Python Refresher

100 Questions & Answers

GenAI: From Foundations to Production | ITER · July 2026 Cohort

RPT Consulting · contact@rptconsulting.com

Section	Questions	Topic focus
Basics	20	Types, operators, comprehensions, built-ins
Functions	15	Args, decorators, generators, closures
Classes & OOP	15	Inheritance, properties, dataclasses, ABCs
File I/O & Env	10	File ops, JSON, CSV, dotenv, pathlib
Error Handling	10	try/except, custom exceptions, retry, mock
Advanced	30	Async, Pydantic, regex, testing, deployment

Basics (20 questions)

Q1 · Basics

What are Python's key features that make it ideal for AI development?

```
Python is interpreted, dynamically typed, and has a vast ecosystem (NumPy, PyTorch, LangChain). Its readable syntax reduces boilerplate, interactive notebooks (Jupyter/Colab) enable rapid experimentation, and near-universal API SDK support makes it the default language for LLM development.
```

Q2 · Basics

What is the difference between a list and a tuple?

```
Lists are mutable (you can change their contents): my_list = [1, 2, 3]; my_list[0] = 9. Tuples are immutable: my_tuple = (1, 2, 3). Tuples are faster, hashable (usable as dict keys), and signal 'this data should not change' — useful for fixed configs like (model, temperature).
```

Q3 · Basics

How does Python handle variable scoping (LEGB rule)?

```
Python resolves names in this order: Local (inside current function) → Enclosing (outer function in closures) → Global (module level) → Built-in (print, len, etc.). Use 'global x' or 'nonlocal x' inside a function to write to outer scopes.
```

Q4 · Basics

What is the difference between '==' and 'is'?

```
'==' checks value equality: [1,2] == [1,2] is True. 'is' checks identity (same object in memory): a is b is True only if a and b point to the exact same object. Never use 'is' to compare strings or numbers in production — use '==' instead.
```

Q5 · Basics

How do f-strings work and why are they preferred?

```
F-strings embed expressions directly: name = 'Claude'; print(f'Hello {name}!'). They are faster than .format() and %, support expressions (f'{2**10}'), method calls (f'{text.upper()}'), and format specs (f'{cost:.4f}'). Essential for building dynamic prompts in LLM code.
```

Q6 · Basics

What are Python's mutable vs immutable types?

```
Immutable: int, float, str, tuple, bool, frozenset. Mutable: list, dict, set, bytearray, and custom objects. Immutables are safe as default argument values and dict keys. Never use mutable defaults: def f(x=[]) accumulates state across calls — use def f(x=None) instead.
```

Q7 · Basics

How do you unpack values in Python?

```
a, b, c = (1, 2, 3) # positional unpacking
first, *rest = [1, 2, 3, 4] # star unpacking: first=1, rest=[2,3,4]
_, important, _ = ('ignore', 'keep', 'ignore') # discard with _
Unpacking is used heavily when processing API responses and iterating over dict.items().
```

Q8 · Basics

What is a dictionary and how do you safely access keys?

```
A dict maps keys to values: config = {'model': 'gpt-4o', 'temperature': 0.7}. Safe access: config.get('max_tokens', 1024) — returns 1024 if key absent instead of raising KeyError. Check existence: 'model' in config. Merge dicts (Python 3.9+): merged = dict1 | dict2.
```

Q9 · Basics

What are sets and when do you use them?

```
Sets store unique, unordered items: unique_words = set(tokens). Fast membership test (O(1)): 'hello' in unique_words. Set operations: a | b (union), a & b (intersection), a - b (difference). Use sets to deduplicate chunk IDs, model names, or retrieved document sources.
```

Q10 · Basics

How does string formatting work for building prompts?

```
# Template with .format()
prompt = 'Summarise {doc} in {n} words.'.format(doc=text, n=50)
# F-string (preferred)
prompt = f'Summarise {text} in {max_words} words.'
# Multi-line prompts
prompt = (
f'You are a {role}.\n'
f'Answer: {question}'
)
Mastering string formatting is essential for prompt engineering.
```

Q11 · Basics

What are list comprehensions and when should you use them?

```
A concise way to build lists: squares = [x**2 for x in range(10) if x % 2 == 0]. Use for simple transformations; avoid for side-effect-heavy logic. Dict comprehension: {k: v.upper() for k, v in env.items()}. In AI code: token_costs = {m: count * rates[m] for m, count in usage.items()}.
```

Q12 · Basics

What is the difference between range() and enumerate()?

```
range(n) yields integers 0..n-1. enumerate(iterable) yields (index, value) pairs. for i, chunk in enumerate(chunks): print(f'Chunk {i}: {chunk["text"][:40]}') Always prefer enumerate() over range(len(x)) — it's more Pythonic and avoids index errors.
```

Q13 · Basics

How do *args and **kwargs work?

```
def func(*args, **kwargs) accepts any positional (*args → tuple) and keyword (**kwargs → dict) args. def log(*messages, level='info'): [print(m) for m in messages]. Used in LangChain wrappers and decorator patterns. Call with unpacking: func(*my_list, **my_dict).
```

Q14 · Basics

What are Python's comparison and logical operators?

```
Comparison: ==, !=, <, >, <=, >=. Logical: and, or, not (short-circuit). Chaining: 0 <= temperature <= 2 is valid Python (checks both conditions). Ternary: label = 'safe' if score > 0.8 else 'unsafe'. Truthiness: empty list/dict/str/0/None are all falsy.
```

Q15 · Basics

How do you read command-line arguments in Python?

```
import sys; args = sys.argv # sys.argv[0] is the script name
For structured args use argparse:
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--model', default='gpt-4o')
args = parser.parse_args()
print(args.model)
Used in CLI tools for AI scripts and batch processing pipelines.
```

Q16 · Basics

What is the difference between print() and logging?

```
print() writes to stdout — fine for quick debugging. The logging module provides levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), timestamps, and file output. import logging; logging.basicConfig(level=logging.INFO); logging.info('Token count: %d', n). Always use logging in production AI apps — LangSmith tracing builds on this pattern.
```

Q17 · Basics

How does Python's 'in' operator work across data structures?

```
'x in container' checks membership. For list/tuple: O(n) scan. For set/dict: O(1) hash lookup. For strings: substring check — 'gpt' in model_name. Practical: if model in SUPPORTED_MODELS: ... Always use sets for large membership checks (e.g., checking stop-words in tokenised text).
```

Q18 · Basics

What are Python's built-in functions you must know?

```
len(), type(), isinstance(), range(), enumerate(), zip(), map(), filter(), sorted(), reversed(), min(), max(), sum(), abs(), round(), int(), float(), str(), list(), dict(), set(), tuple(), open(), print(), input(), id(), dir(), help(). In AI code: zip() pairs prompts with responses; sorted() ranks retrieval results.
```

Q19 · Basics

How does 'zip()' work and where is it used in AI pipelines?

```
zip() pairs elements from multiple iterables: list(zip([1,2,3], ['a','b','c'])) → [(1,'a'),(2,'b'),(3,'c')]. AI use: for prompt, response in zip(prompts, responses): evaluate(prompt, response). Unzip: prompts, responses = zip(*pairs). Also used to pair embeddings with their source texts.
```

Q20 · Basics

What is None and how do you check for it?

```
None is Python's null value — a singleton. Always check with 'is': if result is None: ... Never use == for None checks (custom __eq__ can mislead). Optional type hint: from typing import Optional; def get_key(name: str) -> Optional[str]. Common in API wrappers where a response may or may not contain a value.
```

Functions (15 questions)

Q21 · Functions

What are default argument values and their gotcha?

```
def greet(name, role='user'): return f'{role}: {name}'. Gotcha — mutable defaults are shared across calls: def f(x=[]): x.append(1); return x gives [1], [1,1], [1,1,1] on successive calls. Fix: def f(x=None): x = x if x is not None else []. Always use None as default for mutables.
```

Q22 · Functions

What are lambda functions and when should you use them?

```
Lambdas are anonymous one-liner functions: double = lambda x: x * 2. Use for short callbacks: sorted(chunks, key=lambda c: c['word_count'], reverse=True). Avoid for complex logic — use a named function instead for readability. Common in map/filter: list(map(lambda t: t.strip(), tokens)).
```

Q23 · Functions

What are decorators and how do they work?

```
A decorator wraps a function to add behaviour. Example:
def timer(func):
import time
def wrapper(*args, **kwargs):
t = time.time()
result = func(*args, **kwargs)
print(f'{func.__name__} took {time.time()-t:.2f}s')
return result
return wrapper

@timer
```

```
def call_llm(prompt): ...
Used in LangChain's @tool decorator to register agent tools.
```

Q24 · Functions

What are generator functions and when are they useful?

```
Generators yield values lazily, saving memory. def read_jsonl(path): with
open(path) as f: for line in f: yield json.loads(line). Consume with: for record
in read_jsonl('log.jsonl'): process(record). Essential when streaming LLM
responses or processing large document corpora chunk by chunk.
```

Q25 · Functions

What is the difference between map(), filter(), and list comprehensions?

```
map(func, iterable) applies func to each element. filter(pred, iterable) keeps
items where pred is True. tokens = list(map(str.lower, raw_tokens)). clean =
list(filter(lambda t: len(t) > 2, tokens)). List comprehensions are usually more
readable: clean = [t.lower() for t in raw_tokens if len(t) > 2].
```

Q26 · Functions

How do closures work in Python?

```
A closure captures variables from its enclosing scope:
def make_prompt_builder(system_msg):
def build(user_input):
return f'{system_msg}\nUser: {user_input}'
return build

chat = make_prompt_builder('You are a helpful AI.')
chat('What is RAG?')
Closures are the foundation of decorators and factory functions in LangChain.
```

Q27 · Functions

What is functools.partial and where is it used?

```
partial pre-fills some arguments of a function:
from functools import partial
def call_api(prompt, model, temperature): ...
call_gpt4 = partial(call_api, model='gpt-4o', temperature=0.7)
call_gpt4('Explain attention') # only needs prompt
Used to create specialised API callers without repetition.
```

Q28 · Functions

What are type hints and why do they matter for AI code?

```
Type hints annotate expected types without enforcement:
def chunk_text(text: str, size: int = 200) -> list[dict]: ...
from typing import Optional, Union, Any
Benefits: IDE autocomplete, early bug detection with mypy, self-documenting code.
LangChain and Pydantic rely heavily on type hints for schema validation.
```

Q29 · Functions

How does recursion work and what is the recursion limit?

```
A function calling itself: def factorial(n): return 1 if n <= 1 else n *
factorial(n-1). Python's default limit is 1000 frames (sys.getrecursionlimit()).
```

```
For deep recursion use iteration or sys.setrecursionlimit(). Recursive
tree-of-thought prompt structures can use recursive builders — watch the limit.
```

Q30 · Functions

What is the difference between return and yield?

```
return exits the function and returns one value. yield suspends the function and
returns a value, resuming on next(). A function with yield becomes a generator.
Use return for single results; use yield for streaming: def stream_chunks(text,
n): i=0; while i<len(text): yield text[i:i+n]; i+=n.
```

Q31 · Functions

How do you write a retry decorator with exponential backoff?

```
import time, functools
def retry(max_tries=3, base_delay=1.0):
def decorator(func):
@functools.wraps(func)
def wrapper(*args, **kwargs):
for i in range(max_tries):
try: return func(*args, **kwargs)
except Exception as e:
if i == max_tries-1: raise
time.sleep(base_delay * 2**i)
return wrapper
return decorator
@retry(max_tries=4)
def call_llm(prompt): ...
```

Q32 · Functions

What is memoization and how do you apply it?

```
Memoization caches function results to avoid recomputation:
from functools import lru_cache
@lru_cache(maxsize=256)
def embed_text(text: str) -> tuple: ...
Critical for AI apps: avoid re-embedding identical text chunks. Note: lru_cache
only works with hashable arguments — convert lists to tuples first.
```

Q33 · Functions

What are positional-only and keyword-only parameters?

```
def func(pos_only, /, normal, *, kw_only): ...
pos_only must be passed positionally (no name). kw_only (after *) must use keyword
syntax. Example: def create_client(api_key, /, *, model='gpt-4o', timeout=30): ...
Enforces clean call signatures in SDK wrapper functions.
```

Q34 · Functions

How does Python's sorted() work with custom keys?

```
sorted(iterable, key=func, reverse=False). key is applied to each item before
comparison. Sort retrieval results by relevance score:
ranked = sorted(results, key=lambda r: r['score'], reverse=True)
Sort by multiple fields:
from operator import itemgetter
sorted(docs, key=itemgetter('date', 'score'))
```

```
sorted() always returns a new list; list.sort() sorts in-place.
```

Q35 · Functions

What is the walrus operator (:=) and when is it useful?

```
The walrus operator assigns and tests in one expression (Python 3.8+):
while chunk := file.read(4096): process(chunk)
if (n := len(tokens)) > 1000: print(f'Too long: {n} tokens')
Useful in while loops reading streams (LLM streaming responses) and avoiding
double computation.
```

Classes & OOP (15 questions)

Q36 · Classes & OOP

What are dunder (magic) methods and which ones should you know?

```
__init__ (constructor), __repr__ (developer string), __str__ (user string),
__len__, __getitem__, __setitem__, __contains__ (in operator), __call__ (make
instance callable), __enter__/__exit__ (context manager), __eq__/__lt__
(comparisons). Example: def __repr__(self): return f'Chunk(id={self.id},
words={self.word_count})'.
```

Q37 · Classes & OOP

What is the difference between @classmethod, @staticmethod, and instance methods?

```
Instance method: def process(self, text) — has access to self. @classmethod: def
from_file(cls, path) — has access to the class (cls), used as alternate
constructors. @staticmethod: def validate_model(name) — no self/cls, just a
utility grouped in the class namespace. Use classmethod for factory patterns
common in LangChain loaders.
```

Q38 · Classes & OOP

How does inheritance work in Python?

```
class AnthropicClient(BaseLLMClient):
def __init__(self, api_key):
super().__init__(api_key) # call parent __init__
self.client = Anthropic(api_key=api_key)
def complete(self, prompt): ... # override parent method
Python supports multiple inheritance: class C(A, B). MRO (Method Resolution Order)
is C → A → B.
```

Q39 · Classes & OOP

What are properties (@property) and when do you use them?

```
Properties expose computed values as attributes:
class TokenBudget:
def __init__(self, max_tokens): self._used = 0; self.max = max_tokens
@property
def remaining(self): return self.max - self._used
@remaining.setter
def remaining(self, v): self._used = self.max - v
budget.remaining # calls getter; budget.remaining = 500 # calls setter.
```

Q40 · Classes & OOP

What is dataclass and why is it useful for config objects?

```
from dataclasses import dataclass, field
@dataclass
class LLMConfig:
model: str = 'gpt-4o'
temperature: float = 0.7
max_tokens: int = 1024
stop_sequences: list = field(default_factory=list)
Auto-generates __init__, __repr__, __eq__. Add frozen=True for immutability.
Perfect for passing typed config to API wrapper classes.
```

Q41 · Classes & OOP

What is the ABC module and why use abstract base classes?

```
from abc import ABC, abstractmethod
class BaseLLM(ABC):
@abstractmethod
def complete(self, prompt: str) -> str: ...
@abstractmethod
def embed(self, text: str) -> list[float]: ...
Subclasses must implement all abstract methods or raise TypeError. LangChain's
BaseChatModel, BaseRetriever, etc. all use this pattern.
```

Q42 · Classes & OOP

How do you implement a context manager with a class?

```
class APISession:
def __enter__(self):
self.client = connect(); return self
def __exit__(self, exc_type, exc_val, tb):
self.client.close(); return False

with APISession() as session:
response = session.client.complete(prompt)
Use for DB connections, file handles, LangSmith tracing spans — ensures cleanup
even on errors.
```

Q43 · Classes & OOP

What is composition vs. inheritance and which is preferred?

```
Inheritance: 'is-a' relationship — RAGPipeline inherits from Pipeline.
Composition: 'has-a' relationship — RAGPipeline has a Retriever, a Reranker, and
an LLM. Prefer composition for flexibility: swap components without rewriting
class hierarchies. LangChain's LCEL (|) pipe syntax is pure composition.
```

Q44 · Classes & OOP

How does __slots__ improve memory efficiency?

```
class Embedding:
__slots__ = ['vector', 'text', 'doc_id']
def __init__(self, v, t, d): self.vector=v; self.text=t; self.doc_id=d
__slots__ prevents creating a __dict__ per instance, saving ~40-50 bytes each.
Critical when storing millions of embedding objects in memory for a vector store.
```

Q45 · Classes & OOP

What is method chaining and how do you implement it?

```
Return self from methods to chain calls:
class PromptBuilder:
def __init__(self): self._parts = []
def system(self, msg): self._parts.append(('system', msg)); return self
def user(self, msg): self._parts.append(('user', msg)); return self
def build(self): return self._parts

msgs = PromptBuilder().system('You are helpful.').user('What is RAG?').build()
LangChain's chain = prompt | llm | parser uses this pattern.
```

Q46 · Classes & OOP

What are class variables vs. instance variables?

```
class LLMClient:
call_count = 0 # class variable, shared by all instances
def __init__(self, model): self.model = model # instance variable, unique per
object
def complete(self, prompt): LLMClient.call_count += 1; ...
Access class var: LLMClient.call_count or self.__class__.call_count. Mutable
class variables (lists/dicts) are shared — often a bug if you expect per-instance
state.
```

Q47 · Classes & OOP

How does Python's operator overloading work?

```
class Vector:
def __add__(self, other): return Vector(self.x+other.x, self.y+other.y)
def __mul__(self, scalar): return Vector(self.x*scalar, self.y*scalar)
def __abs__(self): return (self.x**2 + self.y**2)**0.5

v1 + v2 # calls __add__. Numpy arrays and PyTorch tensors use this heavily —
understanding it helps when debugging embedding arithmetic.
```

Q48 · Classes & OOP

What is multiple inheritance and what is MRO?

```
class C(A, B): pass — C inherits from both A and B. MRO (Method Resolution Order)
defines which parent's method wins: C.__mro__ shows the lookup chain. Python uses
C3 linearization: C → A → B → object. Call all parent inits cleanly with super()
— it follows MRO automatically. Mixins (small utility classes) are the common
multiple-inheritance pattern.
```

Q49 · Classes & OOP

What is __call__ and how does it make objects callable?

```
class SimilarityScorer:
def __init__(self, threshold=0.8): self.threshold = threshold
def __call__(self, vec_a, vec_b):
score = cosine_similarity(vec_a, vec_b)
return score >= self.threshold, score

scorer = SimilarityScorer(0.75)
is_match, score = scorer(emb1, emb2)
LangChain runnable objects use __call__ to make chains behave like functions.
```

Q50 · Classes & OOP

How do you serialize and deserialize Python objects?

```
import json, pickle
# JSON (human-readable, safe):
json.dumps({'model': 'gpt-4o', 'temp': 0.7})
json.loads(json_string)
# Pickle (Python-only, fast, supports arbitrary objects):
import pickle
pickle.dump(obj, open('cache.pkl','wb'))
obj = pickle.load(open('cache.pkl','rb'))
Never unpickle untrusted data. Use JSON for configs; pickle for caching embeddings
locally.
```

File I/O & Env (10 questions)

Q51 · File I/O & Env

How do you read and write text files safely?

```
with open('data.txt', 'r', encoding='utf-8') as f: text = f.read()
with open('out.txt', 'w', encoding='utf-8') as f: f.write(result)
Always specify encoding='utf-8' — default varies by OS. Modes: 'r' read, 'w' write
(truncate), 'a' append, 'rb'/'wb' binary. The 'with' block ensures the file closes
even if an exception occurs.
```

Q52 · File I/O & Env

How do you work with JSON files in Python?

```
import json
# Read:
with open('config.json') as f: config = json.load(f)
# Write:
with open('output.json', 'w') as f: json.dump(results, f, indent=2)
# From/to string:
text = json.dumps(obj, ensure_ascii=False)
obj = json.loads(text)
indent=2 makes files human-readable. ensure_ascii=False preserves Unicode
(critical for non-English data).
```

Q53 · File I/O & Env

How do you use pathlib for file system operations?

```
from pathlib import Path
p = Path('data/chunks.jsonl')
p.exists(); p.parent.mkdir(parents=True, exist_ok=True)
p.read_text(encoding='utf-8'); p.write_text(content)
List files: list(Path('docs').glob('*.pdf'))
Prefer pathlib over os.path — cleaner, cross-platform, and object-oriented.
```

Q54 · File I/O & Env

How do you load environment variables with python-dotenv?

```
# .env file: OPENAI_API_KEY=sk-...
from dotenv import load_dotenv
import os
```

```
load_dotenv() # loads .env into os.environ
key = os.getenv('OPENAI_API_KEY')
# Fail loudly if missing:
key = os.environ['OPENAI_API_KEY'] # raises KeyError if absent
Add .env to .gitignore immediately. Never hardcode keys in source files.
```

Q55 · File I/O & Env

How do you read CSV files without pandas?

```
import csv
with open('data.csv', newline='', encoding='utf-8') as f:
reader = csv.DictReader(f)
rows = [row for row in reader]
# Write:
with open('out.csv', 'w', newline='', encoding='utf-8') as f:
writer = csv.DictWriter(f, fieldnames=['prompt','response','tokens'])
writer.writeheader(); writer.writerows(records)
csv.DictReader/DictWriter use column names as dict keys — safer than index-based
access.
```

Q56 · File I/O & Env

How do you use os and sys modules for environment inspection?

```
import os, sys
os.getcwd() # current directory
os.listdir('.') # list files
os.makedirs('output', exist_ok=True) # create dirs safely
os.environ.get('HOME', '/tmp') # read env var with default
sys.argv # command-line args
sys.path # module search paths
sys.version # Python version string
```

Q57 · File I/O & Env

How do you read a large file line by line without loading it all into memory?

```
with open('large_corpus.txt', encoding='utf-8') as f:
for line in f:
process(line.rstrip())
Python's file objects are iterators — each iteration reads one line. For JSONL
logs: for line in f: record = json.loads(line). Never use f.readlines() on
multi-GB files — it loads everything into RAM.
```

Q58 · File I/O & Env

How do you use tempfile for safe temporary storage?

```
import tempfile, os
with tempfile.NamedTemporaryFile(mode='w', suffix='.txt', delete=False) as tmp:
tmp.write(intermediate_result)
tmp_path = tmp.name
# Use tmp_path, then clean up:
os.unlink(tmp_path)
Also: tempfile.mkdtemp() for a temporary directory. Used in AI pipelines when
writing intermediate embeddings or converted documents.
```

Q59 · File I/O & Env

What is a virtual environment and why is it essential?

```
A venv isolates project dependencies from system Python and other projects.
python -m venv .venv # create
source .venv/bin/activate # activate (Linux/Mac)
.venv\Scripts\activate # activate (Windows)
pip install langchain openai # install into venv
pip freeze > requirements.txt # lock versions
Every module of this course must be run inside its own venv.
```

Q60 · File I/O & Env

How does pip work and how do you manage dependencies?

```
pip install package # latest
pip install package==1.2.3 # exact version
pip install -r requirements.txt # from lockfile
pip list # installed packages
pip show langchain # details + dependencies
pip uninstall package
For reproducibility, always pin versions in requirements.txt before sharing or
deploying.
```

Error Handling (10 questions)

Q61 · Error Handling

What is the full try/except/else/finally structure?

```
try:
result = call_api(prompt)
except RateLimitError as e:
log_and_retry(e)
except ValueError as e:
raise # re-raise unchanged
except Exception as e:
logger.error('Unexpected: %s', e)
raise
else:
save(result) # runs only if no exception
finally:
close_connection() # always runs, even on exception or return
```

Q62 · Error Handling

How do you create a custom exception hierarchy?

```
class AppError(Exception): pass
class LLMError(AppError): pass
class RateLimitError(LLMError):
def __init__(self, msg, retry_after=60):
super().__init__(msg); self.retry_after = retry_after
class ContextTooLongError(LLMError): pass
class AuthError(LLMError): pass
Catch specific before general: except RateLimitError before except LLMError.
```

Q63 · Error Handling

What is the difference between raise, raise e, and raise from e?

```
raise # re-raises current exception (preserves original traceback)
raise ValueError('bad input') # raises new exception
raise LLMError('failed') from original_err # chains exceptions, shows both
tracebacks
In production, use 'raise ... from e' when wrapping library exceptions so the root
cause is visible.
```

Q64 · Error Handling

How do you use contextlib for cleaner error handling?

```
from contextlib import suppress, contextmanager

# Suppress specific errors:
with suppress(FileNotFoundError):
os.remove('cache.pkl')

# Custom context manager:
@contextmanager
def timer(label):
t = time.time()
try: yield
finally: print(f'{label}: {time.time()-t:.2f}s')

with timer('embedding'): embed_documents(docs)
```

Q65 · Error Handling

How do you validate function inputs with assertions?

```
def embed(text: str, model: str = 'text-embedding-3-small') -> list:
assert isinstance(text, str), f'Expected str, got {type(text)}'
assert len(text.strip()) > 0, 'text must not be empty'
assert model in VALID_MODELS, f'Unknown model: {model}'
Assertions are disabled with python -O — for production, use explicit if/raise
instead. Pydantic models provide even stronger validation (used in Module 2).
```

Q66 · Error Handling

What is exception chaining and why does it matter?

```
try:
response = requests.get(url)
except requests.ConnectionError as e:
raise LLMError(f'API unreachable: {url}') from e
The 'from e' preserves the original traceback in __cause__. When debugging
production failures, exception chains show the full causal story — critical when a
high-level pipeline error masks a low-level network or auth issue.
```

Q67 · Error Handling

How do you log exceptions properly?

```
import logging
logger = logging.getLogger(__name__)

try:
result = call_llm(prompt)
except Exception:
```

```
logger.exception('LLM call failed for prompt: %.60s', prompt)
raise
logger.exception() automatically appends the full traceback. Use __name__ so log
messages show the module name — essential in multi-file projects.
```

Q68 · Error Handling

What are warnings and how do they differ from exceptions?

```
import warnings
warnings.warn('Using deprecated model', DeprecationWarning, stacklevel=2)
Warnings don't stop execution — they alert developers about future breakage.
Filter in tests: warnings.filterwarnings('error') # treat warnings as errors
LangChain emits many DeprecationWarnings as its API evolves — learn to read them.
```

Q69 · Error Handling

How do you implement a circuit breaker pattern?

```
class CircuitBreaker:
def __init__(self, max_failures=5, reset_after=60):
self.failures = 0; self.max = max_failures
self.open_until = 0; self.reset_after = reset_after
def call(self, func, *args):
if time.time() < self.open_until:
raise RuntimeError('Circuit open — API unavailable')
try:
r = func(*args); self.failures = 0; return r
except Exception:
self.failures += 1
if self.failures >= self.max: self.open_until = time.time() + self.reset_after
raise
```

Q70 · Error Handling

How do you test that code raises the correct exception?

```
import pytest

def test_empty_prompt_raises():
with pytest.raises(ValueError, match='empty'):
call_llm('')

def test_bad_model_raises():
with pytest.raises(ValueError) as exc_info:
call_llm('hello', model='gpt-99')
assert 'gpt-99' in str(exc_info.value)
pytest.raises is the standard pattern for exception testing in CI evaluation
pipelines (Module 6).
```

Advanced (30 questions)

Q71 · Advanced

What are itertools and which functions are most useful for AI data pipelines?

```
import itertools
itertools.chain(*lists) # flatten list of lists
```

```
itertools.islice(generator, 100) # take first 100 items from a generator
itertools.batched(iterable, 32) # Python 3.12; batch into groups of 32
itertools.product(models, temperatures) # Cartesian product for grid search
itertools.cycle(loaders) # round-robin over data sources
```

Q72 · Advanced

What are collections.defaultdict and collections.Counter?

```
from collections import defaultdict, Counter
# defaultdict: auto-creates missing keys
inverted_index = defaultdict(list)
for chunk_id, word in pairs: inverted_index[word].append(chunk_id)

# Counter: frequency counting
word_freq = Counter(tokens)
word_freq.most_common(10) # top 10 words
Both are essential for building search indexes and analyzing token distributions.
```

Q73 · Advanced

What is async/await and why is it critical for production LLM code?

```
import asyncio
async def fetch_completion(prompt):
response = await async_client.complete(prompt)
return response

async def process_batch(prompts):
tasks = [fetch_completion(p) for p in prompts]
return await asyncio.gather(*tasks) # run all concurrently

asyncio.run(process_batch(prompts))
Concurrent API calls dramatically reduce batch processing time — used in Module 4
RAG pipelines.
```

Q74 · Advanced

What is the difference between threading and multiprocessing?

```
threading: multiple threads share memory, limited by GIL — best for I/O-bound
tasks (API calls, file reads). multiprocessing: separate processes, bypass GIL —
best for CPU-bound tasks (embedding computation, tokenization). from
concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
with ThreadPoolExecutor(max_workers=8) as ex: results = list(ex.map(call_api,
prompts))
```

Q75 · Advanced

How do you use Pydantic for data validation?

```
from pydantic import BaseModel, Field, field_validator

class LLMConfig(BaseModel):
model: str = 'gpt-4o'
temperature: float = Field(0.7, ge=0.0, le=2.0)
max_tokens: int = Field(1024, gt=0)
```

```
@field_validator('model')
@classmethod
def check_model(cls, v):
if v not in VALID_MODELS: raise ValueError(f'Unknown: {v}')
return v

cfg = LLMConfig(temperature=1.5) # validated instantly
```

Q76 · Advanced

What are Python protocols (structural subtyping)?

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Embedder(Protocol):
def embed(self, text: str) -> list[float]: ...

class OpenAIEmbedder:
def embed(self, text): return client.embed(text) # no explicit inheritance needed!

isinstance(OpenAIEmbedder(), Embedder) # True
Protocols enable duck typing with type safety — used in LangChain's retriever
interfaces.
```

Q77 · Advanced

What is the __init__.py file and how do Python packages work?

```
A directory with __init__.py is a package. Subdirectories become sub-packages.
my_project/
__init__.py
llm/
__init__.py
client.py
rag/
pipeline.py

from my_project.llm.client import OpenAIClient
__init__.py can re-export: from .client import OpenAIClient # allows 'from
my_project.llm import OpenAIClient'
```

Q78 · Advanced

What is monkey patching and why is it risky?

```
Replacing or adding attributes on an existing class/module at runtime:
import openai
orig = openai.ChatCompletion.create
def patched(*a, **kw): log(); return orig(*a, **kw)
openai.ChatCompletion.create = patched
Useful for testing/mocking but dangerous in production — hard to debug, breaks
upgrades. Prefer proper mocking (unittest.mock) in tests instead.
```

Q79 · Advanced

How do you profile Python code for performance bottlenecks?

```
import cProfile, pstats
with cProfile.Profile() as pr:
run_pipeline(docs)
stats = pstats.Stats(pr)
stats.sort_stats('cumulative').print_stats(20)

# For line-level profiling:
# pip install line_profiler
# @profile decorator on slow function

# Quick timing:
import timeit; timeit.timeit('embed(text)', number=100, globals=globals())
Profile before optimizing — embedding and chunking are usually the bottlenecks.
```

Q80 · Advanced

What is __all__ in a Python module?

```
__all__ = ['MyClass', 'helper_func'] # defines public API
When someone does 'from module import *', only names in __all__ are imported.
Without __all__, everything not starting with _ is exported — often too much.
Defining __all__ makes your package's public surface explicit and IDE-friendly.
```

Q81 · Advanced

How do you implement the Singleton pattern in Python?

```
class ConfigManager:
_instance = None
def __new__(cls, *args, **kwargs):
if cls._instance is None:
cls._instance = super().__new__(cls)
return cls._instance

c1 = ConfigManager(); c2 = ConfigManager(); assert c1 is c2 # same object
Used for shared API client pools and global config objects in AI applications.
```

Q82 · Advanced

What are Python's dunder attributes on modules and classes?

```
__name__: module name ('__main__' when run directly, else module path)
__file__: path to the .py file
__doc__: docstring
__dict__: all attributes
__module__: class's defining module
__bases__: parent classes
Common pattern: if __name__ == '__main__': main() — runs only when executed
directly, not when imported. Use in every script.
```

Q83 · Advanced

What is copy vs deepcopy and when does it matter?

```
import copy
shallow = copy.copy(obj) # copies container, not nested objects
deep = copy.deepcopy(obj) # copies everything recursively

Example: docs = [{'text': 'hello', 'meta': {'page': 1}}]
```

```
shallow_copy = copy.copy(docs)
shallow_copy[0]['meta']['page'] = 99 # also changes docs[0]['meta']['page']!
Use deepcopy when you need fully independent data — e.g., cloning prompt templates
with nested dicts.
```

Q84 · Advanced

How does Python's garbage collector work?

```
Python uses reference counting as the primary GC mechanism — when an object's
refcount hits 0, it's freed immediately. A cyclic GC handles reference cycles (A
→ B → A). import gc; gc.collect() # force a collection cycle
In AI apps with large embedding arrays or model caches, use del obj and
gc.collect() to free GPU/CPU memory between batches.
```

Q85 · Advanced

What are metaclasses and why do they exist?

```
A metaclass is the class of a class — it controls how classes are created. type is
the default metaclass. Custom metaclass:
class SingletonMeta(type):
_instances = {}
def __call__(cls, *args, **kwargs):
if cls not in cls._instances: cls._instances[cls] = super().__call__(*args,
**kwargs)
return cls._instances[cls]

class Config(metaclass=SingletonMeta): ...
Pydantic's BaseModel uses a metaclass to process field annotations at class
creation time.
```

Q86 · Advanced

How do you use dataclasses with post_init and field validators?

```
from dataclasses import dataclass, field
@dataclass
class RAGConfig:
chunk_size: int = 200
overlap: int = 50
top_k: int = 5

def __post_init__(self):
if self.overlap >= self.chunk_size:
raise ValueError('overlap must be less than chunk_size')
if self.top_k < 1:
raise ValueError('top_k must be >= 1')
__post_init__ runs after __init__ — perfect for cross-field validation.
```

Q87 · Advanced

What is a descriptor protocol in Python?

```
Descriptors implement __get__, __set__, __delete__ to control attribute access:
class PositiveFloat:
def __set_name__(self, owner, name): self.name = name
def __get__(self, obj, type=None): return obj.__dict__.get(self.name)
```

```
def __set__(self, obj, value):
if value <= 0: raise ValueError(f'{self.name} must be > 0')
obj.__dict__[self.name] = value

class LLMConfig:
temperature = PositiveFloat()
Pydantic's field validation uses descriptors internally.
```

Q88 · Advanced

How do you write unit tests with pytest?

```
# test_pipeline.py
import pytest
from rag.pipeline import chunk_text, build_index

def test_chunk_returns_list(): assert isinstance(chunk_text('hello world', 2),
list)
def test_chunk_size(): chunks = chunk_text('a ' * 100, chunk_size=10, overlap=2)
assert all(c['word_count'] <= 10 for c in chunks)
def test_empty_text_raises():
with pytest.raises(ValueError): chunk_text('')

Run: pytest -v. CI pipeline in Module 6 runs these automatically on every commit.
```

Q89 · Advanced

How do you use unittest.mock to test API calls?

```
from unittest.mock import MagicMock, patch

def test_llm_call_uses_retry():
with patch('myapp.client.openai.chat.completions.create') as mock_create:
mock_create.side_effect = [RateLimitError('429'), {'content': 'Hello'}]
result = call_llm_with_retry('hi', max_retries=2)
assert result == 'Hello'
assert mock_create.call_count == 2
Mocking avoids real API calls in tests — keeps tests fast and free.
```

Q90 · Advanced

What are Python's string methods essential for text preprocessing?

```
text.strip() / .lstrip() / .rstrip() # remove whitespace
text.split('\n') # split on delimiter; text.splitlines()
text.replace(' ', ' ') # replace substring
text.lower() / .upper() / .title()
text.startswith('##') / .endswith('.')
' '.join(tokens) # join list to string
re.sub(r'\s+', ' ', text) # collapse whitespace with regex
All are used in document ingestion and chunk pre-processing pipelines.
```

Q91 · Advanced

How does re (regex) module work for text cleaning?

```
import re
re.sub(r'<[^>]+>', '', html) # strip HTML tags
```

```
re.sub(r'\s+', ' ', text).strip() # normalise whitespace
re.findall(r'\b\w+\b', text) # extract words
re.split(r'(?<=[.!?])\s+', text) # split into sentences
pattern = re.compile(r'\d{4}-\d{2}-\d{2}') # compile for repeated use
Essential for cleaning scraped web text before feeding to an LLM.
```

Q92 · Advanced

What is the difference between __repr__ and __str__?

```
__repr__: unambiguous developer representation — should ideally be eval()-able:
def __repr__(self): return f'Chunk(id={self.id!r}, words={self.word_count})'.
__str__: readable user-facing string — used by print() and str(). If only __repr__
is defined, str() falls back to it. Always define __repr__ on custom classes — it
makes debugging LangChain pipelines much easier.
```

Q93 · Advanced

How do you handle Unicode and encoding issues?

```
# Always open files with explicit encoding:
with open(f, encoding='utf-8', errors='replace') as fp: ...
# Detect encoding of unknown files:
pip install chardet
import chardet
chardet.detect(raw_bytes)
# Normalise Unicode:
import unicodedata; unicodedata.normalize('NFC', text)
Multi-language AI datasets commonly mix encodings — always normalise before
embedding.
```

Q94 · Advanced

What are Python's built-in sorting algorithms and their complexity?

```
Python uses Timsort (O(n log n) worst, O(n) best for sorted data). list.sort() and
sorted() both use Timsort. sorted(results, key=lambda r: r['score'], reverse=True)
# rank by relevance
heapq.nlargest(5, results, key=lambda r: r['score']) # top-K without full sort
heapq.nlargest is more efficient than sort when K << N — important for large
retrieval sets.
```

Q95 · Advanced

What is the Global Interpreter Lock (GIL) and how does it affect AI code?

```
The GIL allows only one Python thread to execute bytecode at a time. Impact:
CPU-bound threading doesn't speed up Python computation. Workarounds: use
multiprocessing for CPU-bound work (tokenization, chunking in parallel), use
asyncio/threading for I/O-bound work (API calls, file reads). NumPy and PyTorch
release the GIL during heavy computation — that's why they're fast in threads.
```

Q96 · Advanced

How do you use __slots__ with inheritance?

```
class Base:
__slots__ = ['x', 'y']
def __init__(self, x, y): self.x=x; self.y=y
```

```
class Child(Base):
__slots__ = ['z'] # only declare NEW slots
def __init__(self, x, y, z): super().__init__(x, y); self.z=z

Child inherits x, y from Base.__slots__ and adds z. If Base doesn't use __slots__,
Child still gets a __dict__ from Base — __slots__ must be end-to-end.
```

Q97 · Advanced

How does Python's import system work?

```
import searches sys.path in order: current dir → PYTHONPATH entries → stdlib →
site-packages. from package import module triggers package/__init__.py first.
Circular imports (A imports B, B imports A) cause ImportError — resolve by moving
shared code to a third module or using local imports inside functions.
importlib.import_module('langchain.llms') imports dynamically at runtime — useful
for plugin systems.
```

Q98 · Advanced

What are common Python anti-patterns to avoid in AI code?

```
1. Mutable default args: def f(x=[]) — use None.
2. Bare except: except: — always name the exception.
3. Using + in loop to build strings — use list + ''.join().
4. Hardcoded API keys in source files.
5. Loading entire file into memory unnecessarily — iterate lines.
6. Not closing files — always use 'with'.
7. Comparing to None with ==.
8. Catching and silently swallowing exceptions in production.
```

Q99 · Advanced

How do you document Python code professionally?

```
Use Google-style or NumPy-style docstrings:
def embed(text: str, model: str = 'text-embedding-3-small') -> list[float]:
'''Embed a text string using the specified OpenAI model.

Args:
text: Input string to embed. Must be non-empty.
model: OpenAI embedding model name.

Returns:
List of floats representing the embedding vector.

Raises:
ValueError: If text is empty.
'''
Generate docs: pip install sphinx or mkdocs.
```

Q100 · Advanced

How do you prepare a Python project for sharing and deployment?

```
Project structure:
my_project/
src/my_project/ # source code
```

```
tests/ # pytest tests
.env.example # template (no real keys)
requirements.txt # pip freeze
README.md
.gitignore

.gitignore must include: .env, .venv/, __pycache__/, *.pyc, *.pkl, .DS_Store
Before sharing: remove all API keys, add README with setup steps, ensure tests
pass with pytest. For Docker deployment: add a Dockerfile with pip install -r
requirements.txt.
```