Python for Cheminformatics & Bioinformatics
50 Quiz Questions with Answers and Rationales

AI-Driven Drug Development Training

February 2026

Contents

1 Variables & Data Types (Questions 1-5)

Question 1: Compound Data Storage

What is the output of the following code?

```
compound_name = "Aspirin"
mw = 180.16
pic50 = "8.28"
print(type(mw) == type(pic50))
```

Options:

True

False

Error

None

Answer: B) False

Rationale: `mw` is a float (`float`) while `pic50` is stored as a string (`str`). The `type()` function returns different classes, so the comparison returns `False`. In real QSAR data, always ensure numeric values are stored as numbers, not strings.

Question 2: Tuple Unpacking for Molecular Data

What will be the value of `logp`?

```
compound = ("Ibuprofen", 206.28, 3.97)
name, mw, logp = compound
print(logp)
```

Options:

A) "Ibuprofen"

B) 206.28

C) 3.97

D) Error

Answer: C) 3.97

Rationale: Tuple unpacking assigns each element to the corresponding variable in order. `name` gets "Ibuprofen", `mw` gets 206.28, and `logp` gets 3.97.

Question 3: Falsy Values in Drug Data

What is the output?

```
ic50 = None  # Missing measurement
activity = 0  # No inhibition
smiles = ""  # Empty SMILES
print(bool(ic50), bool(activity), bool(smiles))
```

Options:

A) True True True

B) False False False

C) None 0 ""

D) Error

Answer: B) False False False
Rationale: In Python, `None`, `0`, and empty strings are "falsy" values. When handling drug discovery data, these values often indicate missing or invalid measurements that need special handling.

Question 4: Type Conversion for IC50

What is the result of `int(5.8) + int(-2.3)`?
Options:

A) 3

B) 4

C) 3.5

D) Error

Answer: A) 3
Rationale: `int()` truncates towards zero (not floor!). `int(5.8) = 5` and `int(-2.3) = -2`. Therefore, `5 + (-2) = 3`. Be careful when converting IC50 or other bioactivity values.

Question 5: Multi-line SMILES/Sequences

Which statement correctly creates a multi-line DNA sequence string?
Options:

A) `seq = "ATGC\nGCTA"`

B)
```
seq = '''ATGC
GCTA'''
```

C) `seq = """ATGC\nGCTA"""`

D) All of the above

Answer: D) All of the above
Rationale: Multi-line strings can be created using escape character `\n`, triple single quotes, or triple double quotes. This is useful for storing FASTA sequences or long SMILES strings.

2 Operators for Molecular Calculations (Questions 6-10)

Question 6: Division Operators

What is the output?

```
total_atoms = 17
heavy_atoms = 5
print(total_atoms // heavy_atoms, total_atoms % heavy_atoms, total_atoms / heavy_atoms)
```

Options:

A) 3 2 3.4

B) 3.4 2 3

C) 3 2 3.0

D) 3.0 2.0 3.4

Answer: A) 3 2 3.4
Rationale: `//` is floor division (integer quotient), `%` is modulo (remainder), `/` is true division (float). When counting atoms or codons, floor division is often useful.

Question 7: Lipinski Rule Checking

What is the output?

```
mw = 450
logp = 4.5
passes_mw = mw <= 500
passes_logp = logp <= 5
print(passes_mw and passes_logp)
```

Options:

A) True

B) False

C) None

D) Error

Answer: A) True

Rationale: Both conditions evaluate to `True`: MW 450 $\leq$ 500 and LogP 4.5 $\leq$ 5. The `and` operator returns `True` only when both operands are `True`. This pattern is used for Lipinski Rule of Five checks.

Question 8: pIC50 Calculation

What is the value of `10 ** (9 - 8)`?

Options:

A) 1

B) 10

C) 100

D) 0.1

Answer: B) 10

Rationale: `10 ** (9 - 8) = 10 ** 1 = 10`. This formula converts pIC50 to IC50 in nM: IC50_nM = 10 ** (9 - pIC50). For pIC50=8, IC50=10 nM.

Question 9: GC Content Logic

What is the output for calculating GC content?

```
seq = "ATGCGC"
gc_count = seq.count("G") + seq.count("C")
gc_percent = gc_count / len(seq) * 100
print(gc_percent)
```

Options:

A) 66.67 (approximately)

B) 50.0

C) 33.33

D) Error

Answer: A) 66.67 (approximately)

Rationale: The sequence has 2 G's and 2 C's out of 6 nucleotides. GC% = (4/6) $\times$ 100 = 66.67%. GC content is important in bioinformatics for primer design and genome analysis.

Question 10: Identity vs Equality

What is the output?

```
smiles1 = "CCO"
smiles2 = "CCO"
list1 = [smiles1]
list2 = [smiles2]
print(smiles1 == smiles2, list1 == list2, list1 is list2)
```

Options:

A) True True True

B) True True False

C) True False False

D) False False False

Answer: B) True True False

Rationale: `==` compares values (both strings are "CCO", both lists contain "CCO"). `is` compares identity (memory location). The lists have equal content but are different objects.

---

3 Strings for Sequences & SMILES (Questions 11-15)

Question 11: DNA Sequence Slicing

What is the output?

```
dna = "ATGCGATCG"
print(dna[:3], dna[-3:], dna[::3])
```

Options:

A) ATG TCG AGC

B) ATG TCG AGAG

C) ATG GCG ATG

D) Error

Answer: A) ATG TCG AGC

Rationale: `dna[:3]` = first 3 chars = "ATG" (start codon!). `dna[-3:]` = last 3 chars = "TCG". `dna[::3]` = every 3rd char = "A", "C", "T" = "AGC". Slicing is essential for codon extraction.

Question 12: DNA Transcription

What is the output?

```
dna = "ATGC"
rna = dna.replace("T", "U")
print(rna)
```

Options:

A) AUGC

B) ATGC

C) UAGC

D) Error

Answer: A) AUGC

Rationale: DNA transcription replaces thymine (T) with uracil (U). The `replace()` method creates a new string with all T's replaced by U's.

Question 13: SMILES Ring Detection

What is the output?

```
smiles = "c1ccccc1"  # Benzene
has_ring = any(c.isdigit() for c in smiles)
print(has_ring)
```

Options:

A) True

B) False

C) 1

D) Error

Answer: A) True

Rationale: In SMILES notation, digits indicate ring closure points. Benzene's SMILES "c1ccccc1" contains "1" twice (ring opening and closing). The `any()` function returns `True` if any digit is found.

Question 14: F-strings for Compound Data

What is the output?

```
name = "Aspirin"
pic50 = 5.28
print(f"{name}: pIC50 = {pic50:.1f}")
```

Options:

A) Aspirin: pIC50 = 5.28

B) Aspirin: pIC50 = 5.3

C) Aspirin: pIC50 = 5

D) Error

Answer: B) Aspirin: pIC50 = 5.3

Rationale: The format specifier `:.1f` rounds to 1 decimal place. 5.28 rounds to 5.3. F-strings are ideal for formatting molecular property reports.

Question 15: String Immutability

What happens when you run this code?

```
seq = "ATGC"
seq[0] = "G"  # Try to create mutation
```

Options:

A) TypeError - strings are immutable

B) seq becomes "GTGC"

C) seq becomes "GATGC"

D) None

Answer: A) TypeError - strings are immutable

Rationale: Strings cannot be modified in place. To "mutate" a sequence, create a new string: `seq = "G" + seq[1:]`. This is important when simulating mutations in bioinformatics.

4 Lists for Compound Libraries (Questions 16-20)

Question 16: List References

What is the output?

```
library_a = ["CCO", "CC", "CCC"]
library_b = library_a  # Reference, not copy!
library_a.append("CCCC")
print(len(library_b))
```

Options:

A) 3

B) 4

C) Error

D) None

Answer: B) 4

Rationale: `library_b = library_a` creates a reference to the same list object. Modifying `library_a` affects `library_b`. Use `library_b = library_a.copy()` for independent copies.

Question 17: List extend() vs append()

What is the output?

```
compounds = ["Aspirin", "Ibuprofen"]
compounds.extend(["Caffeine", "Metformin"])
print(len(compounds))
```

Options:

A) 2

B) 3

C) 4

D) Error

Answer: C) 4

Rationale: `extend()` adds each element individually. Compare: `append()` would add the entire list as one element, giving length 3.

Question 18: List Comprehension for Filtering

What is the output?

```
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1]
actives = [p for p in pic50_values if p >= 6.0]
print(len(actives))
```

Options:

A) 2

B) 3

C) 4

D) 5

Answer: B) 3

Rationale: List comprehension filters pIC50 values ≥ 6.0: [6.8, 7.3, 8.1]. Only 3 compounds pass the activity threshold. This is a common pattern for filtering active compounds.

Question 19: Nested Lists for Descriptor Matrix

What is the output?

```
# [MW, LogP, HBD, HBA]
descriptors = [
    [180.16, 1.19, 1, 4],    # Aspirin
    [206.28, 3.97, 1, 2],    # Ibuprofen
    [194.19, -0.07, 0, 6]    # Caffeine
]
print(descriptors[1][1])
```

Options:

A) 180.16

B) 1.19

C) 3.97

D) 206.28

Answer: C) 3.97
Rationale: `descriptors[1]` accesses Ibuprofen's row, `[1]` gets LogP (index 1). Nested lists can represent descriptor matrices before converting to NumPy/Pandas.

Question 20: List Sorting

What is the output?

```
compounds = [("Aspirin", 5.2), ("Drug_X", 8.1), ("Ibuprofen", 6.8)]
compounds.sort(key=lambda x: x[1], reverse=True)
print(compounds[0][0])
```

Options:

A) Aspirin

B) Drug_X

C) Ibuprofen

D) 8.1

Answer: B) Drug_X
Rationale: Sorting by pIC50 (index 1) in descending order puts Drug_X (8.1) first. This pattern ranks compounds by potency.

5 Tuples & Sets for Molecular Data (Questions 21-25)

Question 21: Tuple Immutability

What happens?

```
compound = ("Aspirin", 180.16, 5.2)
compound[2] = 6.0  # Try to update pIC50
```

Options:

A) TypeError

B) compound becomes ("Aspirin", 180.16, 6.0)

C) None

D) SyntaxError

Answer: A) TypeError
Rationale: Tuples are immutable – they cannot be modified after creation. Use tuples for fixed compound records, lists for mutable collections.

Question 22: Extended Tuple Unpacking

What is the output?

```
data = (1, 2, 3, 4, 5, 6, 7)
first, *middle, last = data
print(len(middle))
```

Options:

A) 7

B) 5

C) 2

D) 1

Answer: B) 5
Rationale: `*middle` captures all elements between first (1) and last (7). middle = [2, 3, 4, 5, 6], length 5. Useful for parsing variable-length data records.

Question 23: Set Intersection for Common Compounds

What is the output?

```
library_a = {"CMP001", "CMP002", "CMP003"}
library_b = {"CMP002", "CMP003", "CMP004"}
common = library_a & library_b
print(common)
```

Options:

A) {“CMP001”, “CMP004”}

B) {“CMP001”, “CMP002”, “CMP003”, “CMP004”}

C) {“CMP002”, “CMP003”}

D) Error

Answer: C) {“CMP002”, “CMP003”}
Rationale: The `&` operator finds set intersection (common elements). This is useful for finding compounds shared between screening libraries.

Question 24: Set Uniqueness

What is the output?

```
scaffolds = {"benzene", "pyridine", "benzene", "furan", "pyridine"}
print(len(scaffolds))
```

Options:

A) 5

B) 3

C) 2

D) Error

Answer: B) 3
Rationale: Sets automatically remove duplicates. Only unique scaffolds remain: benzene, pyridine, furan. Use sets to count unique molecular scaffolds.

Question 25: Set Difference

What is the output?

```
all_compounds = {"A", "B", "C", "D"}
tested = {"B", "D"}
untested = all_compounds - tested
print(untested)
```

Options:

A) {"B", "D"}

B) {"A", "C"}

C) {"A", "B", "C", "D"}

D) Error

Answer: B) {"A", "C"}
Rationale: Set difference (`-`) returns elements in the first set not in the second. Useful for tracking which compounds still need testing.

6 Dictionaries for Compound Databases (Questions 26-30)

Question 26: Dictionary get() Method

What is the output?

```
compound = {"name": "Aspirin", "MW": 180.16}
logp = compound.get("LogP", "N/A")
print(logp)
```

Options:

A) None

B) N/A

C) Error (KeyError)

D) 0

Answer: B) N/A
Rationale: `get(key, default)` returns the default value if key doesn't exist. This avoids KeyError and is useful for handling missing molecular properties.

Question 27: Dictionary Iteration

What is printed?

```
props = {"MW": 180.16, "LogP": 1.19, "HBD": 1}
for item in props:
    print(item, end=" ")
```

Options:

A) 180.16 1.19 1

B) MW LogP HBD

C) ("MW", 180.16) ("LogP", 1.19) ("HBD", 1)

D) Error

Answer: B) MW LogP HBD

Rationale: Iterating over a dictionary iterates over keys only. Use `props.values()` for values or `props.items()` for key-value pairs.

Question 28: Nested Dictionary

What is the output?

```
compounds = {
    "Aspirin": {"MW": 180.16, "pIC50": 5.2},
    "Ibuprofen": {"MW": 206.28, "pIC50": 6.8}
}
print(compounds["Aspirin"]["pIC50"])
```

Options:

A) 180.16

B) 5.2

C) 6.8

D) Error

Answer: B) 5.2

Rationale: First access gets Aspirin's dict, second access gets its pIC50 value. Nested dicts are common for storing compound databases.

Question 29: Dict Comprehension

What is the output?

```
import math
ic50_nm = {"A": 10, "B": 100, "C": 1000}
pic50 = {k: 9 - math.log10(v) for k, v in ic50_nm.items()}
print(round(pic50["A"], 1))
```

Options:

A) 7.0

B) 8.0

C) 9.0

D) 10.0

Answer: B) 8.0

Rationale: pIC50 = 9 - log10(IC50_nM). For IC50=10 nM: pIC50 = 9 - log10(10) = 9 - 1 = 8.0. Dict comprehension efficiently converts all values.

Question 30: Codon Table Lookup

What is the output?

```
codon_table = {"AUG": "M", "UGG": "W", "UAA": "*"}
protein = codon_table.get("AUG", "X") + codon_table.get("UGG", "X")
print(protein)
```

Options:

A) MW

B) AUG UGG

C) XX

D) Error

Answer: A) MW
Rationale: AUG codes for Methionine (M), UGG codes for Tryptophan (W). The codon table dictionary enables RNA-to-protein translation.

7 Control Flow for Data Processing (Questions 31-35)

Question 31: Activity Classification

What is the output?

```
pic50 = 7.5
if pic50 >= 8:
    print("Highly Active")
elif pic50 >= 6:
    print("Active")
else:
    print("Inactive")
```

Options:

A) Highly Active

B) Active

C) Inactive

D) Error

Answer: B) Active
Rationale: pIC50 7.5 is < 8 (not highly active) but ≥ 6 (active). The first True condition determines the output.

Question 32: Loop with Range

What is the sum?

```
total = 0
for i in range(0, 10, 2):  # 0, 2, 4, 6, 8
    total += i
print(total)
```

Options:

A) 20

B) 25

C) 30

D) 45

Answer: A) 20
Rationale: `range(0, 10, 2)` generates 0, 2, 4, 6, 8. Sum = 0+2+4+6+8 = 20. Step parameter is useful for processing every nth item.

Question 33: Break Statement

What is printed?

```
pic50_values = [5.2, 5.8, 7.5, 6.2, 8.1]
for p in pic50_values:
    if p >= 7.0:
        print(f"Found potent: {p}")
        break
```

Options:

A) Found potent: 8.1

B) Found potent: 7.5

C) Found potent: 7.5
Found potent: 8.1

D) Nothing printed

Answer: B) Found potent: 7.5
Rationale: `break` exits the loop immediately after finding the first potent compound (pIC50 $\geq$ 7.0). Only 7.5 is printed, 8.1 is never reached.

Question 34: Continue Statement

What is printed?

```
for compound in ["valid", None, "active", "", "potent"]:
    if not compound:
        continue
    print(compound, end=" ")
```

Options:

A) valid None active potent

B) valid active potent

C) None

D) Error

Answer: B) valid active potent
Rationale: `continue` skips falsy values (None, “”). Only truthy strings are printed. This pattern filters invalid data entries.

Question 35: Loop with Else

What is printed?

```
pic50_values = [5.2, 5.8, 5.5]
for p in pic50_values:
    if p >= 6.0:
        print("Found active")
        break
else:
    print("No actives found")
```

Options:

A) No actives found

B) Found active

C) Nothing

D) Error

Answer: A) No actives found
Rationale: The `else` clause executes when the loop completes without hitting `break`. Since no pIC50 $\geq$ 6.0, the else block runs.

8 Functions for Molecular Calculations (Questions 36-40)

Question 36: Default Parameters

What is the output?

```
def classify_activity(pic50, threshold=6.0):
    return "Active" if pic50 >= threshold else "Inactive"

print(classify_activity(5.5))
```

Options:

A) Active

B) Inactive

C) Error

D) None

Answer: B) Inactive
Rationale: Default threshold is 6.0. pIC50 5.5 $<$ 6.0, so returns "Inactive". Default parameters make functions more flexible.

Question 37: Return Values

What is the output?

```
def validate_smiles(smiles):
    if not smiles:
        return  # implicit None
    print("Valid")

result = validate_smiles("")
print(result)
```

Options:

A) None

B) Valid
None

C) Error

D) Valid

Answer: A) None
Rationale: Empty string is falsy, so function returns early without printing. Functions without explicit return value return `None`.

Question 38: *args for Multiple Compounds

What is the output?

```
def average_pic50(*values):
    return sum(values) / len(values)

print(average_pic50(5.2, 6.8, 7.3))
```

Options:

A) 19.3

B) 6.43 (approximately)

C) Error

D) (5.2, 6.8, 7.3)

Answer: B) 6.43 (approximately)
Rationale: `*values` collects all arguments into a tuple. Average = (5.2 + 6.8 + 7.3) / 3 = 6.43. Use *args for variable-length inputs.

Question 39: **kwargs for Properties

What is the output?

```
def create_compound(**props):
    return props

compound = create_compound(name="Aspirin", MW=180.16)
print(type(compound))
```

Options:

A) `<class 'tuple'>`

B) `<class 'dict'>`

C) `<class 'list'>`

D) Error

Answer: B) `<class 'dict'>`
Rationale: `**kwargs` collects keyword arguments into a dictionary. This pattern creates flexible compound property containers.

Question 40: Lambda for Conversion

What is the output?

```
import math
ic50_to_pic50 = lambda ic50_nm: 9 - math.log10(ic50_nm)
print(ic50_to_pic50(100))
```

Options:

A) 7.0

B) 8.0

C) 9.0

D) 2.0

Answer: A) 7.0
Rationale: pIC50 = 9 - log10(100) = 9 - 2 = 7.0. Lambda functions are concise for simple conversions used in map/filter operations.

---

9 File & Error Handling (Questions 41-45)

Question 41: Reading FASTA Files

What is the correct way to read a file?

```
with open("sequence.fasta", "r") as f:
    content = f.read()
```

Options:

A) The file is automatically closed after the with block

B) You must call f.close() manually

C) The file stays open

D) Error

Answer: A) The file is automatically closed after the with block
Rationale: The `with` statement is a context manager that ensures proper file cleanup, even if an exception occurs.

Question 42: File Modes

What happens when you try to write to a file opened in read mode?

```
with open("compounds.csv", "r") as f:
    f.write("Aspirin,180.16")
```

Options:

A) Data is written successfully

B) io.UnsupportedOperation error

C) FileNotFoundError

D) Nothing happens

Answer: B) io.UnsupportedOperation error
Rationale: Mode "r" is read-only. Use "w" for writing or "a" for appending. Always check file modes when handling compound data files.

Question 43: Try/Except for Invalid SMILES

What is printed?

```
def parse_smiles(smiles):
    if not smiles:
        raise ValueError("Empty SMILES")
    return smiles

try:
    result = parse_smiles("")
except ValueError:
    print("Error")
print("Done")
```

Options:

A) Error

B) Done

C) Error
Done

D) Nothing

Answer: C) Error
Done

Rationale: ValueError is caught, "Error" is printed, then execution continues to "Done". Exception handling allows graceful recovery from invalid data.

Question 44: Finally Block

What is printed?

```
def process_compound():
    try:
        return "Processed"
    finally:
        print("Cleanup")

result = process_compound()
```

Options:

A) Processed

B) Cleanup

C) Both (Cleanup first, then returns "Processed")

D) Error

Answer: C) Both (Cleanup first, then returns "Processed")

Rationale: `finally` always executes, even after a return statement. This ensures cleanup (closing files, releasing resources) happens reliably.

Question 45: Specific Exception Handling

What is printed?

```
try:
    ic50 = float("invalid")
except ValueError:
    print("Invalid IC50")
except TypeError:
    print("Wrong type")
```

Options:

A) Invalid IC50

B) Wrong type

C) Both

D) Error (uncaught)

Answer: A) Invalid IC50
Rationale: Converting "invalid" to float raises ValueError, not TypeError. The first matching except block handles it.

---

10 Advanced Python for Data Analysis (Questions 46-50)

Question 46: Generator Expression

What is the difference?

```
list_comp = [x**2 for x in range(1000000)]
gen_exp = (x**2 for x in range(1000000))
```

Options:

A) Both use the same memory

B) Generator uses less memory (lazy evaluation)

C) List uses less memory

D) They produce different values

Answer: B) Generator uses less memory (lazy evaluation)
Rationale: Generators compute values on-demand, not all at once. For large compound libraries, generators prevent memory issues.

Question 47: Map Function

What is the output?

```
import math
ic50_values = [10, 100, 1000]
pic50_values = list(map(lambda x: 9 - math.log10(x), ic50_values))
print(pic50_values)
```

Options:

A) `[8.0, 7.0, 6.0]`

B) `[1.0, 2.0, 3.0]`

C) `[10, 100, 1000]`

D) Error

Answer: A) `[8.0, 7.0, 6.0]`
Rationale: map() applies the IC50-to-pIC50 conversion to each value: 9-log10(10)=8, 9-log10(100)=7, 9-log10(1000)=6.

Question 48: Filter Function

What is the output?

```
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1]
potent = list(filter(lambda p: p >= 7.0, pic50_values))
print(potent)
```

Options:

A) `[5.2, 6.8, 4.9]`

B) `[6.8, 7.3, 8.1]`

C) `[7.3, 8.1]`

D) `[8.1]`

Answer: C) `[7.3, 8.1]`
Rationale: filter() keeps elements where the lambda returns True (pIC50 $\geq$ 7.0). Only 7.3 and 8.1 pass the potency threshold.

Question 49: Zip Function

What is the output?

```
names = ["Aspirin", "Ibuprofen", "Caffeine"]
pic50s = [5.2, 6.8, 4.8]
compounds = list(zip(names, pic50s))
print(compounds[0])
```

Options:

A) Aspirin

B) 5.2

C) `("Aspirin", 5.2)`

D) `["Aspirin", 5.2]`

Answer: C) `("Aspirin", 5.2)`
Rationale: zip() pairs elements from multiple iterables into tuples. First element is ("Aspirin", 5.2). Useful for combining parallel data arrays.

Question 50: Enumerate Function

What is the output?

```
smiles_list = ["CCO", "CC(=O)O", "c1ccccc1"]
for idx, smiles in enumerate(smiles_list, start=1):
    print(f"{idx}: {smiles}")
    break  # Only print first
```

Options:

A) 0: CCO

B) 1: CCO

C) CCO: 1

D) Error

Answer: B) 1: CCO
Rationale: enumerate() with start=1 begins counting from 1 (useful for 1-indexed output). Provides both index and value during iteration.

Summary: Topics Covered

Section	Questions	Key Concepts
Variables & Types	1-5	Data types, type conversion, falsy values
Operators	6-10	Arithmetic, comparison, logical operators
Strings	11-15	Slicing, methods, immutability, f-strings
Lists	16-20	Mutability, comprehensions, sorting
Tuples & Sets	21-25	Immutability, set operations
Dictionaries	26-30	Key-value pairs, nested dicts, comprehensions
Control Flow	31-35	if/elif/else, loops, break/continue
Functions	36-40	Parameters, *args, **kwargs, lambda
File/Error Handling	41-45	with statement, try/except/finally
Advanced Topics	46-50	Generators, map, filter, zip, enumerate

All examples use cheminformatics and bioinformatics contexts:

- SMILES strings and molecular properties
- IC50/pIC50 conversions
- DNA/RNA sequence manipulation
- Lipinski Rule of Five
- Compound library management
- FASTA file handling