số nguyên.
– Floor division operator (//): trả về kết quả phép chia làm tròn xuống số nguyên gần nhất.
Nói ngắn gọn: % lấy dư, / chia bình thường ra float, // chia làm tròn xuống ra int.
"% trả về phần dư, / chia ra float, // chia và làm tròn xuống số nguyên."

47. Why doesn't Python deallocate all memory upon exit?

Khi Python thoát, nó không giải phóng toàn bộ bộ nhớ vì có một số lý do.
Thứ nhất, một số đối tượng có vòng tham chiếu (circular references), tức là các đối tượng tham chiếu lẫn nhau. Việc này khiến Python khó xác định thứ tự giải phóng mà không gây lỗi.
Thứ hai, một số bộ nhớ được cấp phát bởi thư viện C hoặc hệ thống mà Python không thể trực tiếp giải phóng được.
Tuy nhiên, Python có cơ chế cleanup rất hiệu quả: nó sẽ cố gắng giải phóng hầu hết các đối tượng do Python quản lý, nhưng không thể đảm bảo 100% vì những lý do trên.
Tóm lại, Python giải phóng phần lớn bộ nhớ tự động, nhưng một số trường hợp đặc biệt vẫn còn tồn tại, và điều này là bình thường.
"Python không giải phóng toàn bộ bộ nhớ khi thoát vì có vòng tham chiếu và một số bộ nhớ do thư viện C cấp phát.
Nhưng Python vẫn cố gắng dọn dẹp phần lớn bộ nhớ do nó quản lý."

48. Why is a set known as unordered? Is it mutable or immutable?

Trong Python, set là một kiểu dữ liệu tập hợp các phần tử không có thứ tự, vì vậy bạn không thể truy cập các phần tử theo vị trí index như list hoặc tuple.
Set được gọi là unordered vì Python không lưu trữ các phần tử theo một thứ tự cố định, và thứ tự hiển thị có thể thay đổi.
Về tính chất, set là mutable, tức là bạn có thể thêm hoặc xóa phần tử sau khi tạo.
Tuy nhiên, các phần tử bên trong set phải immutable, nghĩa là bạn không thể có list hoặc dict làm phần tử trong set, nhưng có thể dùng số, string, tuple…
Nói ngắn gọn: set không có thứ tự, có thể thay đổi kích thước, nhưng các phần tử bên trong phải bất biến."
"Set là tập hợp không có thứ tự, mutable, nhưng các phần tử bên trong phải immutable. Bạn có thể thêm hoặc xóa phần tử, nhưng không thể truy cập theo index."

49. What is the difference between DataFrames and Series?

Trong Python, đặc biệt là thư viện pandas, có hai kiểu dữ liệu chính để lưu trữ và xử lý dữ liệu: Series và DataFrame.
Series là một mảng 1 chiều, giống như một cột trong bảng dữ liệu.
Nó có thể lưu trữ bất kỳ kiểu dữ liệu nào như số, chuỗi… và mỗi phần tử có một index duy nhất.
DataFrame là một bảng dữ liệu 2 chiều, gồm nhiều cột, mỗi cột là một Series.
DataFrame có hàng và cột, cho phép lưu trữ dữ liệu phức tạp với nhiều loại dữ liệu khác nhau theo từng cột.
Nói ngắn gọn: Series = 1 chiều, DataFrame = 2 chiều, DataFrame giống như tập hợp của nhiều Series có cùng index.
"Series là mảng 1 chiều với index, còn DataFrame là bảng 2 chiều, gồm nhiều Series với hàng và cột."

50. What does len() do?

51. What are mutable and immutable types in Python?

Trong Python, mutable và immutable là cách phân loại kiểu dữ liệu dựa trên khả năng thay đổi giá trị sau khi được tạo:
Mutable types là những kiểu dữ liệu mà bạn có thể thay đổi giá trị bên trong sau khi đã tạo.
Ví dụ: list, set, dict. Bạn có thể thêm, xóa hoặc thay đổi phần tử mà không cần tạo đối tượng mới.
Immutable types là những kiểu dữ liệu mà giá trị của chúng không thể thay đổi sau khi tạo.
Ví dụ: int, float, string, tuple. Nếu bạn muốn thay đổi giá trị, Python sẽ tạo một đối tượng mới.
Nói ngắn gọn: mutable = có thể thay đổi trực tiếp, immutable = không thể thay đổi trực tiếp, phải tạo mới nếu muốn sửa.
"Mutable là kiểu dữ liệu có thể thay đổi, ví dụ list, set, dict.
Immutable là kiểu không thể thay đổi, ví dụ int, string, tuple."

52. What are hashable and unhashable types in Python?

Trong Python, hashable và unhashable là cách phân loại kiểu dữ liệu dựa trên khả năng dùng làm key trong dictionary hoặc element trong set:
Hashable types là những kiểu dữ liệu có giá trị không thay đổi và có thể trả về giá trị hash cố định.
Điều này có nghĩa là bạn có thể dùng chúng làm key trong dictionary hoặc phần tử của set.
Ví dụ: int, float, string, tuple (nếu tuple chỉ chứa các phần tử immutable).
Unhashable types là những kiểu dữ liệu có thể thay đổi, nên Python không thể tạo giá trị hash cố định.
Vì vậy, chúng không thể dùng làm key trong dictionary hoặc phần tử trong set.
Ví dụ: list, dict, set.
Nói ngắn gọn: hashable = giá trị không đổi, dùng được làm key; unhashable = giá trị có thể thay đổi, không dùng được làm key.
"Hashable là kiểu dữ liệu bất biến, có thể làm key trong dictionary hoặc phần tử set.
Unhashable là kiểu có thể thay đổi, không dùng được làm key."

OOP

Các câu hỏi về class, instance và method

Class và instance khác nhau như thế nào trong Python?

Trong Python, class và instance là hai khái niệm khác nhau nhưng liên quan chặt chẽ:
Class là bản thiết kế (blueprint) của object.
- Class định nghĩa các thuộc tính (attributes) và phương thức (methods) mà các object của class sẽ có. Nó không chứa dữ liệu cụ thể của từng object, chỉ là khuôn mẫu.
Instance là một object được tạo ra từ class.
- Mỗi instance có dữ liệu riêng, nhưng chia sẻ phương thức với các instance khác của cùng class. Bạn tạo instance bằng cách gọi class như một hàm
Tóm lại:
Class = blueprint, instance = object cụ thể tạo từ blueprint đó. Class định nghĩa hành vi và structure, còn instance lưu trữ dữ liệu thực tế.
"Class là bản thiết kế, định nghĩa thuộc tính và phương thức. Instance là object thực tế được tạo ra từ class, có dữ liệu riêng nhưng chia sẻ hành vi của class."

__init__ và __del__ có vai trò gì?

Trong Python, __init__ và __del__ là hai method đặc biệt liên quan đến vòng đời của object:

__init__ (constructor)

- Được gọi ngay sau khi object được tạo.
- Dùng để khởi tạo dữ liệu, thiết lập giá trị mặc định hoặc thực hiện các thao tác chuẩn bị cho object.

__del__ (destructor)

- Được gọi trước khi object bị xóa khỏi memory (khi reference count = 0).
- Thường dùng để giải phóng tài nguyên, ví dụ đóng file, kết nối database, hoặc dọn dẹp object.

Lưu ý:

__del__ không được đảm bảo gọi ngay lập tức khi object mất reference, vì Python dùng garbage collector.

Nên hạn chế logic phức tạp trong __del__.

Tóm lại:

__init__ để khởi tạo object, __del__ để dọn dẹp trước khi object bị hủy."

"__init__ dùng để khởi tạo object, gán giá trị ban đầu; __del__ được gọi khi object bị xóa để dọn dẹp tài nguyên."

Bạn có thể tạo instance biến của class mà không dùng __init__ không?

Trong Python bạn vẫn có thể tạo instance của class mà không cần dùng __init__, bởi vì __init__ chỉ là constructor để khởi tạo object, không bắt buộc phải có.

Nếu class có __init__, Python sẽ gọi nó khi tạo instance; nếu không có, Python chỉ tạo instance rỗng.

Tóm lại:

__init__ tiện lợi để khởi tạo dữ liệu ngay khi tạo instance, nhưng không bắt buộc, bạn vẫn có thể tạo instance và gán thuộc tính thủ công sau đó.

"Có thể tạo instance mà không có __init__; chỉ cần class và bạn có thể gán attributes sau khi tạo object."

Sự khác biệt giữa classmethod, staticmethod và instance method trong Python là gì? Khi nào bạn dùng từng loại?

Trong Python, có ba loại phương thức trong class: instance method, classmethod và staticmethod, mỗi loại có cách hoạt động và mục đích sử dụng khác nhau.

Instance method

- Đây là loại phổ biến nhất. Phương thức này luôn nhận self làm tham số đầu tiên, đại diện cho instance (đối tượng) của class.
- Bạn dùng nó khi cần truy cập hoặc thay đổi dữ liệu của instance.

Class method

- Nhận cls làm tham số đầu tiên, đại diện cho class, không phải instance.
- Bạn dùng khi cần thao tác trên class, chia sẻ dữ liệu hoặc factory method.

Static method

- Không nhận self hay cls. Nó không cần instance hay class để hoạt động.

- Dùng khi phương thức không truy cập dữ liệu của class hay instance, chỉ thực hiện một chức năng nào đó liên quan đến class.
Tóm tắt cách dùng:
Instance method: khi làm việc với dữ liệu của từng đối tượng.
Class method: khi thao tác với dữ liệu chung của class hoặc tạo các factory method.
Static method: khi chỉ cần logic liên quan nhưng không phụ thuộc class hoặc instance.
"Instance method dùng cho dữ liệu của đối tượng, class method dùng cho dữ liệu của class, static method chỉ thực hiện logic mà không cần class hay instance."

__new__ và __init__ khác nhau như thế nào?

Trong Python, __new__ và __init__ đều liên quan đến việc tạo và khởi tạo đối tượng, nhưng có vai trò khác nhau:
__new__
- Là constructor thực sự, chịu trách nhiệm tạo ra instance mới của class.
- __new__ được gọi trước __init__, nhận class làm tham số đầu tiên (cls) và phải trả về một instance.
- Bạn thường override __new__ khi muốn tùy chỉnh quá trình tạo đối tượng, ví dụ singleton hoặc tạo immutable object.
__init__
- Là initializer, chịu trách nhiệm khởi tạo giá trị, thuộc tính của instance đã được tạo.
- __init__ nhận instance (self) làm tham số đầu tiên.
- Bạn thường override __init__ để gán giá trị mặc định, khởi tạo dữ liệu khi tạo đối tượng.
Tóm lại:
__new__ → tạo ra đối tượng mới.
__init__ → khởi tạo và gán giá trị cho đối tượng vừa tạo."
"__new__ tạo instance mới, còn __init__ khởi tạo giá trị cho instance đó."

Giải thích cơ chế method resolution order (MRO) trong Python.

Trong Python, Method Resolution Order (MRO) là trật tự mà Python tìm kiếm phương thức hoặc thuộc tính khi gọi trên một đối tượng, đặc biệt quan trọng khi một class kế thừa nhiều class (multiple inheritance).
Python sử dụng thuật toán C3 linearization để xác định MRO.
Khi bạn gọi một phương thức trên instance, Python sẽ:
Tìm trong instance class trước.
Sau đó tìm lần lượt trong các class cha theo thứ tự MRO.
Khi tìm thấy, Python sẽ dừng và dùng phương thức đó.
Có thể xem MRO của một class bằng cách dùng .__mro__ hoặc hàm mro().
Tóm lại: MRO giúp Python xác định trật tự ưu tiên trong việc tìm phương thức hoặc thuộc tính, tránh nhầm lẫn trong kế thừa đa lớp.
"MRO là thứ tự Python tìm phương thức trong kế thừa đa lớp; nó đi theo thuật toán C3, từ class hiện tại sang các class cha theo trật tự được xác định."

Khi nào bạn muốn dùng composition thay vì inheritance trong thiết kế OOP?

Trong OOP, chúng ta có hai cách chính để tái sử dụng code: inheritance (kế thừa) và composition (thành phần hóa).
Composition là khi một class sử dụng các object của class khác như là thuộc tính thay vì kế thừa trực tiếp.
Bạn nên dùng composition thay vì inheritance trong các trường hợp sau:
Không muốn phụ thuộc chặt chẽ vào class cha
Khi kế thừa, class con bị ràng buộc với class cha. Nếu class cha thay đổi, class con cũng có thể bị ảnh hưởng.
Composition giúp giảm sự phụ thuộc này, class có thể dùng các object khác mà không cần quan tâm đến nội bộ của chúng.
Khi mối quan hệ "has-a" rõ ràng hơn "is-a"
Inheritance dùng cho mối quan hệ is-a (ví dụ: Cat is-a Animal).
Composition dùng cho mối quan hệ has-a (ví dụ: Car has-a Engine).
Muốn tăng tính linh hoạt và mở rộng
Với composition, bạn có thể thay đổi hành vi của đối tượng bằng cách thay đổi các object bên trong mà không cần thay đổi class.
Điều này giúp code dễ bảo trì, mở rộng mà không phá vỡ cấu trúc class hiện tại.
Ở đây, Car dùng Engine thông qua composition, không cần kế thừa Engine.
Tóm lại:
Dùng inheritance khi mối quan hệ is-a và muốn tái sử dụng code class cha.
Dùng composition khi mối quan hệ has-a, muốn linh hoạt, dễ bảo trì và tránh phụ thuộc chặt chẽ vào class cha.
"Dùng composition khi mối quan hệ là has-a, cần linh hoạt và tránh phụ thuộc chặt chẽ; inheritance chỉ dùng cho mối quan hệ is-a."

Làm thế nào để ngăn chặn tạo nhiều instance của một class (singleton) trong Python?

Trong Python, để đảm bảo một class chỉ có một instance duy nhất, tức là Singleton pattern, bạn có một vài cách phổ biến:
1. Dùng __new__ để kiểm soát instance:
__new__ là constructor thực sự, có thể kiểm tra xem instance đã tồn tại chưa.
Nếu chưa, tạo mới; nếu đã tồn tại, trả về instance hiện có.
2. Dùng decorator hoặc metaclass:
Metaclass có thể override __call__ để kiểm soát việc tạo instance.
Đây là cách mạnh mẽ khi bạn muốn áp dụng Singleton cho nhiều class cùng lúc.
3. Dùng module Python như Singleton:
Trong Python, mỗi module chỉ được import một lần, nên bạn có thể đặt biến và hàm trong module, module sẽ tự động behave như singleton.
Tóm lại:
__new__ là cách phổ biến và trực quan.
Singleton giúp đảm bảo chỉ có một instance trong toàn bộ ứng dụng, hữu ích cho kết nối DB, logging, config, v.v."
"Dùng __new__ để kiểm tra và trả về cùng một instance, hoặc dùng metaclass/decorator để tạo Singleton, đảm bảo class chỉ có một instance duy nhất."

Các câu hỏi về kế thừa (inheritance)

Python có hỗ trợ kế thừa đa cấp (multilevel inheritance) như thế nào?

Trong Python, multilevel inheritance nghĩa là bạn có một chuỗi các lớp kế thừa, tức là class con kế thừa từ class trung gian, class trung gian lại kế thừa từ class cha, và có thể tiếp tục như vậy nhiều cấp.

Python xử lý kế thừa đa cấp tự động, và thứ tự gọi method tuân theo MRO (Method Resolution Order).

Multilevel inheritance cho phép xây dựng cây kế thừa nhiều cấp, giúp tái sử dụng code và mở rộng hành vi từ các lớp cha một cách tuần tự.

"Multilevel inheritance là khi class con kế thừa từ class trung gian, class trung gian kế thừa từ class cha, cho phép subclass dùng method từ tất cả các lớp cha theo MRO."

Khi override method, làm thế nào để gọi method của superclass?

Khi bạn override một method trong subclass nhưng vẫn muốn gọi method gốc của superclass, bạn dùng hàm super().

super() trả về một object tạm thời đại diện cho superclass, cho phép bạn gọi method đã bị override.

Tóm lại:

Dùng super().method_name() trong subclass để gọi method gốc của superclass khi override.

"Khi override method, dùng super().method_name() để gọi method gốc của superclass."

Sự khác nhau giữa single inheritance và multiple inheritance trong Python?

Trong Python, single inheritance và multiple inheritance khác nhau về số lượng superclass mà một class có thể kế thừa:

Single inheritance

- Một subclass chỉ kế thừa từ một superclass duy nhất.
- Cấu trúc đơn giản, dễ hiểu, ít xảy ra xung đột tên.

Multiple inheritance

- Một subclass có thể kế thừa từ nhiều superclass cùng lúc.
- Giúp tái sử dụng code từ nhiều class, nhưng có thể gây conflict nếu các superclass có method/attribute cùng tên.

Trong multiple inheritance, Python dùng MRO (Method Resolution Order) để xác định thứ tự gọi method.

Single inheritance đơn giản, multiple inheritance linh hoạt nhưng cần quản lý xung đột method/attribute.

Tóm lại:

Single inheritance: 1 superclass → subclass đơn giản, dễ hiểu.

Multiple inheritance: nhiều superclass → subclass linh hoạt, nhưng phải hiểu MRO để tránh xung đột.

"Single inheritance: subclass chỉ kế thừa 1 superclass. Multiple inheritance: subclass kế thừa nhiều superclass, dùng MRO để xác định thứ tự method."

Python có hỗ trợ multiple inheritance. Bạn sẽ giải quyết vấn đề diamond problem như thế nào?

Python hỗ trợ multiple inheritance, tức là một class có thể kế thừa từ nhiều class cha.
Một vấn đề thường gặp là diamond problem: khi một class con kế thừa từ hai class cha, và cả hai class cha lại cùng kế thừa từ một class ông nội, Python cần xác định phiên bản phương thức nào sẽ được gọi nếu method đó tồn tại ở nhiều lớp.
Python giải quyết diamond problem bằng C3 linearization, hay còn gọi là Method Resolution Order (MRO):
- Python sẽ tạo một thứ tự duy nhất để tìm method.
- Khi gọi một phương thức, Python sẽ tìm theo thứ tự MRO từ class con → class cha theo thứ tự linearized → class ông nội → object.
- Điều này tránh việc gọi phương thức nhiều lần từ cùng một class ông nội.
Tóm lại:
Python tự động giải quyết diamond problem nhờ MRO (C3 linearization).
Bạn có thể dùng .__mro__ hoặc mro() để kiểm tra thứ tự tìm phương thức.
Nhờ vậy, method được gọi đúng và không bị trùng lặp, dù kế thừa đa lớp.
"Python giải quyết diamond problem bằng MRO (C3 linearization), xác định thứ tự duy nhất để gọi phương thức, tránh trùng lặp và đảm bảo method được gọi đúng."

7. Khi override một method từ class cha, super() hoạt động như thế nào?

Khi bạn override một method từ class cha trong Python, bạn có thể muốn gọi lại phiên bản method gốc của class cha để tái sử dụng logic hoặc mở rộng chức năng. Đây là lúc super() xuất hiện.
super() trả về một đối tượng proxy đại diện cho class cha tiếp theo theo MRO.
Khi bạn gọi super().method(), Python sẽ tìm method trong class cha theo thứ tự MRO và gọi nó.
Điều này giúp tránh việc gọi trực tiếp tên class cha, làm code linh hoạt hơn, đặc biệt với multiple inheritance.
Tóm lại:
super() giúp gọi method của class cha theo thứ tự MRO.
Nó cực kỳ hữu ích với multiple inheritance và khi bạn muốn mở rộng thay vì thay thế hoàn toàn method của class cha.
"super() gọi method của class cha theo MRO, giúp mở rộng thay vì thay thế hoàn toàn method khi override, đặc biệt hữu ích với multiple inheritance."

8. Nếu một subclass cần mở rộng method của superclass nhưng vẫn gọi method gốc, bạn viết thế nào?

Khi một subclass muốn mở rộng method từ superclass nhưng vẫn giữ logic gốc, bạn sẽ override method trong subclass và sử dụng super() để gọi method của superclass.
Tóm lại:
Override method trong subclass.
Dùng super().method_name() để gọi method gốc từ superclass.
Sau đó thêm logic mới, như vậy bạn vừa mở rộng vừa không mất logic gốc.
"Override method trong subclass, dùng super().method() để gọi method gốc, sau đó thêm hành vi mới để mở rộng mà không mất logic gốc."

Các câu hỏi về encapsulation và properties

Làm sao để khai báo biến private trong Python?

Trong Python, để khai báo một biến private, bạn đặt tên biến bắt đầu bằng hai dấu gạch dưới __, ví dụ __name.
Python không có cơ chế private thật sự như Java hay C++, nhưng cơ chế này gọi là name mangling.
Khi bạn đặt __name, Python sẽ đổi tên nội bộ biến thành _ClassName__name để giảm nguy cơ trùng tên và truy cập ngoài class.
Cách truy cập trực tiếp bằng _ClassName__var vẫn có thể, nhưng về nguyên tắc không nên làm.
Private giúp ẩn dữ liệu quan trọng và tránh truy cập ngoài ý muốn, nhưng vẫn linh hoạt theo Pythonic way.
Tóm lại:
Private variable: __var
Giúp bảo vệ dữ liệu bên trong class, giảm rủi ro lỗi logic khi truy cập ngoài.
"Đặt tên biến bắt đầu bằng __ để làm private; Python sẽ đổi tên nội bộ để hạn chế truy cập ngoài class."

Giải thích cách truy cập biến protected (bắt đầu bằng _) có an toàn không?

Trong Python, biến protected thường được đặt tên bắt đầu bằng một dấu gạch dưới _, ví dụ _name.
Đây là quy ước, không phải cơ chế ép buộc, có nghĩa là Python vẫn cho phép truy cập trực tiếp từ bên ngoài class, nhưng về nguyên tắc bạn không nên làm vậy.
Mục đích là để báo hiệu cho lập trình viên rằng đây là thuộc tính nội bộ, nên chỉ truy cập bên trong class hoặc subclass.
Nếu bạn truy cập trực tiếp, về mặt kỹ thuật Python không cấm, nhưng có thể gây lỗi logic nếu class thay đổi cách quản lý biến này.
Tóm lại:
Truy cập biến protected về mặt ngôn ngữ là được, nhưng không an toàn về mặt thiết kế, nên chỉ truy cập trong class hoặc subclass, coi nó như một "warning" cho developer.
"Biến protected (_var) có thể truy cập từ ngoài, nhưng về nguyên tắc bạn không nên làm; chỉ dùng trong class hoặc subclass để giữ an toàn thiết kế."

Bạn dùng @property khi nào?

Trong Python, @property được dùng khi bạn muốn truy cập một method như một attribute, tức là biến một method thành property, giúp ẩn logic tính toán hoặc validation bên trong mà vẫn giữ cách gọi giống attribute.
Dùng @property:
- Khi muốn ẩn dữ liệu thực sự nhưng vẫn expose giá trị dễ dùng.
- Khi muốn tính toán động mỗi lần truy cập attribute.
- Khi muốn thêm validation hoặc logic khi gán mà không phá vỡ API hiện tại.
Tóm lại: @property giúp giữ interface gọn gàng, cho phép truy cập attribute nhưng có thể chèn logic tính toán hoặc kiểm tra dữ liệu bên trong.
"Dùng @property khi muốn gọi method như attribute, để tính toán hoặc validate dữ liệu mà vẫn giữ interface gọn gàng."

Trong Python, các biến private thực sự có phải private không? Giải thích cơ chế name mangling.

Trong Python, biến private không thực sự "private" như trong các ngôn ngữ khác như Java hay C++.
Thực tế, Python chỉ mô phỏng tính private bằng cơ chế gọi là name mangling.
Khi bạn đặt tên biến với hai dấu gạch dưới ở đầu (ví dụ __secret), Python sẽ tự động đổi tên biến đó bên trong class thành _ClassName__variableName.
Cơ chế này giúp tránh xung đột tên khi kế thừa, chứ không thật sự ngăn truy cập.
Tóm lại:
Biến private trong Python chỉ là convention (quy ước), không phải giới hạn thật sự.
Name mangling chỉ đổi tên nội bộ để tránh trùng lặp, chứ không ngăn cấm truy cập từ bên ngoài.
"Biến private trong Python không thật sự private; Python chỉ đổi tên nội bộ bằng name mangling, ví dụ __var thành _ClassName__var, để tránh trùng tên chứ không ngăn truy cập."

Khi nào nên dùng @property và @property.setter?

@property và @property.setter được dùng khi bạn muốn biến một phương thức (method) trong class thành một thuộc tính (attribute) mà vẫn kiểm soát được việc truy cập và gán giá trị.
Nói đơn giản, chúng giúp bạn ẩn logic xử lý phức tạp phía sau cú pháp thuộc tính, tức là người dùng có thể truy cập như obj.attr thay vì obj.get_attr().
Khi nào nên dùng:
- Khi bạn muốn bảo vệ hoặc kiểm soát dữ liệu nội bộ của object.
- Khi muốn thêm logic kiểm tra, tính toán hoặc logging trong lúc truy cập hoặc gán giá trị.
- Khi cần chuyển đổi từ public attribute sang property mà không phá code cũ, vì cú pháp truy cập không thay đổi.
Tóm lại:
@property → dùng để định nghĩa getter.
@property.setter → dùng để định nghĩa setter cho cùng một thuộc tính.
Giúp viết code sạch, dễ đọc và an toàn hơn.
"Dùng @property khi muốn truy cập method như thuộc tính, và @property.setter khi cần kiểm soát việc gán giá trị, ví dụ để kiểm tra, validate hoặc ẩn logic nội bộ."

Bạn làm thế nào để kiểm soát truy cập hoặc validation cho các attribute mà không phá vỡ API của class?

Cách phổ biến nhất trong Python để kiểm soát truy cập hoặc validation cho các attribute mà không phá vỡ API của class là dùng @property và @property.setter.
Điều này cho phép bạn giữ nguyên cú pháp truy cập như thuộc tính thông thường (ví dụ obj.attr), nhưng bên trong vẫn có thể thêm logic kiểm tra, validate hoặc tính toán động.
Khi làm như vậy:
API cũ vẫn hoạt động bình thường, người dùng class không cần thay đổi cách truy cập thuộc tính.
Bạn vẫn có thể thêm logic kiểm tra, logging, hoặc tính toán mà không phá vỡ code cũ.
Ngoài ra, trong một số trường hợp phức tạp hơn, bạn có thể dùng:

Descriptor (__get__, __set__, __delete__) nếu muốn kiểm soát sâu hơn.
Hoặc metaclass để can thiệp ở cấp độ toàn class.
Tóm lại:
Sử dụng @property và @property.setter là cách Pythonic, gọn gàng và an toàn nhất để kiểm soát attribute mà không làm thay đổi API.
"Dùng @property và @property.setter để thêm logic kiểm tra hoặc validation cho attribute mà vẫn giữ cú pháp truy cập như cũ, không phá vỡ API."

Các câu hỏi về polymorphism (đa hình) và duck typing

Python là ngôn ngữ duck-typed. Bạn có thể giải thích ý nghĩa này trong OOP không?

Python là ngôn ngữ duck-typed, nghĩa là kiểu dữ liệu được xác định dựa trên hành vi (behavior) chứ không phải khai báo kiểu. Nếu một object có phương thức và thuộc tính cần thiết, ta có thể dùng nó mà không quan tâm đến kiểu thực sự.
Trong OOP, điều này cho phép lập trình linh hoạt hơn — không cần kế thừa cùng class, chỉ cần tuân theo giao diện hành vi (interface by convention).
"Duck typing nghĩa là Python dựa vào hành vi của object, không cần kiểu cụ thể — nếu object có method phù hợp, ta có thể dùng nó."

Làm thế nào để nhiều class cùng implement method giống nhau mà gọi cùng tên được?

Trong OOP, nhiều class có thể định nghĩa (implement) cùng một method tên giống nhau. Khi gọi, Python sẽ tự động gọi method của class tương ứng với object thực tế — đó là cơ chế đa hình (polymorphism).
"Nhờ cơ chế đa hình, nhiều class có thể định nghĩa cùng tên method. Khi gọi, Python sẽ tự chọn method đúng theo object thật."

Giải thích operator overloading cơ bản, ví dụ với __add__.

Operator overloading là cách cho phép định nghĩa lại hành vi của các toán tử (+, -, , …) cho class do ta tạo ra. Python thực hiện điều này qua các magic method như __add__, __sub__, v.v.
"Operator overloading cho phép định nghĩa lại cách các toán tử hoạt động cho object. Ví dụ __add__ giúp tùy chỉnh hành vi của dấu +."

Python là duck-typed, vậy polymorphism trong Python khác với Java/C++ như thế nào?

Trong Python, vì là duck-typed language, nên polymorphism không phụ thuộc vào kiểu dữ liệu tĩnh (static type) như trong Java hay C++.
Cụ thể, Python không quan tâm đến kiểu của đối tượng — chỉ cần đối tượng đó có method hoặc hành vi mong muốn, thì nó được xem là hợp lệ.
Nguyên tắc này gọi là 'Duck Typing' — nếu nó 'kêu như vịt và đi như vịt', thì Python coi nó là một 'vịt'.
Trong khi đó, ở Java/C++, polymorphism thường yêu cầu:
Các lớp phải kế thừa cùng base class hoặc implement cùng interface,
Và kiểu của biến được xác định trước tại compile time.
Tóm lại:
Polymorphism trong Python linh hoạt và tự nhiên hơn — nó dựa vào hành vi của đối tượng, không dựa vào kiểu dữ liệu tĩnh.

"Python dùng duck typing, nên polymorphism dựa vào hành vi chứ không dựa vào kiểu. Chỉ cần đối tượng có method cần thiết là được, không cần cùng class hay interface như Java/C++."

Bạn có thể implement polymorphism mà không cần inheritance không? Ví dụ?

Trong Python hoàn toàn có thể thực hiện polymorphism mà không cần inheritance, nhờ vào duck typing.
Polymorphism có nghĩa là cùng một hàm hoặc phương thức có thể hoạt động với nhiều kiểu đối tượng khác nhau, miễn là chúng có cùng hành vi (method).
Python không yêu cầu các đối tượng đó phải kế thừa từ cùng một lớp cha.
"Trong Python, nhờ duck typing, ta có thể có polymorphism mà không cần inheritance — chỉ cần các object có cùng method, ví dụ như nhiều class khác nhau cùng có hàm fly()."

Giải thích operator overloading và khi nào bạn nên dùng nó.

Operator overloading nghĩa là *cho phép ta định nghĩa lại cách các toán tử (như +, -, , ==, …) hoạt động với các đối tượng do mình tạo ra.
Nói cách khác, ta có thể tùy chỉnh hành vi của các toán tử khi làm việc với class của mình.
Bạn nên dùng operator overloading khi:
Muốn code của mình trực quan và dễ đọc hơn (ví dụ: v1 + v2 thay vì v1.add(v2)),
Và khi nó phù hợp với ngữ cảnh logic của đối tượng (ví dụ: cộng vector, so sánh ma trận, nối chuỗi tuỳ chỉnh, v.v).
Tuy nhiên, không nên lạm dụng, vì nếu overload không hợp lý sẽ khiến code khó hiểu và khó bảo trì.
"Operator overloading cho phép ta định nghĩa lại cách các toán tử hoạt động với class của mình. Ví dụ, ta có thể định nghĩa __add__ để cộng hai vector. Dùng khi muốn code tự nhiên và dễ đọc hơn, nhưng không nên lạm dụng."

Các câu hỏi nâng cao / thiết kế

Bạn làm thế nào để thiết kế một class có thread-safe trong Python?

Để thiết kế một class thread-safe trong Python, tức là đảm bảo các method hoặc dữ liệu trong class không bị lỗi khi nhiều thread truy cập đồng thời, bạn cần sử dụng cơ chế đồng bộ hóa.
Trong Python, thường dùng threading.Lock hoặc các synchronization primitives khác:
- Lock (mutex) để bảo vệ các phần code critical section, nơi dữ liệu chia sẻ có thể bị thay đổi.
- RLock nếu cần lock có thể được acquire nhiều lần trong cùng một thread.
- Semaphore hoặc Event trong các trường hợp phức tạp hơn.
Ngoài ra, các lưu ý khác:
Tránh sử dụng biến global không được bảo vệ.
Sử dụng thread-safe data structures như queue.Queue nếu có thể.
Giữ critical section ngắn gọn và đơn giản để giảm blocking.
Tóm lại:
Để class thread-safe, chủ yếu là dùng lock hoặc các cơ chế đồng bộ để kiểm soát truy cập dữ liệu chia sẻ giữa các thread, và đảm bảo không có race condition xảy ra.

"Sử dụng threading.Lock hoặc các synchronization primitives để bảo vệ các critical section, đảm bảo dữ liệu chia sẻ không bị race condition khi nhiều thread truy cập."

Python có metaclass. Bạn dùng metaclass khi nào?

Trong Python, metaclass là class của một class, tức là nó quyết định cách class được tạo ra.
Khi bạn tạo một class, Python thực sự dùng metaclass để xây dựng class đó.
Bạn sẽ dùng metaclass khi cần can thiệp vào việc tạo class, ví dụ:
Thêm attributes hoặc methods tự động cho tất cả các class dùng metaclass đó.
Thực hiện validation hoặc kiểm tra cấu trúc class khi class được định nghĩa.
Triển khai Singleton pattern hoặc các design pattern phức tạp mà chỉ có metaclass mới thực hiện được ở cấp class.
Tóm lại:
Metaclass thường được dùng khi bạn cần can thiệp vào cách Python tạo class, thực hiện validation, tự động thêm logic hoặc pattern cấp class, chứ không dùng cho các logic bình thường trong method của object.
"Metaclass là class của class. Dùng khi cần can thiệp cách tạo class, ví dụ thêm attributes tự động, kiểm tra cấu trúc class, hoặc triển khai pattern như Singleton."

Làm sao để tối ưu memory footprint cho các object (ví dụ dùng __slots__)?

Trong Python, mỗi object thông thường có __dict__ để lưu trữ attributes, nên nếu bạn tạo nhiều object, memory footprint có thể tăng đáng kể.
Để tối ưu bộ nhớ, bạn có thể dùng __slots__.
__slots__ cho phép không tạo __dict__ cho object, mà chỉ cấp bộ nhớ cho các attribute đã định nghĩa trong slots. Điều này giúp tiết kiệm bộ nhớ và đôi khi tăng hiệu năng truy cập attributes.
Lưu ý:
__slots__ không áp dụng cho class cần kế thừa phức tạp trừ khi kế thừa cũng dùng slots.
Nên dùng khi tạo nhiều instance của cùng class và muốn giảm memory footprint.
Tóm lại:
Sử dụng __slots__ là cách Pythonic để tiết kiệm bộ nhớ cho object, đặc biệt với class có nhiều instance, bằng cách hạn chế attributes được phép và loại bỏ dict của object.
"Dùng __slots__ để hạn chế attributes và loại bỏ __dict__ của object, giúp tiết kiệm bộ nhớ khi tạo nhiều instance."

Bạn có thể giải thích cách Python quản lý reference và garbage collection liên quan đến object?

Trong Python, quản lý bộ nhớ dựa vào reference counting và garbage collection.
Reference counting:
- Mỗi object trong Python có một counter đếm số references đang trỏ tới nó.
- Khi một reference mới tới object được tạo, counter tăng; khi reference bị xóa, counter giảm.
- Khi counter = 0, nghĩa là object không còn được tham chiếu, Python tự động giải phóng bộ nhớ cho object đó.
Garbage collection (GC):
- Reference counting không xử lý được cyclic references (vòng tham chiếu giữa các object).

- Vì vậy Python có một garbage collector bổ sung, dựa trên tracing để phát hiện và xóa các object trong vòng tham chiếu mà không còn khả năng truy cập.

Tóm lại:

Python quản lý memory tự động bằng reference counting cho phần lớn object, còn garbage collector xử lý các trường hợp vòng tham chiếu phức tạp. Nhờ vậy, lập trình viên ít phải lo về memory leaks.

"Python dùng reference counting để giải phóng object khi không còn reference, và dùng garbage collector để xử lý cyclic references, giúp quản lý memory tự động."

1. Write a Python program that declares a string variable, prints the address of the variable, declares another int variable, and a pointer to it.

```python
# Declare a string variable
my_string = "Hello, Python!"

# Print the "address" of the string variable
print("Address of my_string:", hex(id(my_string)))

# Declare an integer variable
my_int = 42

# "Pointer" to the integer variable (in Python, just another reference)
my_int_pointer = my_int

# Print the address of the integer variable via the pointer
print("Address of my_int via pointer:", hex(id(my_int_pointer)))

# Print the value to show it works
print("Value via pointer:", my_int_pointer)
```

2. Write a program to find the sum of all even numbers from 1 to 100.

```python
# Initialize sum
sum_even = 0

# Loop through numbers from 1 to 100
for number in range(1, 101):
    if number % 2 == 0:  # Check if the number is even
        sum_even += number

# Print the result
print("Sum of all even numbers from 1 to 100 is:", sum_even)
```

3. Write a program to find the largest and smallest elements in an array.

python Copy code

```python
# Sample array
arr = [12, 45, 2, 89, 34, 7, 99, 1]

# Find the largest element
largest = max(arr)

# Find the smallest element
smallest = min(arr)

# Print the results
print("Array:", arr)
print("Largest element:", largest)
print("Smallest element:", smallest)
```

4. Write a program to check if a number is prime.

python Copy code

```python
# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False  # 0 and 1 are not prime
    for i in range(2, int(n**0.5) + 1):  # Check up to square root of n
        if n % i == 0:
            return False
    return True

# Input from user
num = int(input("Enter a number: "))

# Check and print result
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

5. Write a program to calculate the factorial of a number using recursion.

```python
# Recursive function to calculate factorial
def factorial(n):
    if n == 0 or n == 1:  # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive call

# Input from user
num = int(input("Enter a number: "))

# Calculate factorial
result = factorial(num)

# Print result
print(f"The factorial of {num} is {result}")
```

6. Write a program to count the number of words in a given string.

```python
# Input string from user
text = input("Enter a string: ")

# Split the string into words using whitespace as separator
words = text.split()

# Count the number of words
word_count = len(words)

# Print the result
print("Number of words in the string:", word_count)
```

7. Write a program to generate random numbers within a specified range.

```python
import random

# Input range from user
lower = int(input("Enter the lower bound: "))
upper = int(input("Enter the upper bound: "))

# Input how many random numbers to generate
count = int(input("Enter how many random numbers you want: "))

# Generate and print random numbers
print("Random numbers:")
for _ in range(count):
    print(random.randint(lower, upper))
```

8. Write a program to implement a bubble sort algorithm.

```python
# Bubble sort function
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap if the element is greater than the next
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Sample array
arr = [64, 34, 25, 12, 22, 11, 90]

print("Original array:", arr)

# Sort the array
bubble_sort(arr)

print("Sorted array:", arr)
```

9. Write a program to generate permutations of a given string.

python Copy code

```python
from itertools import permutations

# Input string from user
s = input("Enter a string: ")

# Generate all permutations
perm = permutations(s)

# Print permutations
print("All permutations of the string:")
for p in perm:
    print(''.join(p))
```