Week 2 Lab Session: Hands-On Programming Lab and Jupyter Notebook Workspace Guide

This **Comprehensive Lab Worksheet** serves as a complete hands-on guide for the Week 2 Lab Session of **Machine Learning with Python for Managers**. It is designed to bridge foundational programming theories with interactive execution in the **Jupyter Notebook** environment.

By completing this lab, you will write, execute, and debug the exact Python code structures, sequence containers, and logic controls.

Lab Objectives

By the end of this 60-minute practical session, you will be able to:

1. **Navigate the Jupyter Workspace**: Master cell executions, manage kernel sessions, and write structural documentation using Markdown.
2. **Manipulate Variables and Basic Types**: Inspect dynamic assignments using type() and perform precise arithmetic operations including floor division (//) and modulus (%).
3. **Control Mutability & Sequence Collections**: Work with ordered, mutable lists and safeguard immutable tuples, strings, and ranges.
4. **Develop Logic Control and Loops**: Code nested decision gates (if-elif-else), definite iteration (for), indefinite iteration (while), and loop control statements (break, continue, else).

Part 1: Configuring the Jupyter Laboratory

Jupyter Notebook is an open-source web application that allows managers and data scientists to combine live, executable code, visualizations, and structured narrative text into a single, self-documenting file.

Task 1.1: Documenting with Markdown

In a professional environment, raw code must be accompanied by business context. Jupyter uses **Markdown cells** for this purpose.

1. Launch your Jupyter Notebook and create a new notebook file named Week2_HandsOn_Lab.ipynb.

2. Select your first cell, and in the toolbar dropdown, change the cell type from **Code** to **Markdown**.

3. Type the following formatted text block inside the cell:

```
# Week 2 Practical Laboratory
## Python Basics for Machine Learning Managers

* **Topics Covered**: Variables, Arithmetic, Strings, Lists, Tuples, Ranges, and
Control Flow.
---
*Note: This cell is written in Markdown and will not be executed as computer code.*
```

4. Press Shift + Enter or Ctrl + Enter to render the Markdown text into clean, styled HTML.

Task 1.2: Launching and Running Code Cells

1. Double-click the cell below your Markdown block (ensure its type is set to **Code** in the toolbar).

2. Type the basic output check:

```
print("Hello from Jupyter! This cell contains active executable code.")
```

3. Press Shift + Enter. The active **Kernel** in the background executes the command and prints the output string directly beneath the cell. Note the execution count number appearing in the square brackets (e.g., In [1]:), which tracks your run history during the current session.

Part 2: Variables, Arithmetic Operators, and Precision Modelling

Python is **dynamically typed**, meaning you do not need to declare whether a variable is an integer or a decimal float before initializing it. The interpreter dynamically allocates memory and assigns the data type based on the value on the right-hand side of the assignment operator (=).

Task 2.1: Dynamic Variable Assignment and Type Inspection

Run the following script to initialize singular variable parameters and inspect their underlying database classes using the `type()` function:

```
# Initialize primitive variable assignments
a = 10             # Integer representation
b = 10.5           # Floating-point representation
c = 3 + 4j         # Complex number representation (imaginary unit 'j')


# Output variables and inspect their data types
print("--- Variable Type Verification ---")
print(f"Variable a holds: {a} | Data Type: {type(a)}")
print(f"Variable b holds: {b} | Data Type: {type(b)}")
print(f"Variable c holds: {c} | Data Type: {type(c)}")
```

Task 2.2: Advanced Arithmetic and Float Precision Operations

Division in Python has distinct operational paradigms. Standard division (/) always returns a decimal float, while floor division (//) truncates the decimal portion entirely. The modulus operator (%) returns only the remainder.

Run this script to observe how Python handles mathematical calculations and float precision:

```
# Inputs for calculation
x = 10
y = 3
z = 5.1

# 1. Standard Division vs. Floor Division
std_div = x / y       # Returns a continuous decimal
floor_div = x // y    # Truncates to the integer floor (does not round)
remainder = x % y     # Returns the mathematical remainder (modulus)

print("--- Division Paradigms ---")
print(f"Standard Division (10 / 3)   = {std_div}")
print(f"Floor Division (10 // 3)     = {floor_div} (Truncated integer portion)")
print(f"Modulus Remainder (10 % 3)   = {remainder} (Leftover after division)")

# Floor division demonstration with floats (Professor Batra's example)
float_floor = x // z
print(f"Floor Division with Float (10 // 5.1) = {float_floor} (Evaluates to 1.0)")

# 2. Exponentiation and Exponent Powers
power_calc = x ** y   # 10 raised to the power of 3 (10 * 10 * 10)
print(f"\nPower Exponentiation (10 ** 3) = {power_calc}")

# 3. Floating-Point Precision and Rounding Functions
num = 3.787
abs_val = abs(-15)    # Absolute value removes negative signs

print(f"\n--- Precision and Absolute Functions ---")
print(f"Absolute Value of -15        = {abs_val}")
print(f"Original Decimal Number      = {num}")
print(f"Rounded to 1 Decimal Place   = {round(num, 1)}") # Evaluates to 3.8
print(f"Rounded to 2 Decimal Places  = {round(num, 2)}") # Evaluates to 3.79 (rounds up
because of 7)
```

Task 2.3: In-Place Assignment Shorthand

Assignment operators let you update variables incrementally without rewriting long formulas.

```
# Initial value of counter
counter = 10

counter += 5   # Shorthand for counter = counter + 5 -> 15
print(f"Incremented counter (+= 5): {counter}")

counter *= 2   # Shorthand for counter = counter * 2 -> 30
print(f"Multiplied counter (*= 2): {counter}")
```

```
counter -= 10  # Shorthand for counter = counter - 10 -> 20
print(f"Decremented counter (-= 10): {counter}")

counter /= 2   # Shorthand for counter = counter / 2 -> 10.0
print(f"Divided counter (/= 2): {counter}")
```

Part 3: Sequence Collection Operations

Python manages multi-dimensional data using sequential containers: Strings, Lists, Tuples, and Ranges.

Task 3.1: Mutable List Manipulation

Lists are **ordered** and **mutable** collections defined with square brackets []. Because they are mutable, elements can be added, updated, sorted, or removed dynamically during execution.

```
# 1. Initialize a heterogeneous list with mixed data types
my_list = [10, 20, 30, 40, 50, "python", True]

print("--- Core Indexing and Slicing ---")
print(f"Full List: {my_list}")
print(f"First Element (Index 0)         : {my_list[0]}")
print(f"Fourth Element (Index 3)        : {my_list[3]}")
print(f"Absolute Last Element (Index -1): {my_list[-1]}")
print(f"Second-to-Last Element (Index -2): {my_list[-2]}")

# 2. List Modification and Splicing
my_list[2] = 100  # Overwrite index 2 (replaces 30 with 100)
print(f"\nAfter Direct In-Place Update    : {my_list}")

# 3. Adding Elements
my_list.append(500)             # Append 500 to the end of the list
my_list.insert(1, 1000)         # Insert 1000 at index 1 (shifts list)
print(f"After append(500) and insert    : {my_list}")

# 4. Extending Lists vs. Nesting Lists
list_a = [1, 2, 3]
list_b = [4, 5, 6]

# Extend merges list_b directly into list_a
list_a.extend(list_b)
print(f"\nExtended List (list_a.extend)   : {list_a}")

# 5. Removing Elements and Popping
# remove() deletes by value; pop() deletes by index
list_a.remove(4)                # Deletes the first occurrence of 4
popped_val = list_a.pop(1)      # Removes and returns element at index 1

print(f"After removing value 4          : {list_a}")
print(f"Popped Element (Index 1 was 2)  : {popped_val}")
print(f"Final Mutated List              : {list_a}")
```

Task 3.2: Immutable String Processing and Text Parsing

Strings are **immutable** sequences of text characters. When you modify a string, Python does not change the original characters in memory; instead, it generates an entirely new string.

```
# Initialize messy customer name record
s = "  hello world  "

# 1. Case Operations
upper_s = s.upper()          # Converts to uppercase
lower_s = s.lower()          # Converts to lowercase
title_s = s.title()          # Title Case: capitalizes first letter of every word
cap_s = s.capitalize()      # Sentence Case: capitalizes only first letter of string
swap_s = upper_s.swapcase()  # Reverses casing

print("--- String Case Modifications ---")
print(f"Original Text        : '{s}'")
print(f"Uppercase (.upper)   : '{upper_s}'")
print(f"Title Case (.title)  : '{title_s}'")
print(f"Capitalized          : '{cap_s}'")
print(f"Swap Case (.swapcase): '{swap_s}'")

# 2. Whitespace Stripping
strip_all = s.strip()        # Removes spaces on both sides
strip_left = s.lstrip()      # Removes leading spaces (left side)
strip_right = s.rstrip()     # Removes trailing spaces (right side)

print(f"\n--- Whitespace Stripping ---")
print(f"Cleaned Strip        : '{strip_all}'")
print(f"Left Strip Only      : '{strip_left}'")

# 3. Text Splitting and Reassembling Sequences
tech_stack = "python,java,c++"
languages = tech_stack.split(",") # Splits by comma and returns a list

print(f"\n--- Splitting and Joining Strings ---")
print(f"Raw Concatenated Text: '{tech_stack}'")
print(f"Split List Result    : {languages}")

# Reassemble the list into a single string with a custom hyphen delimiter
joined_stack = "-".join(languages)
print(f"Reassembled String   : '{joined_stack}'")

# 4. Substring Analysis
banana_str = "banana"
count_a = banana_str.count("a") # Counts occurrences of letter 'a'
print(f"\nCharacter Count: Letter 'a' appears in '{banana_str}' {count_a} times.")
```

Task 3.3: Tuple Locking and Repetition Warnings

Tuples are **ordered, immutable** collections defined with parentheses ().

- **Professor's Warning**: The multiplication operator (*) on tuples does not perform element-wise arithmetic; instead, it duplicates the sequence's elements in place.

```
# Initialize a tuple representing fixed geographical coordinates
t1 = (12.9716, 77.5946)
print("--- Tuple Operations ---")
print(f"Geographical Tuple: {t1} | Data Type: {type(t1)}")
print(f"Latitude coordinate (Index 0): {t1[0]}")

# Attempt to modify a tuple (This will trigger a TypeError)
try:
    t1[0] = 13.0123
except TypeError as error:
    print(f"\n[MUTABILITY CHECK] Caught Expected Error: {error}")
    print("Verification: Tuples cannot be modified after creation!")

# The Tuple Repetition Operator (*) - Professor Batra's Caution
base_tuple = (1, 2, 3)
repetition_tuple = base_tuple * 3

print(f"\nMultiplication Check (base_tuple * 3):")
print(f"Resulting Tuple: {repetition_tuple}")
print("Caution: The * operator replicates sequence items; it does not multiply them!")
```

Task 3.4: Memory-Efficient Ranges

Range sequences (`range(start, stop, step)`) generate integer lists on demand rather than storing them in memory, making them highly efficient for running large loops.

```
# Create distinct range sequences
r1 = range(5)              # 0 to 4 (default step of 1, start of 0)
r2 = range(2, 7)           # 2 to 6 (stop is exclusive)
r3 = range(4, 15, 3)       # 4 to 14, jumping by steps of 3 (4, 7, 10, 13)

print("--- Range Sequence Conversions ---")
print(f"Range object r3: {r3}")
print(f"r3 converted to physical list: {list(r3)}") # Generates and displays numbers
```

Part 4: Logic Gates & Dynamic Looping (15 Minutes)

Control structures allow your code to evaluate conditions and run repetitive tasks dynamically.

Task 4.1: Multi-Tier Decision System (if-elif-else)

Run this script to evaluate user input against a standard academic grading system:

```
# Define student score parameter (simulate dynamic database values)
marks = 78

print("--- Rule-Based Decision Execution ---")
if marks >= 90:
    grade = "A (Excellent)"
elif marks >= 75:
    grade = "B (Very Good)"
elif marks >= 60:
    grade = "C (Good)"
else:
    grade = "F (Fail)"

print(f"Student Score: {marks} | Final Grade: {grade}")
```

Task 4.2: Definite Iteration (for) and Indefinite Iteration (while)

1. **The** for **Loop**: Used when the number of iterations is known beforehand or bounded by a sequence.
2. **The** while **Loop**: Run based on dynamic conditions. It requires an updating counter to prevent infinite loops.

```
# 1. Squaring elements in a collection using a for loop
numbers = [1, 2, 3]
print("--- For Loop Execution ---")
for num in numbers:
    squared_val = num ** 2
    print(f"Number: {num} -> Squared Value: {squared_val}")

# 2. Variable-driven loop using a while loop
print("\n--- While Loop Execution ---")
count = 1
while count <= 5:
    print(f"While Loop Counter is currently: {count}")
    count += 1  # Crucial update step to avoid infinite loop execution
```

Task 4.3: Interrupts and Process Validation (break, continue, else)

Run this script to understand loop control:

- break: Terminates the loop immediately.
- continue: Skips the remaining code in the current iteration and jumps back to the top of the loop.
- **Loop** else **block**: Executes only if the loop completes its entire run normally without hitting a break statement.

```
# List of transaction records to audit
transactions = [120.0, 450.0, -10.0, 300.0, 95.0]

print("--- Transaction Scan Initiated ---")
for tx in transactions:
    # 1. Skip anomalous negative charges using continue
    if tx < 0:
        print(f"Alert: Anomalous charge detected (${tx}). Skipping record.")
        continue # Immediately jumps to next transaction

    # 2. Terminate entire audit immediately if charge is too high
    if tx > 400:
        print(f"Critical Break: Flagged transaction of ${tx} exceeds safety limit!")
        print("Halting entire audit loop for immediate manual review.")
        break # Exits the loop entirely

    print(f"Transaction processed successfully: ${tx}")

# This else block executes only if the loop runs to completion without hitting a break
else:
    print("\nBatch Audit Success: All transactions validated without safety violations.")
```

Lab Challenge: Automated Portfolio Yield Auditor

To apply everything you have learned, write a Python script in a new cell that processes a list of client investments. Your code must audit and calculate the total compound interest generated over 3 years for each client.

Challenge Constraints:

1. **Initialize the Dataset**: Create a list containing 4 client records:
   `portfolios = [["C1", 50000, 0.06, "ACTIVE"], ["C2", -5000, 0.04, "ACTIVE"], ["C3", 80000, 0.08, "EXEMPT"], ["C4", 120000, 0.12, "ACTIVE"]]`
2. **Loop and Process**:
   - If a client's status is "EXEMPT", use continue to bypass their calculation and print: "Client [ID]: Status EXEMPT. Skipping calculation."
   - If a client's initial investment is negative (less than 0), print a warning and use continue to move to the next client.
   - If any individual initial investment exceeds $100,000, print a critical alert and use break to halt the entire batch scan for compliance verification.
3. **Interest Calculation**:

For valid active clients, calculate the forecasted portfolio value after 3 years using annual compound growth:

$$Value = Investment \times (1 + Interest\ Rate)^3$$

- Round the final portfolio value to 2 decimal places using round().
- Print a summary: "`Client [ID]: Clean Initial $[Investment] grew to $[Forecasted Value] at [Rate]%.`"

4. **Validation Guard**:
   - Add a loop else block that prints a final batch success message if the entire audit finishes without hitting a security limit.

Challenge Solution Code (Self-Study Guide)

Try writing the script on your own first. If you get stuck, run the following solution code in your Jupyter cell to verify your results:

```
# Initial portfolio data setup
portfolios = [
    ["C1", 50000, 0.06, "ACTIVE"],
    ["C2", -5000, 0.04, "ACTIVE"],   # Anomalous negative value, should be skipped
    ["C3", 80000, 0.08, "EXEMPT"],   # Exempt client, should be bypassed
    ["C4", 120000, 0.12, "ACTIVE"]   # Exceeds $100k, should trigger a break
]

print("--- Initiating Portfolio Audit Loop ---")
for client in portfolios:
    client_id = client[0]
    investment = client[1]
    rate = client[2]
    status = client[3]

    # 1. Exempt bypass check
    if status == "EXEMPT":
        print(f"Client {client_id}: Status EXEMPT. Skipping calculation.")
        continue

    # 2. Negative value data guard
    if investment < 0:
        print(f"Data Warning: Client {client_id} has a negative investment (${investment})!
Skipping.")
        continue

    # 3. Security threshold check
    if investment > 100000:
        print(f"CRITICAL LIMIT BREACH: Client {client_id} holds ${investment}, exceeding the
$100,000 limit!")
```

```
        print("Halt: Terminating entire batch loop for compliance verification.")
        break

    # 4. Calculation of compound interest
    years = 3
    final_val = investment * ((1 + rate) ** years)
    rounded_val = round(final_val, 2)

    print(f"Client {client_id}: Clean Initial ${investment} grew to ${rounded_val} at
{rate*100:.1f}%.")

# 5. Validation Guard (only prints if loop finishes without hitting the break)
else:
    print("\nSuccess: Batch scan completed. All active portfolios validated successfully.")
```