PHARM. INFORMATICS & COMPUTATIONAL SCIENCE SERIES

Hands-On Lecture Note

Advanced Data Structures & Scientific Computing for Pharmacy

Structured Multi-Element Objects • Indexing & Slicing • String Manipulation • NumPy Arrays • Vector Mathematics • Biological Data Tracking

Module Focus: Python for Pharmaceutical Data Analysis

Recommended Tools: Python 3.10+, NumPy, Jupyter Notebook / Google Colab

Audience: Undergraduate & Postgraduate Pharmacy Students (B.Pharm / Pharm.D / M.Pharm)

Prepared for classroom, laboratory, and self-paced hands-on learning

Prepared by Dr. Moushumi Barman

Table of Contents

1. Introduction: Why Data Structures Matter in Pharmacy Science

Modern pharmacy practice and pharmaceutical research generate enormous volumes of structured and semi-structured data: patient dosing records, drug formulation compositions, HPLC/assay readings, pharmacokinetic time-series, adverse-event logs, and inventory ledgers. Handling this information efficiently requires more than spreadsheets — it requires programmatic data structures that can be searched, filtered, transformed, and computed upon at scale. This lecture note introduces the core data structures of the Python programming language — lists, tuples, and dictionaries — and extends into scientific computing with NumPy, the foundational numerical library used throughout computational pharmacy, bioinformatics, and pharmacometrics.

By the end of this module, learners will be able to represent pharmaceutical datasets programmatically, extract and mutate specific data elements, manipulate textual drug information, and perform vectorized mathematical operations on numeric arrays such as dose-response curves, plasma concentration profiles, and multi-batch quality-control results.

1.1 Learning Objectives

- Differentiate between lists, tuples, and dictionaries and select the correct structure for a pharmacy data scenario.
- Perform indexing, slicing, and mutation on one- and two-dimensional data collections.
- Apply advanced string methods to clean and parse drug names, batch codes, and prescription text.
- Create and inspect multi-dimensional NumPy arrays for pharmaceutical datasets.
- Execute vectorized mathematical and statistical operations across arrays (dose calculations, concentration conversions, assay statistics).
- Align datasets from multiple sources (e.g., dispensing records and lab results) using shared identifiers.
- Design a structured array / record array to track biological and pharmacological variables over time.

1.2 Setting Up the Environment

All examples in this note use standard Python 3 syntax and the NumPy library. Install NumPy with the command below, then verify the installation:

```
# Terminal / command prompt
pip install numpy pandas matplotlib

# Inside Python or Jupyter Notebook
import numpy as np
```

```
print(np.__version__)   # confirms NumPy is installed correctly
```

> ⚙ **Instructional Note**
> *Throughout this note, code cells are designed to be typed and executed sequentially in a single Jupyter Notebook or Google Colab session, so that variables defined in one example remain available in the next.*

1.3 Primitive Data Types Underlying Every Structure

Before combining values into lists, tuples, or dictionaries, it is worth reviewing the primitive (scalar) data types that populate them, since NumPy in particular is strict about the type of data an array holds.

Type	Example	Typical Pharmacy Use
int	500	Dose in mg, patient age in years
float	72.5	Weight in kg, plasma concentration
str	'Amoxicillin'	Drug name, batch code, instructions
bool	True / False	Allergy flag, follow-up-required flag

A single Python list may freely mix these types, but a NumPy array is homogeneous — all elements share one dtype, which is precisely what allows NumPy to perform fast vectorized mathematics, a distinction explored fully in Section 4.

2. Structured Multi-Element Objects

Python provides three foundational “container” types for grouping related data: lists, tuples, and dictionaries. Each has different rules regarding order, mutability, and access, making them suited to different pharmaceutical data-handling tasks — from a simple list of drug names to a structured dictionary representing a full patient record.

2.1 Lists — Mutable Ordered Collections

A list is an ordered, mutable (changeable) collection of items, defined using square brackets. Lists are ideal for representing a sequence of related values that may grow, shrink, or be re-ordered — such as a running log of dispensed medications or a series of blood-pressure readings.

```
# A list of antihypertensive drugs stocked in the pharmacy
antihypertensives = ['Amlodipine', 'Losartan', 'Enalapril',
'Hydrochlorothiazide']

print(antihypertensives)
print(type(antihypertensives))          # <class 'list'>
print(len(antihypertensives))           # 4

# Adding and removing items
antihypertensives.append('Telmisartan')      # add to end
antihypertensives.insert(1, 'Ramipril')       # insert at position 1
antihypertensives.remove('Enalapril')         # remove by value
popped = antihypertensives.pop()              # removes & returns last
item
print(antihypertensives)
```

Common list operations relevant to pharmacy datasets include sorting drug names alphabetically, reversing chronological dosing logs, and concatenating multiple batches of records.

```
systolic_bp = [138, 145, 129, 152, 133]     # weekly systolic BP readings
(mmHg)

systolic_bp.sort()                # ascending order in place
print(systolic_bp)                # [129, 133, 138, 145, 152]

systolic_bp.sort(reverse=True)    # descending order
average_bp = sum(systolic_bp) / len(systolic_bp)
print('Average systolic BP:', round(average_bp, 1))
```

2.2 Tuples — Immutable Ordered Collections

A tuple is an ordered but immutable collection, defined with parentheses. Once created, its elements cannot be changed, added, or removed. Tuples are well suited to data that should remain constant throughout a program — for example, a fixed drug identity record (name, strength, dosage form) or a coordinate pair such as (batch_id, expiry_date).

```
# A tuple representing a fixed drug record: (name, strength_mg,
dosage_form)
drug_record = ('Paracetamol', 500, 'Tablet')

print(drug_record[0])          # 'Paracetamol'
print(drug_record[1])          # 500

# Attempting to modify raises an error — tuples are immutable
try:
    drug_record[1] = 650
except TypeError as e:
    print('Error:', e)         # 'tuple' object does not support item
assignment

# Tuple unpacking — a very common pharmacy-data pattern
name, strength, form = drug_record
print(f'{name} is available as a {strength} mg {form}')
```

Because tuples cannot be modified accidentally, they are frequently used as dictionary keys and as fixed-format records returned from functions, such as a function that returns (mean, standard_deviation) for an assay result.

2.3 Dictionaries — Key-Value Mapped Data

A dictionary stores data as key–value pairs, defined with curly braces. Dictionaries are unordered-by-index but preserve insertion order in modern Python, and they allow extremely fast lookup by a meaningful key rather than a numeric position — ideal for representing a patient profile, a drug monograph, or a formulation recipe.

```
# A dictionary representing a single patient's pharmacotherapy profile
patient = {
    'patient_id': 'PT-1042',
    'age': 58,
    'weight_kg': 72.5,
    'diagnosis': 'Type 2 Diabetes Mellitus',
    'medications': ['Metformin', 'Glimepiride'],
    'creatinine_mg_dl': 1.1
}
```

```
print(patient['diagnosis'])            # access by key
print(patient.get('allergies', 'None recorded'))   # safe access with
default

# Adding / updating entries
patient['allergies'] = ['Sulfa drugs']
patient['weight_kg'] = 73.0             # update existing value

# Iterating over a dictionary
for key, value in patient.items():
    print(f'{key}: {value}')
```

Dictionaries may be nested to represent hierarchical pharmaceutical data, such as a formulary containing multiple drugs, each with its own set of properties:

```
formulary = {
    'Amoxicillin': {'class': 'Penicillin', 'route': 'Oral',
'half_life_hr': 1.3},
    'Ceftriaxone': {'class': 'Cephalosporin', 'route': 'IV',
'half_life_hr': 8.0},
}

print(formulary['Ceftriaxone']['half_life_hr'])   # 8.0
```

2.4 Choosing the Right Structure

Feature	List	Tuple	Dictionary
Syntax	[ ]	( )	{ key: value }
Ordered	Yes	Yes	Yes (insertion order)
Mutable	Yes	No	Yes
Access method	Index (position)	Index (position)	Key (label)
Typical pharmacy use	Dose logs, batch lists	Fixed drug records	Patient/drug profiles
Duplicates allowed	Yes	Yes	Keys: No, Values: Yes

✏️ **Hands-On Exercise: Structured Objects**

1. Create a list named 'nsaids' containing five NSAID drug names, then append 'Diclofenac' and sort the list alphabetically.

2. Create a tuple 'vial_info' = ('Insulin Glargine', 100, 'units/mL') and print each unpacked value with a descriptive label.

3. Build a dictionary 'patient2' with keys for patient_id, age, and a list of current medications, then add a new key 'renal_function' with value 'Normal'.

2.5 Sets — Unique Unordered Collections

A closely related fourth structure, the set, stores only unique, unordered elements and is defined with curly braces or the set() function. Sets are useful in pharmacy data work for de-duplicating drug lists, comparing formularies, and quickly checking membership.

```
ward_a_drugs = {'Amoxicillin', 'Paracetamol', 'Omeprazole', 'Metformin'}
ward_b_drugs = {'Paracetamol', 'Metformin', 'Insulin', 'Losartan'}

print(ward_a_drugs & ward_b_drugs)   # intersection: drugs used on BOTH
wards
print(ward_a_drugs | ward_b_drugs)   # union: all distinct drugs across
both wards
print(ward_a_drugs - ward_b_drugs)   # difference: drugs unique to ward A
print('Insulin' in ward_b_drugs)     # fast membership test -> True
```

Sets do not preserve order and cannot contain duplicate values, which makes them unsuitable for sequential dosing logs but ideal for questions of overlap, uniqueness, and membership across drug lists.

3. Practical Operations on Structured Data

Beyond simply creating collections, pharmacy data scientists must repeatedly extract sub-sections of data (indexing and slicing), edit specific records in place (mutation), and clean or parse textual fields such as drug names, prescriber notes, or batch codes (string manipulation). This section develops each skill using pharmaceutical examples.

3.1 Indexing and Nested Indexing (Matrix-like Access)

Indexing retrieves a single element from a collection using its position, starting at 0. Negative indices count from the end of the collection. When lists are nested inside other lists — forming a matrix-like, row-and-column structure — elements are accessed by chaining indices.

```
batches = ['B101', 'B102', 'B103', 'B104', 'B105']

print(batches[0])       # 'B101'  – first element
print(batches[-1])      # 'B105'  – last element
print(batches[2])       # 'B103'  – third element

# Nested list representing a dose-response matrix:
# rows = drug concentrations (mg/L), columns = 3 replicate absorbance
readings
assay_matrix = [
    [0.102, 0.098, 0.101],   # concentration 1
    [0.215, 0.220, 0.209],   # concentration 2
    [0.398, 0.405, 0.392],   # concentration 3
]

print(assay_matrix[1])       # entire second row: [0.215, 0.220, 0.209]
print(assay_matrix[1][2])    # single cell: row 1, column 2 -> 0.209
print(assay_matrix[-1][0])   # last row, first column -> 0.398
```

⚙ Best Practice

Matrix-like nested lists are conceptually useful, but for real numerical work — statistics, linear algebra, broadcasting — NumPy arrays (Section 4) are far more efficient and expressive than nested Python lists.

3.2 Array/List Slicing Techniques

Slicing extracts a sub-sequence using the syntax start:stop:step, where the 'stop' index is excluded. Slicing is essential for isolating time windows in pharmacokinetic data, e.g., the first six hourly plasma-concentration readings.

```
plasma_conc = [0.0, 5.2, 9.8, 12.1, 10.4, 8.7, 6.9, 5.3, 4.0, 3.1]   #
ng/mL over 10 hours

print(plasma_conc[0:4])     # first 4 readings: hours 0-3
print(plasma_conc[3:7])     # readings for hours 3-6
print(plasma_conc[-3:])     # last 3 readings
print(plasma_conc[::2])     # every 2nd reading (even hours)
print(plasma_conc[::-1])    # reversed sequence

# Slicing the nested assay_matrix by row
first_two_concentrations = assay_matrix[0:2]
print(first_two_concentrations)
```

3.3 Data Element Mutation

Mutation refers to changing the contents of a mutable object (like a list or dictionary) in place, without creating a new object. This is critical when correcting a data-entry error, updating a patient's latest vitals, or replacing an out-of-range assay reading.

```
# Correcting a single value — direct index assignment
plasma_conc[0] = 0.5              # replace the erroneous baseline
reading
print(plasma_conc)

# Mutating a slice — replace multiple elements at once
plasma_conc[7:10] = [5.0, 3.8, 2.9]
print(plasma_conc)

# Mutating a nested (matrix-like) element
assay_matrix[2][1] = 0.410        # correct one cell of the assay matrix
print(assay_matrix)

# Mutating dictionary values (patient defined in Section 2.3)
patient['weight_kg'] += 1.5       # patient gained weight since last
visit
patient['medications'].append('Empagliflozin')   # add new medication to
the list
print(patient['medications'])
```

⚙ Common Pitfall

Because lists and dictionaries are mutable, assigning one variable to another (list_b = list_a) does NOT create a copy — both names point to the same object. Use list_a.copy() or list(list_a) to create an independent copy before mutating.

3.4 Advanced String Manipulation Methods

Pharmaceutical text data — drug names, batch codes, prescription instructions — often arrives inconsistently formatted. Python's string methods allow cleaning, parsing, and reformatting this text programmatically.

```
raw_entry = 'amoxicillin_500MG-tab #B2201'

clean = raw_entry.strip()                          # remove leading/trailing
whitespace
print(clean)                                    # 'amoxicillin_500MG-tab #B2201'

print(clean.upper())                            # 'AMOXICILLIN_500MG-TAB #B2201'
print(clean.lower())                            # 'amoxicillin_500mg-tab #b2201'
print(clean.title())                            # 'Amoxicillin_500Mg-Tab #B2201'

print(clean.replace('_', ' '))                  # 'amoxicillin 500MG-tab #B2201'
print(clean.split('-'))                            # ['amoxicillin_500MG', 'tab
#B2201']
print(clean.startswith('amoxicillin'))          # True
print(clean.find('500MG'))                      # index position of '500MG'

# Parsing a batch code into components
batch_code = 'B-2024-0456-EXP0627'
parts = batch_code.split('-')
print(parts)                        # ['B', '2024', '0456', 'EXP0627']
year, lot_number = parts[1], parts[2]
print(f'Manufactured year: {year}, Lot: {lot_number}')

# Formatted string construction (f-strings) for a dispensing label
drug, dose, freq = 'Metformin', 500, 'twice daily'
label = f'{drug} {dose} mg — take {freq} with food'
print(label)

# join() to rebuild a delimited string from a list
ingredients = ['Paracetamol', 'Caffeine', 'Chlorpheniramine']
print(', '.join(ingredients))    # 'Paracetamol, Caffeine, Chlorpheniramine'

# isnumeric / isalpha checks — useful for validating data entry fields
```

```
print('500'.isnumeric())          # True
print('MG'.isalpha())             # True
```

Method	Purpose	Pharmacy Example
.strip()	Remove leading/trailing whitespace	Cleaning imported CSV fields
.split(sep)	Break a string into a list	Parsing 'Drug-Strength-Form' codes
.join(list)	Combine list into one string	Building an ingredient list for a label
.replace(a,b)	Substitute characters/substrings	Standardizing units 'MG' → 'mg'
.format() / f-string	Insert variables into text	Generating prescription labels
.find() / .index()	Locate a substring position	Extracting strength from a drug code
.upper()/.lower()/.title()	Normalize case	Matching drug names across databases

✎ Hands-On Exercise: Indexing, Slicing, Mutation & Strings

1. Given hourly_glucose = [88, 92, 105, 130, 118, 99, 95, 90], slice out readings for hours 2 through 5 (inclusive).

2. Replace the 3rd element of hourly_glucose with the corrected value 128 using index assignment.

3. Given the string ' Ibuprofen_400mg-CAP ', clean it, standardize the case, and split it into ['Ibuprofen', '400mg', 'CAP'].

4. Multi-Dimensional Data Management with NumPy

4.1 Why NumPy for Pharmaceutical Computing

NumPy (Numerical Python) provides the ndarray object — a fast, memory-efficient, multi-dimensional array that supports vectorized mathematical operations. Unlike nested Python lists, NumPy arrays enforce a single data type, support element-wise arithmetic without explicit loops, and integrate with the broader scientific Python ecosystem (pandas, SciPy, matplotlib) used in pharmacokinetics, biostatistics, and computational chemistry.

```
import numpy as np

# A one-dimensional array of plasma drug concentrations (ng/mL)
conc = np.array([0.0, 5.2, 9.8, 12.1, 10.4, 8.7, 6.9])
print(conc)
print(type(conc))        # <class 'numpy.ndarray'>
print(conc.dtype)        # float64
print(conc.shape)        # (7,)  — 7 elements, one dimension
```

4.2 Creating and Inspecting NumPy Arrays

NumPy offers many array-construction functions beyond converting a list, including arrays of zeros, ones, evenly spaced values, and random samples — useful for simulations, placeholders, and generating time points for dosing schedules.

```
# Two-dimensional array: rows = patients, columns = weekly weight (kg)
readings
weights = np.array([
    [70.2, 70.5, 70.1, 69.8],
    [58.4, 58.6, 58.9, 59.0],
    [82.1, 81.7, 81.5, 81.0],
])
print(weights.shape)     # (3, 4) -> 3 patients, 4 weekly readings
print(weights.ndim)      # 2  (two dimensions: rows and columns)
print(weights.size)      # 12 total elements

# Convenience constructors
zeros_arr = np.zeros((3, 4))          # placeholder array of zeros, shape
(3,4)
ones_arr  = np.ones(5)                # array of ones, shape (5,)
dose_times = np.arange(0, 24, 4)      # [0, 4, 8, 12, 16, 20] -> sampling
times (hr)
```

```
linspace_times = np.linspace(0, 24, 7)  # 7 evenly spaced points from 0
to 24 hr
print(dose_times)
print(linspace_times)

# Reshaping a flat array into a matrix (e.g., 12 assay readings -> 3 x 4
plate layout)
readings = np.arange(12)
plate = readings.reshape(3, 4)
print(plate)
```

4.3 Indexing and Slicing N-Dimensional Arrays

NumPy extends indexing and slicing to multiple dimensions using a single comma-separated syntax [row, column], which is more concise and far faster than chained indexing on nested Python lists.

```
print(weights[0, :])       # all weekly readings for patient 0 (row 0)
print(weights[:, 1])       # week-2 reading for ALL patients (column 1)
print(weights[1, 2])       # single value: patient 1, week 3
print(weights[0:2, 1:3])   # sub-matrix: patients 0-1, weeks 2-3

# Boolean (conditional) indexing – extremely powerful for filtering
clinical data
high_weight_mask = weights[:, -1] > 70      # patients whose final
reading exceeds 70 kg
print(high_weight_mask)                      # array([ True, False,
True])
print(weights[high_weight_mask])             # rows meeting the
condition

# Fancy indexing – select specific rows by position
selected_patients = weights[[0, 2]]          # patients 0 and 2 only
print(selected_patients)
```

> ⚙ **Common Pitfall**
> *Unlike Python list slices, NumPy array slices are VIEWS of the original data, not copies. Modifying a slice will modify the original array. Use .copy() explicitly when an independent copy is required — for example before running an experimental correction on a subset of assay data.*

4.4 Combining and Reshaping Arrays

Pharmacy datasets are frequently assembled from multiple smaller arrays — for example, stacking individual patient dose vectors into one cohort matrix, or splitting a combined dataset back into per-patient records. NumPy's stacking and reshaping functions support both directions.

```
patient_a = np.array([120, 122, 118, 121])   # 4 weekly systolic readings
patient_b = np.array([135, 138, 130, 128])
patient_c = np.array([142, 140, 139, 137])

cohort_matrix = np.vstack([patient_a, patient_b, patient_c])   # stack as
rows
print(cohort_matrix.shape)     # (3, 4)

combined_wide = np.hstack([patient_a, patient_b])               #
concatenate end-to-end
print(combined_wide.shape)     # (8,)

# Splitting a matrix back into individual patient rows
rows = np.vsplit(cohort_matrix, 3)
for r in rows:
    print(r.flatten())

# Transposing swaps rows and columns — useful for switching between
# 'patients as rows' and 'time points as rows' layouts
print(cohort_matrix.T.shape)   # (4, 3)
```

✎ Hands-On Exercise: Building & Indexing NumPy Arrays

1. Create a 4x3 NumPy array 'doses' representing 4 patients' morning, afternoon, and evening insulin doses (units), then print its shape and ndim.
2. Extract the afternoon dose column for all patients using slicing.
3. Use boolean indexing to select all patients whose morning dose exceeds 10 units.

5. Mathematical and Vector Operations on Arrays

5.1 Element-wise Arithmetic and Broadcasting

NumPy performs arithmetic element-by-element across an entire array without explicit loops — a feature called vectorization. Broadcasting allows operations between arrays of different but compatible shapes, such as adding a single correction value to every element of a matrix.

```
conc_ngml = np.array([0.0, 5.2, 9.8, 12.1, 10.4, 8.7, 6.9])

conc_ugml = conc_ngml / 1000              # unit conversion: ng/mL ->
ug/mL (vectorized)
print(conc_ugml)

dose_adjustment = conc_ngml * 1.15         # apply a 15% bioavailability
correction to ALL points
print(dose_adjustment)

baseline_corrected = conc_ngml - conc_ngml[0]   # subtract baseline from
every reading
print(baseline_corrected)

# Broadcasting: subtract a per-patient baseline (column vector) from a 2D
matrix
baseline = weights[:, 0].reshape(-1, 1)     # shape (3,1)
weight_change = weights - baseline           # broadcast across all 4
columns
print(weight_change)
```

5.2 Vector and Matrix Operations (Dot Products, Norms)

Vector operations underpin many pharmacometric calculations, such as computing total drug exposure (area under the curve approximations), weighted dose scores, and similarity between dosing regimens.

```
time_hr   = np.array([0, 1, 2, 4, 6, 8, 12])
conc_vals = np.array([0.0, 5.2, 9.8, 12.1, 10.4, 8.7, 5.9])

# Trapezoidal-style AUC approximation using vectorized operations
dt = np.diff(time_hr)                          # time intervals between
samples
avg_conc = (conc_vals[:-1] + conc_vals[1:]) / 2 # average concentration
per interval
auc_segments = dt * avg_conc                    # area of each trapezoid
```

```
auc_total = np.sum(auc_segments)                # total AUC (ng*hr/mL)
print('Total AUC:', round(auc_total, 2))

# Dot product — weighted dosing score across 3 active ingredients
ingredient_amounts = np.array([250, 125, 50])       # mg of each
ingredient
potency_weights    = np.array([1.0, 0.8, 1.5])      # relative potency
factor
weighted_score = np.dot(ingredient_amounts, potency_weights)
print('Weighted potency score:', weighted_score)

# Vector norm — magnitude of a multi-analyte residual vector (e.g., QC
deviation)
target = np.array([100, 100, 100])
measured = np.array([98.5, 101.2, 99.0])
deviation_norm = np.linalg.norm(measured - target)
print('QC deviation magnitude:', round(deviation_norm, 3))

# Matrix multiplication — converting ingredient masses to molar
quantities
masses_mg = np.array([[500, 250], [325, 162.5]])     # 2 batches x 2
ingredients (mg)
molar_mass = np.array([180.16, 151.16])               # g/mol for each
ingredient
moles = masses_mg / 1000 / molar_mass                 # element-wise,
broadcasting over columns
print(moles)
```

5.3 Statistical and Aggregate Functions

NumPy provides built-in aggregate functions that are indispensable for quality control, descriptive statistics of clinical trial data, and summarizing assay replicates.

```
assay = np.array([98.2, 101.5, 99.8, 100.4, 97.9, 102.1])   # % label
claim, 6 replicates

print('Mean:', np.mean(assay))
print('Median:', np.median(assay))
print('Std Dev:', round(np.std(assay, ddof=1), 3))   # sample standard
deviation
print('Variance:', round(np.var(assay, ddof=1), 3))
print('Min / Max:', np.min(assay), '/', np.max(assay))
print('Range:', np.ptp(assay))
```

```
# Row-wise and column-wise aggregation on the 2D 'weights' matrix
print('Mean weight per patient (row-wise):', np.mean(weights, axis=1))
print('Mean weight per week (column-wise):', np.mean(weights, axis=0))

# Percentage relative standard deviation (%RSD) — a key QC acceptance
metric
rsd_percent = (np.std(assay, ddof=1) / np.mean(assay)) * 100
print(f'%RSD: {rsd_percent:.2f}%')
```

Function	Description	Typical Use
np.mean() / np.median()	Central tendency	Average assay result
np.std() / np.var()	Spread of data	QC precision (%RSD)
np.sum() / np.cumsum()	Total / running total	Cumulative drug exposure
np.min() / np.max()	Extremes	Peak plasma concentration (Cmax)
np.dot() / @	Dot / matrix product	Weighted scores, dose vectors
np.linalg.norm()	Vector magnitude	Multi-analyte deviation
np.diff()	Successive differences	Time intervals, rate of change

✎ Hands-On Exercise: Vector & Statistical Operations

1. Given hplc_areas = np.array([45210, 45890, 44950, 46102, 45500]), compute the mean, standard deviation, and %RSD. Is %RSD below the common 2% acceptance limit?

2. Compute the AUC (0-8h) for time_hr = [0,2,4,6,8] and conc_vals = [0,15,22,18,10] using the trapezoidal approach shown in Section 5.2.

3. Using the 'weights' 2D array, compute each patient's total weight change from week 1 to week 4 using vectorized subtraction.

6. Dataset Alignment and Structured Arrays

6.1 Aligning Multi-Source Pharmacy Datasets

Real pharmacy data rarely arrives as a single clean array — dispensing logs, laboratory results, and vital-sign records are often collected separately and must be aligned by a shared identifier (e.g., patient ID) before joint analysis. Dictionaries provide a natural mechanism for this alignment when working in pure Python/NumPy without pandas.

```
# Source 1: dispensing records keyed by patient ID
dispensing = {
    'PT-101': {'drug': 'Metformin', 'dose_mg': 500},
    'PT-102': {'drug': 'Atorvastatin', 'dose_mg': 20},
    'PT-103': {'drug': 'Losartan', 'dose_mg': 50},
}

# Source 2: laboratory results keyed by the SAME patient ID
lab_results = {
    'PT-101': {'hba1c': 7.2, 'creatinine': 0.9},
    'PT-102': {'hba1c': 5.9, 'creatinine': 1.0},
    'PT-103': {'hba1c': 6.1, 'creatinine': 1.3},
}

# Aligning both sources into one unified record per patient
unified = {}
for pid in dispensing:                      # iterate using the shared
key
    if pid in lab_results:                   # only align records present
in BOTH sources
        unified[pid] = {**dispensing[pid], **lab_results[pid]}

for pid, record in unified.items():
    print(pid, '->', record)
```

This shared-key alignment pattern — matching records via a common identifier and merging their fields — is the same underlying logic used by database JOIN operations and by pandas.merge() in more advanced tooling. Understanding it at the dictionary level builds essential intuition before moving to larger pharmacoinformatics platforms.

6.2 Structured Arrays for Biological Tracking

NumPy's structured arrays (record arrays) allow each 'row' to contain multiple named fields of different data types — combining the labeled clarity of a dictionary with the numerical performance of NumPy.

This is ideal for tracking longitudinal biological/pharmacological variables such as patient vitals, drug levels, and lab markers in one compact, typed array.

```
import numpy as np

# Define the structure: field name + data type for each column
dtype = np.dtype([
    ('patient_id', 'U10'),      # unicode string, max 10 characters
    ('age', 'i4'),              # 32-bit integer
    ('weight_kg', 'f4'),        # 32-bit float
    ('systolic_bp', 'i4'),
    ('glucose_mgdl', 'f4'),
])

# Initialize a structured array for biological tracking (5 patients)
cohort = np.zeros(5, dtype=dtype)

cohort[0] = ('PT-101', 58, 72.5, 138, 145.2)
cohort[1] = ('PT-102', 47, 65.0, 122, 98.4)
cohort[2] = ('PT-103', 63, 80.2, 150, 175.6)
cohort[3] = ('PT-104', 39, 58.9, 118, 89.1)
cohort[4] = ('PT-105', 71, 68.4, 160, 210.3)

print(cohort)
print(cohort.dtype.names)               #
('patient_id','age','weight_kg', ...)

# Field-wise access behaves like a labeled column
print(cohort['glucose_mgdl'])           # all glucose values as one
array
print(np.mean(cohort['systolic_bp']))   # mean systolic BP across the
cohort

# Boolean filtering on a structured array — flag patients for follow-up
at_risk = cohort[cohort['glucose_mgdl'] > 150]
print(at_risk['patient_id'])            # array(['PT-103', 'PT-105'])

# Mutating a specific field for a specific record (e.g., updated visit
data)
cohort[2]['weight_kg'] = 79.6
print(cohort[2])
```

⚙ Going Further

Structured arrays keep each field's data type explicit and memory-compact, which matters when tracking thousands of patients or high-frequency biosensor data. For most day-to-day pharmacy analytics, the pandas DataFrame (built on top of NumPy) offers a friendlier interface — but understanding structured arrays reveals what a DataFrame is doing internally.

✎ Hands-On Exercise: Alignment & Structured Arrays

1. Extend the 'dispensing' and 'lab_results' dictionaries with a fourth patient 'PT-104' present in only one source, then modify the alignment loop to report patients missing lab data instead of silently skipping them.

2. Add two more fields, 'sex' ('U1') and 'on_insulin' (bool), to the cohort dtype and repopulate the structured array with appropriate values.

3. Using the cohort structured array, compute the average weight of patients under age 60, using boolean indexing combined with field access.

7. Capstone Hands-On Laboratory: Patient Cohort Dashboard

This capstone exercise integrates every concept covered in this lecture note — lists, tuples, dictionaries, indexing, slicing, mutation, string parsing, and NumPy vector/statistical operations — into a single mini pharmacy-analytics workflow.

7.1 Scenario

A community pharmacy is monitoring five patients enrolled in a hypertension management program. Blood pressure is recorded weekly for four weeks, alongside each patient's prescribed medication. The pharmacy team wants a small analytics script that (1) organizes the raw data, (2) cleans the drug-name text, (3) computes each patient's average and trend in blood pressure, and (4) flags patients whose most recent reading remains above target (systolic > 140 mmHg).

7.2 Guided Solution Outline

```
import numpy as np

# Step 1: Raw data organized with a dictionary of patient records (tuples
+ lists)
raw_patients = {
    'PT-201': {'drug_raw': '  losartan_50MG-tab ', 'bp': [148, 144, 141,
139]},
    'PT-202': {'drug_raw': ' Amlodipine_5MG-TAB', 'bp': [152, 150, 147,
145]},
    'PT-203': {'drug_raw': 'hydrochlorothiazide_25mg-Tab ', 'bp': [136,
134, 130, 128]},
    'PT-204': {'drug_raw': ' Enalapril_10MG-tab ', 'bp': [160, 155, 150,
143]},
    'PT-205': {'drug_raw': 'Telmisartan_40mg-TAB', 'bp': [130, 128, 126,
124]},
}

# Step 2: Clean drug names using string methods
for pid, rec in raw_patients.items():
    clean_name = rec['drug_raw'].strip().split('_')[0].title()
    rec['drug_clean'] = clean_name

# Step 3: Build a NumPy structured array for the BP tracking cohort
dtype = np.dtype([('patient_id', 'U10'), ('drug', 'U20'),
                  ('week1', 'i4'), ('week2', 'i4'), ('week3', 'i4'),
('week4', 'i4')])
cohort = np.zeros(len(raw_patients), dtype=dtype)
```

```
for i, (pid, rec) in enumerate(raw_patients.items()):
    w1, w2, w3, w4 = rec['bp']
    cohort[i] = (pid, rec['drug_clean'], w1, w2, w3, w4)

# Step 4: Vectorized analytics
bp_matrix = np.array([list(row)[2:] for row in cohort])   # numeric 5x4
sub-matrix
avg_bp = np.mean(bp_matrix, axis=1)
trend = bp_matrix[:, -1] - bp_matrix[:, 0]                 # negative =
improving
flagged = cohort['week4'] > 140

for i in range(len(cohort)):
    status = 'FOLLOW-UP NEEDED' if flagged[i] else 'On Target'
    print(f"{cohort['patient_id'][i]} | {cohort['drug'][i]:<15} "
          f"| Avg BP: {avg_bp[i]:.1f} | Trend: {trend[i]:+d} | {status}")
```

7.3 Expected Learning Outcome

Learners should observe that patients PT-202 and PT-204 remain above the 140 mmHg systolic target at week 4 and are flagged for follow-up, while the remaining three patients show a downward (improving) trend and are within target. This mirrors the real workflow of combining record cleaning, structured storage, and vectorized computation used in pharmacy data analytics and clinical decision-support tools.

✎ Hands-On Exercise: Capstone Extension

1. Modify the script to also compute the standard deviation of each patient's four weekly readings using np.std, and add it as a printed column.

2. Add a sixth patient with missing week-3 data (use np.nan) and adjust the average calculation using np.nanmean so the missing value does not distort the result.

3. Sort and print the patient cohort by average blood pressure, highest to lowest, using np.argsort on the avg_bp array.

8. Summary, Best Practices & Further Reading

8.1 Key Takeaways

- Lists are ordered and mutable — best for growing/changing sequences such as dosing logs.
- Tuples are ordered and immutable — best for fixed records such as (drug, strength, form).
- Dictionaries map keys to values — best for labeled records such as patient or drug profiles.
- Indexing retrieves single elements; slicing retrieves ranges; both use zero-based positions.
- Mutation changes data in place — always be mindful of shared references versus independent copies.
- String methods (.strip, .split, .join, .replace, f-strings) are essential for cleaning pharmaceutical text data.
- NumPy arrays enable fast, vectorized mathematics across large numeric datasets without explicit loops.
- Broadcasting, dot products, and aggregate functions (mean, std, sum) underlie most pharmacometric and QC calculations.
- Dictionaries (by shared key) and structured arrays (by named field) both provide practical tools for aligning and organizing multi-source biological data.

8.2 Best Practices Checklist

1. Always inspect data shape and type (len(), type(), .shape, .dtype) immediately after loading or constructing a structure.
2. Prefer tuples for data that should never change accidentally (e.g., reference ranges, fixed identifiers).
3. Use .copy() explicitly whenever an independent duplicate of a list, dictionary, or array is required.
4. Validate and clean text fields (drug names, batch codes) with string methods before using them as dictionary keys or join fields.
5. Prefer vectorized NumPy operations over manual Python loops for any calculation across more than a few dozen numeric values.
6. Document units explicitly in variable names (e.g., conc_ngml, weight_kg) to prevent dosing and conversion errors.
7. When aligning multiple datasets, always check for missing or mismatched identifiers rather than assuming a perfect match.

8.3 Suggested Further Reading

- Official NumPy User Guide — numpy.org/doc/stable/user/

- Python Software Foundation — official documentation on Data Structures (docs.python.org/3/tutorial/datastructures.html)
- McKinney, W. — Python for Data Analysis (O'Reilly) — for the transition from NumPy to pandas DataFrames.
- Rowland, M. & Tozer, T.N. — Clinical Pharmacokinetics and Pharmacodynamics — for the underlying pharmacometric concepts used in the AUC and dosing examples.
- SciPy Lecture Notes — scipy-lectures.org — for further scientific computing techniques beyond this module.

⚙ Closing Note

This lecture note is intended as a hands-on companion — learners are strongly encouraged to type and execute every code example rather than reading passively, and to complete each exercise box before proceeding to the next section.

Appendix: Quick Reference Cheat Sheet

The tables below summarize the most frequently used operations covered in this lecture note for rapid lookup during laboratory sessions and assignments.

A.1 Structured Object Operations

Operation	List	Tuple	Dictionary
Create	my_list = []	my_tuple = ()	my_dict = {}
Add item	.append(x) / .insert(i,x)	not supported	d[key] = value
Remove item	.remove(x) / .pop(i)	not supported	del d[key] / .pop(key)
Access	my_list[i]	my_tuple[i]	d[key] / d.get(key)
Length	len(my_list)	len(my_tuple)	len(d)
Membership test	x in my_list	x in my_tuple	key in d
Iterate	for x in my_list	for x in my_tuple	for k, v in d.items()

A.2 Indexing & Slicing Syntax

Syntax	Meaning
seq[i]	Element at position i (0-based)
seq[-1]	Last element
seq[a:b]	Elements from index a up to (not including) b
seq[a:b:s]	Elements from a to b, stepping by s
seq[::-1]	Reversed sequence
arr[r, c]	NumPy 2D element at row r, column c
arr[r1:r2, c1:c2]	NumPy 2D sub-matrix slice
arr[mask]	Boolean-indexed selection (condition-based filtering)

A.3 Core NumPy Functions

Category	Functions
Creation	np.array(), np.zeros(), np.ones(), np.arange(), np.linspace()

Category	Functions
Shape / Inspect	.shape, .ndim, .size, .dtype, .reshape()
Combine / Split	np.vstack(), np.hstack(), np.vsplit(), .T
Arithmetic	+, -, *, /, ** (all element-wise / broadcasted)
Linear algebra	np.dot(), @, np.linalg.norm()
Statistics	np.mean(), np.median(), np.std(), np.var(), np.sum()
Filtering	Boolean masks: arr[arr > value]
Structured data	np.dtype([(name, type), ...]), np.zeros(n, dtype=dtype)

A.4 String Methods Quick Reference

Method	Effect
.strip() / .lstrip() / .rstrip()	Remove surrounding whitespace
.upper() / .lower() / .title()	Change letter case
.split(sep) / .join(list)	Break apart / reassemble text
.replace(old, new)	Substitute text
.find(sub) / .index(sub)	Locate a substring
.startswith() / .endswith()	Prefix / suffix test
f'{var}'	Embed variables directly into text

> **⚙ Study Tip**
> *Keep this appendix open in a second window during laboratory sessions; nearly every hands-on exercise in this lecture note can be completed using only the operations summarized here.*