SLP1001: A Comprehensive Guide to Sleeping in Class **Spring 2026**

Lecture 0: Python for Data Science

Instructor: Prof. Yitong Guo & Prof. Zhiyan Jin *Date: 19/Jan/2026*

1 Variables, Data Types and Operators

1.1 Variables and Constants

- A variable is a named space in the memory where a programmer can store data and later retrieve the data using the variable name.
- The value of a variable can be changed later in a program.
- You cannot use the following words as variables:

False	None	True	and	as	assert	break
class	continue	def	del	elif	else	except
finally	for	from	global	if	import	in
is	lambda	nonlocal	not	or	pass	raise
return	try	while	with	yield		

- Fixed values such as numbers and letters are called constants, since their values won't change.

1.2 Assignment

In Python, variables are initiated by assigning values to them.

- **Assignment statement:** Assigned values can be retrieved from located memory.
- **Cascaded assignment:** Multiple variables can be set as the same value using single assignment statement.
- **Simultaneous assignment:** Values of two variables can be simultaneously assigned.

```python
# Assigning values to variables
x = 20                  # Assign value 20 to variable x
y = x + 10              # Retrived the value of x from located memory
print(y)                # Output: 30

a = b = c = 2 + 7 + 2   # Cascaded assignment
print(a, b, c)          # Output: (11, 11, 11)

#Simultaneous assignment
WeChatID_g, WeChatID_j = "_MyLittleUniverse", "JZYapril"
```

Listing 1: Assignment Example

1.3 Basic Data Types

In Python, every value has a "type" that tells the computer how to interact with it.

- **Integer (int):** Whole numbers.
- **Float (float):** Numbers with decimal points.
- **String (str):** Text enclosed in quotes.
- **Boolean (bool):** True or False values.

```
# Assigning values to variables
age = 20                # Integer
gpa = 3.8               # Float
name = "Alice"          # String
is_enrolled = True      # Boolean

print(type(gpa))        # Output: <class 'float'>
```

Listing 2: Data Types Example

1.4 Type Conversion (Casting)

Sometimes data is received in the wrong format (e.g., a number stored as text). We can convert between types using casting functions.

- **int():** Converts a value to an integer (truncates decimals).
- **float():** Converts a value to a float.
- **str():** Converts a value to a string (text).

```
# Converting Float to Integer (removes decimal)
price = 9.99
price_int = int(price)      # Result: 9

# Converting String to Number (essential for math on text data)
data_entry = "50"
amount = int(data_entry)    # Result: 50 (now we can do math)

# Converting Number to String (essential for combining text)
score = 95
message = "Your score is " + str(score)  # Result: "Your score is 95"
```

Listing 3: Type Conversion Example

1.5 Operators

We use operators to perform calculations and make comparisons.

- **Arithmetic:** `+` (add), `-` (subtract), `*` (multiply), `/` (divide), `**` (power).
- **Comparison:** `==` (equal), `!=` (not equal), `>` (greater than), `>=` (greater than or equal to).

```python
x = 10
y = 3

result = x + y          # 13
is_equal = (x == y)     # False
power = x ** 2          # 100 (10 squared)
```

Listing 4: Operators Example

2 Input and Output

Programs need to interact with the user. We use `print()` to show results and `input()` to get data from the keyboard.

```python
# Outputting text
print("Welcome to SLP1001!!!!")

# Getting user input (input always returns a String)
user_name = input("Enter your name: ")
print("Hello, " + user_name)
```

Listing 5: Input/Output Example

3 Flow Control and Loops

We use flow control to make decisions (logic) and loops to repeat tasks (iteration).

3.1 Indentation

- Increase after if/for statements.
- Maintain to indicate the scope of the block.
- Decrease to indicate the end of a block.
- Blank lines and comments are ignored.

3.2 Logical Operators

- Logical operators can be used to combine several logical expressions into a single expression.
- Python has three logical operators: not, and, or.
- `not True == False`, `not False == True`, `False and True == False`, `False or True == True`.

3.3 Handle Dangerous Part of Code (try/except)

- You surround a dangerous part of code with try/except.
- If the code in try block works, the except block is skipped.
- If the code in try block fails, the except block will be executed.

```python
my_string = "Hi Yitong, hi Zhiyan."
try:
    convert_string = int(my_string)
except:
    convert_string = -1
print(convert_string)                # The Output is: -1

my_string = "1234"
try:
    convert_string = int(my_string)
except:
    convert_string = -1
print(convert_string)                # The Output is: 1234
```

Listing 6: Try/except to Capture Errors

3.4 Conditional Statements (if/elif/else)

This allows the code to execute different blocks based on conditions.

```python
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 60:
    print("Grade: Pass")
else:
    print("Grade: Fail")
print("Finished")
```

Listing 7: Conditional Logic Example

3.5 Loops (For and While)

- **For Loop:** Iterates over a sequence (like a range of numbers).
- **While Loop:** Keeps running as long as a condition is True.

```python
# For Loop: Print numbers 0 to 4
for i in range(5):
    print(i)

# While Loop: Countdown
count = 3
while count > 0:
    print(count)
    count = count - 1  # Decrease count
```

Listing 8: Loop Examples

Practice: Given a set of numbers, write a program to calculate their sum using for loop.

4 Functions

A function is a reusable block of code that performs a specific task. Using functions keeps our code organized and avoids repetition.

- Built-in functions which are part of Python, such as `print()`, `int()`, `float()`, etc.

- **Definition:** Our own functions can be defined using the `def` keyword.
- **Parameters:** Inputs the function expects.
- **Return:** The result the function sends back.
- **Argument:** An argument is a value we pass into the function as its input when we call.
- **Parameters:** A parameter is a variable which we use in the function definition that is a "handle" that allows the code in the function to access the arguments for a particular function invocation

```python
def calculate_area(width = 1, height = 2):
    """Calculates the area of a rectangle."""
    area = width * height
    return area

# Using the function
rect_area = calculate_area(5, 10)
print(rect_area)  # Output: 50
```

Listing 9: Function Example

5 Lists

A List is a collection of items stored in a single variable. It is ordered and mutable (changeable). This is the fundamental structure for storing datasets.

- **Definition of Lists:** A collection allows us to put many values in a single "variable"
- **Indexing:** Accessing items by position (starting at 0).
- **Appending:** Adding new items to the end.

```python
# Creating a list of numbers
data_points = [10, 20, 30, 45]

# Accessing elements
first_item = data_points[0]  # 10

# Changing an element
data_points[2] = 5           # List becomes [10, 20, 5, 45]

# Adding a new data point
data_points.append(50)       # List becomes [10, 20, 5, 45, 50]

# Length of list can be obtained using len()
print(len(data_points))      # Output: 5

# Slicing list
print(data_points[1:3], data_points[3:])
# Output: [20, 35], [45, 50]

# Sort list
data_points.sort()
print(data_points)           # Output: [2, 10, 20, 45, 50]

# Split a string into a list
my_string = "Zhiyan played baseball last weekend"
print(my_string.split())     # Ouput: ["Zhiyan", "played", "baseball", "last", "weekend"]
```

Listing 10: List Example

Practice: Write a program to instruct the user to input several numbers and calculate their average using list methods.

6 Dictionary

A Dictionary is a powerful data collection that stores values in a "bag" where each item is associated with a specific label (key). Unlike lists, dictionaries are unordered and use keys rather than positions for indexing.

- **Key-Value Pairs:** Dictionaries consist of literals surrounded by curly braces containing `key:value` pairs.
- **Fast Lookup:** They allow for fast, database-like operations in Python.
- **The get() Method:** This method checks if a key exists and returns a default value if it is not found, preventing errors.
- **Iterating:** You can loop through a dictionary to access keys, values, or both as tuples using `.items()`.

```python
# Creating a dictionary (literals use curly braces)
counts = {"chuck": 1, "fred": 42, "jan": 100}

# Accessing values based on keys
print(counts["chuck"])            # Output: 1

# Adding or updating an element
counts["age"] = 21
counts["age"] = 23                # Value is over-written

# The get() method for safe lookups
# Returns 0 since "eee" does not exist
print(counts.get("eee", 0))       # Output: 0

# Getting lists of keys and values
print(list(counts.keys()))        # ["jan", "fred", "chuck", "age"]
print(list(counts.values()))      # [100, 42, 1, 23]
# Iterating through key-value pairs
for key, value in counts.items():
    print(key, value)
```

Listing 11: Dictionary Example

Practice: Write a program that sorts the elements of a dictionary by the value of each element rather than the key.

7 Object Oriented Programming

Object-Oriented Programming (OOP) is a paradigm where a program is composed of many cooperating objects that make use of each other's capabilities. In Python, everything—including numbers and strings—is treated as an object.

7.1 Objects and Classes

An **object** represents a uniquely identifiable entity in the real world (e.g., a student, a circle, or even a loan). Every object has three key characteristics:

- **Identity:** A unique integer ID assigned at runtime by Python (retrieved using `id()`).
- **Attributes:** Data fields represented by variables that store the object's properties.

- **Methods:** Functions defined within a class that allow an object to perform actions.

A **class** serves as a blueprint or template that defines the variables and methods common to all objects of the same kind. Creating a specific object from a class is known as **instantiation**.

7.2 Class Definition and `self`

Python classes use the `__init__()` method, known as the **initializer**, to set the initial state of an object upon creation. All methods within a class must include `self` as their first parameter, which refers to the specific instance invoking the method.

```python
import math

class Circle:
    # Initializer to construct a circle object
    def __init__(self, radius=1):
        self.radius = radius  # Instance variable

    def getArea(self):
        return self.radius * self.radius * math.pi

# Creating an instance (instantiation)
my_circle = Circle(5)
print(my_circle.radius)      # Accessing a data field
print(my_circle.getArea())   # Invoking an instance method
```

Listing 12: Class Definition Example

7.3 Information Hiding (Private Fields)

Directly accessing data fields is discouraged as it makes code vulnerable to bugs. **Data hiding** prevents direct external access by using two leading underscores (`__`) to define private data fields. These can only be accessed via "getter" and "setter" methods.

```python
class Circle:
    def __init__(self, radius=1):
        self.__radius = radius  # Private data field

    def getRadius(self):        # Getter method
        return self.__radius
```

Listing 13: Private Data Fields

7.4 Inheritance

Inheritance allows you to define a general class (**superclass**) and extend it into specialized classes (**subclasses**) that inherit its properties and methods.

- **Method Overriding:** A subclass can modify a method implementation defined in its superclass to suit its specific needs.
- **Dynamic Binding:** Python decides which method implementation to invoke at runtime by searching from the most specific class up to the most general (the `object` class).

```python
class GeometricObject:
    def __init__(self, color="green"):
        self.color = color

class Circle(GeometricObject):  # Circle inherits GeometricObject
    def __init__(self, radius, color):
        super().__init__(color) # Initialize superclass properties
        self.radius = radius
```

Listing 14: Inheritance Syntax

Practice: Design a `Rectangle` class with data fields for `width` and `height`, providing methods for `getArea()` and `getPerimeter()`.

8 Basic Linear Algebra

8.1 Vectors and Matrices

- **Vector (v):** A 1D array of numbers. Geometrically, it represents a point or an arrow in space.

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} \in \mathbb{R}^m$$

- **Matrix (A):** A 2D array of numbers. It can be viewed as a collection of column vectors.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} | & | & | & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & | & | \end{bmatrix} = \begin{bmatrix} - & r_1 & - \\ - & r_2 & - \\ - & \vdots & - \\ - & r_m & - \end{bmatrix} \in \mathbb{R}^{m \times n}$$

8.2 Vector-Vector Multiplication (Dot Product)

The dot product of two vectors returns a single scalar value. It measures how much two vectors point in the same direction.

$$a^\top \cdot b = \sum a_i b_i = a_1 b_1 + a_2 b_2 + \dots$$

Example:

$$\begin{bmatrix} 1 \\ 3 \end{bmatrix}^\top \cdot \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 2 \end{bmatrix} = (1 \times 4) + (3 \times 2) = 4 + 6 = 10$$

8.3 Matrix-Vector Multiplication

When we multiply a matrix A by a vector x, we are transforming the vector.

Visualization: weighted summation of the columns of the matrix. The values in the vector x act as "weights" for the columns of A.

$$Ax = \begin{bmatrix} | & | & | & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1 c_1 + x_2 c_2 + \cdots + x_n c_n$$

Example:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} \mathbf{2} \\ \mathbf{5} \end{bmatrix} = \mathbf{2} \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \mathbf{5} \times \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 15 \\ 20 \end{bmatrix} = \begin{bmatrix} 17 \\ 24 \end{bmatrix}$$

8.4 Matrix-Matrix Multiplication

Matrix multiplication is essentially performing Matrix-Vector multiplication multiple times (once for each column of the second matrix). Let $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times r}$,

$$B = \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \cdots & b_r \\ | & | & & | \end{bmatrix}, AB = \begin{bmatrix} Ab_1, Ab_2, \cdots, Ab_r \end{bmatrix}$$

Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, AB = \begin{bmatrix} A \begin{bmatrix} 5 \\ 5 \end{bmatrix}, A \begin{bmatrix} 6 \\ 8 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

8.5 Transpose

The transpose of a matrix, denoted as $A^\top$, is created by flipping the matrix over its main diagonal. The rows of A become the columns of $A^\top$.

If $A \in \mathbb{R}^{m \times n}$, then $A^\top \in \mathbb{R}^{n \times m}$.

Example:

$$A = \begin{bmatrix} \mathbf{1} & 2 & 3 \\ 4 & \mathbf{5} & 6 \end{bmatrix} \xrightarrow{\text{Transpose}} A^\top = \begin{bmatrix} \mathbf{1} & 4 \\ 2 & \mathbf{5} \\ 3 & 6 \end{bmatrix}$$

8.6 Identity Matrix and Inverse

8.6.1 The Identity Matrix (I)

In scalar math, the number 1 is the "multiplicative identity" because $5 \times 1 = 5$. In linear algebra, we have the **Identity Matrix** (I). It is a square matrix with 1s on the main diagonal and 0s everywhere else.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying any matrix A by I leaves A unchanged: $AI = A$.

8.6.2 Matrix Inverse (A^{-1})

The inverse of a square matrix A is a unique matrix denoted as A^{-1}. When a matrix is multiplied by its inverse, the result is the Identity Matrix.

$$AA^{-1} = I$$

This is conceptually similar to reciprocals in scalar math ($5 \times \frac{1}{5} = 1$).

Example: Let $A = \begin{bmatrix} 2 & 1 \\ 5 & 3 \end{bmatrix}$. The inverse is $A^{-1} = \begin{bmatrix} 3 & -1 \\ -5 & 2 \end{bmatrix}$.

Verification:

$$AA^{-1} = \begin{bmatrix} 2 & 1 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ -5 & 2 \end{bmatrix} = \begin{bmatrix} (2)(3) + (1)(-5) & (2)(-1) + (1)(2) \\ (5)(3) + (3)(-5) & (5)(-1) + (3)(2) \end{bmatrix} = \begin{bmatrix} 6 - 5 & -2 + 2 \\ 15 - 15 & -5 + 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

9 Introduction to Numpy

In this section we will see how we translate these linear algebra concepts into code using Numpy.

9.1 Setup

```python
import numpy as np

# Define Vectors
v1 = np.array([1, 3])
v2 = np.array([4, 2])
x = np.array([2, 5])

# Define Matrices
A = np.array([[1, 3],
              [2, 4]])
B = np.array([[5, 6],
              [7, 8]])
```

9.2 Operations

9.2.1 Vector-Vector (Dot Product)

```python
# Method 1: using @ operator
dot_prod = v1 @ v2   # 1*4 + 3*2 = 10

# Method 2: using np.dot
dot_prod_2 = np.dot(v1, v2)
```

9.2.2 Matrix-Vector

Note how the dimensions change: $(2 \times 2) \cdot (2 \times 1) \to (2 \times 1)$.

```python
# Result will be a new vector
transformed_vector = A @ x
# Result: array([17, 24])
```

9.2.3 Matrix-Matrix

```python
# Matrix Multiplication
C = A @ B
# Result:
# [[19, 22],
#  [43, 50]]
```

9.2.4 Transpose

We use the `.T` attribute to flip dimensions.

```
# Rows become columns
A_transposed = A.T
# Result: [[1, 2], [3, 4]]
```

9.2.5 Matrix Inverse

We use `np.linalg.inv()` to find the matrix that reverses A.

```
# Calculate Inverse
A_inv = np.linalg.inv(A)

# Verification: A @ A_inv should be Identity Matrix
identity_check = A @ A_inv
# Result: [[1., 0.], [0., 1.]]
```

9.3 Common Data Science Functions in Numpy

Beyond linear algebra, Numpy provides essential tools for statistical analysis and data generation that we will use frequently before moving to Pandas.

9.3.1 Descriptive Statistics

```
data = np.array([10, 20, 30, 40, 50])

# Basic Stats
print(np.mean(data))   # Average: 30.0
print(np.median(data)) # Median: 30.0
print(np.std(data))    # Standard Deviation: 14.14...
print(np.max(data))    # Maximum value
```

9.3.2 Filtering and Logic (np.where)

This is the Numpy equivalent of "If-Else" for entire arrays. It is crucial for cleaning data.

```
scores = np.array([85, 40, 90, 55])

# Syntax: np.where(condition, value_if_true, value_if_false)
results = np.where(scores >= 60, "Pass", "Fail")
# Result: ['Pass', 'Fail', 'Pass', 'Fail']
```

9.3.3 Unique Values

Useful for finding distinct categories in a dataset.

```
labels = np.array(["cat", "dog", "cat", "bird", "dog"])
unique_labels = np.unique(labels)
# Result: ["bird", "cat", "dog"]
```

9.3.4 Random Sampling (More Examples Later)

Essential for splitting data into ”Training” and ”Testing” sets later in the course.

```
# Generate 5 random numbers between 0 and 1
rand_nums = np.random.rand(5)

# Generate 3 random integers between 0 and 100
rand_ints = np.random.randint(0, 100, 3)
```

10 Introduction to Pandas

While Numpy is excellent for numerical calculation, real-world data often contains mixed types (text, dates, numbers) and missing values. In the next section, we will introduce **Pandas**, which is built on top of Numpy but designed for:

- **DataFrames:** Tabular data structures (like Excel sheets).
- **Data Loading:** Reading CSV, Excel, and JSON files.
- **Data Wrangling:** Handling missing data, merging datasets, and string manipulation.

10.1 Core Data Structures

10.1.1 Series

A Series is a one-dimensional labeled array. Unlike Numpy arrays, items in a Series can be indexed by labels (names) instead of just integers.

```
import pandas as pd
import numpy as np

# Creating a Series with custom index labels
data = pd.Series([10, 20, 30], index=["a", "b", "c"])

print(data["a"])  # Access by label (Output: 10)
```

10.1.2 DataFrame

A DataFrame is a 2D table with rows and columns. It is essentially a collection of Series sharing the same index.

```
# Creating a DataFrame from a Dictionary
df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["New York", "London", "Paris"]
})

print(df)
# Output:
#       Name  Age      City
# 0    Alice   25  New York
# 1      Bob   30    London
# 2  Charlie   35     Paris
```

Listing 15: Creating a DataFrame

10.2 Essential Functionality

10.2.1 Inspecting Data

Before analyzing, we must understand the structure of our data.

```python
# View the first 2 rows
print(df.head(2))
# Output:
#     Name  Age      City
# 0  Alice   25  New York
# 1    Bob   30    London

# Get summary of data types and non-null counts
print(df.info())
# Output:
# <class 'pandas.core.frame.DataFrame'>
# Data columns (total 3 columns): ...

# Get statistical summary for numeric columns
print(df.describe())
# Output:
#              Age
# count   3.000000
# mean   30.000000 ...
```

10.3 Selection and Indexing

Pandas provides two powerful methods for selecting data: `.loc` (Label-based) and `.iloc` (Integer-based).

```python
# .loc: Select by Column Name
ages = df.loc[:, "Age"]
# Result: Series of [25, 30, 35]

# .iloc: Select by Position (Row 0, Column 1)
first_age = df.iloc[0, 1]
# Output: 25
```

10.4 Missing Data

Real-world data is rarely clean. Missing values are represented as `NaN` (Not a Number).

```python
# DataFrame with missing values
df_missing = pd.DataFrame({
    "A": [1, 2, np.nan],
    "B": [5, np.nan, np.nan]
})

# 1. Detect missing values (Returns Boolean DataFrame)
print(df_missing.isna())
# Output:
#        A      B
# 0  False  False
# 1  False   True
# 2   True   True

# 2. Drop rows with ANY missing values
df_clean = df_missing.dropna()
# Result: Only row 0 remains

# 3. Fill missing values (e.g., with 0)
df_filled = df_missing.fillna(0)
```

```
# Result: All NaNs replaced with 0.0
```

Listing 16: Handling Missing Values

10.5 Data Loading and File Systems

10.5.1 Reading and Writing Text Data

The most common format for data storage is CSV (Comma Separated Values).

```
# Reading a CSV file
df = pd.read_csv("sample_data.csv")

# Writing to a CSV file (index=False prevents saving row numbers)
df.to_csv("output_data.csv", index=False)
```

Listing 17: Loading Data

10.5.2 Other Types Data Loading

- **Excel:** `pd.read_excel("data.xlsx", sheet_name="Sheet1")`
- **JSON:** `pd.read_json("data.json")`
- **SQL:** `pd.read_sql(query, connection)`

11 Matplotlib for Visualization (More Examples Later)

Matplotlib is the most widely used visualization library in Python. It provides control over every aspect of a figure. In this section, we categorize plots based on the data relationships they visualize.

11.1 Basic Operations

In Matplotlib, a plot consists of a **Figure** (the overall window/page) and one or more **Axes** (the individual plots inside).

```
import matplotlib.pyplot as plt

# 1. Create Figure and Axes
fig, ax = plt.subplots(2, 2)

# 2. Plot
ax[0, 0].text(0.5, 0.5, "I am Zhiyan Jin", fontsize=20, ha='center')

ax[0, 1].text(0.5, 0.5, "I am previewing for \n tomorrow's final.", fontsize=15, ha='center'
    )

ax[1, 0].text(0.5, 0.5, "I am Yitong Guo", fontsize=20, ha='center')

ax[1, 1].text(0.5, 0.5, "I'm invincible \n when staying up late.", fontsize=15, ha='center')

# 3. Show
plt.show()
```

Listing 18: Basic Line Plot

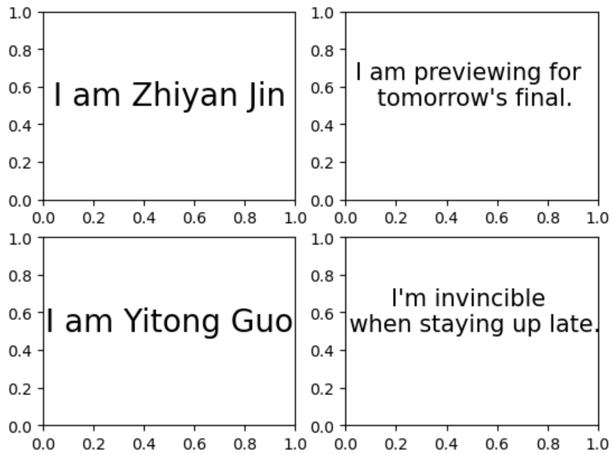

Figure 1: Results of the Code Above

11.2 Pairwise Data

These plots visualize the relationship between variables (usually x and y).

- **plot(x, y):** Connects points with lines. Best for time series or mathematical functions.
- **scatter(x, y):** Draws unconnected points. Best for finding correlations between two variables.
- **bar(x, height):** Vertical rectangles. Best for comparing categorical data.

```
import matplotlib.pyplot as plt
import numpy as np

# Prepare Data
x = np.linspace(0, 10, 10)
y = 2 * x + 1

# Create Figure
fig, ax = plt.subplots(3)

# 1. Plot (Line)
ax[0].plot(x, y, linewidth=2.0, label="Line Plot")
ax[0].legend()

# 2. Scatter (Points)
noise = np.random.normal(0, 2, 10)
ax[1].scatter(x, y + noise, color="blue", label="Scatter Plot")
ax[1].legend()

# 3. Bar (Categorical)
categories = ["A", "B", "C", "D", "E"]
values = [5, 7, 3, 8, 4]
ax[2].bar(categories, values, label="Bar Plot", color="green")
ax[2].legend()

plt.show()
```

Listing 19: Pairwise Plots Example

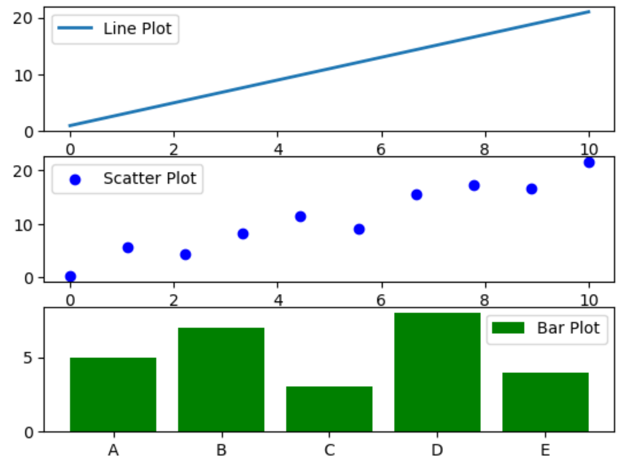

Figure 2: Results of the Code Above

11.3 Statistical Distributions

These plots help us understand the spread, central tendency, and outliers of a dataset.

- **hist(x):** Histogram. Bins data to show frequency distribution.
- **boxplot(X):** Shows the median, quartiles (25%, 75%), and outliers.
- **pie(x):** Proportional sectors (use sparingly in data science).

```
data = np.random.randn(1000)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Histogram
ax1.hist(data, bins=100, edgecolor="white")
ax1.set_title("Histogram (Distribution)")

# Boxplot
ax2.boxplot(data)
ax2.set_title("Boxplot (Outliers)")

plt.show()
```

Listing 20: Statistical Plots Example

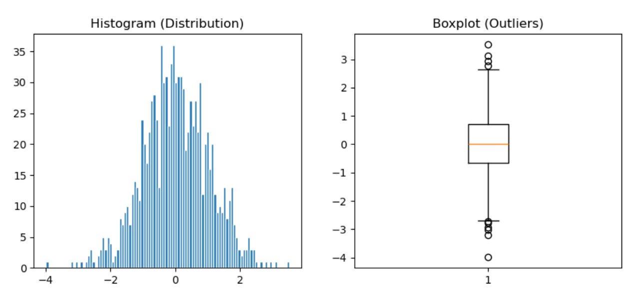

Figure 3: Results of the Code Above

12 Advanced Visualization: Seaborn (More Examples Later)

While Matplotlib handles the low-level drawing, **Seaborn** is a high-level library built on top of Matplotlib. It is specifically designed for statistical data exploration and works seamlessly with Pandas DataFrames.

```python
import seaborn as sns
sns.set_theme(style="ticks")

# Load the penguins dataset
penguins = sns.load_dataset("penguins")

# Show the joint distribution using kernel density estimation
g = sns.jointplot(
    data=penguins,
    x="bill_length_mm", y="bill_depth_mm", hue="species",
    kind="kde",
)
```

Listing 21: Seaborn Scatter Plot

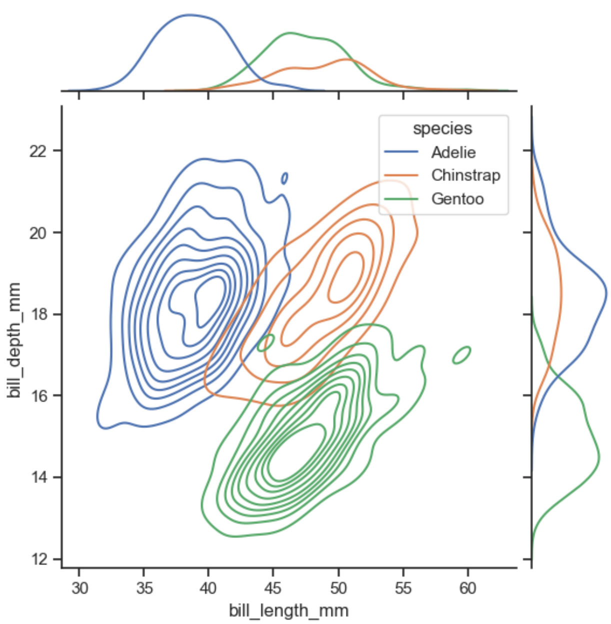

Figure 4: Results of the Code Above