Lab 3: Python Basics, Files, and Functions

CS6.201: Introduction to Software Systems

ISS TAs

International Institute of Information Technology, Hyderabad

Outline

1. Introduction
2. The `uv` Package Manager
3. Basic Input/Output
4. Variables and Data Types
5. Operators
6. Conditional Statements
7. Loops
8. Data Structures
9. String Manipulation
10. Functions
11. File Operations
12. Conclusion

Introduction

Python is a high-level, interpreted programming language known for its **readability** and **versatility**. It is widely used in:

- Web development
- Data science
- Artificial intelligence
- Automation

Estimated Duration: 1.5 hours

Learning Objectives

By the end of this lab, you should be able to:

1. **Manage Projects**: Understand the utility of `uv` for dependency and project management.
2. **Use Core Syntax**: Correctly use variables, data types, and operators.
3. **Control Flow**: Implement conditional logic and iteration using loops.
4. **Organize Data**: Use data structures such as lists and dictionaries.
5. **Modularize Code**: Define and invoke functions to create reusable code blocks.
6. **Handle Files**: Read from and write to the file system.

Part 1: The `uv` Package Manager

Modern software development requires robust tools for managing dependencies and environments.

For this course, we utilize **uv**, a high-performance Python package and project manager written in Rust.

It serves as a unified replacement for traditional tools such as `pip`, `pip-tools`, and `virtualenv`, offering significantly faster execution times.

Key Commands: Initialize a Project

To initialize a new project, use the `init` command. This generates a `pyproject.toml` file, which is the standard configuration file for Python projects.

```
uv init [project_name]

```

If executed within an existing directory without a project name, it initializes the project in the current working directory.

Key Commands: Adding Packages

Add Packages

External libraries are essential for extending Python's capabilities. To install a library (e.g. `requests`):

```
uv add requests

```

This command automatically updates the `pyproject.toml` file and resolves version constraints in `uv.lock`.

Key Commands: Removing Packages

Remove Packages

To remove an unnecessary library:

```
uv remove requests

```

Key Commands: Run Scripts

To execute a Python script within the project's virtual environment:

```
uv run python script.py

```

Alternatively, you can run specific tools or modules:

```
uv run pytest

```

This ensures scripts have access to all installed dependencies.

Hands-On Exercise 1: Project Setup

1. Open your terminal.
2. Create a new directory named `lab3_exercise` and navigate into it.
3. Run `uv init` to initialize the project.
4. Inspect the generated `pyproject.toml` file.

Part 2: Basic Input/Output (Output)

Input and Output (I/O) operations allow a program to interact with the external world.

Output: The `print()` Function

The `print()` function displays output to the standard output device (console).

```python
print("Hello, World!")

```

```
Hello, World!
```

```python
name = "Student"
print("Welcome,", name)

```

```
Welcome, Student
```

Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings provide a concise and readable way to embed expressions inside string literals. An f-string is prefixed with the letter `f`.

```python
language = "Python"
version = 3.12
message = f"We are learning {language} version {version}."
print(message)

```

```
We are learning Python version 3.12.
```

The expressions inside the curly braces `{}` are evaluated at runtime.

Input: The input() Function

The input() function pauses program execution to wait for the user to type text and press Enter. It returns the entered data as a **string**.

```
user_name = input("Enter your name: ")
print(f"Greeting, {user_name}.")

```

Input: Type Conversion

Type Conversion

Since `input()` always returns a string, mathematical operations on the input require explicit type conversion.

```python
age_str = input("Enter your age: ")
# Convert the string to an integer
age = int(age_str)

print(f"Next year you will be {age + 1}")

```

Hands-On Exercise 2: Interactive Greeting

Create a Python script that:

1. Prompts the user for their favorite color.
2. Prompts the user for their birth year.
3. Calculates their approximate age.
4. Prints a message using the color and age.

Part 3: Variables and Data Types

A variable is a symbolic name that refers to an object. You do *not* need to declare the type of a variable before using it.

Fundamental Data Types

```python
# Integer (int): Whole numbers
items = 10

# Floating Point (float): Real numbers with decimals
price = 19.99

# String (str): Sequence of characters
brand = "Acme Corp"

# Boolean (bool): Truth values
is_available = True

# NoneType (None): Absence of a value
discount = None

```

The `type()` Function

The `type()` function returns the class type of the argument passed.

```python
print(f"Type of items: {type(items)}")
print(f"Type of price: {type(price)}")

```

```
Type of items: <class 'int'>
Type of price: <class 'float'>
```

Dynamic Typing

Dynamic Typing

Python is dynamically typed — the type of a variable is determined at runtime and can change during execution.

```python
x = 100              # x is an integer
print(type(x))

x = "Now I am a string"  # x is now a string
print(type(x))

```

Part 4: Arithmetic Operators

Python supports standard arithmetic operations.

```python
a, b = 10, 3

# Addition
print(f"{a} + {b} = {a + b}")
# Subtraction
print(f"{a} - {b} = {a - b}")
# Multiplication
print(f"{a} * {b} = {a * b}")
# True Division
print(f"{a} / {b} = {a / b}")
# Floor Division
print(f"{a} // {b} = {a // b}")
# Modulo
print(f"{a} % {b} = {a % b}")
# Exponentiation
print(f"{a} ** {b} = {a ** b}")

```

Output:

```
10 + 3 = 13
10 - 3 = 7
10 * 3 = 30
10 / 3 = 3.3333...
10 // 3 = 3
10 % 3 = 1
10 ** 3 = 1000
```

Logical and Comparison Operators

Logical Operators

```python
is_adult = True
has_ticket = False

# Logical AND: True if both operands are True
can_enter = is_adult and has_ticket
print(f"Can enter? {can_enter}")

```

```
Can enter? False
```

Comparison Operators

```python
print(f"10 > 3: {10 > 3}")    # True
print(f"10 == 10: {10 == 10}")  # True
print(f"10 != 5: {10 != 5}")   # True

```

Part 5: Conditional Statements

Conditional statements allow a program to execute different blocks of code based on conditions.

The `if-elif-else` Structure

```python
battery_level = 15

if battery_level > 80:
    status = "Healthy"
elif battery_level > 20:
    status = "Okay"
else:
    status = "Low Battery"

print(f"Battery Status: {status}")

```

```
Battery Status: Low Battery
```

Indentation

It is critical to note that Python uses **indentation** (whitespace) to define the scope of code blocks, such as loops, functions, and classes.

Standard practice is to use **4 spaces** per indentation level.

Hands-On Exercise 3: Grade Calculator

Write a program that:

1. Asks the user for a numerical score (0–100).
2. Prints `'Pass'` if the score is 50 or above.
3. Prints `'Fail'` if the score is below 50.

Part 6: The for Loop

Loops are used to iterate over a sequence or logical condition, allowing code to be executed repeatedly.

The `for` loop iterates over a sequence (list, tuple, dictionary, set, or string).

```python
# range(5) generates numbers from 0 to 4
for i in range(5):
    print(f"Index: {i}", end=" ")
print()

```

```
Index: 0 Index: 1 Index: 2 Index: 3 Index: 4
```

The `while` Loop

The `while` loop executes a set of statements as long as a condition is true.

```python
countdown = 3
while countdown > 0:
    print(f"{countdown}...")
    countdown -= 1
print("Liftoff.")

```

```
3...
2...
1...
Liftoff.
```

Hands-On Exercise 4: Countdown

Write a loop that prints all even numbers from 10 down to 0.

Part 7: Data Structures — Lists

Efficient data organization is crucial for software performance.

Lists: An ordered, mutable collection of items, defined using square brackets [].

```python
tasks = ["Analyze requirements", "Design system",
         "Implement code"]

# Accessing elements by index (0-based)
first_task = tasks[0]

# Modifying the list
tasks.append("Deploy")
tasks[1] = "Refine architecture"

print(f"Tasks: {tasks}")

```

```
Tasks: ['Analyze requirements', 'Refine architecture',
        'Implement code', 'Deploy']
```

List Slicing: Basics

Slicing extracts a portion of a list using the syntax `list[start:stop:step]`.

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Basic slicing: elements from start to stop-1
first_three = numbers[0:3]      # [0, 1, 2]

# Omitting start/stop defaults to beginning/end
last_four = numbers[-4:]        # [6, 7, 8, 9]

print(f"First three: {first_three}")
print(f"Last four: {last_four}")

```

```
First three: [0, 1, 2]
Last four: [6, 7, 8, 9]
```

List Slicing: Advanced

We can also use the step parameter in slicing.

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Using step to skip elements
every_second = numbers[::2]    # [0, 2, 4, 6, 8]

# Reverse a list using negative step
reversed_list = numbers[::-1]  # [9, 8, ..., 0]

# Extracting a middle portion
middle = numbers[3:7]          # [3, 4, 5, 6]

print(f"Every second: {every_second}")
print(f"Reversed: {reversed_list}")

```

Strings as Sequences

Strings are immutable sequences of Unicode characters.

```python
language = "Python"
print(f"First character: {language[0]}")
print(f"Last character: {language[-1]}")

```

```
First character: P
Last character: n
```

Dictionaries

A **dictionary** is an unordered collection of key-value pairs.

```python
user_profile = {
    "username": "student_01",
    "role": "Administrator",
    "access_level": 5
}

print(f"User: {user_profile['username']}")
print(f"Role: {user_profile['role']}")

# Adding a new key-value pair
user_profile["active"] = True

```

```
User: student_01
Role: Administrator
```

Hands-On Exercise 5: Shopping List

1. Create a Python list containing three grocery items.
2. Add a fourth item to the list.
3. Print the second item in the list.

Part 8: String Manipulation (Basics)

Python includes a rich set of methods for processing strings.

```python
raw_text = "   Data Processing   "

# strip() removes leading and trailing whitespace
clean_text = raw_text.strip()

# Text case transformations
print(clean_text.upper())  # DATA PROCESSING
print(clean_text.lower())  # data processing

```

```
DATA PROCESSING
data processing
```

String Manipulation (Split & Join)

We can easily convert between strings and lists.

```python
clean_text = "Data Processing"

# split() splits a string into a list of words
words = clean_text.split()
print(f"Words: {words}")

# join() combines items in an iterable into a string
sentence = "-".join(words)
print(f"Joined: {sentence}")

```

```
Words: ['Data', 'Processing']
Joined: Data-Processing
```

Part 9: Functions — Defining a Function

Functions are reusable blocks of code that perform a specific task. They help in utilizing the concept of **code modularity**.

Functions are defined using the `def` keyword.

```python
def generate_greeting(name, specific_greeting="Hello"):
    """Generates a greeting message.

    Args:
        name (str): The name of the person.
        specific_greeting (str): Defaults to "Hello".
    Returns:
        str: The formatted greeting string.
    """
    message = f"{specific_greeting}, {name}."
    return message

```

Invoking Functions

```python
msg1 = generate_greeting("Alice")
msg2 = generate_greeting("Bob", "Greetings")

print(msg1)
print(msg2)

```

```
Hello, Alice.
Greetings, Bob.
```

Function Return Values

Return Rules

A function can return a value using the `return` statement. If no `return` statement is specified, the function returns `None` by default.

```python
def calculate_square(n):
    return n * n

result = calculate_square(5)
print(f"The square of 5 is {result}")

```

```
The square of 5 is 25
```

Recursion

Recursion is a programming technique where a function calls itself to solve a problem.

A recursive function must have:

- A **base case** — a condition that stops the recursion.
- A **recursive case** — where the function calls itself with a modified argument.

Recursion Example 1: Factorial

The factorial of a non-negative integer n (denoted $n!$) is the product of all positive integers $\leq n$.

```python
def factorial(n):
    """Calculates the factorial of n recursively."""
    # Base case: factorial of 0 or 1 is 1
    if n <= 1:
        return 1
    # Recursive case: n! = n * (n-1)!
    return n * factorial(n - 1)

fact_5 = factorial(5)
print(f"5! = {fact_5}")

```

```
5! = 120
```

Recursion Example 2: Fibonacci (Code)

The Fibonacci sequence: each number is the sum of the two preceding ones.

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2)$$

```python
def fibonacci(n):
    """Returns the nth Fibonacci number (0-indexed)."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

```

Recursion Example 2: Fibonacci (Execution)

Executing the recursive Fibonacci function:

```python
fib_7 = fibonacci(7)
print(f"The 7th Fibonacci number is {fib_7}")

fib_sequence = [fibonacci(i) for i in range(8)]
print(f"First 8 Fibonacci numbers: {fib_sequence}")

```

```
The 7th Fibonacci number is 13
First 8 Fibonacci numbers: [0, 1, 1, 2, 3, 5, 8, 13]
```

Recursion — Important Considerations

- **Base Case**: Always ensure a base case exists to prevent infinite recursion.
- **Stack Depth**: Python has a default recursion limit (usually 1000). Deep recursion may cause a `RecursionError`.
- **Efficiency**: Simple recursive solutions (like Fibonacci) can be inefficient due to repeated calculations.

Hands-On Exercise 6: Calculator Function

Write a function called `multiply` that takes two arguments, calculates their product, and returns the result. Test it by calling it with two numbers.

Part 10: File Operations — Writing

File handling allows programs to persist data on the storage system.

We use the `open()` function along with the `with` statement. The `with` statement ensures the file is properly closed.

Writing to a File

```python
filename = "lab3_log.txt"

with open(filename, "w") as f:
    f.write("System execution started.\n")
    f.write("Initialization complete.\n")

```

File Operations — Reading

Reading from a File

```python
import os

if os.path.exists(filename):
    with open(filename, "r") as f:
        content = f.read()
        print("File Contents:")
        print(content)

```

```
File Contents:
System execution started.
Initialization complete.
```

Hands-On Exercise 7: File Logger

Task

Write a script that asks the user for a sentence and appends it to a file named `my_journal.txt`. Ensure that each new entry opens on a new line.

Conclusion

This laboratory session has covered the essential building blocks of Python programming.

Summary:

- **Environment**: We learned to use `uv` for efficient project management.
- **Syntax**: We explored variables, types, and operational logic.
- **Structure**: We implemented control flows using conditionals and loops.
- **Modularity**: We practiced writing functions and handling files.

Thank you!