Python for Cheminformatics & Bioinformatics
Lab Solutions

AI-Driven Drug Development Training

February 2026

Contents

1 Lab 1: Variables & Data Types

1.1 Exercise 1.1 – Compound Data Storage

```
# Store drug compound data
name = "Ibuprofen"
smiles = "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O"
molecular_weight = 206.28
pic50 = 6.1
is_active = pic50 >= 6.0  # True

# Print using f-strings
print(f"Compound: {name}")
print(f"SMILES: {smiles}")
print(f"Molecular Weight: {molecular_weight} Da")
print(f"pIC50: {pic50}")
print(f"Is Active: {is_active}")

# Formatted output
print(f"\n{name}: MW={molecular_weight:.2f}, pIC50={pic50:.2f}, Active={is_active}
    ")
```

1.2 Exercise 1.2 – DNA Sequence

```
# Store DNA sequence
dna_sequence = "ATGCGATCGATCGATCGATCG"

# Calculate properties
seq_length = len(dna_sequence)
adenine_count = dna_sequence.count("A")
thymine_count = dna_sequence.count("T")

print(f"Sequence: {dna_sequence}")
print(f"Length: {seq_length} bp")
print(f"Adenines (A): {adenine_count}")
print(f"Thymines (T): {thymine_count}")
```

1.3 Exercise 1.3 – Type Conversion

```
import math

# IC50 as string (from file/input)
ic50_str = "5.2"

# Convert to float
ic50_nm = float(ic50_str)

# Calculate pIC50
ic50_M = ic50_nm * 1e-9  # Convert nM to M
pic50 = -math.log10(ic50_M)

# Store as both float and formatted string
pic50_float = pic50
pic50_formatted = f"{pic50:.2f}"
```

```
print(f"IC50: {ic50_nm} nM")
print(f"pIC50 (float): {pic50_float}")
print(f"pIC50 (string): {pic50_formatted}")
```

2 Lab 2: Operators

2.1 Exercise 2.1 – IC50 Conversion

```
import math

# IC50 values in nanomolar
ic50_values = [10.0, 100.0, 1000.0]

# Convert each to pIC50
# Formula: pIC50 = -log10(IC50 * 10^-9)
print("IC50 (nM) -> pIC50 Conversion:")
print("-" * 30)

for ic50_nm in ic50_values:
    ic50_M = ic50_nm * 1e-9
    pic50 = -math.log10(ic50_M)
    print(f"IC50: {ic50_nm:>8.1f} nM -> pIC50: {pic50:.2f}")

# Alternative: pIC50 = 9 - log10(IC50_nM)
print("\nUsing simplified formula:")
for ic50_nm in ic50_values:
    pic50 = 9 - math.log10(ic50_nm)
    print(f"IC50: {ic50_nm:>8.1f} nM -> pIC50: {pic50:.2f}")
```

2.2 Exercise 2.2 – Lipinski Check

```
# Compound properties
MW = 450
LogP = 3.5
HBD = 2
HBA = 8

# Lipinski Rule of Five
rule_mw = MW <= 500
rule_logp = LogP <= 5
rule_hbd = HBD <= 5
rule_hba = HBA <= 10

# Combined check using AND
passes_lipinski = rule_mw and rule_logp and rule_hbd and rule_hba

print("Lipinski Rule of Five Check:")
print(f"  MW <= 500: {MW} -> {rule_mw}")
print(f"  LogP <= 5: {LogP} -> {rule_logp}")
print(f"  HBD <= 5: {HBD} -> {rule_hbd}")
print(f"  HBA <= 10: {HBA} -> {rule_hba}")
print(f"\nPasses Lipinski: {passes_lipinski}")
```

2.3 Exercise 2.3 – GC Content (Rosalind)

```
# DNA sequence
sequence = "AGCTATAG"

# Count G and C
g_count = sequence.count("G")
c_count = sequence.count("C")
total = len(sequence)

# Calculate GC percentage
gc_content = (g_count + c_count) / total * 100

print(f"Sequence: {sequence}")
print(f"G count: {g_count}")
print(f"C count: {c_count}")
print(f"Total length: {total}")
print(f"GC Content: {gc_content:.1f}%")
```

2.4 Exercise 2.4 – Activity Classification

```
# pIC50 value
pic50 = 7.2

# Classification using comparison operators
is_highly_potent = pic50 >= 8
is_potent = pic50 >= 7
is_moderate = pic50 >= 6
is_weak = pic50 < 6

# Determine classification
if is_highly_potent:
    classification = "Highly potent"
elif is_potent:
    classification = "Potent"
elif is_moderate:
    classification = "Moderate"
else:
    classification = "Weak"

print(f"pIC50: {pic50}")
print(f"Classification: {classification}")
```

3 Lab 3: Strings

3.1 Exercise 3.1 – DNA Transcription

```
# DNA sequence
dna = "ATGCGATCGATCG"

# Transcribe to RNA (replace T with U)
rna = dna.replace("T", "U")

print(f"DNA: {dna}")
print(f"RNA: {rna}")
```

3.2 Exercise 3.2 – Reverse Complement (Rosalind REVC)

```
# DNA sequence
dna = "AAAACCCGGT"

# Create complement mapping
complement = {"A": "T", "T": "A", "G": "C", "C": "G"}

# Method 1: Using loop
comp_seq = ""
for nucleotide in dna:
    comp_seq += complement[nucleotide]

# Reverse the complement
reverse_comp = comp_seq[::-1]

print(f"Original:    {dna}")
print(f"Complement:  {comp_seq}")
print(f"Reverse Comp: {reverse_comp}")

# Method 2: Using translate (more efficient)
trans_table = str.maketrans("ATGC", "TACG")
reverse_comp_v2 = dna.translate(trans_table)[::-1]
print(f"Reverse Comp (v2): {reverse_comp_v2}")
```

3.3 Exercise 3.3 – SMILES Analysis

```
# Aspirin SMILES
smiles = "CC(=O)OC1=CC=CC=C1C(=O)O"

# Check for ring (digits indicate ring closure)
has_ring = any(char.isdigit() for char in smiles)

# Count carbons (uppercase C only, not lowercase aromatic c)
carbon_count = smiles.count("C")

# Count oxygens
oxygen_count = smiles.count("O")

# Check aromaticity (lowercase letters = aromatic)
is_aromatic = any(char.islower() for char in smiles)

print(f"SMILES: {smiles}")
print(f"Contains ring: {has_ring}")
print(f"Carbon count: {carbon_count}")
print(f"Oxygen count: {oxygen_count}")
print(f"Is aromatic: {is_aromatic}")
```

3.4 Exercise 3.4 – Nucleotide Count (Rosalind DNA)

```
# DNA sequence from Rosalind
sequence = "AGCTTTTCATTCTGACTGCAACGGGCAATA"

# Count each nucleotide
a_count = sequence.count("A")
c_count = sequence.count("C")
```

```
g_count = sequence.count("G")
t_count = sequence.count("T")

# Print in Rosalind format (A C G T separated by spaces)
print(f"Sequence: {sequence}")
print(f"A:{a_count} T:{t_count} G:{g_count} C:{c_count}")

# Rosalind output format
print(f"\nRosalind format: {a_count} {c_count} {g_count} {t_count}")
```

4 Lab 4: Conditionals

4.1 Exercise 4.1 – Drug-Likeness Checker

```
# Compound properties
mw = 520
logp = 4.2
hbd = 3
hba = 8

# Count violations
violations = 0
if mw > 500:
    violations += 1
    print(f"Violation: MW ({mw}) > 500")
if logp > 5:
    violations += 1
    print(f"Violation: LogP ({logp}) > 5")
if hbd > 5:
    violations += 1
    print(f"Violation: HBD ({hbd}) > 5")
if hba > 10:
    violations += 1
    print(f"Violation: HBA ({hba}) > 10")

# Determine drug-likeness
if violations <= 1:
    print(f"\nResult: Drug-like ({violations} violation(s))")
else:
    print(f"\nResult: Non-drug-like ({violations} violations)")
```

4.2 Exercise 4.2 – Codon Identifier

```
# Test codon
codon = "ATG"

# Using if/elif/else
if codon == "ATG":
    codon_type = "Start codon (Methionine)"
elif codon in ["TAA", "TAG", "TGA"]:
    codon_type = "Stop codon"
else:
    codon_type = "Coding codon"

print(f"Codon: {codon}")
```

```
print(f"Type: {codon_type}")

# Using match-case (Python 3.10+)
print("\nUsing match-case:")
match codon:
    case "ATG":
        print("Start codon (Methionine)")
    case "TAA" | "TAG" | "TGA":
        print("Stop codon")
    case _:
        print("Coding codon")
```

4.3 Exercise 4.3 – Activity Classifier

```
# Test values
test_values = [8.5, 7.2, 6.5, 5.3, 4.1]

def classify_activity(pic50):
    """Classify compound based on pIC50."""
    if pic50 >= 8:
        return "Highly Active"
    elif pic50 >= 7:
        return "Active"
    elif pic50 >= 6:
        return "Moderately Active"
    elif pic50 >= 5:
        return "Weakly Active"
    else:
        return "Inactive"

# Test all values
print("Activity Classification:")
print("-" * 35)
for pic50 in test_values:
    classification = classify_activity(pic50)
    print(f"pIC50 {pic50}: {classification}")
```

5 Lab 5: Loops

5.1 Exercise 5.1 – Batch IC50 Conversion

```
import math

# IC50 values in nanomolar
ic50_list = [1.0, 10.0, 100.0, 1000.0, 10000.0]

print("Batch IC50 to pIC50 Conversion:")
print("-" * 40)

for ic50_nm in ic50_list:
    pic50 = 9 - math.log10(ic50_nm)
    print(f"IC50: {ic50_nm:>8.1f} nM -> pIC50: {pic50:.2f}")
```

5.2 Exercise 5.2 – Nucleotide Counter (Rosalind DNA)

```
# DNA sequence
seq = "AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGT"

# Method 1: Using dictionary and loop
counts = {"A": 0, "C": 0, "G": 0, "T": 0}

for nucleotide in seq:
    if nucleotide in counts:
        counts[nucleotide] += 1

print(f"Sequence: {seq[:30]}...")
print(f"Counts: {counts}")
print(f"Rosalind format: {counts['A']} {counts['C']} {counts['G']} {counts['T']}")
```

5.3 Exercise 5.3 – Filter Active Compounds

```
# pIC50 values
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1, 5.9, 6.2]

print("Filtering Active Compounds (pIC50 >= 6):")
print("-" * 40)

for i, pic50 in enumerate(pic50_values):
    if pic50 < 6:
        continue  # Skip inactive
    print(f"Compound {i+1}: pIC50 = {pic50} (Active)")
```

5.4 Exercise 5.4 – Find First Potent Compound

```
# pIC50 values
pic50_values = [5.2, 5.8, 6.1, 7.5, 8.2, 6.8]

print("Finding first highly potent compound (pIC50 >= 7.5):")
print("-" * 50)

index = 0
while index < len(pic50_values):
    pic50 = pic50_values[index]
    print(f"Checking index {index}: pIC50 = {pic50}")

    if pic50 >= 7.5:
        print(f"\nFound! Index {index}, pIC50 = {pic50}")
        break

    index += 1
else:
    print("No highly potent compound found.")
```

5.5 Exercise 5.5 – Read Until Stop Codon (While Loop)

```
# Codons list (TAA is a stop codon)
codons = ["ATG", "CGA", "TCG", "GGC", "TAA", "AAA"]
```

```python
stop_codons = ["TAA", "TAG", "TGA"]

print("Reading codons until stop codon:")
print("-" * 50)

sequence = ""
idx = 0
while idx < len(codons) and codons[idx] not in stop_codons:
    print(f"Codon {idx}: {codons[idx]} -> Added to sequence")
    sequence += codons[idx]
    idx += 1

if idx < len(codons):
    print(f"\nStop codon {codons[idx]} found at index {idx}")

print(f"\nFinal sequence: {sequence}")
print(f"Sequence length: {len(sequence)} nucleotides")
# Output: ATGCGATCGGGC (12 nucleotides)
```

5.6 Exercise 5.6 – Compound Screening (While Loop)

```python
# pIC50 values from screening
pic50_values = [4.5, 5.2, 6.8, 5.1, 7.3, 4.9, 8.1, 5.9]
target_actives = 3
threshold = 6.0

print(f"Screening until {target_actives} active compounds found (pIC50 >= {
    threshold}):")
print("-" * 60)

active_count = 0
screened = 0
actives_found = []

while screened < len(pic50_values) and active_count < target_actives:
    pic50 = pic50_values[screened]
    status = "Active" if pic50 >= threshold else "Inactive"
    print(f"Compound {screened + 1}: pIC50 = {pic50} -> {status}")

    if pic50 >= threshold:
        active_count += 1
        actives_found.append((screened + 1, pic50))

    screened += 1

print(f"\nScreening complete!")
print(f"Total compounds screened: {screened}")
print(f"Active compounds found: {active_count}")
print(f"Active compound details: {actives_found}")
# Output: Screened 7 compounds to find 3 actives
```

6 Lab 6: Functions

6.1 Exercise 6.1 – pIC50 Converter Function

```python
import math

def ic50_to_pic50(ic50_nm):
    """
    Convert IC50 from nanomolar to pIC50.

    Args:
        ic50_nm: IC50 value in nanomolar

    Returns:
        pIC50 value (float)
    """
    return 9 - math.log10(ic50_nm)

# Test with various values
test_values = [10, 100, 1000]

print("IC50 to pIC50 Conversion:")
for ic50 in test_values:
    pic50 = ic50_to_pic50(ic50)
    print(f"IC50 {ic50} nM -> pIC50 {pic50:.2f}")
```

6.2 Exercise 6.2 – GC Content Function

```python
def gc_content(sequence):
    """
    Calculate GC content percentage of a DNA sequence.

    Args:
        sequence: DNA sequence string (uppercase)

    Returns:
        GC percentage as float
    """
    sequence = sequence.upper()
    g_count = sequence.count("G")
    c_count = sequence.count("C")
    total = len(sequence)

    if total == 0:
        return 0.0

    return (g_count + c_count) / total * 100

# Test sequences
test_sequences = ["AGCTATAG", "GCGCGCGC", "ATATAT"]

print("GC Content Calculations:")
for seq in test_sequences:
    gc = gc_content(seq)
    print(f"{seq}: {gc:.1f}%")
```

6.3 Exercise 6.3 – Lipinski Calculator

```python
def check_lipinski(mw, logp, hbd, hba):
```

```
    """
    Check Lipinski Rule of Five.

    Args:
        mw: Molecular weight
        logp: LogP value
        hbd: H-bond donors
        hba: H-bond acceptors

    Returns:
        Tuple of (passes: bool, violations: int)
    """
    violations = 0

    if mw > 500:
        violations += 1
    if logp > 5:
        violations += 1
    if hbd > 5:
        violations += 1
    if hba > 10:
        violations += 1

    passes = violations <= 1
    return passes, violations

# Test compounds
compounds = [
    ("Aspirin", 180.16, 1.19, 1, 4),
    ("Drug_A", 520, 4.5, 2, 8),
    ("Drug_B", 450, 6.2, 7, 12)
]

print("Lipinski Rule of Five Checker:")
print("-" * 50)
for name, mw, logp, hbd, hba in compounds:
    passes, violations = check_lipinski(mw, logp, hbd, hba)
    status = "PASS" if passes else "FAIL"
    print(f"{name}: {status} ({violations} violations)")
```

6.4 Exercise 6.4 – Reverse Complement Function

```
def reverse_complement(dna):
    """
    Generate reverse complement of a DNA sequence.

    Args:
        dna: DNA sequence string

    Returns:
        Reverse complement string
    """
    complement_map = {"A": "T", "T": "A", "G": "C", "C": "G"}

    # Generate complement
    complement = ""
    for nucleotide in dna.upper():
```

```
        complement += complement_map.get(nucleotide, nucleotide)

    # Reverse
    return complement[::-1]

# Test with Rosalind REVC sample
test_dna = "AAAACCCGGT"
result = reverse_complement(test_dna)

print(f"Input:  {test_dna}")
print(f"Output: {result}")
print(f"Expected: ACCGGGTTTT")
print(f"Correct: {result == 'ACCGGGTTTT'}")
```

7 Lab 6B: Error Handling

7.1 Exercise 6B.1 – Safe IC50 Conversion

```
import math

def safe_ic50_to_pic50(ic50_nm):
    """
    Safely convert IC50 to pIC50 with error handling.

    Args:
        ic50_nm: IC50 value in nanomolar

    Returns:
        pIC50 value or None on error
    """
    try:
        # Check for non-numeric input
        ic50_nm = float(ic50_nm)

        # Check for invalid values
        if ic50_nm <= 0:
            raise ValueError(f"IC50 must be positive, got {ic50_nm}")

        return 9 - math.log10(ic50_nm)

    except TypeError:
        print("Error: Input must be a number")
        return None
    except ValueError as e:
        print(f"Error: {e}")
        return None

# Test cases
print("Testing safe_ic50_to_pic50:")
print("-" * 40)
test_cases = [10, -5, 0, "invalid", 100]

for test in test_cases:
    result = safe_ic50_to_pic50(test)
    print(f"Input: {test!r:>10} -> pIC50: {result}")
```

7.2 Exercise 6B.2 – SMILES Validator

```python
from rdkit import Chem

def validate_smiles(smiles):
    """
    Validate a SMILES string using RDKit.

    Args:
        smiles: SMILES string

    Returns:
        Tuple of (is_valid: bool, message: str)
    """
    try:
        if not smiles or not isinstance(smiles, str):
            raise ValueError("SMILES must be a non-empty string")

        mol = Chem.MolFromSmiles(smiles)

        if mol is None:
            raise ValueError(f"Invalid SMILES: {smiles}")

        return True, "Valid SMILES"

    except ValueError as e:
        return False, str(e)
    except Exception as e:
        return False, f"Unexpected error: {e}"

# Test SMILES
test_smiles = [
    "CCO",                          # Ethanol - valid
    "CC(=O)OC1=CC=CC=C1C(=O)O",     # Aspirin - valid
    "invalid_smiles",               # Invalid
    "C(C)(C)(C)(C)C",               # Invalid valence
    ""                              # Empty
]

print("SMILES Validation:")
print("-" * 50)
for smiles in test_smiles:
    is_valid, message = validate_smiles(smiles)
    status = "VALID" if is_valid else "INVALID"
    print(f"{smiles[:30]:30} -> {status}")
```

8 Lab 7: Lists

8.1 Exercise 7.1 – Compound Library

```python
# Create SMILES list for common drugs
smiles_library = [
    "CC(=O)OC1=CC=CC=C1C(=O)O",   # Aspirin
    "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O",   # Ibuprofen
    "CN1C=NC2=C1C(=O)N(C(=O)N2C)C",   # Caffeine
    "CC(=O)NC1=CC=C(C=C1)O",   # Acetaminophen
    "C1=CC=C(C=C1)CC(C(=O)O)N"   # Phenylalanine
```

```
]

print(f"Initial library: {len(smiles_library)} compounds")

# Add a new compound
smiles_library.append("CCO")  # Ethanol
print(f"After append: {len(smiles_library)} compounds")

# Remove a compound by value
smiles_library.remove("CCO")
print(f"After remove: {len(smiles_library)} compounds")

# Insert at specific position
smiles_library.insert(1, "C1CCCCC1")  # Cyclohexane
print(f"After insert at index 1: {len(smiles_library)} compounds")

# Print first and last
print(f"\nFirst compound: {smiles_library[0]}")
print(f"Last compound: {smiles_library[-1]}")
```

8.2 Exercise 7.2 – pIC50 Statistics

```
# pIC50 values
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1, 5.9, 6.2, 7.8]

# Calculate statistics
min_val = min(pic50_values)
max_val = max(pic50_values)
sorted_vals = sorted(pic50_values)

# Count actives (pIC50 >= 6)
active_count = sum(1 for p in pic50_values if p >= 6)

print(f"pIC50 values: {pic50_values}")
print(f"Min: {min_val}")
print(f"Max: {max_val}")
print(f"Sorted: {sorted_vals}")
print(f"Active compounds (pIC50 >= 6): {active_count}")
```

9 Lab 7B: Tuples & Sets

9.1 Exercise 7B.1 – Compound Records

```
# Create tuples for compounds
aspirin = ("Aspirin", "CC(=O)OC1=CC=CC=C1C(=O)O", 5.2)
ibuprofen = ("Ibuprofen", "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O", 6.1)
caffeine = ("Caffeine", "CN1C=NC2=C1C(=O)N(C(=O)N2C)C", 4.8)

compounds = [aspirin, ibuprofen, caffeine]

# Unpack and print
print("Compound Records:")
print("-" * 60)
for compound in compounds:
    name, smiles, pic50 = compound  # Tuple unpacking
```

```
    print(f"Name: {name}")
    print(f"SMILES: {smiles}")
    print(f"pIC50: {pic50}")
    print()

# Try to modify (will fail)
# aspirin[2] = 6.0  # TypeError: 'tuple' object does not support item assignment
print("Note: Tuples are immutable - cannot modify values!")
```

9.2 Exercise 7B.2 – Library Comparison

```
# Two compound libraries
library_A = {"CMP001", "CMP002", "CMP003", "CMP004"}
library_B = {"CMP003", "CMP004", "CMP005", "CMP006"}

print(f"Library A: {library_A}")
print(f"Library B: {library_B}")
print()

# Set operations
all_compounds = library_A | library_B  # Union
common = library_A & library_B  # Intersection
only_in_A = library_A - library_B  # Difference
only_in_B = library_B - library_A  # Difference

print(f"All unique compounds (union): {all_compounds}")
print(f"Common compounds (intersection): {common}")
print(f"Only in Library A: {only_in_A}")
print(f"Only in Library B: {only_in_B}")
```

10 Lab 8: List Comprehensions

10.1 Exercise 8.1 – Filter Active Compounds

```
# pIC50 values
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1, 5.9]

# Filter using list comprehension (pIC50 >= 6.0)
active_compounds = [p for p in pic50_values if p >= 6.0]

print(f"All pIC50 values: {pic50_values}")
print(f"Active (pIC50 >= 6.0): {active_compounds}")
```

10.2 Exercise 8.2 – Batch Conversion

```
import math

# IC50 values in nanomolar
ic50_list = [10, 100, 1000]

# Method a) List comprehension
pic50_comp = [9 - math.log10(ic50) for ic50 in ic50_list]
print(f"Using list comprehension: {pic50_comp}")
```

```
# Method b) map() with lambda
pic50_map = list(map(lambda ic50: 9 - math.log10(ic50), ic50_list))
print(f"Using map() with lambda: {pic50_map}")
```

10.3 Exercise 8.3 – Conditional Comprehension

```
# pIC50 values
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1]

# Create tuples with activity classification
classified = [
    (p, "Active") if p >= 6 else (p, "Inactive")
    for p in pic50_values
]

print("Activity Classification:")
for pic50, status in classified:
    print(f"  pIC50 {pic50}: {status}")
```

11 Lab 9: Dictionaries

11.1 Exercise 9.1 – Compound Database

```
# Compound database with nested properties
compound_db = {
    "Aspirin": {
        "SMILES": "CC(=O)OC1=CC=CC=C1C(=O)O",
        "MW": 180.16,
        "pIC50": 5.2,
        "is_active": False
    },
    "Ibuprofen": {
        "SMILES": "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O",
        "MW": 206.28,
        "pIC50": 6.1,
        "is_active": True
    },
    "Caffeine": {
        "SMILES": "CN1C=NC2=C1C(=O)N(C(=O)N2C)C",
        "MW": 194.19,
        "pIC50": 4.8,
        "is_active": False
    }
}

# Print database
for name, props in compound_db.items():
    print(f"{name}: pIC50={props['pIC50']}, Active={props['is_active']}")
```

11.2 Exercise 9.2 – Codon Table (Rosalind)

```
# Partial codon table
codon_table = {
    "ATG": "M",  # Start/Methionine
```

```
    "TGG": "W",  # Tryptophan
    "TAA": "Stop",
    "TAG": "Stop",
    "TGA": "Stop",
    "TTT": "F", "TTC": "F",  # Phenylalanine
    "GCT": "A", "GCC": "A", "GCA": "A", "GCG": "A"  # Alanine
}

# Translate short sequence
dna_sequence = "ATGGCTTGA"

# Split into codons and translate
protein = ""
for i in range(0, len(dna_sequence), 3):
    codon = dna_sequence[i:i+3]
    amino_acid = codon_table.get(codon, "?")
    if amino_acid == "Stop":
        break
    protein += amino_acid

print(f"DNA: {dna_sequence}")
print(f"Protein: {protein}")
```

11.3 Exercise 9.3 – Dict Comprehension

```
# Filter to only active compounds using dict comprehension
active_compounds = {
    name: props
    for name, props in compound_db.items()
    if props["is_active"]
}

print("Active Compounds Only:")
for name, props in active_compounds.items():
    print(f"  {name}: pIC50={props['pIC50']}")
```

12 Lab 10: File Handling

12.1 Exercise 10.1 – Write Compound CSV

```
# Compound data
compounds = [
    ("Aspirin", "CC(=O)OC1=CC=CC=C1C(=O)O", 5.2),
    ("Ibuprofen", "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O", 6.1),
    ("Caffeine", "CN1C=NC2=C1C(=O)N(C(=O)N2C)C", 4.8),
    ("Acetaminophen", "CC(=O)NC1=CC=C(C=C1)O", 5.5),
    ("Naproxen", "COC1=CC2=CC(C(C)C(O)=O)=CC=C2C=C1", 6.8)
]

# Write to CSV
with open("compounds.csv", "w") as f:
    f.write("name,smiles,pIC50\n")  # Header
    for name, smiles, pic50 in compounds:
        f.write(f"{name},{smiles},{pic50}\n")
```

```
print("Wrote compounds.csv successfully!")
```

12.2 Exercise 10.2 – Read FASTA

```
def parse_fasta(filename):
    """
    Parse a FASTA file and return dict of {header: sequence}.
    """
    sequences = {}
    current_header = None
    current_seq = ""

    with open(filename, "r") as f:
        for line in f:
            line = line.strip()
            if line.startswith(">"):
                # Save previous sequence
                if current_header:
                    sequences[current_header] = current_seq
                # Start new sequence
                current_header = line[1:]  # Remove '>'
                current_seq = ""
            else:
                current_seq += line

        # Save last sequence
        if current_header:
            sequences[current_header] = current_seq

    return sequences

# Example usage (assuming file exists)
# sequences = parse_fasta("sequences.fasta")
# for header, seq in sequences.items():
#     print(f">{header}: {len(seq)} bp")
```

12.3 Exercise 10.3 – Filter and Export

```
# Read compound CSV and filter actives
actives = []

with open("compounds.csv", "r") as f:
    header = f.readline()  # Skip header
    for line in f:
        parts = line.strip().split(",")
        name, smiles, pic50 = parts[0], parts[1], float(parts[2])
        if pic50 >= 6.0:
            actives.append((name, smiles, pic50))

# Write filtered results
with open("actives.csv", "w") as f:
    f.write("name,smiles,pIC50\n")
    for name, smiles, pic50 in actives:
        f.write(f"{name},{smiles},{pic50}\n")

print(f"Exported {len(actives)} active compounds to actives.csv")
```

13 Lab 11: NumPy

13.1 Exercise 11.1 – Descriptor Matrix

```
import numpy as np

# Molecular descriptors: [MW, LogP, HBD, HBA]
descriptors = np.array([
    [180.16, 1.19, 1, 4],    # Aspirin
    [206.28, 3.97, 1, 2],    # Ibuprofen
    [194.19, -0.07, 0, 6],   # Caffeine
    [151.16, 0.46, 2, 3],    # Acetaminophen
    [230.26, 3.18, 1, 3]     # Naproxen
])

# Calculate mean and std per descriptor (column)
means = descriptors.mean(axis=0)
stds = descriptors.std(axis=0)

print("Descriptor Statistics:")
print(f"{'Descriptor':<10} {'Mean':>10} {'Std':>10}")
print("-" * 32)
labels = ["MW", "LogP", "HBD", "HBA"]
for i, label in enumerate(labels):
    print(f"{label:<10} {means[i]:>10.2f} {stds[i]:>10.2f}")
```

13.2 Exercise 11.2 – Normalization

```
import numpy as np

# Normalize to 0-1 range using vectorized operations
min_vals = descriptors.min(axis=0)
max_vals = descriptors.max(axis=0)

normalized = (descriptors - min_vals) / (max_vals - min_vals)

print("Normalized Descriptors (0-1 range):")
print(normalized.round(3))
```

13.3 Exercise 11.3 – Boolean Filtering

```
import numpy as np

# Filter compounds where MW < 500 AND LogP < 5
mw_col = 0
logp_col = 1

mask = (descriptors[:, mw_col] < 500) & (descriptors[:, logp_col] < 5)

filtered = descriptors[mask]

print(f"Compounds passing filter (MW<500 AND LogP<5): {mask.sum()}")
print("Filtered descriptors:")
print(filtered)
```

14 Lab 11B: Pandas

14.1 Exercise 11B.1 – Create Compound DataFrame

```
import pandas as pd

# Create DataFrame
df = pd.DataFrame({
    'Name': ['Aspirin', 'Ibuprofen', 'Caffeine', 'Acetaminophen', 'Naproxen', '
       Drug_X'],
    'SMILES': [
        'CC(=O)OC1=CC=CC=C1C(=O)O',
        'CC(C)CC1=CC=C(C=C1)C(C)C(=O)O',
        'CN1C=NC2=C1C(=O)N(C(=O)N2C)C',
        'CC(=O)NC1=CC=C(C=C1)O',
        'COC1=CC2=CC(C(C)C(O)=O)=CC=C2C=C1',
        'CC(C)CC2CCC(CC2)C(C)C(=O)O'
    ],
    'MW': [180.16, 206.28, 194.19, 151.16, 230.26, 350.5],
    'LogP': [1.19, 3.97, -0.07, 0.46, 3.18, 4.5],
    'pIC50': [5.2, 6.1, 4.8, 5.5, 6.8, 7.2]
})

# Add activity class column
df['Activity'] = df['pIC50'].apply(lambda x: 'Active' if x >= 6.0 else 'Inactive')

print(df)
```

14.2 Exercise 11B.2 – Data Analysis

```
import pandas as pd

# Calculate mean pIC50
mean_pic50 = df['pIC50'].mean()
print(f"Mean pIC50: {mean_pic50:.2f}")

# Count by activity class
activity_counts = df['Activity'].value_counts()
print(f"\nActivity counts:\n{activity_counts}")

# Filter drug-like compounds (MW < 500, LogP < 5)
drug_like = df[(df['MW'] < 500) & (df['LogP'] < 5)]
print(f"\nDrug-like compounds: {len(drug_like)}")

# Sort by pIC50 descending
sorted_df = df.sort_values('pIC50', ascending=False)
print(f"\nSorted by pIC50:\n{sorted_df[['Name', 'pIC50']]}")
```

14.3 Exercise 11B.3 – GroupBy Analysis

```
import pandas as pd

# Group by activity class
grouped = df.groupby('Activity').agg({
    'MW': 'mean',
```

```
    'LogP': 'mean',
    'pIC50': ['mean', 'count']
})

print("Analysis by Activity Class:")
print(grouped)

# Export to CSV
grouped.to_csv('activity_analysis.csv')
print("\nExported to activity_analysis.csv")
```

15 Lab 12: JSON & Regex

15.1 Exercise 12.1 – Parse PubChem JSON

```
import json

# PubChem-like JSON data
json_data = '{"CID": 2244, "name": "Aspirin", "MW": 180.16}'

# Parse JSON
compound = json.loads(json_data)

# Extract fields
cid = compound["CID"]
name = compound["name"]
mw = compound["MW"]

print(f"CID: {cid}")
print(f"Name: {name}")
print(f"Molecular Weight: {mw}")
```

15.2 Exercise 12.2 – Find Restriction Sites

```
import re

# DNA sequence
sequence = "ATGAATTCGCGAATTCTA"

# Find EcoRI restriction site (GAATTC)
pattern = r"GAATTC"
matches = list(re.finditer(pattern, sequence))

print(f"Sequence: {sequence}")
print(f"Pattern: {pattern} (EcoRI site)")
print(f"Found {len(matches)} matches:")

for match in matches:
    print(f"  Position {match.start()}: {match.group()}")
```

15.3 Exercise 12.3 – SMILES Validation

```
import re
```

```python
def analyze_smiles(smiles):
    """Analyze SMILES string for structural features."""
    results = {}

    # Check for aromatic ring (lowercase c, n, o, s)
    results['aromatic'] = bool(re.search(r'[cnos]', smiles))

    # Check for ring closure (digits)
    results['has_ring'] = bool(re.search(r'\d', smiles))

    # Check for double bond (=)
    results['has_double_bond'] = bool(re.search(r'=', smiles))

    return results

# Test SMILES
test_smiles = [
    "CCO",  # Ethanol
    "c1ccccc1",  # Benzene
    "CC(=O)OC1=CC=CC=C1C(=O)O"  # Aspirin
]

print("SMILES Analysis:")
print("-" * 60)
for smiles in test_smiles:
    result = analyze_smiles(smiles)
    print(f"{smiles}")
    print(f"  Aromatic: {result['aromatic']}")
    print(f"  Has ring: {result['has_ring']}")
    print(f"  Has double bond: {result['has_double_bond']}")
    print()
```

16 Rosalind Challenge Solutions

16.1 DNA – Counting DNA Nucleotides

```python
def count_nucleotides(dna):
    """Count A, C, G, T in a DNA string."""
    return dna.count('A'), dna.count('C'), dna.count('G'), dna.count('T')

# Sample
dna = "AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC"
a, c, g, t = count_nucleotides(dna)
print(f"{a} {c} {g} {t}")
```

16.2 RNA – Transcribing DNA into RNA

```python
def transcribe(dna):
    """Transcribe DNA to RNA (T -> U)."""
    return dna.replace('T', 'U')

# Sample
dna = "GATGGAACTTGACTACGTAAATT"
rna = transcribe(dna)
print(rna)  # GAUGGAACUUGACUACGUAAAUU
```

16.3 REVC – Complementing a Strand of DNA

```
def reverse_complement(dna):
    """Return the reverse complement of a DNA string."""
    complement = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G'}
    return ''.join(complement[n] for n in reversed(dna))

# Sample
dna = "AAAACCCGGT"
print(reverse_complement(dna))  # ACCGGGTTTT
```

16.4 GC – Computing GC Content

```
def gc_content(sequence):
    """Calculate GC content percentage."""
    gc = sequence.count('G') + sequence.count('C')
    return (gc / len(sequence)) * 100

# Sample
seq = "
    CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGACTGGGAACCTGCGGGCAGTAGGTGGAAT
    "
print(f"{gc_content(seq):.6f}")
```

16.5 HAMM – Counting Point Mutations

```
def hamming_distance(s1, s2):
    """Count mismatches between two strings."""
    return sum(c1 != c2 for c1, c2 in zip(s1, s2))

# Sample
s1 = "GAGCCTACTAACGGGAT"
s2 = "CATCGTAATGACGGCCT"
print(hamming_distance(s1, s2))  # 7
```