Python for Cheminformatics & Bioinformatics

Lessons: Theory & Concepts

Nirajan Bhattarai

AI-Driven Drug Development Training

February 2026

Lessons Overview

Learning Objectives

By the end of this course, you will be able to:

1. Write Python code for basic data manipulation
2. Work with molecular data (SMILES, formulas, properties)
3. Process biological sequences (DNA, RNA, protein)
4. Use control flow (conditionals, loops) for data filtering
5. Organize data using lists, dictionaries, and sets
6. Create reusable functions for cheminformatics tasks
7. Read/write files (CSV, FASTA, JSON)
8. Use NumPy arrays and Pandas DataFrames
9. Apply concepts to Rosalind bioinformatics problems

Prerequisites & Setup

Prerequisites:

- Basic computer literacy
- No prior programming experience required
- Interest in drug discovery / life sciences

Software Setup:

- Python 3.8+ installed
- IDE: VS Code, PyCharm, or Jupyter Notebook
- Libraries: `pip install numpy pandas`

Resources:

- Rosalind.info – Bioinformatics problems
- ChEMBL – Bioactivity database
- PubChem – Chemical information

Data Types You'll Encounter

Before we start coding, let's understand the data types used in drug discovery:

Cheminformatics:

- SMILES – Molecular structures
- Molecular Descriptors (MW, LogP)
- Activity Data (IC50, pIC50)
- Lipinski Properties

Bioinformatics:

- DNA Sequences (A, T, G, C)
- RNA Sequences (A, U, G, C)
- Protein Sequences (amino acids)
- FASTA Format

Understanding these data types is essential for the exercises in this course.

SMILES: Simplified Molecular Input Line Entry System

What is SMILES?
A text-based notation for representing chemical structures as strings.

Key Rules:

- Atoms: C, N, O, S, P, F, Cl, Br, I (organic subset)
- Bonds: single (default), double (=), triple (#), aromatic (:)
- Rings: Numbers indicate ring closures (e.g., `c1ccccc1` = benzene)
- Branches: Parentheses for side chains (e.g., `CC(C)C` = isobutane)
- Aromatic: Lowercase letters (c, n, o, s)

Examples:

`CCO`	Ethanol
`CC(=O)O`	Acetic acid
`c1ccccc1`	Benzene
`CC(=O)Oc1ccccc1C(=O)O`	Aspirin

SMILES: Visual Examples

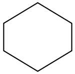
Ethanol

`CCO`

$CH_3 - CH_2 - OH$

Benzene

`c1ccccc1`



Acetic Acid

`CC(=O)O`

CH_3COOH

Why SMILES?

- Compact text representation (easy to store in databases)
- Human and machine readable
- Standard input for cheminformatics tools (RDKit, OpenBabel)
- Enables molecular property calculations

SMARTS: SMILES Arbitrary Target Specification

What is SMARTS?
A pattern matching language for substructure searching in molecules.

Key Features:

- Extension of SMILES with wildcards and logic
- Used to define functional groups and pharmacophores
- Essential for filtering compound libraries

SMARTS Operators:

`*`	Any atom
`[!C]`	Not carbon
`[#7]`	Nitrogen (by atomic number)
`[C,N]`	Carbon OR nitrogen
`[C;R]`	Carbon in a ring
`[$(C=O)]`	Recursive SMARTS

SMARTS: Common Patterns

Functional Group SMARTS:

Pattern	SMARTS	Description
Hydroxyl	`[OX2H]`	Alcohol OH
Carboxylic acid	`[CX3](=O)[OX2H1]`	COOH
Primary amine	`[NX3;H2]`	NH_2
Carbonyl	`[CX3]=[OX1]`	C=O
Aromatic ring	`[a]`	Any aromatic atom
Halogen	`[F,Cl,Br,I]`	Any halogen

Python Example (RDKit):

```python
from rdkit import Chem
mol = Chem.MolFromSmiles("CC(=O)O")  # Acetic acid
pattern = Chem.MolFromSmarts("[CX3](=O)[OX2H1]")
has_cooh = mol.HasSubstructMatch(pattern)  # True
```

SELFIES: Self-Referencing Embedded Strings

What is SELFIES?
A 100% robust molecular string representation for machine learning.

Key Advantages over SMILES:

- **Always valid**: Every SELFIES string is a valid molecule
- **No syntax errors**: Perfect for generative models
- **Bijective**: One-to-one mapping to molecules
- **ML-friendly**: Ideal for VAEs, GANs, transformers

Example Comparison:

SMILES	**SELFIES**
`CCO`	`[C][C][O]`
`c1ccccc1`	`[C][=C][C][=C][C][=C][Ring1][=Branch1]`
`CC(=O)O`	`[C][C][=Branch1][C][=O][O]`

SELFIES: Python Usage

Why SELFIES for Machine Learning?

- Random mutations always produce valid molecules
- No need to validate generated strings
- Better for evolutionary algorithms and generative AI

Python Example:

```
import selfies as sf

# SMILES to SELFIES
smiles = "CC(=O)Oc1ccccc1C(=O)O"  # Aspirin
selfies = sf.encoder(smiles)
print(selfies)
# [C][C][=Branch1][C][=O][O][C][=C][C][=C]...

# SELFIES to SMILES
recovered = sf.decoder(selfies)
print(recovered)  # CC(=O)Oc1ccccc1C(=O)O

# Get alphabet for tokenization
alphabet = sf.get_alphabet_from_selfies([selfies])
```

InChI: International Chemical Identifier

What is InChI?
A standardized, unique identifier for chemical substances by IUPAC.

Structure: `InChI=1S/formula/connections/h/...`

Layers:

- **Formula**: Molecular formula (e.g., C9H8O4)
- **Connections**: Atom connectivity
- **H-atoms**: Hydrogen atom positions
- **Charge/Stereo**: Optional stereochemistry

Example (Aspirin):

```
InChI=1S/C9H8O4/c1-6(10)13-8-5-3-2-4-7(8)9(11)12/h2-5H,1H3,(H,11,12)
```

InChIKey: 27-character hash for database lookup

```
BSYNRYMUTXBXSQ-UHFFFAOYSA-N
```

InChI vs SMILES: When to Use Which?

Feature	**SMILES**	**InChI**
Uniqueness	Not canonical*	Canonical
Human readable	Yes	Partially
Database key	No	Yes (InChIKey)
Substructure search	Yes	No
Round-trip conversion	Yes	Lossy
ML/AI input	Common	Rare
Web search	Difficult	Easy (InChIKey)

**Canonical SMILES exist but vary by toolkit*

Best Practices:

- Use **SMILES** for cheminformatics workflows
- Use **InChIKey** for database lookups and deduplication
- Use **SELFIES** for generative machine learning
- Use **SMARTS** for substructure filtering

Other Molecular Representations

Molecular Fingerprints (for similarity):

- **ECFP/Morgan**: Circular fingerprints (most common)
- **MACCS**: 166 predefined structural keys
- **RDKit FP**: Topological fingerprints
- **Atom-pair**: Pairs of atoms with distance

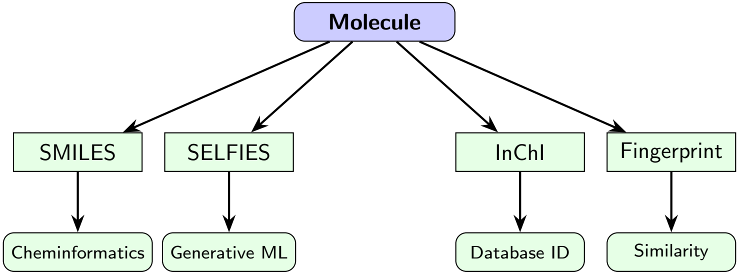
3D Representations:

- **SDF/MOL**: 3D coordinates + connectivity
- **PDB**: Protein Data Bank format
- **XYZ**: Simple coordinate format

Graph Representations (for GNNs):

- **Adjacency matrix**: Atom connectivity
- **Node features**: Atom types, charges
- **Edge features**: Bond types, orders

Molecular Representations: Summary



Quick Reference:

Task	**Use**
Property calculation	SMILES + RDKit
Molecule generation	SELFIES
Database search	InChIKey
Similarity search	Morgan/ECFP fingerprints
Pattern matching	SMARTS

Molecular Descriptors & Properties

Key Molecular Properties:

Property	Description	Typical Range
MW	Molecular Weight (Da)	150–500 Da
LogP	Lipophilicity (octanol/water)	-2 to 5
HBD	H-bond Donors	0–5
HBA	H-bond Acceptors	0–10
TPSA	Topological Polar Surface Area	0–140 Å^2
RotBonds	Rotatable Bonds	0–10

Lipinski's Rule of Five (Drug-likeness):

- MW $\leq$ 500 Da
- LogP $\leq$ 5
- HBD $\leq$ 5
- HBA $\leq$ 10

Compounds with $\leq$1 violation are likely orally bioavailable.

Bioactivity Data: IC50, Ki, and pIC50

Measuring Drug Potency:

- **IC50**: Concentration for 50% inhibition
- **Ki**: Inhibition constant (binding affinity)
- **EC50**: Concentration for 50% effect (agonists)

Units: Usually reported in nM (nanomolar) or μM (micromolar)

pIC50 Conversion:

$$\text{pIC50} = -\log_{10}(\text{IC50}_{\text{M}}) = 9 - \log_{10}(\text{IC50}_{\text{nM}})$$

Activity Classification:

pIC50	IC50 (nM)	Classification
≥ 8	≤ 10	Highly Active
6–8	10–1000	Active
5–6	1000–10000	Moderate
< 5	> 10000	Inactive

DNA: Deoxyribonucleic Acid

The Blueprint of Life:

- Double helix structure
- Four nucleotides: **A**denine, **T**hymine, **G**uanine, **C**ytosine
- Base pairing: A–T (2 H-bonds), G–C (3 H-bonds)
- Direction: 5' $\rightarrow$ 3' (reading direction)

```
5' A-T-G-C-G-A-T-C-G-A 3'
        | | | | | | | | | |
3' T-A-C-G-C-T-A-G-C-T 5'
```

Key Metrics:

- **GC Content**: % of G and C bases (stability indicator)
- **Length**: Number of base pairs (bp) or nucleotides (nt)
- **Codons**: 3-nucleotide sequences encoding amino acids

RNA: Ribonucleic Acid

DNA's Working Copy:

- Single-stranded molecule
- Four nucleotides: **A**denine, **U**racil, **G**uanine, **C**ytosine
- **Key difference**: Uracil (U) replaces Thymine (T)

Transcription (DNA → RNA):

`DNA: ATGCGATCG` → `RNA: AUGCGAUCG`

Types of RNA:

mRNA	Messenger RNA (carries genetic code)
tRNA	Transfer RNA (brings amino acids)
rRNA	Ribosomal RNA (protein synthesis)
siRNA	Small interfering RNA (gene silencing)

In Python: `rna = dna.replace("T", "U")`

Proteins: Amino Acid Sequences

The Workhorses of the Cell:

- Built from 20 standard amino acids
- Encoded by codons (3 nucleotides = 1 amino acid)
- **Start codon**: AUG (Methionine)
- **Stop codons**: UAA, UAG, UGA

Translation (RNA → Protein):

$$\texttt{AUG-GCC-UAU-...-UAA} \rightarrow \texttt{M-A-Y-...}$$

Amino Acid Properties:

Hydrophobic	A, V, L, I, M, F, W, P
Polar	S, T, N, Q, Y, C
Charged (+)	K, R, H
Charged (-)	D, E
Special	G (flexible), P (rigid)

FASTA Format: Sequence Storage

Standard Format for Biological Sequences:

```
>Rosalind_6404 Human hemoglobin alpha
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSH
GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLL
SHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
>Rosalind_5959 E. coli beta-galactosidase
MTMITDSLAVVLQRRDWENPGVTQLNRLAAHPPFASWRNSEEARTDRPSQQ
LRSLNGEWRFAWFPAPEAVPESWLECDLPEADTVVVPSNWQMHGYDAPIYT
```

Format Rules:

- Header line starts with > followed by sequence ID
- Sequence data on following lines (typically 60–80 chars/line)
- Multiple sequences in one file

Common source: Rosalind.info bioinformatics problems

Key Operations in Bioinformatics:

- **Transcribe**: DNA $\rightarrow$ RNA (replace T with U)
- **Translate**: RNA $\rightarrow$ Protein (use codon table)
- **Reverse Complement**: Get complementary DNA strand
- **GC Content**: Calculate sequence stability

Summary: Data Types Reference

Data Type	**Python Type**	**Example**
SMILES	`str`	`"CC(=O)Oc1ccccc1C(=O)O"`
MW	`float`	`180.16`
LogP	`float`	`1.19`
IC50 (nM)	`float`	`5.2`
pIC50	`float`	`8.28`
Drug-like	`bool`	`True`
DNA sequence	`str`	`"ATGCGATCG"`
RNA sequence	`str`	`"AUGCGAUCG"`
Protein sequence	`str`	`"MAMAPRTEIN"`
GC content	`float`	`55.5`
Sequence length	`int`	`1542`

All biological sequences are strings in Python!

Lesson 1: Learning Objectives

Learning Objectives:

- Understand how variables store data in memory
- Identify Python's core data types (int, float, str, bool)
- Perform type conversions between data types

Description:
Variables are the foundation of programming – named containers that hold data. Understanding data types ensures correct operations and prevents errors in scientific computing. **Applications:**

- Store compound properties (MW, LogP, SMILES)
- Represent bioactivity measurements (IC50, pIC50)
- Handle DNA/RNA sequence data

Lesson 1: Variables & Data Types

Concept: Variables store data in memory.

Python data types: `int`, `float`, `str`, `bool`, `NoneType`

Type Conversion: `int()`, `float()`, `str()`, `bool()`

Drug Discovery Scenarios:

- Compound Info (name, MW, LogP, SMILES)
- Bioactivity Data (IC50, Ki, pIC50)
- DNA/RNA Sequences (nucleotide strings)

Lesson 1 Code Example

```python
# Compound Info
compound_name = "Aspirin"
mw = 180.16  # Molecular Weight (Da)
logP = 1.19  # Lipophilicity
smiles = "CC(=O)OC1=CC=CC=C1C(=O)O"

# Bioactivity Data
ic50_nM = 5.2  # IC50 in nanomolar
pic50 = 8.28   # -log10(IC50 in M)
is_active = True

# DNA Sequence
dna_seq = "ATGCGATCGATCG"
seq_length = len(dna_seq)

# Type Conversion
ic50_str = str(ic50_nM)
mw_int = int(mw)

print(f"{compound_name}: MW={mw}, pIC50={pic50}")
```

Lesson 2: Learning Objectives

Learning Objectives:

- Use arithmetic operators for scientific calculations
- Apply comparison operators to filter data
- Combine conditions using logical operators

Description:
Operators are symbols that perform computations and comparisons. They enable mathematical transformations, data filtering, and decision-making in code.
Applications:

- Convert IC50 to pIC50 ($-\log_{10}$)
- Check Lipinski Rule of Five compliance
- Calculate GC content in DNA sequences

Lesson 2: Operators

Arithmetic: `+`, `-`, `*`, `/`, `%`, `**`

Comparison: `>`, `<`, `==`, `!=`, `>=`, `<=`

Logical: `and`, `or`, `not`

Drug Discovery scenarios: Calculate pIC50, check Lipinski rules, filter active compounds

Lesson 2 Code Example

```python
import math

# IC50 to pIC50 conversion
ic50_nM = 10.0  # nanomolar
ic50_M = ic50_nM * 1e-9  # convert to molar
pic50 = -math.log10(ic50_M)  # pIC50 = 8.0
print(f"IC50: {ic50_nM} nM -> pIC50: {pic50:.2f}")

# Lipinski Rule of Five checks
mw, logP, hbd, hba = 450, 3.5, 2, 6
lipinski_ok = (mw <= 500) and (logP <= 5) and (hbd <= 5) and
    (hba <= 10)
print(f"Passes Lipinski: {lipinski_ok}")

# GC Content calculation
seq = "ATGCGCGCTA"
gc_count = seq.count("G") + seq.count("C")
gc_percent = (gc_count / len(seq)) * 100
print(f"GC Content: {gc_percent:.1f}%")
```

Lesson 3: Learning Objectives

Learning Objectives:

- Manipulate strings using indexing and slicing
- Apply string methods for text processing
- Parse and transform sequence data

Description:
Strings are sequences of characters essential for representing biological sequences (DNA, RNA, proteins) and chemical notations (SMILES). String manipulation is fundamental to bioinformatics. **Applications:**

- Transcribe DNA to RNA (T $\rightarrow$ U)
- Generate reverse complement sequences
- Parse SMILES for molecular features

Lesson 3: Strings

Strings store text – essential for sequences and SMILES.

Methods: indexing/slicing, `len()`, `upper()`, `lower()`, `replace()`, `split()`, `count()`, `find()`

Drug Discovery scenarios: DNA/RNA sequences, SMILES strings, protein sequences

Lesson 3 Code Example

```python
# DNA Sequence manipulation
dna = "ATGCGATCGATCG"
print(f"Length: {len(dna)}")
print(f"First 3 (codon): {dna[:3]}")  # ATG
print(f"Last codon: {dna[-3:]}")       # TCG

# Transcription: DNA -> RNA (T -> U)
rna = dna.replace("T", "U")
print(f"RNA: {rna}")  # AUGCGAUCGAUCG

# Count nucleotides
print(f"A: {dna.count('A')}, T: {dna.count('T')}")
print(f"G: {dna.count('G')}, C: {dna.count('C')}")

# SMILES analysis
smiles = "CC(=O)OC1=CC=CC=C1C(=O)O"
has_ring = any(c.isdigit() for c in smiles)
print(f"Has ring: {has_ring}")  # True
```

Lesson 4: Learning Objectives

Learning Objectives:

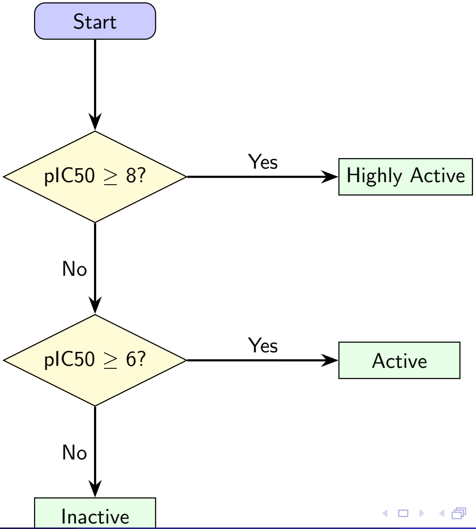
- Control program flow with if/elif/else statements
- Use match-case for pattern matching (Python 3.10+)
- Build nested conditional logic

Description:
Conditionals allow programs to make decisions based on data values. They enable classification, filtering, and rule-based logic essential for compound screening.
Applications:

- Classify compounds by activity level
- Check drug-likeness (Lipinski violations)
- Identify start/stop codons in sequences

Lesson 4: Conditional Statements

`if`/`elif`/`else` for branching
`match-case` (Python 3.10+) for pattern matching



Lesson 4 Code Example (if/else)

```python
# Classify compound activity by pIC50
pic50 = 7.5
if pic50 >= 8:
    activity = "Highly Active"
elif pic50 >= 6:
    activity = "Active"
elif pic50 >= 5:
    activity = "Moderate"
else:
    activity = "Inactive"
print(f"pIC50 {pic50}: {activity}")

# Check Lipinski Rule of Five
mw, logP, hbd, hba = 450, 4.2, 2, 6
violations = 0
if mw > 500: violations += 1
if logP > 5: violations += 1
if hbd > 5: violations += 1
if hba > 10: violations += 1
drug_like = "Yes" if violations <= 1 else "No"
print(f"Drug-like: {drug_like} ({violations} violations)")
```

Lesson 4 Code Example (match-case)

```python
# Identify codon type (Python 3.10+)
codon = "ATG"
match codon:
    case "ATG":
        print("Start codon (Methionine)")
    case "TAA" | "TAG" | "TGA":
        print("Stop codon")
    case _:
        print("Coding codon")

# Classify nucleotide
nucleotide = "G"
match nucleotide:
    case "A" | "G":
        base_type = "Purine"
    case "C" | "T":
        base_type = "Pyrimidine"
    case _:
        base_type = "Unknown"
print(f"{nucleotide} is a {base_type}")
```

Lesson 5: Learning Objectives

Learning Objectives:

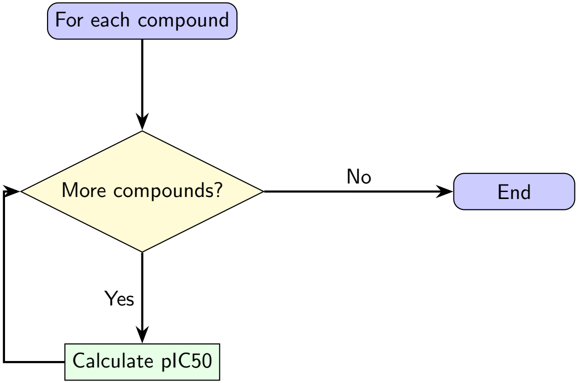
- Iterate over sequences using for loops
- Use while loops for conditional repetition
- Control loop flow with break and continue

Description:
Loops automate repetitive tasks by executing code blocks multiple times. They are essential for processing compound libraries and analyzing sequence data at scale. **Applications:**

- Process compound libraries (batch MW calculation)
- Count nucleotide frequencies (Rosalind DNA)
- Screen compounds against activity thresholds

Lesson 5: Loops

`for` loop: iterate sequences/range
`while` loop: repeat until condition False
`break`/`continue`: control loop flow



Lesson 5 Code Example

```python
import math

# Process compound library - calculate pIC50
ic50_values = [5.2, 120.0, 8.7, 2.1, 450.0]  # nM
for ic50 in ic50_values:
    pic50 = 9 - math.log10(ic50)
    print(f"IC50: {ic50:>6.1f} nM -> pIC50: {pic50:.2f}")

# Count nucleotides in DNA sequence
dna = "ATGCGATCGATCG"
counts = {"A": 0, "T": 0, "G": 0, "C": 0}
for nucleotide in dna:
    counts[nucleotide] += 1
print(f"Nucleotide counts: {counts}")

# Find active compounds (break/continue)
compounds = [("CPD1", 7.2), ("CPD2", 5.1), ("CPD3", 8.5)]
for name, pic50 in compounds:
    if pic50 < 6: continue  # skip inactive
    if pic50 > 8: break     # found highly active
    print(f"{name}: Active (pIC50={pic50})")
```

Lesson 6: Learning Objectives

Learning Objectives:

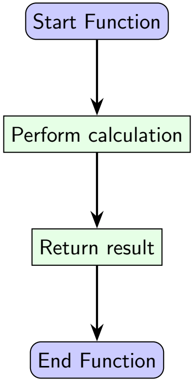
- Define reusable functions with parameters
- Use return statements to output results
- Apply *args and **kwargs for flexible inputs

Description:
Functions encapsulate reusable code blocks, promoting modularity and reducing duplication. They are the building blocks of scientific pipelines and analysis tools.
Applications:

- Create IC50 $\rightarrow$ pIC50 converters
- Build Lipinski property calculators
- Implement sequence analysis functions (GC, REVC)

Lesson 6: Functions

`def name(params):` define function
`return` for return value
Default arguments, *args, **kwargs


$$\begin{tikzcd}[row sep=large]
\text{Start Function} \arrow[d] \\
\text{Perform calculation} \arrow[d] \\
\text{Return result} \arrow[d] \\
\text{End Function}
\end{tikzcd}$$

Lesson 6 Code Example

```python
# Function: IC50 to pIC50 conversion
def ic50_to_pic50(ic50_nm):
    """Convert IC50 (nM) to pIC50."""
    return -math.log10(ic50_nm * 1e-9)

# Function with validation
def calculate_molecular_weight(smiles):
    """Calculate MW from SMILES."""
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None, "Invalid SMILES"
    return Descriptors.MolWt(mol), "Success"

# *args - process multiple compounds
def average_activity(*pic50_values):
    return sum(pic50_values) / len(pic50_values)

# **kwargs - compound properties
def print_compound(**props):
    for key, value in props.items():
        print(f"{key}: {value}")
```

Lesson 6B: Learning Objectives

Learning Objectives:

- Handle runtime errors with try/except blocks
- Use else and finally for cleanup operations
- Raise custom exceptions for validation

Description:
Error handling prevents program crashes from invalid data or unexpected conditions. Robust error handling is critical when processing real-world chemical/biological data with missing or malformed entries. **Applications:**

- Handle invalid SMILES parsing gracefully
- Manage missing data in compound datasets
- Validate FASTA file formats

Lesson 6B: Error Handling (try/except)

Concept: Handle runtime errors gracefully

Keywords: `try`, `except`, `else`, `finally`, `raise`

Common Exceptions:

- `ValueError` – invalid value conversion
- `TypeError` – wrong type operation
- `ZeroDivisionError` – division by zero
- `FileNotFoundError` – file doesn't exist
- `KeyError` – dict key not found
- `IndexError` – list index out of range

Lesson 6B Code Example

```python
# Basic try/except
try:
    num = int(input("Enter number: "))
    result = 10 / num
except ValueError:
    print("Invalid input!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Execution complete")

# Raising exceptions
def divide(a, b):
    if b == 0:
        raise ValueError("Divisor cannot be zero")
    return a / b
```

Lesson 7: Learning Objectives

Learning Objectives:

- Create and modify lists using built-in methods
- Access elements via indexing and slicing
- Perform common list operations (append, remove, sort)

Description:
Lists are ordered, mutable collections that store sequences of items. They are the primary data structure for managing compound libraries and activity datasets.
Applications:

- Store SMILES strings for compound libraries
- Manage pIC50 activity measurements
- Build queues for batch processing

Lesson 7: Python Lists – Basics

Lists store ordered sequences.

Methods: `append`, `extend`, `insert`, `remove`, `pop`, `clear`, `index`, `count`, `copy`

Scenario Examples: SMILES list, pIC50 values, compound IDs, sequence fragments

Lesson 7 Code Example

```python
smiles_list = ["CCO", "CC(=O)O", "c1ccccc1"]
smiles_list.append("CCN")
smiles_list.insert(1, "CC")
smiles_list.remove("CCO")
print(smiles_list[0], smiles_list[-1])
```

Lesson 7B: Learning Objectives

Learning Objectives:

- Use tuples for immutable data records
- Apply sets for unique element collections
- Perform set operations (union, intersection, difference)

Description:
Tuples provide immutable sequences ideal for fixed records. Sets offer fast membership testing and mathematical set operations for comparing collections.
Applications:

- Store compound records (name, SMILES, pIC50)
- Find unique molecular scaffolds
- Compare compound libraries (common/unique hits)

Lesson 7B: Tuples & Sets

Tuples: Immutable ordered sequences

- Created with `()` or `tuple()`
- Cannot modify after creation
- Use for fixed data (coordinates, RGB colors)

Sets: Unordered collection of unique elements

- Created with {} or `set()`
- No duplicates allowed
- Fast membership testing
- Set operations: union, intersection, difference

Lesson 7B Code Example

```python
# Tuples - immutable (compound data)
compound = ("Aspirin", "CC(=O)OC1=CC=CC=C1C(=O)O", 180.16)
name, smiles, mw = compound  # unpacking

# Sets - unique scaffolds
scaffolds = {"benzene", "pyridine", "benzene"}  # 2 unique
scaffolds.add("furan")

# Set operations for compound comparison
lib_A = {"CMP001", "CMP002", "CMP003"}
lib_B = {"CMP002", "CMP003", "CMP004"}
print(lib_A | lib_B)  # union: all compounds
print(lib_A & lib_B)  # intersection: common
print(lib_A - lib_B)  # unique to lib_A
```

Lesson 8: Learning Objectives

Learning Objectives:

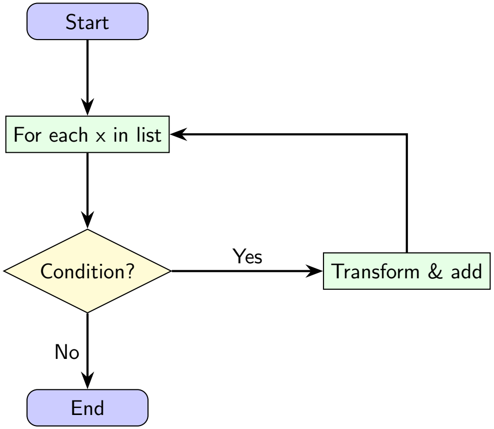
- Write concise list comprehensions
- Apply map() and filter() for transformations
- Combine functional programming techniques

Description:
List comprehensions and functional tools (map, filter) enable concise, readable data transformations. They replace verbose loops with elegant one-liners.
Applications:

- Filter active compounds (pIC50 > 6)
- Batch convert IC50 to pIC50 values
- Extract drug-like compounds (MW < 500)

Lesson 8: List Comprehensions & Map/Filter

Concepts: Transform, filter, map, lambda functions



Lesson 8 Code Example

```
pic50_values = [5.2, 6.8, 7.3, 4.9, 8.1]

# List comprehension: filter active compounds
actives = [p for p in pic50_values if p >= 6.0]

# Lambda + map: convert pIC50 to IC50 (nM)
ic50_nm = list(map(lambda p: 10**(9-p), pic50_values))

# Filter: highly potent (pIC50 > 7)
potent = list(filter(lambda p: p > 7, pic50_values))

print(actives, ic50_nm, potent)
```

Lesson 8B: Learning Objectives

Learning Objectives:

- Create anonymous functions with lambda
- Understand variable scope (local, global, nonlocal)
- Use closures for stateful functions

Description:
Lambda functions are compact, inline functions for simple operations. Understanding scope ensures correct variable access and prevents bugs in complex programs. **Applications:**

- Sort compounds by activity with custom keys
- Create quick property calculators
- Build stateful counters for batch processing

Lesson 8B: Lambda Functions & Variable Scope

Lambda: Anonymous single-expression functions

`lambda args:  expression`

Variable Scope:

- **Local** – inside function
- **Enclosing** – outer function (nested)
- **Global** – module level
- **Built-in** – Python built-ins

Use `global` keyword to modify global variables
Use `nonlocal` for enclosing scope

Lesson 8B Code Example

```
# Lambda functions for molecular properties
to_pic50 = lambda ic50: -math.log10(ic50 * 1e-9)
is_active = lambda p: p >= 6.0

# Sorting compounds by activity
compounds = [("Aspirin", 5.2), ("Ibuprofen", 6.8), ("Drug_X"
    , 7.5)]
compounds.sort(key=lambda x: x[1], reverse=True)

# Variable scope in processing
processed_count = 0  # global

def process_batch():
    global processed_count
    processed_count += 1

def create_counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
```

Lesson 9: Learning Objectives

Learning Objectives:

- Create and manipulate key-value dictionaries
- Access, update, and iterate over dict items
- Use dict comprehensions for transformations

Description:
Dictionaries store data as key-value pairs, enabling fast lookups by name. They are ideal for structured data like compound databases and lookup tables.
Applications:

- Build compound databases (name $\rightarrow$ properties)
- Create codon translation tables
- Store molecular descriptor lookups

Lesson 9: Dictionaries

Key-value storage, unordered, mutable.

Methods: `keys()`, `values()`, `items()`, `get()`, `update()`, `pop()`, `popitem()`, `clear`

Scenarios: Compound database, codon table, property lookup

Lesson 9 Code Example

```python
compound_db = {
    "Aspirin": {"SMILES": "CC(=O)OC1=CC=CC=C1C(=O)O", "pIC50": 5.2},
    "Caffeine": {"SMILES": "CN1C=NC2=C1C(=O)N(C(=O)N2C)C", "pIC50": 4.8}
}

# Add new compound
compound_db["Ibuprofen"] = {"SMILES": "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O", "pIC50": 6.1}

# Filter actives (dict comprehension)
actives = {k: v for k, v in compound_db.items() if v["pIC50"] >= 5.0}

print(actives)
```

Lesson 10: Learning Objectives

Learning Objectives:

- Read and write text files using open()
- Use context managers (with) for safe file handling
- Parse structured file formats (CSV, FASTA)

Description:
File I/O enables programs to read input data and save results. Essential for working with compound datasets (CSV, SDF) and biological sequences (FASTA).
Applications:

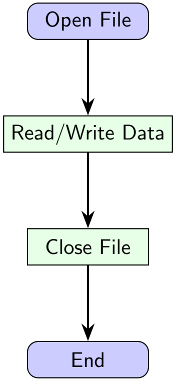
- Read/write compound CSV files
- Parse FASTA sequence files
- Export filtered results for downstream analysis

Lesson 10: File Handling

Read/write text files.

Methods: `open()`, `read()`, `readline()`, `readlines()`, `write()`, `writelines()`, `close()`

Use `with` for automatic closing



Lesson 10 Code Example

```python
# Write compound data to CSV
with open("compounds.csv", "w") as f:
    f.write("name,smiles,pIC50\n")
    f.write("Aspirin,CC(=O)OC1=CC=CC=C1C(=O)O,5.2\n")

# Read FASTA file
with open("sequence.fasta", "r") as f:
    header = f.readline().strip()  # >sequence_id
    sequence = ""
    for line in f:
        sequence += line.strip()
```

Lesson 11: Learning Objectives

Learning Objectives:

- Create and manipulate NumPy arrays
- Perform vectorized mathematical operations
- Use boolean indexing for data filtering

Description:
NumPy provides efficient N-dimensional arrays for numerical computing. Vectorized operations are orders of magnitude faster than Python loops for large datasets. **Applications:**

- Store molecular descriptor matrices
- Normalize and scale feature data
- Compute statistics on activity arrays

Lesson 11: NumPy Arrays

NumPy arrays for efficient numerical computation

Key Data Structures:

- `ndarray` – N-dimensional array (homogeneous data)
- Supports 1D (vector), 2D (matrix), nD (tensor)

Array Creation:

- `np.array()` – from list/tuple
- `np.zeros()`, `np.ones()` – filled arrays
- `np.arange()`, `np.linspace()` – sequences
- `np.random.rand()` – random arrays

Lesson 11: NumPy Properties & Operations

Array Properties:

- `shape` – dimensions (rows, cols)
- `dtype` – data type (int64, float64)
- `ndim` – number of dimensions
- `size` – total elements

Key Operations:

- Element-wise: `+`, `-`, `*`, `/`, `**`
- Aggregation: `sum()`, `mean()`, `std()`, `min()`, `max()`
- Reshaping: `reshape()`, `flatten()`, `transpose()`
- Indexing: slicing, boolean masks, fancy indexing

Lesson 11 Code Example

```python
import numpy as np

# Array creation
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
zeros = np.zeros((2, 3))       # 2x3 array of zeros
ones = np.ones((3, 3))         # 3x3 array of ones
seq = np.arange(0, 10, 2)      # [0, 2, 4, 6, 8]

# Properties
print(arr.shape, arr.dtype, arr.ndim)

# Operations
print(arr * 2)                 # element-wise multiply
print(arr.sum(axis=0))         # sum per column
print(arr.mean(axis=1))        # mean per row

# Boolean indexing
mask = arr > 5
print(arr[mask])               # [6, 7, 8, 9]
```

Lesson 11B: Learning Objectives

Learning Objectives:

- Create Series and DataFrame structures
- Filter, group, and aggregate tabular data
- Read/write data from CSV, Excel, and SDF files

Description:
Pandas provides labeled data structures for data analysis. DataFrames are the standard for handling compound datasets with mixed data types and missing values. **Applications:**

- Manage compound libraries with properties
- Analyze bioactivity data (groupby, statistics)
- Merge descriptor and activity datasets

Lesson 11B: Pandas Data Structures

Pandas provides powerful data structures for data analysis

Key Data Structures:

- `Series` – 1D labeled array (like a column)
- `DataFrame` – 2D labeled table (rows & columns)

Why Pandas?

- Handles heterogeneous data (mixed types)
- Built-in handling of missing values (NaN)
- Powerful indexing and filtering
- Easy file I/O (CSV, Excel, JSON, SQL)

Lesson 11B: Series & DataFrame

Series: 1D array with labels (index)

- Created from list, dict, or scalar
- Access by label: `s['a']` or position: `s[0]`

DataFrame: 2D table with row/column labels

- Created from dict, list of dicts, or 2D array
- Columns = Series
- Access column: `df['col']` or `df.col`
- Access row: `df.loc['label']` or `df.iloc[0]`

Lesson 11B Code Example (Creation)

```python
import pandas as pd

# Series - activity values with compound IDs
activities = pd.Series([5.2, 6.8, 7.3], index=['CMP001', '
    CMP002', 'CMP003'])
print(activities['CMP002'])  # 6.8

# DataFrame from compound data
df = pd.DataFrame({
    'Name': ['Aspirin', 'Ibuprofen', 'Caffeine'],
    '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)C(=O)N2C'],
    'pIC50': [5.2, 6.1, 4.8],
    'MW': [180.16, 206.28, 194.19]
})
print(df)

# DataFrame from list of dicts
data = [{'x': 1, 'y': 2}, {'x': 3, 'y': 4}]
df2 = pd.DataFrame(data)
```

Lesson 11B Code Example (Operations)

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Aspirin', 'Ibuprofen', 'Caffeine', 'Drug_X'],
    'pIC50': [5.2, 6.1, 4.8, 7.5],
    'MW': [180.16, 206.28, 194.19, 320.5]
})

# Basic properties
print(df.shape, df.columns, df.dtypes)

# Selection
print(df['Name'])                # single column
print(df[['Name', 'pIC50']])# multiple columns
print(df.loc[0])                 # row by label
print(df.iloc[0:2])              # rows by position

# Filtering
actives = df[df['pIC50'] > 6.0]
drug_like = df[df['MW'] < 500]
```

Lesson 11B Code Example (Analysis)

```
# Aggregation
print(df['pIC50'].mean())     # average activity
print(df['pIC50'].max())      # most potent
print(df.describe())          # summary statistics

# GroupBy by activity class
df['Class'] = df['pIC50'].apply(lambda x: 'Active' if x >= 6
    else 'Inactive')
grouped = df.groupby('Class')['MW'].mean()

# Adding columns
df['IC50_nM'] = 10**(9 - df['pIC50'])

# Sorting by activity
df_sorted = df.sort_values('pIC50', ascending=False)

# Missing data handling
df['LogP'] = [1.2, None, -0.5, 2.3]
df['LogP'].fillna(df['LogP'].mean(), inplace=True)
```

Lesson 11B Code Example (File I/O)

```python
import pandas as pd

# Read compound data from CSV
df = pd.read_csv('compounds.csv')

# Write filtered actives
df[df['pIC50'] > 6].to_csv('actives.csv', index=False)

# Read from SDF (via RDKit)
from rdkit import Chem
from rdkit.Chem import PandasTools
df = PandasTools.LoadSDF('molecules.sdf')

# Quick data exploration
print(df.head())        # first 5 compounds
print(df.info())        # column types
print(df.describe())    # statistical summary
```

Lesson 12: Learning Objectives

Learning Objectives:

- Parse and generate JSON data
- Write regex patterns for text matching
- Extract and validate patterns in sequences

Description:
JSON is the standard format for web APIs (PubChem, ChEMBL). Regular expressions enable powerful pattern matching for sequence motifs and data validation. **Applications:**

- Query ChEMBL/PubChem REST APIs
- Find restriction sites in DNA sequences
- Validate SMILES and sequence formats

Lesson 12: JSON & Regex

JSON: exchange data between systems (PubChem API, ChEMBL)
`json.loads()`, `json.dumps()`

Regex: pattern matching for SMILES, sequences
`re.search()`, `re.findall()`, `re.sub()`

Lesson 12 Code Example

```python
import json
import re

# JSON - PubChem-like data
data = '{"name": "Aspirin", "CID": 2244, "MW": 180.16}'
compound = json.loads(data)
print(compound["name"])

# Regex - find DNA motifs
seq = "ATGCGATCGATCG"
matches = re.findall(r"GATC", seq)  # restriction site
```

Lesson 13: Learning Objectives

Learning Objectives:

- Import and use modules and packages
- Create custom reusable modules
- Organize code with proper structure

Description:
Modules organize code into reusable files. Packages bundle related modules. This enables building maintainable scientific pipelines and sharing code across projects.
Applications:

- Use RDKit for cheminformatics
- Create molecular utility libraries
- Build reusable bioinformatics toolkits

Lesson 13: Modules & Packages

Module: Single Python file with reusable code

Package: Directory containing multiple modules

Import Styles:

- `import module`
- `from module import function`
- `from module import *`
- `import module as alias`

Creating Modules: Any `.py` file is a module

`__name__`: Use `if __name__ == "__main__":`

Lesson 13 Code Example

```python
# Cheminformatics modules
from rdkit import Chem
from rdkit.Chem import Descriptors
import math

# Create molecule utilities (mol_utils.py)
# def calc_lipinski(smiles):
#     mol = Chem.MolFromSmiles(smiles)
#     return {
#         'MW': Descriptors.MolWt(mol),
#         'LogP': Descriptors.MolLogP(mol),
#         'HBD': Descriptors.NumHDonors(mol),
#         'HBA': Descriptors.NumHAcceptors(mol)
#     }

# Main guard
if __name__ == "__main__":
    print("Running QSAR pipeline...")
```

Course Summary

Section 1: Python Basics (Lessons 1–6B)

- Variables (molecules, bioactivity), Data Types
- Operators (IC50 conversion, MW calculation)
- Strings (SMILES, DNA sequences)
- Conditionals (drug-likeness, activity classification)
- Loops (compound libraries, sequence processing)
- Functions (property calculators), Error Handling

Section 2: Collections & Data (Lessons 7–12)

- Lists (SMILES), Tuples (compound records), Sets (scaffolds), Dicts (compound DB)
- List Comprehensions, Lambda for filtering
- File Handling (CSV, FASTA, SDF)
- NumPy (descriptor matrices), Pandas (compound DataFrames)
- JSON (ChEMBL API), Regex (sequence motifs)

Next Steps

Practice Resources:

- Rosalind.info for bioinformatics problems
- ChEMBL/PubChem for real compound data
- RDKit tutorials for cheminformatics

Topics to Explore Next:

- Object-Oriented Programming (Molecule classes)
- Machine Learning (scikit-learn, XGBoost)
- QSAR/QSPR modeling pipelines
- Molecular visualization (Py3Dmol, NGLview)
- Deep learning (PyTorch, molecular graphs)
- Docking & virtual screening

Questions?