01 Variables and Data Types

Overview

Variables bind names to objects at runtime. Python is dynamically typed: names can be rebound to any type. Common built‑ins: int, float, bool, str, NoneType. Mutable vs immutable types matter for behavior and performance. Use descriptive snake_case names and constants in UPPER_CASE.

Example Code

```
x = 10              # int
pi = 3.14159         # float
name = "Elvin"       # str
is_active = True     # bool
n = None             # NoneType

# multiple assignment and swapping
a, b = 1, 2
a, b = b, a

print(type(x), type(pi), type(name), type(is_active), type(n))
```

Explanation

We assign values without declaring types; Python infers them. Swapping uses tuple unpacking. Immutables (ints, strings, tuples) produce new objects on change; mutables (lists, dicts, sets) are modified in place. This affects function arguments and copying semantics.

Question

What is the output?

```
x = 1
x = x + 2
print(type(x), x)
```

Answer

<class 'int'> 3

02 Numbers and Math

Overview

Python supports integers with arbitrary precision, floats (binary IEEE‑754), complex numbers, and decimals (exact base‑10 via decimal module). Use math for common functions and fractions for rational arithmetic.

Example Code

```
import math, decimal, fractions

x = 7 // 3    # floor division -> 2
y = 7 / 3     # true division  -> 2.333...
z = 7 % 3     # modulo          -> 1
p = 2 ** 10   # exponentiation  -> 1024

print(math.sqrt(16), math.ceil(2.1), math.floor(2.9))

# Decimal for money
D = decimal.Decimal
price = D("0.10") + D("0.20")
print("Decimal price:", price)

# Fractions
f = fractions.Fraction(2, 3) + fractions.Fraction(1, 6)
print("Fraction:", f)
```

Explanation

// truncates towards negative infinity for ints. / always returns float. Decimal avoids binary rounding issues in finance. Fractions keep exact rational results. Prefer math.isclose for float comparisons.

Question

Why is 0.1 + 0.2 != 0.3 in floats, and how to compare properly?

Answer

Binary floating point cannot represent some decimals exactly; use math.isclose(0.1+0.2, 0.3) or the decimal module for exact arithmetic.

03 Strings Basics

Overview

Strings are immutable sequences of Unicode code points. Supports indexing, slicing, concatenation, repetition, and membership tests. Useful methods: lower, upper, title, replace, find, split, join.

Example Code

```
s = "Python"
print(s[0], s[-1])     # indexing
print(s[1:4])          # slicing 'yth'
print(len(s))
print("py" in s.lower())

name = "elvin babanli"
print(name.title())
print(name.replace(" ", "_"))
```

Explanation

Indexing accesses characters; negative indices count from the end. Slices are half□open [start:end). Immutability means operations create new strings. Join is more efficient than string concatenation in loops.

Question

What happens if you assign s[0] = 'X'? Why?

Answer

TypeError: strings are immutable; you must create a new string like 'X' + s[1:].

04 String Methods and Formatting

Overview

Format values with f-strings, str.format, or %-format (legacy). Control width, precision, alignment. Sanitize/escape user data if building SQL/HTML; prefer parameterized queries and templates.

Example Code

```
user = "Elvin"; score = 97.456
print(f"{user} scored {score:.2f}")
print("{u} scored {s:.1f}".format(u=user, s=score))
print("%s scored %.0f" % (user, score))

# alignment
print(f"|{user:^10}|{score:>8.2f}|")
```

Explanation

f-strings evaluate expressions inline, very readable. Format specs like .2f control decimal places; >, <, ^ align within a field of given width.

Question

Format PI=3.14159 as '3.14' using f-string.

Answer

```
PI = 3.14159; print(f"{PI:.2f}")
```

05 Input and Output

Overview

input() returns text; convert types as needed. print() supports sep, end, file. For large outputs, join strings or write to files.

Example Code

```
name = input("Enter name: ")
age = int(input("Age: "))
print("Hello", name, "age:", age, sep=" | ", end="!\n")
with open("hello.txt","w", encoding="utf-8") as f:
    print(f"Hi {name}", file=f)
```

Explanation

input always returns str. Casting ensures correct arithmetic. print can redirect to a file. For performance, avoid concatenating many small strings; use join or write once.

Question

Why convert input to int for numeric operations?

Answer

Because input() returns str; numeric ops require int/float.

06 Conditionals

Overview

Use if/elif/else to branch logic. Values have truthiness: 0, '', [], {}, set(), None evaluate to False. Chain comparisons like 0 < x < 10.

Example Code

```
x = 42
if x > 50:
    print("big")
elif x == 42:
    print("exactly 42")
else:
    print("small")

s = ""
if not s:
    print("empty string")
```

Explanation

Truthiness simplifies checks. Chain comparisons are more readable than combining with and. Remember to handle edge cases in elif chains.

Question

Write a condition that checks if s is a non‑empty string.

Answer

```
if isinstance(s, str) and s: ...
```

07 Match Case (Pattern Matching)

Overview

Python 3.10+ structural pattern matching simplifies complex branching by matching values and shapes.

Example Code

```
def classify_http(code):
    match code:
        case 200:
            return "OK"
        case 400 | 404:
            return "Client error"
        case 500:
            return "Server error"
        case _:
            return "Unknown"

print(classify_http(404))
```

Explanation

The match statement compares the subject to patterns in order; pipes combine alternatives; _ is the wildcard. Pattern matching also destructures sequences and dicts.

Question

Match a tuple (x, y) where x==0 and extract y.

Answer

```
match pt:
    case (0, y): print("On Y axis", y)
```

08 Loops

Overview

for iterates over any iterable; while repeats while a condition holds. Use break, continue, else on loops. Prefer enumerate and zip over manual index counters.

Example Code

```
# for with range
for i in range(1, 6):
    print("Number:", i)

# iterate a list
fruits = ["apple","banana","cherry"]
for f in fruits:
    print(f)

# while
count = 0
while count < 3:
    print("Count:", count)
    count += 1

# enumerate
for idx, val in enumerate(fruits, 1):
    print(idx, val)
```

Explanation

range is lazy in Py3. The loop else runs if no break occurred. enumerate gives index+value cleanly; zip pairs multiple iterables.

Question

Print only even numbers from 1..10.

Answer

```
for i in range(1,11):
```

```
if i % 2 == 0:
    print(i)
```

09 Comprehensions

Overview

List/set/dict comprehensions provide concise, readable transformations and filters; often faster than explicit loops.

Example Code

```
squares = [x*x for x in range(10)]
evens = {x for x in range(10) if x % 2 == 0}
mapping = {x: x*x for x in range(5)}
print(squares, evens, mapping)
```

Explanation

Comprehensions evaluate expression for each item; optional if clause filters. Dict/set comprehensions mirror list comprehensions with braces.

Question

Build a dict mapping numbers 1..5 to their cubes.

Answer

```
cubes = {n: n**3 for n in range(1,6)}
```

10 Functions and Arguments

Overview

Functions encapsulate logic. Parameters can be positional, keyword‑only, defaulted, *args, **kwargs. Document behavior and types with docstrings and hints.

Example Code

```
def greet(name, title="Mr/Ms"):
    return f"Hello {title} {name}"

def add(*nums):
    return sum(nums)

def show(**kw):
    print(kw)

print(greet("Elvin"))
print(add(1,2,3))
show(role="admin", active=True)
```

Explanation

Default values are evaluated once at definition. Use None as default for mutable parameters and create inside. *args collects extra positionals, **kwargs collects extra keywords.

Question

Write power(base, exp=2) that returns base**exp.

Answer

```
def power(base, exp=2): return base ** exp
```

11 Recursion

Overview

Recursion solves problems by calling the same function on smaller inputs. Requires a base case to stop and a recursive step to progress. Useful for divide‑and‑conquer (quick sort), tree/graph traversal, and mathematical sequences.

Example Code

```
def factorial(n):
    """Return n! using recursion"""
    if n == 0 or n == 1:   # Base case
        return 1
    return n * factorial(n-1)  # Recursive step

print(factorial(5))  # 120
```

Explanation

Each call reduces n by 1 until it hits 0/1; then the call stack unwinds multiplying results. Recursion depth is limited (sys.getrecursionlimit). Tail recursion is not optimized in CPython, so beware deep recursion.

Question

Write recursive fibonacci(n). What is its time complexity?

Answer

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
# Exponential O(phi^n); memoization reduces to O(n).
```

12 Modules and Packages

Overview

Modules are .py files; packages are directories with __init__.py (or implicit namespaces). Import using absolute or relative paths. __name__ == '__main__' guards script entry points.

Example Code

```
# mypkg/mymod.py
def hello(name): return f"Hello {name}"

# main.py
from mypkg.mymod import hello
if __name__ == "__main__":
    print(hello("Elvin"))
```

Explanation

Absolute imports are clearer and robust. The main guard prevents code from executing on import. Use virtual environments to isolate dependencies.

Question

How do you run a module as a script?

Answer

```
python -m package.module  # runs module's __main__ context
```

13 Exceptions

Overview

Use try/except/else/finally to handle errors. Catch specific exceptions, not bare except. Raise custom exceptions for domain errors. Use context managers for cleanup.

Example Code

```
try:
    val = int("x")   # ValueError
except ValueError as e:
    print("Bad value:", e)
else:
    print("Parsed OK")
finally:
    print("Done")

def divide(a,b):
    if b == 0:
        raise ZeroDivisionError("b cannot be 0")
    return a/b
```

Explanation

Except runs on error; else runs when no exception; finally always runs. Raising communicates failure to callers. Log exceptions with context; avoid swallowing them silently.

Question

Safely parse integer from input string s; if invalid, return 0.

Answer

```
def to_int(s):
    try:
        return int(s)
    except ValueError:
        return 0
```

14 File Handling

Overview

Use open with context managers to read/write text and binary files. Understand newline, encoding, buffering. For large files, stream line by line.

Example Code

```
# write
with open("demo.txt","w",encoding="utf-8") as f:
    f.write("hello\nworld")

# read all
with open("demo.txt","r",encoding="utf-8") as f:
    data = f.read()

# stream
with open("demo.txt","r",encoding="utf-8") as f:
    for line in f:
        print(line.rstrip())
```

Explanation

The with block ensures files close on errors. Specify UTF‑8 encoding. Iterating over the file yields lines efficiently. For CSV/JSON, use dedicated modules.

Question

How to count lines in a large file efficiently?

Answer

```
with open(path,'r',encoding='utf-8') as f:
    count = sum(1 for _ in f)
```

15 Context Managers

Overview

Context managers guarantee setup/teardown. Implement with __enter__/__exit__ or contextlib contextmanager.

Example Code

```
from contextlib import contextmanager
@contextmanager
def opening(path):
    f = open(path, 'r', encoding='utf-8')
    try:
        yield f
    finally:
        f.close()

with opening('demo.txt') as f:
    print(f.readline().strip())
```

Explanation

The decorated function yields a resource; after the with block, __exit__ closes it even on exceptions. This avoids leaks and duplicated try/finally code.

Question

Why use context managers instead of manual open/close?

Answer

They ensure cleanup on all paths, including exceptions, reducing bugs.

16 Iterators and Generators

Overview

Iterators implement __iter__/__next__. Generators yield values lazily; generator expressions create on‑the‑fly sequences. Useful for streaming data without storing all in memory.

Example Code

```
def countdown(n):
    while n>0:
        yield n
        n -= 1

for v in countdown(3):
    print(v)

# generator expression
squares = (x*x for x in range(5))
print(list(squares))
```

Explanation

Each next() resumes execution after the last yield. Using generators reduces memory footprint and can improve performance on pipelines.

Question

Build a generator that yields even numbers up to n (inclusive).

Answer

```
def evens(n):
    for x in range(0, n+1, 2):
        yield x
```

17 Lambda, map, filter, reduce

Overview

Small anonymous functions plus functional tools for transforms, filters, and reductions. Prefer comprehensions for clarity when possible; use reduce for associative operations.

Example Code

```
from functools import reduce
nums = [1,2,3,4]
print(list(map(lambda x: x*x, nums)))
print(list(filter(lambda x: x%2==0, nums)))
print(reduce(lambda a,b: a+b, nums, 0))
```

Explanation

map applies a function to each item; filter keeps those where predicate is True; reduce combines values into one using an accumulator. Initial value prevents errors on empty sequences.

Question

Keep even squares for 0..10 using one expression.

Answer

```
vals = [x*x for x in range(11) if (x*x)%2==0]
```

18 Lists

Overview

Mutable sequence supporting append, extend, insert, remove, pop, sort, reverse, slicing. Be aware of shallow vs deep copies and list multiplication pitfalls.

Example Code

```
arr = [3,1,2]
arr.append(5); arr.extend([6,7])
arr.insert(1, 9)
arr.remove(1)        # first match
arr.sort()           # in place
arr2 = sorted(arr)   # new list

# shallow copy
b = arr[:]

# list of lists pitfall
matrix = [[0]*3]*3
matrix[0][0] = 1
print(matrix)  # unexpected
```

Explanation

matrix uses the same inner list three times; use [[0 for _ in range(3)] for _ in range(3)]. Shallow copy duplicates the outer list only. sorted returns a new list, leaving original intact.

Question

Remove duplicates from a list while preserving order.

Answer

```
seen=set(); out=[]
for x in arr:
    if x not in seen:
        out.append(x); seen.add(x)
```

19 Tuples

Overview

Immutable sequences; good for fixed collections and dict keys. Support unpacking, including starred and nested patterns.

Example Code

```
t = (1,2,3)
x,y,z = t
x, *mid, z = (1,2,3,4,5)
print(mid)

# single-element tuple
single = (42,)
```

Explanation

Immutability enables hashing (if elements are hashable). Unpacking improves readability. Use tuples for heterogeneous records that shouldn't change.

Question

Why can tuples be dict keys but lists cannot?

Answer

Tuples are immutable and hashable (if elements are hashable); lists are mutable and unhashable.

20 Dictionaries

Overview

Mapping of keys to values. Methods: get, items, keys, values, setdefault, update. Merge with | (3.9+). Use dict comprehensions. Keys must be hashable.

Example Code

```
user = {"name":"Elvin","age":23}
user["city"]="Warsaw"
print(list(user.keys()))
print(user.get("missing","default"))

# merge
extra = {"active":True}
merged = user | extra
```

Explanation

Access with square brackets raises KeyError if missing; get returns default. items returns (key,value) pairs for iteration. Merging with | creates a new dict.

Question

Create a dict counting word frequencies in a list words.

Answer

```
freq = {}
for w in words:
    freq[w] = freq.get(w, 0) + 1
```

21 Sets

Overview

Unordered collections of unique items. Support union, intersection, difference, symmetric difference. Great for membership tests and deduplication.

Example Code

```
a = {1,2,3}; b = {3,4,5}
print(a | b, a & b, a - b, a ^ b)
nums = [1,2,2,3]
print(list(set(nums)))
```

Explanation

Set operations are O(n) on average for membership and insertion. Converting to set removes duplicates but loses order; for ordered dedup, use dict.fromkeys or a seen set loop.

Question

Find common elements between two lists a and b.

Answer

```
common = set(a) & set(b)
```

22 Collections Module

Overview

Specialized containers: Counter (multiset), deque (fast appends/pops at both ends), defaultdict (auto defaults), namedtuple (lightweight classes).

Example Code

```
from collections import Counter, deque, defaultdict, namedtuple
print(Counter("banana").most_common(1))
dq = deque([1,2,3]); dq.appendleft(0)
dd = defaultdict(int); dd['x'] += 1
Point = namedtuple('Point','x y'); p = Point(1,2)
```

Explanation

Counter simplifies frequency tasks; deque is efficient for queues; defaultdict avoids KeyError; namedtuple provides attribute access without full classes.

Question

Get the two most common letters in 'abracadabra'.

Answer

```
Counter('abracadabra').most_common(2)
```

23 Itertools

Overview

Powerful iterator utilities: product, permutations, combinations, accumulate, chain, groupby, islice. Efficient and memory-friendly.

Example Code

```
import itertools as it
print(list(it.accumulate([1,2,3])))
print(list(it.chain([1,2],[3,4])))
print(list(it.permutations('abc',2)))
print(list(it.combinations('abc',2)))
print(list(it.product([0,1], repeat=3)))
```

Explanation

---

accumulate computes running totals; chain flattens iterables; permutations/combinations generate arrangements; product builds Cartesian products.

Question

Build an infinite counter starting at 10, step 2, and print first 3 values.

Answer

```
c = it.count(10,2)
for _ in range(3): print(next(c))
```

24 Datetime and Time

Overview

Work with dates/times: now, arithmetic with timedelta, formatting/parsing with strftime/strptime, timezone handling with zoneinfo (3.9+).

Example Code

```
from datetime import datetime, timedelta
now = datetime.now()
tomorrow = now + timedelta(days=1)
print(now.strftime('%Y-%m-%d %H:%M'))
parsed = datetime.strptime('2025-01-15','%Y-%m-%d')
print(parsed.date())
```

Explanation

strftime formats datetimes to strings; strptime parses strings. Timedelta supports addition/subtraction. For tz-aware datetimes, use zoneinfo for correct conversions.

Question

Parse '2024-12-31 23:59' and add 2 minutes; print result.

Answer

```
dt = datetime.strptime('2024-12-31 23:59','%Y-%m-%d %H:%M')
print(dt + timedelta(minutes=2))
```

25 Random

Overview

Pseudo-random utilities: random, randint, choice, shuffle, sample. Seed for reproducibility. Don't use for crypto; use secrets instead.

Example Code

```
import random as R
R.seed(42)
print(R.random())
print(R.randint(1,6))
items = [1,2,3,4,5]
R.shuffle(items); print(items)
print(R.sample(range(100), 5))
```

Explanation

Seeding fixes the random sequence for testing. shuffle mutates the list. sample returns a new list of unique items. For secure tokens, use secrets.token_hex.

Question

Pick 3 distinct random numbers from 1..10.

Answer

```
import random as R
print(R.sample(range(1,11), 3))
```

26 Regular Expressions

Overview

Search, validate, and extract patterns with the re module. Use raw strings r'' for patterns. Common tokens: \d digits, \w word, \s space, . any char, ^/$ anchors, [] classes, () groups, ? * + quantifiers.

Example Code

```
import re
m = re.search(r"(\d+)", "Item42")
print(m.group(1))
print(re.findall(r"\b\w+\b", "Hello, world!"))

# simple email (basic)
pat = r"^[\w.-]+@[\w.-]+\.[a-zA-Z]{2,}$"
print(bool(re.match(pat, "a@b.com")))
```

Explanation

Always compile complex patterns for speed. Use groups to capture subparts. Escape special characters. For readability, use re.VERBOSE with comments.

Question

Write a regex that captures a 3-letter uppercase word at start of line.

Answer

```
pat = r"^[A-Z]{3}\b"
```

27 OOP Basics

Overview

Define classes to model data/behavior. Use __init__ for construction, methods for behavior, and __repr__ for debugging. Favor composition over inheritance by default.

Example Code

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        return f"Hi, I'm {self.name}"
    def __repr__(self):
        return f"Person(name={self.name!r})"

p = Person("Elvin"); print(p.greet()); print(p)
```

Explanation

Attributes belong to each instance. __repr__ returns an unambiguous representation for debugging. Encapsulation is by convention (single underscore).

Question

Add an age attribute and method is_adult (>=18).

Answer

```
class Person:
    def __init__(self, name, age):
        self.name, self.age = name, age
    def is_adult(self):
        return self.age >= 18
```

28 Inheritance and Dunder Methods

Overview

Reuse behavior with inheritance; override methods. Dunder methods customize built‑ins like len(), iteration, arithmetic, context managers.

Example Code

```
class Animal:
    def speak(self): return "..."
class Dog(Animal):
    def speak(self): return "Woof"

d = Dog(); print(d.speak())

class Box:
    def __init__(self): self.items=[]
    def __len__(self): return len(self.items)

b = Box(); b.items.extend([1,2,3]); print(len(b))
```

Explanation

Dog inherits speak and overrides it. __len__ allows len(b) to call b.__len__(). Implementing dunder methods integrates classes with Python idioms.

Question

Implement __iter__ for Box to iterate over items.

Answer

```
def __iter__(self):
    return iter(self.items)
```

29 Dataclasses and Typing

Overview

Dataclasses reduce boilerplate for simple data containers; typing adds static hints that improve readability and tooling.

Example Code

```
from dataclasses import dataclass
from typing import List

@dataclass
class User:
    name: str
    age: int
    tags: List[str]

u = User("Elvin", 23, ["python","ai"])
print(u)
```

Explanation

Dataclasses auto□generate __init__, __repr__, __eq__. Type hints are not enforced at runtime but help linters/IDEs. Use Optional[T] for nullable fields.

Question

Why prefer dataclass over a plain dict for records?

Answer

Dataclasses provide structure, validation points, equality, defaults, and IDE support, reducing bugs versus unstructured dicts.

30 Decorators

Overview

Higher‑order wrappers that extend behavior of functions without modifying their code. Common uses: logging, timing, caching, access control, retries.

Example Code

```
import time

def timer(fn):
    def wrapper(*a, **kw):
        t0 = time.perf_counter()
        try:
            return fn(*a, **kw)
        finally:
            dt = time.perf_counter() - t0
            print(f"{fn.__name__} took {dt:.4f}s")
    return wrapper

@timer
def work():
    sum(range(1_000_00))

work()
```

Explanation

The decorator returns a new function that calls the original plus extra logic. The @ syntax assigns work = timer(work). Use functools.wraps to preserve metadata.

Question

What does @timer print and why is finally used?

Answer

It prints the function duration after execution; finally guarantees printing even if the function raises or returns early.