LABORATORY MANUAL

PYTHON PROGRAMMING - I

BCA | First Semester

Unit-wise practicals • Worked examples • Test cases • Industry-oriented tasks

College	RVS College of engineering and technology, Jamshedpur
Department	BCA
Academic Session	2026-2027
Faculty	Prof. Deepak Kumar Tiwari

How to Use This Manual

This manual is designed for beginners who are learning to write, run, explain, test, and improve Python programs. Each practical contains an aim, concepts, a step-by-step algorithm, an executable reference program, an expected run, test cases, an industry connection, and an extension task**. Students should type and execute the code themselves rather than only copy it.**

Course outcomes

Code	Course outcome	Evidence in this manual
CO1	Demonstrate the use of built-in data types of Python.	Practicals 1-9: primitive types, lists, tuples, sets, dictionaries, and comprehensions.
CO2	Demonstrate significant experience with the Python program development environment.	All practicals; environment setup, execution, debugging, input validation, and test cases.
CO3	Design functions and custom modules for a given requirement.	Practicals 10, 14-18 and the integrated mini-projects.

Unit-wise practical map

Syllabus unit	Main concepts	Practicals
I - Introducing Python	Interpreter, identifiers, basic data types, operators, decision/looping, console I/O	1-2
II - Lists and Tuples	Indexing, slicing, operations, methods, immutability, packing/unpacking	3-4
III - Sets and Dictionaries	Set operations, mathematical sets, dictionary CRUD and methods	5-6
IV - Comprehensions and Functions	List/set/dictionary comprehensions, function arguments, unpacking, recursion	7-9
V - Functional Programming	Lambda, higher-order functions, map, filter, reduce	10-13
VI - Modules, Packages and Namespaces	Built-in/custom modules, imports, packages, scope, globals/locals, inner functions	14-18

Numbering note: The supplied lab list appears to have Units IV and VI interchanged for comprehensions/recursion and custom modules. This manual maps each exercise to the unit where the topic actually appears in the supplied teaching syllabus.

Recommended lab environment

- Install a supported Python 3 release from https://www.python.org/downloads/ or use the Python installation approved by the college.
- Use IDLE, Visual Studio Code, or another editor available in the lab. Save programs with the .py extension.
- Run a file from the terminal with: python filename.py (on some systems, use python3 filename.py).
- Check the interpreter with: python --version. Do not name your file after a standard module, for example random.py, statistics.py, or math.py.
- For the module practicals, keep files in the folder structure shown in the exercise and run the main script from the project root.

Professional coding checklist

- Use meaningful snake_case names, consistent four-space indentation, and short, readable functions.
- Validate external input and handle expected invalid values; do not silently accept impossible data.
- Use f-strings for readable output and docstrings for functions that have a clear purpose.
- Test normal, boundary, empty, and invalid inputs. Compare actual output with expected output.
- Avoid unexplained global state. Prefer parameters and return values so functions are easy to test and reuse.
- Keep examples dependency-free unless a task explicitly asks for an external package.

PRACTICAL 01 | UNIT I

Python Interpreter, Basic Data Types and Console I/O

Course outcome: CO1, CO2

Aim: Write and execute a Python program using identifiers, basic data types, input(), output(), type(), and type conversion.

Concepts used: Interpreter, variables, str, int, float, bool, None, identifiers, input(), print(), f-strings, int() and float().

Algorithm / steps

1. Read a student name, age, and percentage from the console.
2. Convert numeric input strings to int and float.
3. Store a Boolean status and demonstrate type inspection.
4. Display the values using readable formatted output.

Program

Save as practical_01_basics.py

```
student_name = input("Enter student name: ").strip()
age = int(input("Enter age in years: "))
percentage = float(input("Enter percentage: "))
is_registered = True

print("\n--- Student Profile ---")
print(f"Name: {student_name}")
print(f"Age: {age} years")
print(f"Percentage: {percentage:.2f}%")
print(f"Registered: {is_registered}")
print("Types:", type(student_name), type(age), type(percentage))
print("Registration type:", type(is_registered))
```

Sample run / expected result

```
Enter student name: Asha Kumar
Enter age in years: 18
Enter percentage: 82.5

--- Student Profile ---
Name: Asha Kumar
Age: 18 years
Percentage: 82.50%
Registered: True
Types: <class 'str'> <class 'int'> <class 'float'>
Registration type: <class 'bool'>
```

Test cases to verify

Test / input	Expected behaviour
Age = 18, percentage = 82.5	Values are converted to int and float and displayed.
Percentage = 82	Accepted and displayed as 82.00%.
Age entered as abc	Raises ValueError; note this as an input-validation improvement for the next version.

Industry connection: Console input and formatted output are used in command-line utilities, scripts, and quick internal tools. Correct type conversion prevents common data-processing errors.
Extension task: Add a loop and try/except so the program asks again when age or percentage is not numeric; reject negative age or percentage outside 0-100.
Viva questions: What type does input() return? What is the difference between int and float? Why are identifiers case-sensitive?

PRACTICAL 02 | UNIT I

Operators, Precedence, Decisions and Loops

Course outcome: CO1, CO2

Aim: Use arithmetic, comparison, logical operators, decision control, and looping to build a validated marks-to-grade utility.

Concepts used: Arithmetic operators, comparison operators, and/or/not, precedence, while loop, if/elif/else, try/except, break.

Algorithm / steps

1. Repeatedly request a mark until an integer from 0 to 100 is entered.
2. Use comparison operators to select a grade.
3. Print a clear result and test boundary marks.

Program

Save as practical_02_grade.py

```
while True:
    try:
        marks = int(input("Enter marks (0-100): "))
    except ValueError:
        print("Please enter a whole number.")
        continue

    if 0 <= marks <= 100:
        break
    print("Marks must be between 0 and 100.")

if marks >= 90:
    grade = "A+"
elif marks >= 80:
    grade = "A"
elif marks >= 70:
    grade = "B"
elif marks >= 60:
    grade = "C"
elif marks >= 40:
    grade = "D"
else:
    grade = "F"

print(f"Marks: {marks}; Grade: {grade}")
```

Sample run / expected result

Enter marks (0-100): 82
Marks: 82; Grade: A

Test cases to verify

Test / input	Expected behaviour
0	Grade F.
40	Grade D.
90	Grade A+.

101	Rejected and asked again.
hello	Rejected with a validation message; program continues.

Industry connection: Input validation, business rules, and boundary checks are core to forms, billing systems, admission tools, and data-entry applications.
Extension task: Add a pass/fail message and calculate eligibility using marks >= 40 and attendance >= 75. Use and/or only after defining the exact rule.
Viva questions: Why is a while loop used here? What is the difference between = and ==? What does continue do?

PRACTICAL 03 | UNIT II

Lists: Indexing, Slicing and Built-in Methods

Course outcome: CO1, CO2

Aim: Create, inspect, modify, and sort a list while applying common list methods.

Concepts used: Mutable sequences, zero-based indexing, negative indexing, slicing, append, extend, insert, remove, pop, sort, count, index.

Algorithm / steps

1. Create a list of daily sales values.
2. Add a new value and insert a correction.
3. Remove a known value, sort the list, and compute basic summaries.
4. Check for an empty list before calculating an average.

Program

Save as practical_03_lists.py

```
sales = [1250.0, 980.0, 1430.0, 1100.0]
sales.append(1575.0)
sales.insert(1, 1050.0)
sales.remove(980.0)

print("First sale:", sales[0])
print("Last two:", sales[-2:])
print("Count:", len(sales))
print("Sorted:", sorted(sales))
print("Highest:", max(sales))
print("Lowest:", min(sales))
print("Total:", sum(sales))
print("Average:", round(sum(sales) / len(sales), 2) if sales else 0)
```

Sample run / expected result

```
First sale: 1250.0
Last two: [1100.0, 1575.0]
Count: 5
Sorted: [1050.0, 1100.0, 1250.0, 1430.0, 1575.0]
Highest: 1575.0
Lowest: 1050.0
Total: 6405.0
Average: 1281.0
```

Test cases to verify

Test / input	Expected behaviour
Empty list	Do not divide by zero; return or report zero/NA based on the application rule.
Remove a missing value with remove()	A ValueError occurs; use membership checking or handle the exception.
sales[0]	Returns the first element when the list is non-empty.

Industry connection: Lists are used to hold ordered collections such as transaction amounts, filenames, form inputs, and readings before processing.

Extension task: Write a menu-driven to-do list that supports add, show, complete/remove, and exit. Reject blank task names.
Viva questions: Are lists mutable? What does slicing list[-2:] mean? Difference between append() and extend()?

PRACTICAL 04 | UNIT II

Tuples: Packing, Unpacking and Immutable Records

Course outcome: CO1, CO2

Aim: Use tuples to group related values, unpack them into variables, and access tuple methods.

Concepts used: Tuple creation, one-item tuple comma, indexing, slicing, packing, unpacking, count(), index(), immutability.

Algorithm / steps

1. Represent a product record as a tuple.
2. Unpack record fields into separate variables.
3. Access elements using indexes and slices.
4. Use count() and index() to inspect repeated values.

Program

Save as practical_04_tuples.py

```
product = ("KB-104", "Wireless Keyboard", 1299.00, 12)
product_code, product_name, price, stock = product

print("Code:", product_code)
print("Product:", product_name)
print("Price:", price)
print("Stock:", stock)
print("First two fields:", product[:2])

ratings = (4, 5, 4, 3, 5, 4)
print("Number of 4-star ratings:", ratings.count(4))
print("First 5-star index:", ratings.index(5))

single_value_tuple = (10,)
print("Single tuple:", single_value_tuple, type(single_value_tuple))
```

Sample run / expected result

```
Code: KB-104
Product: Wireless Keyboard
Price: 1299.0
Stock: 12
First two fields: ('KB-104', 'Wireless Keyboard')
Number of 4-star ratings: 3
First 5-star index: 1
Single tuple: (10,) <class 'tuple'>
```

Test cases to verify

Test / input	Expected behaviour
Tuple with one item: (10,)	It is a tuple because the comma is present.
Attempt product[2] = 1500	TypeError: tuples do not support item assignment.
Unpack four values into three variables	ValueError due to a mismatch in item counts.

Industry connection: Tuples suit fixed-size records and return values where order is known and accidental item replacement should be avoided. For more complex records, a dataclass or named type can be introduced in a later course.

Extension task: Create a tuple of latitude and longitude, unpack it, then print a formatted coordinate. Explain why the coordinate pair should not be changed in-place.
Viva questions: Why are tuples considered immutable? How do you make a one-element tuple? What is tuple unpacking?

PRACTICAL 05 | UNIT III

Sets and Mathematical Set Operations

Course outcome: CO1, CO2

Aim: Remove duplicates and compare collections using union, intersection, difference, and symmetric difference.

Concepts used: Unique set elements, add, update, discard, remove, union, intersection, difference, symmetric_difference, subset.

Algorithm / steps

1. Represent students enrolled in two workshops as sets.
2. Find students in either workshop and students attending both.
3. Find participants unique to each workshop.
4. Use subset/superset checks where useful.

Program

Save as practical_05_sets.py

```
python_lab = {"Asha", "Ravi", "Meena", "Kabir"}
ai_lab = {"Meena", "Kabir", "Fatima", "Ravi"}

print("Python attendees:", sorted(python_lab))
print("AI attendees:", sorted(ai_lab))
print("Either workshop:", sorted(python_lab | ai_lab))
print("Both workshops:", sorted(python_lab & ai_lab))
print("Only Python:", sorted(python_lab - ai_lab))
print("Only one workshop:", sorted(python_lab ^ ai_lab))

python_lab.add("Ishaan")
python_lab.discard("Unknown Student")  # safe if absent
print("Updated count:", len(python_lab))
```

Sample run / expected result

```
Python attendees: ['Asha', 'Kabir', 'Meena', 'Ravi']
AI attendees: ['Fatima', 'Kabir', 'Meena', 'Ravi']
Either workshop: ['Asha', 'Fatima', 'Kabir', 'Meena', 'Ravi']
Both workshops: ['Kabir', 'Meena', 'Ravi']
Only Python: ['Asha']
Only one workshop: ['Asha', 'Fatima']
Updated count: 5
```

Test cases to verify

Test / input	Expected behaviour
Duplicate values in a set literal	Stored once; sets contain unique elements.
discard() for a missing value	No error is raised.
A = {1, 2}, B = {1, 2, 3}	A is a subset of B; B is a superset of A.

Industry connection: Sets are useful for deduplication, permission comparisons, finding missing IDs, and comparing membership lists from different systems.

Extension task: Given two lists of student IDs, report duplicates in each list and the IDs present in one source but not the other.

Viva questions: Are sets ordered? What is the difference between remove() and discard()? Explain intersection in plain language.

PRACTICAL 06 | UNIT III

Dictionaries: CRUD Operations and Methods

Course outcome: CO1, CO2

Aim: Create and update key-value mappings and safely retrieve values using dictionary methods.

Concepts used: Dictionary literals, keys and values, lookup, get(), update(), pop(), items(), keys(), values(), membership.

Algorithm / steps

1. Create an inventory dictionary with item codes and stock counts.
2. Safely retrieve a key using get().
3. Update stock, add an item, and remove an item.
4. Print the final inventory and total number of units.

Program

Save as practical_06_dictionary.py

```
stock = {"PEN-01": 120, "NOTE-02": 75, "FILE-03": 40}

print("Stock of PEN-01:", stock.get("PEN-01", 0))
print("Stock of missing item:", stock.get("MUG-99", 0))

stock["PEN-01"] -= 10
stock["MUG-99"] = 15
removed_stock = stock.pop("FILE-03", 0)

print("\nCurrent inventory")
for item_code, quantity in sorted(stock.items()):
    print(f"{item_code}: {quantity}")
print("Removed item quantity:", removed_stock)
print("Total units:", sum(stock.values()))
```

Sample run / expected result

```
Stock of PEN-01: 120
Stock of missing item: 0

Current inventory
MUG-99: 15
NOTE-02: 75
PEN-01: 110
Removed item quantity: 40
Total units: 200
```

Test cases to verify

Test / input	Expected behaviour
Lookup absent key with stock["MISSING"]	Raises KeyError; use get() when absence is expected.
pop("FILE-03", 0)	Removes the key if present, otherwise returns 0.
Check "PEN-01" in stock	Returns True when the key exists.

Industry connection: Dictionaries underpin JSON-like records, configuration, counters, product catalogues, API responses, and in-memory indexes.

Extension task: Build a student directory keyed by roll number. Support add, search, update phone/email, and delete; do not store duplicate roll numbers.
Viva questions: What is a key-value pair? Why must dictionary keys be hashable? Difference between d[key] and d.get(key)?

PRACTICAL 07 | UNIT IV

List, Set and Dictionary Comprehensions

Course outcome: CO1, CO2

Aim: Transform and filter data in concise expressions while preserving readability.

Concepts used: Iterable, expression, filter condition, list comprehension, set comprehension, dictionary comprehension.

Algorithm / steps

1. Create a list of squares for even numbers from 1 to 10.
2. Create a set of unique word lengths.
3. Create a dictionary mapping each word to its length.
4. Compare these expressions with a normal loop.

Program

Save as practical_07_comprehensions.py

```
numbers = range(1, 11)
even_squares = [n * n for n in numbers if n % 2 == 0]

words = ["python", "data", "python", "logic", "data"]
unique_lengths = {len(word) for word in words}
word_lengths = {word: len(word) for word in words}

print("Squares of even numbers:", even_squares)
print("Unique word lengths:", sorted(unique_lengths))
print("Word lengths:", word_lengths)
```

Sample run / expected result

```
Squares of even numbers: [4, 16, 36, 64, 100]
Unique word lengths: [4, 5, 6]
Word lengths: {'python': 6, 'data': 4, 'logic': 5}
```

Test cases to verify

Test / input	Expected behaviour
No values meet the condition	The resulting comprehension is an empty list or set.
Duplicate words in set comprehension	Repeated equal values collapse to one set element.
Duplicate dictionary keys from repeated words	The later value replaces the earlier value; here lengths are identical.

Industry connection: Comprehensions are common in data preparation and scripts, but long nested expressions reduce readability. Prefer a loop for complex transformations.

Extension task: From a list of product dictionaries, create a list of names in stock and a dictionary of product code to price. Do not include zero-stock items in the list.

Viva questions: What is the difference among list, set, and dictionary comprehensions? When should a loop be preferred?

PRACTICAL 08 | UNIT IV

Functions, Argument Types and Unpacking

Course outcome: CO3, CO2

Aim: Design reusable functions using positional, keyword, default, variable-length, and unpacked arguments.

Concepts used: def, return, docstrings, positional arguments, keyword arguments, default values, *args, **kwargs, argument unpacking.

Algorithm / steps

1. Create a function to calculate a discounted line-item total.
2. Create an average function that accepts any number of scores using *args.
3. Call the first function with a dictionary unpacked using **.
4. Validate function inputs where the domain rule is clear.

Program

Save as practical_08_functions.py

```
def line_total(unit_price, quantity=1, discount_percent=0):
    """Return the discounted total for one line item."""
    if unit_price < 0 or quantity < 0:
        raise ValueError("Price and quantity must not be negative.")
    if not 0 <= discount_percent <= 100:
        raise ValueError("Discount must be from 0 to 100.")

    subtotal = unit_price * quantity
    return subtotal * (1 - discount_percent / 100)


def average(*scores):
    """Return the arithmetic mean, or None when no scores are given."""
    if not scores:
        return None
    return sum(scores) / len(scores)

item = {"unit_price": 250, "quantity": 3, "discount_percent": 10}
print(f"Line total: Rs. {line_total(**item):.2f}")
print("Average:", average(78, 82, 91))
print("No-score average:", average())
```

Sample run / expected result

```
Line total: Rs. 675.00
Average: 83.66666666666667
No-score average: None
```

Test cases to verify

Test / input	Expected behaviour
line_total(250)	Uses defaults: quantity=1 and discount_percent=0.
line_total(**item)	Uses dictionary keys as keyword argument names.
line_total(-1, 2)	Raises ValueError due to invalid price.
average()	Returns None because no data was supplied.

Industry connection: Functions let teams reuse business rules and test them independently. Explicit parameters and return values also make code easier to review and maintain.
Extension task: Write calculate_shipping(weight, express=False) and demonstrate positional, keyword, and default arguments. Add tests for negative weight.
Viva questions: What is the purpose of return? How do *args and **kwargs differ? What is the difference between a parameter and an argument?

PRACTICAL 09 | UNIT IV

Recursive Functions and Base Cases

Course outcome: CO3, CO2

Aim: Implement a recursive function and explain the base case and recursive case.

Concepts used: Recursion, base case, recursive call, call stack, input domain, termination.

Algorithm / steps

1. Define factorial(n) for non-negative integers.
2. Return 1 for the base case n <= 1.
3. Reduce the problem by calling factorial(n - 1).
4. Test the base case, a typical input, and invalid input.

Program

Save as practical_09_recursion.py

```
def factorial(n):
    """Return n! for a non-negative integer n."""
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError("n must be an integer.")
    if n < 0:
        raise ValueError("n must not be negative.")
    if n <= 1:  # base case
        return 1
    return n * factorial(n - 1)  # recursive case

for value in (0, 1, 5, 7):
    print(f"{value}! = {factorial(value)}")
```

Sample run / expected result

```
0! = 1
1! = 1
5! = 120
7! = 5040
```

Test cases to verify

Test / input	Expected behaviour
factorial(0)	Returns 1 immediately at the base case.
factorial(5)	Returns 120.
factorial(-2)	Raises ValueError.
factorial(2.5)	Raises TypeError.

Industry connection: Recursion is useful for naturally nested problems such as tree traversal, directory structures, and divide-and-conquer algorithms. In production, consider stack-depth and performance limits.
Extension task: Write a recursive function to sum the digits of a non-negative integer. Then compare it with a loop-based solution.
Viva questions: What is a base case? What happens if a recursive function never reaches its base case?

PRACTICAL 10 | UNIT V

Lambda Functions and Sorting with a Key

Course outcome: CO2, CO3

Aim: Use a lambda expression for a small function and sort structured records by a selected field.

Concepts used: Anonymous function, lambda expression, key parameter, sorted(), stable sorting, tuple sort keys.

Algorithm / steps

1. Create a list of student records.
2. Sort by descending score and then ascending name.
3. Use lambda only for the small key expression.
4. Display a ranked report.

Program

Save as practical_10_lambda_sort.py

```
students = [
    {"name": "Ravi", "score": 82},
    {"name": "Asha", "score": 94},
    {"name": "Meena", "score": 94},
    {"name": "Kabir", "score": 76},
]

ranked = sorted(
    students,
    key=lambda student: (-student["score"], student["name"])
)

for rank, student in enumerate(ranked, start=1):
    print(f"{rank}. {student['name']} - {student['score']}")
```

Sample run / expected result

1. Asha - 94
2. Meena - 94
3. Ravi - 82
4. Kabir - 76

Test cases to verify

Test / input	Expected behaviour
Two records have the same score	Name is used as the tie-breaker.
Empty student list	The loop prints no rows and no error occurs.
Use sorted(..., key=lambda x: x["name"])	Sorts records alphabetically by name.

Industry connection: Sorting by a key is a frequent reporting task for leaderboards, product prices, search results, and prioritised work queues.

Extension task: Sort product records by price ascending, then quantity descending. Explain the key tuple and keep the lambda short.

Viva questions: What is a lambda function? What is the purpose of key in sorted()? Why is a named def often better for complex logic?

PRACTICAL 11 | UNIT V

Using map(), filter() and reduce()

Course outcome: CO2, CO3

Aim: Transform, select, and aggregate collections using functional programming tools and lambda expressions.

Concepts used: map(), filter(), reduce() from functools, iterable conversion to list, accumulator, pure transformation.

Algorithm / steps

1. Create a list of order values.
2. Use map() to calculate tax-inclusive values.
3. Use filter() to select high-value orders.
4. Use reduce() to calculate a total.

Program

Save as practical_11_map_filter_reduce.py

```
from functools import reduce

orders = [450, 1200, 800, 2500, 600]
with_tax = list(map(lambda amount: round(amount * 1.05, 2), orders))
high_value = list(filter(lambda amount: amount >= 1000, orders))
total = reduce(lambda running_total, amount: running_total + amount, orders, 0)

print("Orders with 5% tax:", with_tax)
print("Orders >= 1000:", high_value)
print("Total before tax:", total)
```

Sample run / expected result

```
Orders with 5% tax: [472.5, 1260.0, 840.0, 2625.0, 630.0]
Orders >= 1000: [1200, 2500]
Total before tax: 5550
```

Test cases to verify

Test / input	Expected behaviour
Empty list with map/filter	Produces an empty list.
reduce(..., [], 0)	Returns the initial accumulator 0.
Threshold 1000	Only 1200 and 2500 are selected.

Industry connection: These operations are helpful for clear data pipelines. In team code, use a named function instead of a complicated lambda when the transformation needs a descriptive name or debugging.

Extension task: For a list of temperatures in Celsius, use map() to convert to Fahrenheit, filter() to keep values above 30 C, and reduce() to compute a total.

Viva questions: What does map() return in modern Python? Why is reduce imported from functools? What is an accumulator?

PRACTICAL 12 | UNIT V

Higher-Order Functions and Reusable Transformations

Course outcome: CO2, CO3

Aim: Write a function that accepts another function as an argument and applies it to a collection.

Concepts used: Higher-order function, callable object, function as argument, return value, separation of policy and data.

Algorithm / steps

1. Define apply_to_values(values, operation).
2. Pass a named function and a lambda as operations.
3. Compare the outputs without duplicating the iteration code.
4. Ensure the helper function has one clear responsibility.

Program

Save as practical_12_higher_order.py

```
def apply_to_values(values, operation):
    """Apply operation to every value and return the results."""
    return [operation(value) for value in values]


def normalise_price(price):
    """Round a price to two decimal places."""
    return round(price, 2)

prices = [99.999, 125.5, 44.444]
rounded = apply_to_values(prices, normalise_price)
with_fee = apply_to_values(prices, lambda price: round(price * 1.02, 2))

print("Rounded:", rounded)
print("Price plus 2% fee:", with_fee)
```

Sample run / expected result

```
Rounded: [100.0, 125.5, 44.44]
Price plus 2% fee: [102.0, 128.01, 45.33]
```

Test cases to verify

Test / input	Expected behaviour
operation is normalise_price	The named function is called once per value.
operation is lambda price: price * 2	The function applies doubling to all input values.
values is empty	Returns an empty list.

Industry connection: Higher-order functions allow reusable pipelines, configurable validation, callbacks, and pluggable business rules. They reduce duplicated loop logic.

Extension task: Create a reusable transform function that accepts a list of scores and a callable; pass one callable that adds 5 bonus points with a cap of 100.

Viva questions: What makes a function higher-order? What is a callable? When is a named function clearer than a lambda?

PRACTICAL 13 | UNIT V

Data Transformation Pipeline: Integrating Functional Tools

Course outcome: CO1, CO2, CO3

Aim: Combine a list of dictionaries with lambda, map/filter/reduce-style processing to create a simple order report.

Concepts used: List of dictionaries, list comprehension, lambda, map, filter, reduce, aggregation, readable pipeline design.

Algorithm / steps

1. Represent orders as dictionaries.
2. Filter orders whose status is paid.
3. Calculate the line total for each paid order.
4. Aggregate the result and format a report.

Program

Save as practical_13_order_pipeline.py

```
from functools import reduce

orders = [
    {"id": "O-101", "amount": 500, "status": "paid"},
    {"id": "O-102", "amount": 800, "status": "pending"},
    {"id": "O-103", "amount": 1250, "status": "paid"},
]

paid_orders = list(filter(lambda order: order["status"] == "paid", orders))
amounts = list(map(lambda order: order["amount"], paid_orders))
paid_total = reduce(lambda total, amount: total + amount, amounts, 0)

print("Paid order IDs:", [order["id"] for order in paid_orders])
print("Paid amounts:", amounts)
print("Paid total:", paid_total)
```

Sample run / expected result

```
Paid order IDs: ['O-101', 'O-103']
Paid amounts: [500, 1250]
Paid total: 1750
```

Test cases to verify

Test / input	Expected behaviour
No paid orders	IDs and amounts are empty lists; total is 0.
Pending order only	It must not contribute to paid total.
Add a paid order for 100	Paid total increases by 100.

Industry connection: This resembles a simplified reporting pipeline used in operations and finance. Real systems would also validate records, use a database, log errors, and define currency/rounding rules.

Extension task: Add a tax-inclusive amount field and group paid orders into low-value (< 1000) and high-value (>= 1000) lists.

Viva questions: Why filter before aggregating? Why provide the initial value 0 to reduce? What assumptions does this example make about each order?

PRACTICAL 14 | UNIT VI

Built-in Modules and Standard Library Utilities

Course outcome: CO2, CO3

Aim: Import standard-library modules and use them to calculate summary statistics and create a timestamp.

Concepts used: import, from-import, math, statistics, datetime, module namespace, module aliases.

Algorithm / steps

1. Import the required standard-library modules.
2. Calculate mean, median, and population standard deviation.
3. Use math.ceil() for a simple rounding-up task.
4. Print a timestamp for the report.

Program

Save as practical_14_standard_library.py

```
from datetime import datetime
import math
import statistics

response_times_ms = [120, 135, 98, 160, 110, 142]

print("Mean response time:", round(statistics.mean(response_times_ms), 2))
print("Median response time:", statistics.median(response_times_ms))
print("Population standard deviation:", round(statistics.pstdev(response_times_ms), 2))
print("Batches needed for 53 records at 10 per batch:", math.ceil(53 / 10))
print("Report generated at:", datetime.now().astimezone().isoformat(timespec="seconds"))
```

Sample run / expected result

Mean response time: 127.5
Median response time: 127.5
Population standard deviation: 20.64
Batches needed for 53 records at 10 per batch: 6
Report generated at: (current local date and time)

Test cases to verify

Test / input	Expected behaviour
Empty list passed to statistics.mean()	Raises StatisticsError; validate that data exists.
53 records, 10 per batch	math.ceil(5.3) returns 6.
Import statistics	The names are available under the statistics namespace.

Industry connection: The standard library offers tested functionality for dates, paths, math, statistics, JSON, CSV, and logging. Prefer it over writing common utilities from scratch.
Extension task: Use pathlib to check whether a folder exists and create an output directory if it does not. Do not delete user files.
Viva questions: What is a module? What is the difference between import math and from math import ceil? Why should a local file not be named statistics.py?

PRACTICAL 15 | UNIT VI

Create and Import a Custom Module

Course outcome: CO3, CO2

Aim: Move reusable logic into a custom module and import it from a separate main program.

Concepts used: Custom module, import, namespace, reusable functions, __name__, __main__ guard.

Algorithm / steps

1. Create pricing.py with reusable pricing functions.
2. Create main.py in the same directory.
3. Import the required function and call it.
4. Use the main guard so the entry point runs only when main.py is executed directly.

Program

File 1: pricing.py

```
"""Reusable pricing calculations for a small shop."""


def calculate_total(unit_price, quantity, tax_rate=0.0):
    """Return total cost including tax; tax_rate is a decimal."""
    if unit_price < 0 or quantity < 0:
        raise ValueError("Price and quantity must not be negative.")
    if not 0 <= tax_rate <= 1:
        raise ValueError("Tax rate must be from 0 to 1.")
    subtotal = unit_price * quantity
    return round(subtotal * (1 + tax_rate), 2)


def format_currency(amount):
    """Return an amount formatted to two decimal places."""
    return f"Rs. {amount:,.2f}"
```

File 2: main.py (same folder)

```
from pricing import calculate_total, format_currency


def main():
    total = calculate_total(unit_price=499.0, quantity=2, tax_rate=0.05)
    print("Order total:", format_currency(total))


if __name__ == "__main__":
    main()
```

Sample run / expected result

Order total: Rs. 1,047.90

Test cases to verify

Test / input	Expected behaviour
Run python main.py from the folder	The main function runs and prints the order total.
Import calculate_total from pricing	The function is available without executing main.py as

	a script.
tax_rate = 1.2	Raises ValueError due to an invalid rate.

Industry connection: Modules enable reuse and testing. The main guard keeps command-line behaviour from running during import.
Extension task: Add apply_discount() to pricing.py and test normal, zero-quantity, and invalid-rate cases.
Viva questions: What is a custom module? What does the main guard prevent?

PRACTICAL 16 | UNIT VI

Packages and Project Structure

Course outcome: CO3, CO2

Aim: Organise related modules into a package and run the application from the project root.

Concepts used: Package, __init__.py, module paths, imports, project root, separation of files by responsibility.

Algorithm / steps

1. Create a folder named student_app.
2. Add __init__.py and grade_utils.py inside it.
3. Create main.py beside the package folder.
4. Import a function and run main.py from the project root.

Program

Project structure

```
python_lab_project/
    main.py
    student_app/
        __init__.py
        grade_utils.py
```

File: student_app/grade_utils.py

```
"""Grade-related helper functions."""


def average_score(scores):
    if not scores:
        raise ValueError("At least one score is required.")
    if any(score < 0 or score > 100 for score in scores):
        raise ValueError("Scores must be between 0 and 100.")
    return sum(scores) / len(scores)


def pass_count(scores, pass_mark=40):
    return sum(score >= pass_mark for score in scores)
```

File: main.py

```
from student_app.grade_utils import average_score, pass_count


def main():
    scores = [78, 61, 35, 92]
    print(f"Average: {average_score(scores):.2f}")
    print("Students passed:", pass_count(scores))


if __name__ == "__main__":
    main()
```

PRACTICAL 16 (CONTINUED)

Sample run / expected result

Average: 66.50
Students passed: 3

Test cases to verify

Test / input	Expected behaviour
Run from project root	Python finds student_app and imports grade_utils.
Run from an unrelated folder without setting the project path	The package may not be found; run from the project root or configure the environment correctly.
scores=[]	average_score raises ValueError with a clear message.

Industry connection: Packages keep medium-sized applications maintainable as the number of modules grows. Clear folder structure makes it easier to test, reuse, and review features.
Extension task: Add a student_app/report.py module that imports grade_utils and creates a formatted summary. Keep input/output code in main.py.
Viva questions: What is a package? What role does __init__.py commonly play? Why should a project be run from a documented root folder?

PRACTICAL 17 | UNIT VI

Namespaces, Global and Local Scope, and Inner Functions

Course outcome: CO3, CO2

Aim: Observe local and global names, inspect global()/locals(), and create a closure using an inner function.

Concepts used: Namespace, local scope, global scope, global(), locals(), nested/inner functions, nonlocal, LEGB lookup concept.

Algorithm / steps

1. Define a module-level configuration value.
2. Inspect names visible through globals() and locals().
3. Create an inner function that remembers a count using nonlocal.
4. Explain why returning values is usually preferable to modifying globals.

Program

Save as practical_17_scope.py

```
APP_NAME = "Student Report Tool"  # module-level name


def make_counter():
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    return increment


def show_scope(student_name):
    result = f"{APP_NAME}: {student_name}"
    print("Local names include student_name:", "student_name" in locals())
    print("APP_NAME in globals:", "APP_NAME" in globals())
    return result

counter = make_counter()
print(show_scope("Asha"))
print("Counter:", counter(), counter(), counter())
```

Sample run / expected result

```
Local names include student_name: True
APP_NAME in globals: True
Student Report Tool: Asha
Counter: 1 2 3
```

Test cases to verify

Test / input	Expected behaviour
Call counter() repeatedly	The closure remembers count between calls.
Use global count in make_counter	Would refer to a module-level name and would not update the enclosing local count as intended.

Call show_scope("Ravi")	The returned string contains Ravi.

Industry connection: Understanding scope prevents accidental state changes and name conflicts. Closures can hold small pieces of state, while globals() and locals() are mainly useful for inspection and specialised tooling.
Extension task: Build make_ticket_id(prefix) that returns an inner function generating IDs such as LAB-1, LAB-2. Do not use a global counter.
Viva questions: What is a namespace? What does nonlocal do? Why should globals() and locals() not be used as a substitute for clear function design?

PRACTICAL 18 | UNIT VI

Integrated CLI Utility: Module + Collections + Functions

Course outcome: CO1, CO2, CO3

Aim: Combine dictionaries, sets, functions, validation, and a module-style entry point in a small command-line utility.

Concepts used: Collections, functions, input validation, main(), main guard, clear user-facing messages.

Algorithm / steps

1. Store inventory in a dictionary.
2. Implement functions to add stock, sell stock, and display inventory.
3. Use a set to track valid item codes encountered in transactions.
4. Validate quantity and prevent selling more than available stock.

Program

Save as practical_18_inventory.py

```
inventory = {"PEN": 100, "BOOK": 40, "FILE": 25}
transaction_items = set()


def add_stock(item, quantity):
    if quantity <= 0:
        raise ValueError("Quantity must be positive.")
    inventory[item] = inventory.get(item, 0) + quantity
    transaction_items.add(item)


def sell_stock(item, quantity):
    if quantity <= 0:
        raise ValueError("Quantity must be positive.")
    if item not in inventory:
        raise KeyError(f"Unknown item: {item}")
    if quantity > inventory[item]:
        raise ValueError("Insufficient stock.")
    inventory[item] -= quantity
    transaction_items.add(item)


def show_inventory():
    for item, quantity in sorted(inventory.items()):
        print(f"{item}: {quantity}")


def main():
    add_stock("PEN", 20)
    sell_stock("BOOK", 5)
    print("Inventory after transactions:")
    show_inventory()
    print("Items transacted:", sorted(transaction_items))


if __name__ == "__main__":
    main()
```

PRACTICAL 18 (CONTINUED)

Sample run / expected result

```
Inventory after transactions:
BOOK: 35
FILE: 25
PEN: 120
Items transacted: ['BOOK', 'PEN']
```

Test cases to verify

Test / input	Expected behaviour
sell_stock("BOOK", 100)	Raises ValueError and does not reduce the stock.
add_stock("PEN", 0)	Raises ValueError.
sell_stock("UNKNOWN", 1)	Raises KeyError with the unknown code.

Industry connection: The structure resembles the core of a small business command-line tool. In a production system, persistence, authentication, transaction logging, and automated tests would be required.
Extension task: Extend it with a menu, case-normalise item codes, add unit tests, and save/load inventory as JSON using the standard library.
Viva questions: Why are operations wrapped in functions? Why does the code use a dictionary for stock and a set for transacted item codes?

Mini-Projects for Internal Assessment

These projects integrate the semester concepts. Students should submit source files, sample runs, test cases, a short README, and a brief explanation of design decisions. Projects can be completed individually or in small groups as directed by the instructor.

Mini-project 1: Student Marks and Result Analyzer

Problem statement: Build a command-line program that records student names and marks, calculates averages and grades, lists top performers, and reports pass/fail counts.

Required concepts: Lists, dictionaries, loops, validation, functions, sorting with lambda, comprehensions, and a custom module.

Minimum features: Add a student, validate marks from 0 to 100, show all records, calculate average, show highest/lowest score, and create a summary report.

Suggested modules: main.py for the menu; result_utils.py for grade and statistics functions.

Acceptance tests: Reject marks outside 0-100; handle an empty class without division by zero; break tied scores consistently; do not allow blank student names.

Mini-project 2: Inventory and Billing CLI

Problem statement: Build a small shop utility to maintain product codes, stock counts, unit prices, add-stock transactions, sales, discounts, and a final bill.

Required concepts: Dictionary for product records, set for unique transaction codes, lists for line items, functions, argument defaults, modules, and input validation.

Minimum features: Add product, update stock, sell quantity, calculate subtotal/discount/tax, show inventory, and print a bill summary.

Acceptance tests: Prevent negative prices/quantities; reject sale quantities greater than stock; handle unknown product codes; test zero quantity and 0%/100% discount.

Mini-project 3: Workshop Registration Analyzer

Problem statement: Compare two workshop registration lists and generate a unique attendee list, shared attendees, and participants attending only one workshop.

Required concepts: Lists, sets, dictionary counting, comprehensions, functions, and formatted reports.

Minimum features: Normalise names/IDs, remove duplicate registrations, compare workshops, count participants, and export a clean text report.

Acceptance tests: Handle duplicate IDs, empty registration lists, mixed upper/lower case, and students registered for both workshops.

Suggested Practical Evaluation Rubric

Criteria	Suggested marks	What is assessed
Understanding and algorithm	2	Can explain the problem, steps, and choice of data type.
Program correctness	4	Program runs and produces correct outputs for the main case.
Testing and validation	2	Tests boundary/invalid inputs and handles expected errors.
Code quality and explanation	1	Readable names, indentation, function use, and comments/docstrings where helpful.
Viva / reflection	1	Can explain key statements and one design decision.

Viva-Voce Revision Bank

1. What is an interpreter? How is a Python source file executed?
2. How do int, float, str, bool, list, tuple, set, and dict differ?
3. What is the difference between == and is?
4. Explain operator precedence with an example using parentheses.
5. What is the difference between a list and a tuple?
6. When would you choose a set instead of a list?
7. How does dictionary get() help avoid a KeyError?
8. What is a comprehension and when can it hurt readability?
9. What is the difference between a function parameter and an argument?
10. Explain positional, keyword, default, *args, and **kwargs arguments.
11. Why must recursive functions have a base case?
12. What is a lambda function? How is it used as a sort key?
13. Explain map(), filter(), and reduce() in one sentence each.
14. What is a higher-order function?
15. What is the difference between a module and a package?
16. Why should executable code be placed under if __name__ == "__main__":?
17. What do globals(), locals(), and nonlocal mean?
18. Why do professional programs validate input and test boundary cases?

Reference Resources

Use these resources for additional examples and exact language reference. The practical programs in this manual are original teaching examples designed to fit the supplied course syllabus.

- Python Tutorial (official documentation): https://docs.python.org/3/tutorial/
- Built-in Types (official documentation): https://docs.python.org/3/library/stdtypes.html
- Functional Programming HOWTO (official documentation): https://docs.python.org/3/howto/functional.html
- Python Standard Library (official documentation): https://docs.python.org/3/library/
- PEP 8 - Style Guide for Python Code: https://peps.python.org/pep-0008/
- Textbook references supplied in the syllabus: Martin Brown, Python: The Complete Reference, 4th ed.; Yashavant Kanetkar and Aditya Kanetkar, Let Us Python, 4th ed.; Y. Zhang, An Introduction to Python and Computer Programming.