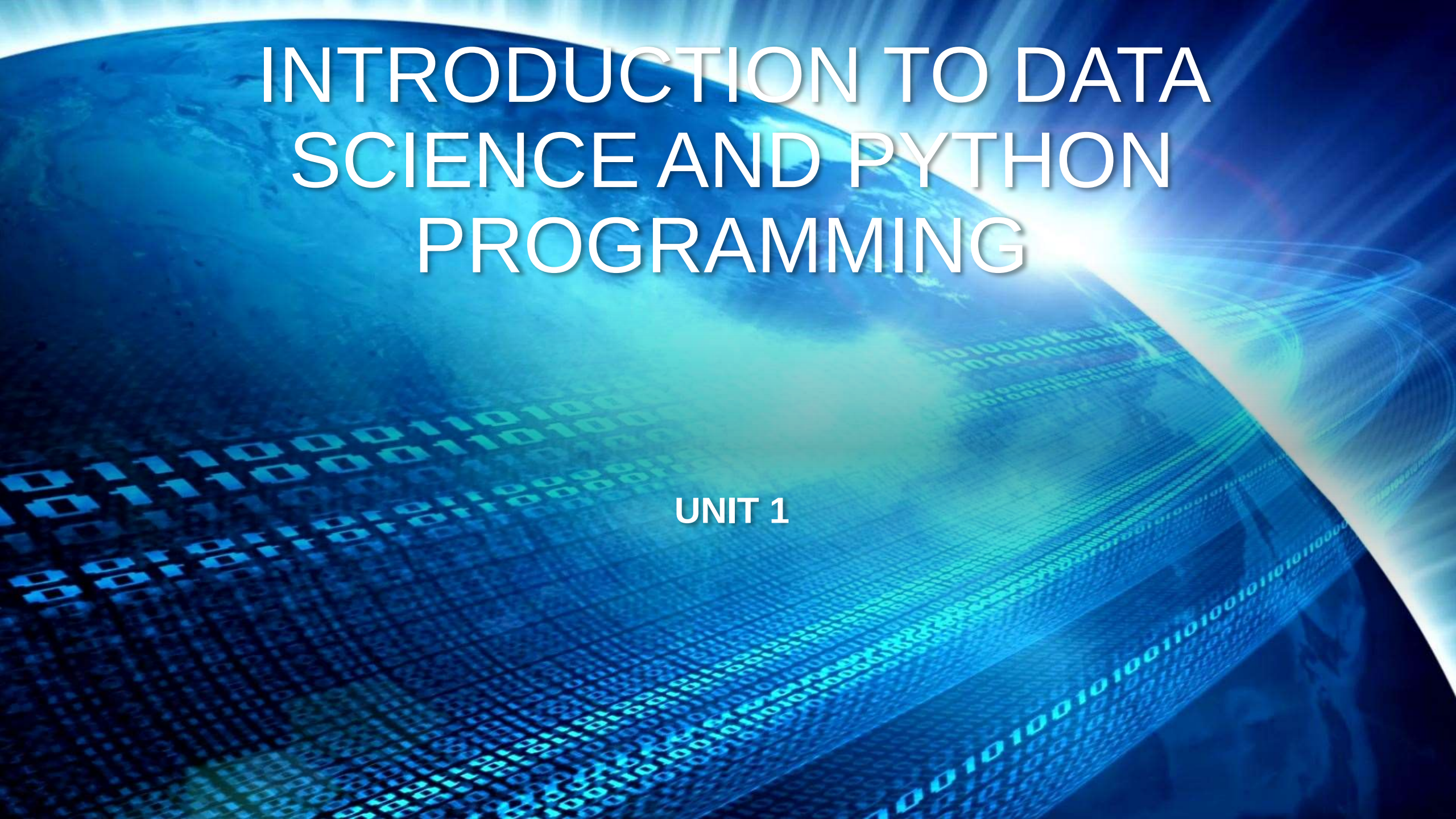

INTRODUCTION TO DATA SCIENCE AND PYTHON PROGRAMMING
UNIT 1

Syllabus

UNIT 1: INTRODUCTION TO DATA SCIENCE AND PYTHON PROGRAMMING

Introduction to Data Science - Why Python? - Essential Python libraries - Python Introduction- Features, Identifiers, Reserved words, Indentation, Comments, Built-in Data types and their Methods: Strings, List, Tuples, Dictionary, Set - Type Conversion- Operators. Decision Making- Looping- Loop Control statement- Math and Random number functions. User defined functions - function arguments & its types.

UNIT 2: FILE, EXCEPTION HANDLING AND OOP

User defined Modules and Packages in Python- Files: File manipulations, File and Directory related methods- Python Exception Handling. OOPs Concepts -Class and Objects, Constructors – Data hiding- Data Abstraction- Inheritance.

UNIT 3: INTRODUCTION TO NUMPY

NumPy Basics: Arrays and Vectorized Computation- The Num Pynd array- Creating ndarrays- Data Types fornd arrays- Arithmetic with Num Py Arrays- Basic Indexing and Slicing - Boolean Indexing-Transposing Arrays and Swapping Axes. Universal Functions: Fast Element-Wise Array Functions- Mathematical and Statistical Methods- Sorting-Unique and Other Set Logic.

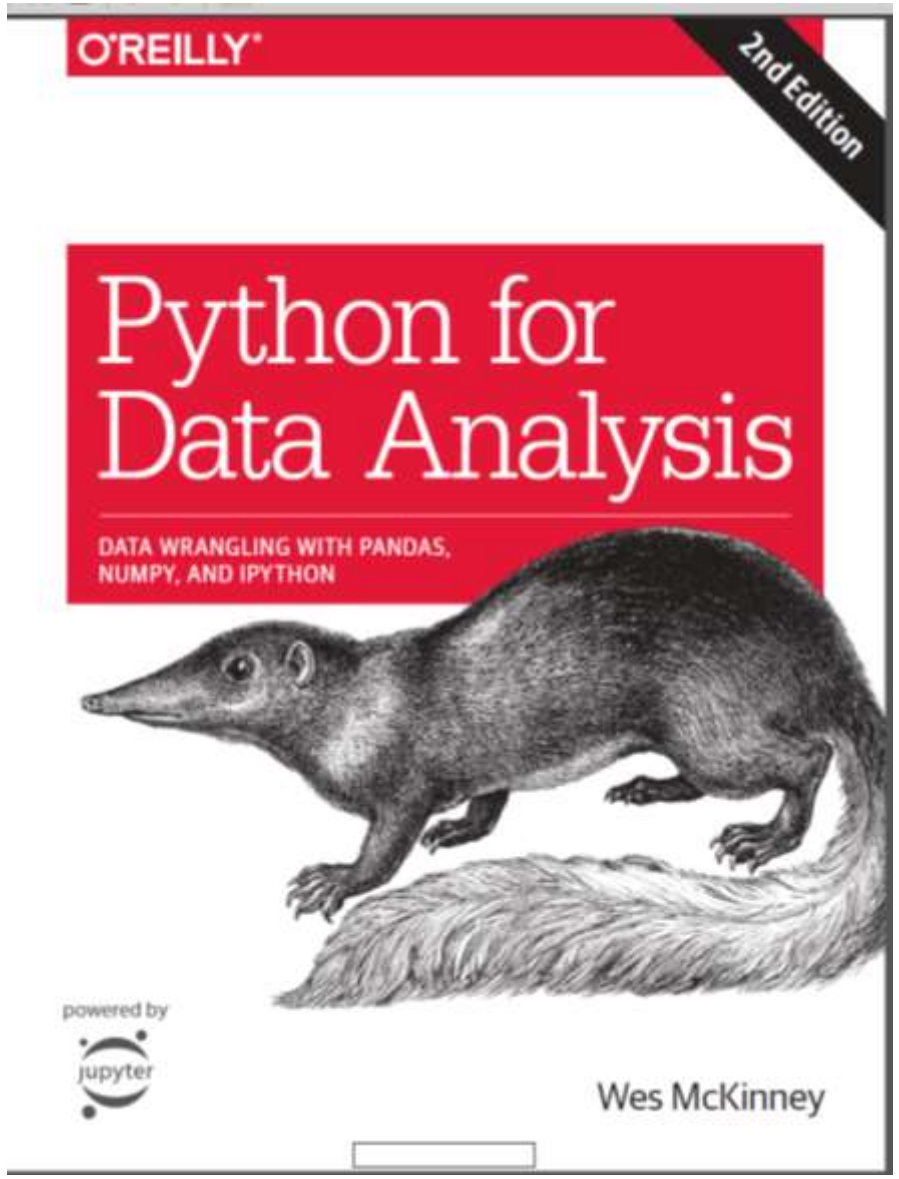
UNIT 4: DATA MANIPULATION WITH PANDAS

Introduction to pandas Data Structures: Series, DataFrame, Essential Functionality: Dropping Entries- Indexing, Selection, and Filtering- Function Application and Mapping- Sorting and Ranking. Summarizing and Computing Descriptive Statistics- Unique Values, Value Counts, and Membership. Reading and Writing Data in Text Format.

UNIT 5: DATA CLEANING, PREPARATION AND VISUALIZATION

Data Cleaning and Preparation: Handling Missing Data - Data Transformation: Removing Duplicates, Transformin g Data Using a Function or Mapping, Replacing Values, Detecting and Filtering Outliers- String Manipulation: Vectorized String Functions in pandas. Plotting with pandas: Line Plots, Bar Plots, Histograms and Density Plots, Scatter or Point Plots

The text book



CONTENT

Introduction to Data Science - Why Python?

Essential Python libraries

Python Introduction Features:

- Identifiers
- Reserved words
- Indentation
- Comments

Built-in Data types and their Methods

Strings, List, Tuples, Dictionary, Set - Type Conversion- Operators.

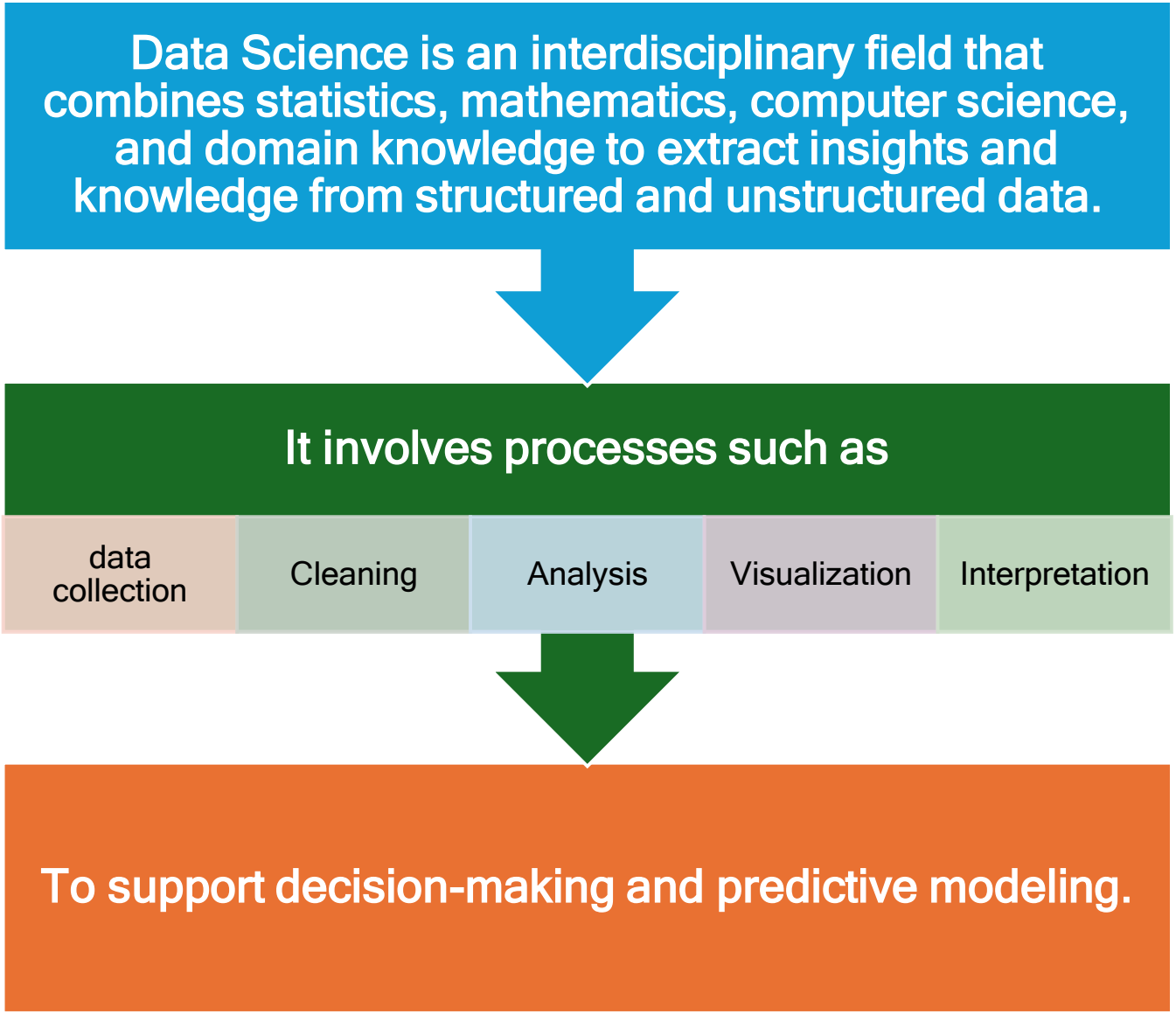
Decision Making

Looping- Loop Control statement- Math and Random number functions.

User defined functions

function arguments & its types.

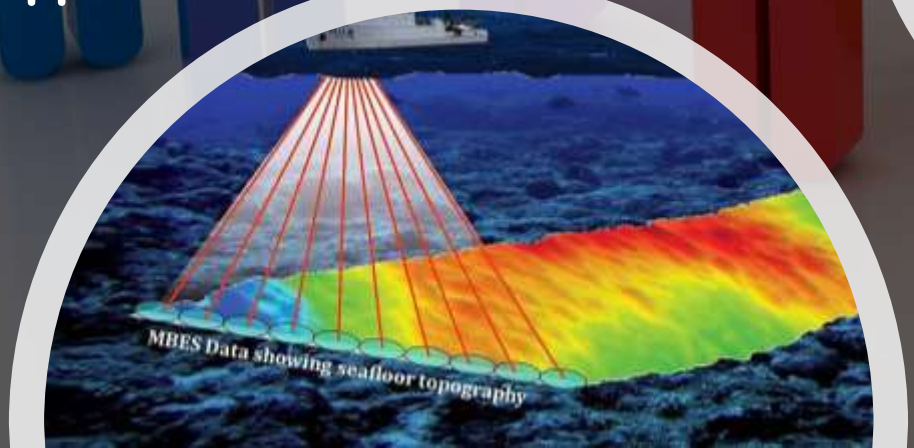
What is Data Science?



Where?

- People's daily lives
  - 4 billion internet users
  - Social media, smart devices, …
- Scientific discovery
  - Rubin Observatory: 20TB/night

Many application domains…



Digital Era

A lot of digital information.

We are interacting with various types of digital devices.

More than four billion Internet users nowadays, and they are a wide variety of social media or smart devices that we interact on a daily basis.

We are creating a lot of digital content and also many different types of apps, and behind the thing, there's more data involved.

scientific discovery. In those domains, we are seeing increasingly a number of sensing devices and also increasing sensing capabilities.

As a result, We are continuously generating a lot of information for scientific discovery.

When?


COLLECTION
STORAGE
NETWORK
RESEARCH
BIG DATA
CLOUD TECHNOLOGY
ANALYSIS
VISUALIZATION
EXABITE
VOLUME

What make a data big data?

Volume:	Velocity:	Variety:	Veracity:	Value:
Refers to the vast amounts of data generated every second. Big data often involves terabytes or even petabytes of information.	The speed at which data is generated, collected, and analyzed. This includes real-time data streams and the need for quick processing.	The different types of data, including structured, semi-structured, and unstructured data from various sources like text, images, video, and more.	The accuracy and reliability of the data. Big data comes with a lot of noise and uncertainty, requiring robust methods for data validation.	The potential insights and benefits that can be derived from the data. The true value lies in the ability to turn vast amounts of data into actionable insights.

How?

- Open source
- Interpreted
- simple syntax
- Object-oriented
- Large standard Library

- High level
- GUI programming
- Portable
- Scalable
- Easy to write

Why Python for Data Analysis?

Ease of Learning and Use

Python has a simple and readable syntax, making it easy for beginners and experienced developers to work with data.

It allows quick prototyping and experimentation, which is crucial for data science.

Rich Ecosystem of Libraries

Python provides powerful libraries for **data manipulation, analysis, and visualization**, such as:

- **NumPy** – Numerical computing and array operations.
- **pandas** – Data manipulation and analysis with DataFrames.
- **Matplotlib & Seaborn** – Data visualization and exploratory analysis.
- **scikit-learn** – Machine learning and statistical modeling.
- **TensorFlow & PyTorch** – Deep learning frameworks.

Open Source and Community Support

Python is **free** and open source, with a large community contributing to its development.

A vast number of online resources, tutorials, and forums (e.g., Stack Overflow, GitHub) make troubleshooting and learning easier.

Why Python for Data Analysis?

Versatility Across Domains

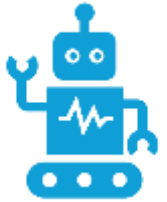
Used not only in **data science** but also in **web development, automation, cybersecurity, and more**.

Integration with **big data tools** (Apache Spark, Hadoop) and cloud platforms (AWS, Google Cloud, Azure).

Integration with Other Languages & Tools

Python works well with **R, SQL, C, Java**, and other technologies.

Compatible with Jupyter Notebook, which allows **interactive data exploration and visualization**.



Industry Adoption

Python is used by companies like **Google, Facebook, Netflix, and NASA** for AI, analytics, and automation.

Strong support from academia and industry ensures continued growth and innovation.

Solving the "Two-Language" Problem with Python

Many organizations traditionally use two different languages for research and production:

Specialized languages (e.g., R, SAS) for **prototyping and data analysis**	**Compiled languages** (e.g., Java, C++) for **building scalable production systems**

Python bridges this gap by serving **both purposes**:

It is **easy to prototype** in Python due to its expressive syntax and extensive data science libraries.	It is also **suitable for production** as it supports integration with databases, web applications, and cloud platforms.	Using Python across the entire workflow **reduces complexity and maintenance costs**, allowing researchers and software engineers to collaborate more efficiently.

Limitation

- **Performance Limitations**
  - Python is an **interpreted** language, so it generally runs **slower** than compiled languages like **C++ or Java**.
  - For tasks requiring **high-speed execution** (e.g., real-time systems, high-frequency trading), compiled languages are often preferred.
- **Global Interpreter Lock (GIL) and Multithreading**
  - The **GIL (Global Interpreter Lock)** restricts Python from executing multiple CPU-bound threads **simultaneously** in a single process.
  - This makes **multithreaded performance suboptimal** for CPU-intensive tasks.
  - However, Python can achieve **parallelism** using:
    - **Multiprocessing** (spawning separate processes instead of threads)
    - **C extensions** (leveraging C/C++ libraries that bypass the GIL)
- **Not Ideal for Some Low-Level Applications**
  - Applications that require **low-level hardware interaction**, **real-time processing**, or **embedded systems** may not be best suited for Python.
  - C, C++, or Rust are often preferred in such cases.

Workarounds for Python's Limitations

Use Just-in-Time (JIT) Compilation:

- Libraries like **Numba** and **PyPy** optimize Python execution speed.

Leverage Compiled Extensions:

- Python integrates well with C/C++ using **Cython** or **JIT compilers**.

Parallel Processing Frameworks:

- Python supports **Dask, Ray, and Spark** for distributed computing.

Hybrid Approach:

- Critical components can be written in C++ while the rest of the application remains in Python.

Essential Python Libraries for Data Science

Core Scientific Computing Libraries

NumPy – Numerical computing, arrays, linear algebra
SciPy – Scientific computing, signal processing, optimization

Data Manipulation & Analysis

pandas – Data manipulation, tabular data (DataFrames)
Dask – Scalable parallel computing for large datasets

Data Visualization

Matplotlib – Basic charts (line plots, bar charts, histograms)
Seaborn – Statistical visualization (heatmaps, violin plots)
Plotly – Interactive visualizations (3D plots, dashboards)

Machine Learning & AI

scikit-learn – ML algorithms (regression, classification, clustering)
TensorFlow & PyTorch – Deep learning frameworks
XGBoost & LightGBM – Gradient boosting for structured data

Big Data & Distributed Computing

Apache Spark (PySpark) – Big data processing
Dask – Parallel computing
Vaex – Fast analysis of large datasets

Essential Python Libraries for Data Science

Natural Language Processing (NLP)

NLTK - Basic text processing
spaCy - Fast NLP (Named Entity Recognition, tokenization)
Transformers (Hugging Face) - State-of-the-art deep learning models

Time Series Analysis

statsmodels - Statistical modeling & forecasting
Prophet (Facebook) - Easy-to-use forecasting

Web Scraping & Automation

BeautifulSoup - HTML parsing
Scrapy - Web scraping framework
Selenium - Browser automation

Database & File Handling

SQLAlchemy - Database ORM
PyODBC & psycopg2 - Connect to SQL databases
h5py - Handling large datasets in HDF5 format

Deployment & API Development

Flask & FastAPI - Building APIs for ML models
Streamlit & Dash - Interactive data science web apps

The Programming Cycle for Python

Write/Edit

Run
with some input

OK?

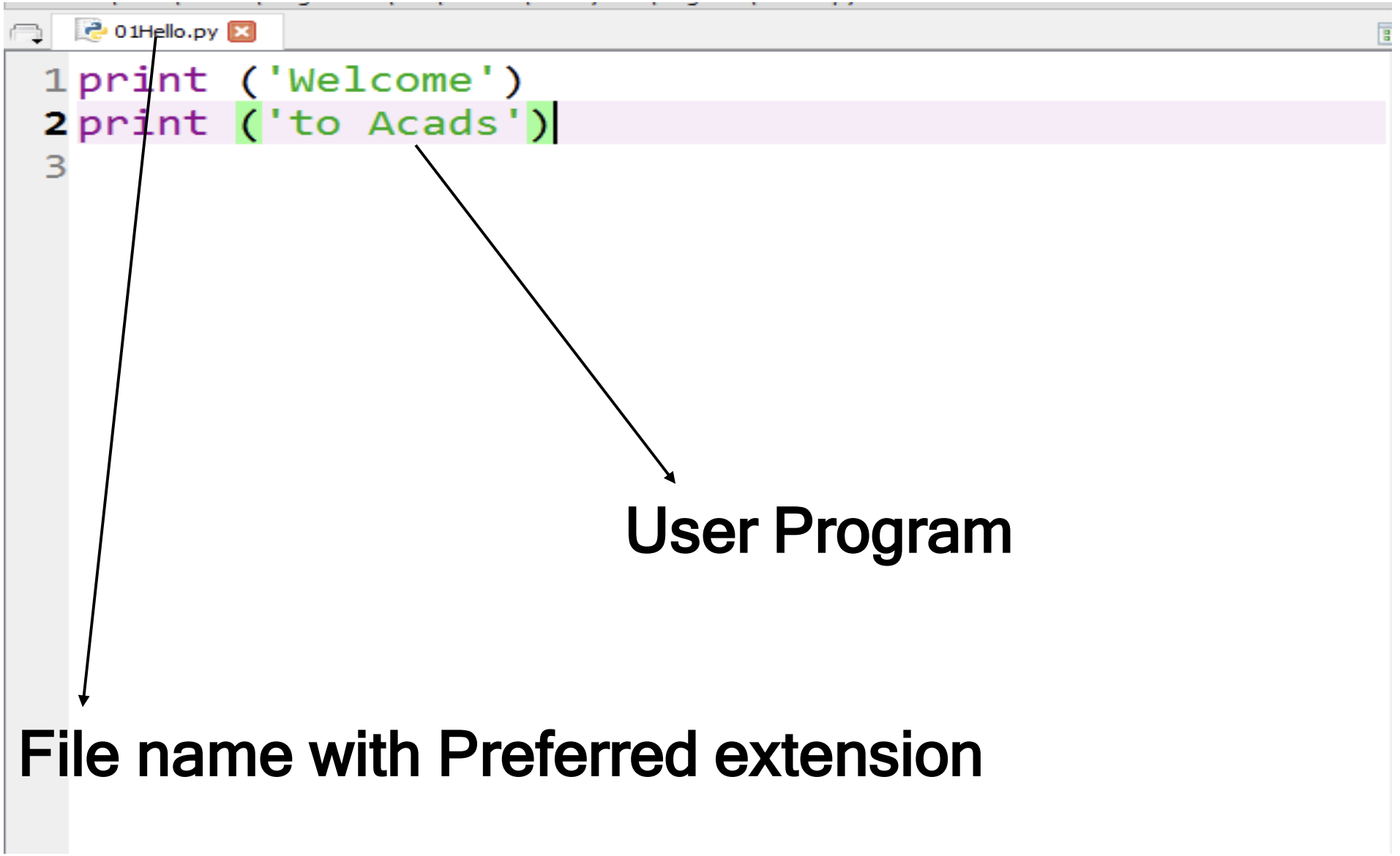
NO

YES

NO

More
Inputs?

YES

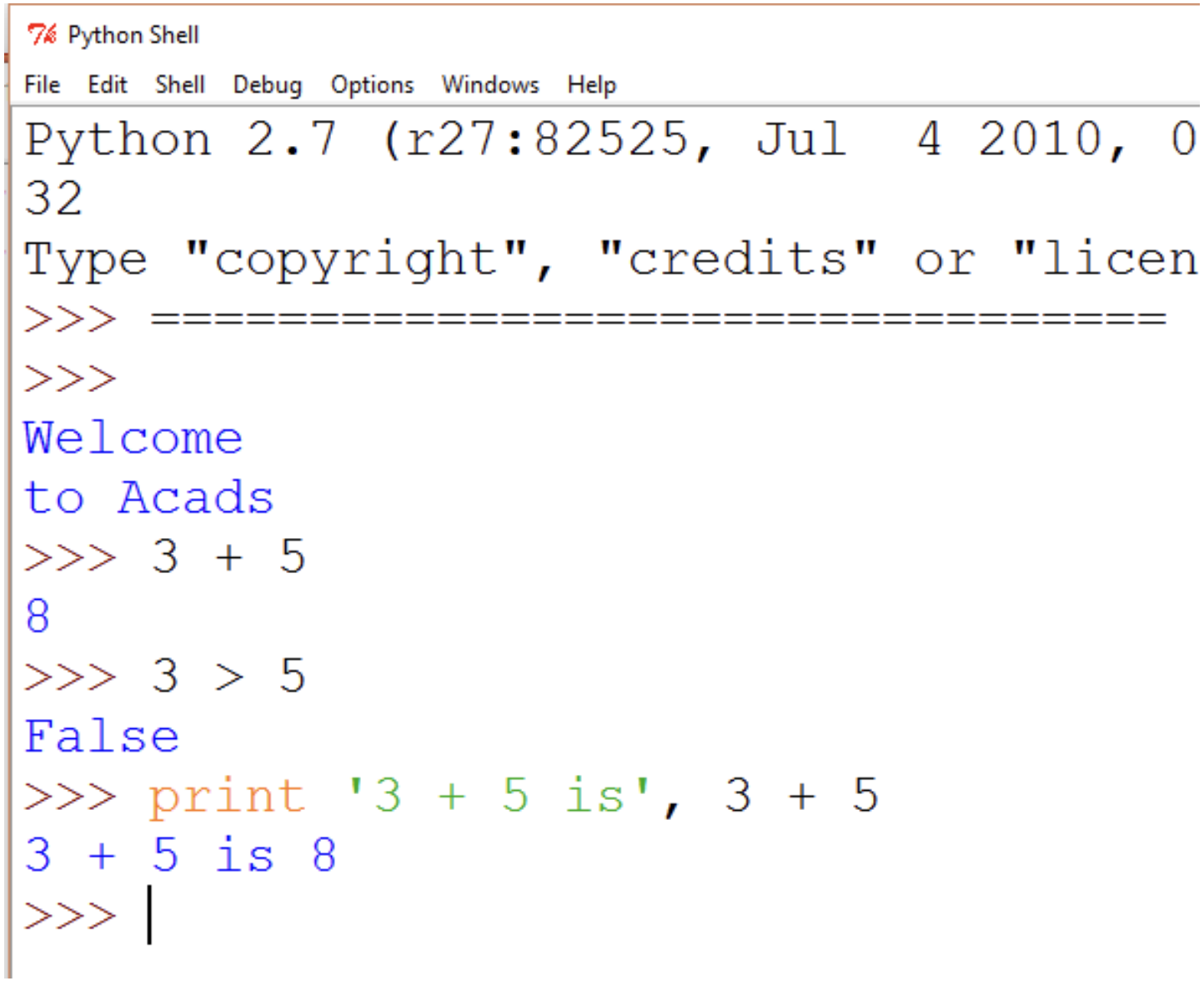

01Hello.py
1 print ('Welcome')
2 print ('to Acads')
3
User Program
File name with Preferred extension

Running Python in the Standard Interpreter

- Python is an **interpreted language**, meaning it executes code **line by line**.
- You can start Python in an interactive shell by running:

```
python
    >>> a = 5
    >>> print(a)
            5
```

- To **exit** the Python shell:
  - Type `exit()` or
  - Press `Ctrl + D`


Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7 (r27:82525, Jul 4 2010, 0
32
Type "copyright", "credits" or "licen
>>> ==========
>>>
Welcome
to Acads
>>> 3 + 5
8
>>> 3 > 5
False
>>> print '3 + 5 is', 3 + 5
3 + 5 is 8
>>>

Running Python in the Standard Interpreter

- **Running Python Scripts**
  - Instead of running code interactively, you can save it in a **.py file** and execute it.
  - Example: Create a file `hello_world.py` with this content:

```
print('Hello world')
```

- Run the script from the terminal:

```
$ python hello_world.py
```

Output:

```
Hello world
```

```
Edit
```

- `$ jupyter notebook`
-
- It opens a **browser-based interface** where you can write and execute Python code in cells

Using IPython (Enhanced Interactive Python)

- IPython is an **improved interactive shell** that provides:
  - Better debugging tools
  - Code completion
  - Magic commands
- Launch IPython by typing:

```
$ ipython
```

- To run a script in **IPython**:

```
In [1]: %run hello_world.py
Hello world
```

- **Differences from the standard Python shell:**
  - IPython uses **In [x]:** instead of >>>
  - It allows **interactive exploration of variables** after script execution

Using Jupyter Notebooks

Jupyter notebooks provide an **interactive web-based coding environment**

Supports code execution, visualization, and documentation using **Markdown**

Start Jupyter by running:

```
$ jupyter notebook
```

It opens a **browser-based interface** where you can write and execute Python code in cells

Interacting with Python Programs

Python program communicates its results to user using print

Most useful programs require information from users

Name and age for a travel reservation system	Python 3 uses input to read user input as a string (str)

Elements of Python

A Python program is a sequence of **definitions** and **commands** (statements)

Commands manipulate **objects**

Each object is associated with a **Type**

Type:

A set of values

A set of operations on these values

Expressions:

An operation (combination of objects and operators)

Indentation Instead of Braces

Unlike languages such as R, C++, Java, and Perl, Python **uses whitespace (tabs or spaces) for code structuring**.

Example:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

A colon (:) denotes the start of an indented block.

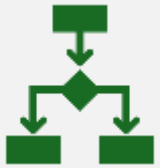
Code within the block must have uniform indentation.

Best Practice: Use **four spaces** for indentation.

- Avoid mixing spaces and tabs.
- Many text editors have settings to replace tab stops with spaces automatically.
- Some use **two spaces**, but four spaces is the widely accepted standard.

No Need for Semicolons

Unlike many other programming languages, Python **does not require semicolons** to terminate statements.



However, semicolons can be used to separate multiple statements on a single line:

```
a = 5; b = 6; c = 7
```

Best Practice: Avoid placing multiple statements on one line as it **reduces readability**.

Everything is an Object

Python maintains a **consistent object model**, meaning **everything in Python is an object.**

Examples of Python objects:

- Numbers, strings, lists, dictionaries
- Functions, classes, modules

Each object has:

- **An associated type** (e.g., string, function)
- **Internal data**

Implication: Functions can be treated like any other object, making Python highly flexible.

Python treats **numbers, strings, functions, lists, dictionaries, and even modules** as objects.

Examples

```
x = 10          # x is an object of
type int
y = "Hello"   # y is an object of
type str
print(type(x))   # Output: <class
'int'>
print(type(y))   # Output: <class
'str'>
```

Functions are also objects and can be assigned to variables:

```
def square(n):
    return n * n


f = square   # Assign function to a
variable
print(f(5))   # Output: 25
```

Comments in Python

- Any text preceded by # is **ignored by the Python interpreter**.
- Used to add explanations or temporarily disable code.
- **Example:**

```
results = []
for line in file_handle:
        # keep the empty lines for now
        # if len(line) == 0:
        # continue
        results.append(line.replace('foo', 'bar'))
```

- Comments can also be placed after a line of executed code:

```
print("Reached this line") # Simple status report
```

- **Best Practice:** Place comments **before** the line of code when providing explanations.

Comments types

Single-line Comments (#)

```
# This is a single-line comment

print("Hello, World!")  # This is
also a comment
```

Multi-line Comments (""" """ or ''' ''')

```
"""
This is a multi-line comment.
It is often used for documentation.
"""`

print("Hello, Python!")
```

Variables and Multiple Assignment

```
age = 20
print(age)
sentence = "my name is avi"
print(sentence)
####
sarah, bob, mike = 16, 22, 17
print(sarah)
print(bob)
print(mike)
```

```
sarah = bob = mike = 17
print(sarah)
print(bob)
print(mike)
####
name, age = "avi", 22
print(name)
print(age)
```

Arithmetic Operators and Strings

```
# Arithmetic operators:
# + - * / %
age1 = 12
age2 = 18
print(age1 + age2)
print(age1 * age2)
print(age1 / age2)
print(age1 % age2)
###
sent1 = "today is a beautiful day"
print(sent1)
```

```
first_name = "avi"
last_name = "jain"
print(first_name + last_name)
print(first_name + " " + last_name)
###
print("hi" * 10)
###
sent = "avi was playing basketball"
print(sent[0])
print(sent[0:3])
```

Binary operators

Operation	Description
`a + b`	Add `a` and `b`
`a - b`	Subtract `b` from `a`
`a * b`	Multiply `a` by `b`
`a / b`	Divide `a` by `b`
`a // b`	Floor-divide `a` by `b`, dropping any fractional remainder
`a ** b`	Raise `a` to the `b` power
`a & b`	`True` if both `a` and `b` are `True`; for integers, take the bitwise AND
`a | b`	`True` if either `a` or `b` is `True`; for integers, take the bitwise OR
`a ^ b`	For booleans, `True` if `a` or `b` is `True`, but not both; for integers, take the bitwise `EXCLUSIVE-OR`

Variables and Argument Passing

- **Reference Binding in Python**
  - Variables in Python are **references** to objects, not copies

```
a = [1, 2, 3]
b = a  # Both 'a' and 'b' point to the same list
a.append(4)
print(b)  # Output: [1, 2, 3, 4]
```

  - Changing a also changes b because they point to the same list object.

Placeholders in Strings

- There are three ways to substitute data into strings in Python:
  - **Concatenation** uses the + operator to combine strings.
  - **Placeholders** use the % operator to insert values into strings.
  - **Formatted strings** use the f prefix to insert values into strings.

Examples

```
name = "jake"
sentence = "%s is 15
years old"
print(sentence % name)
sentence = "%s %s was
the president of the
United States"
print(sentence %
("Barack", "Obama"))


sentence = "%s is %d
years old"
print(sentence &
("avi", 23))
```

```
name = "avi"
print(f"Hello,
{name}")
x = 10
y = 20
print(f"The sum of x
and y is {x + y}")
```

Dynamic references

- In contrast with many compiled languages, such as Java and C++, object references in
- Python have no type associated with them. There is no problem with the following:

```
In [12]: a = 5
In [13]: type(a)
Out[13]: int
In [14]: a = 'foo'
In [15]: type(a)
Out[15]: str
```

Dynamic references, strong types

- Variables are names for objects within a particular namespace; the type information is stored in the object itself.
- Some observers might hastily conclude that Python is not a “typed language.” This is not true; consider this example:

```
In [16]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```

Types in Python

int

Bounded integers, e.g. 732 or -5

float

Real numbers, e.g. 3.14 or 2.0

long

Long integers with unlimited precision

str

Strings, e.g. ‘hello’ or ‘C’

Types in Python

Scalar

- Indivisible objects that do not have internal structure
- **int** (signed integers), **float** (floating point), **bool**
- (Boolean), ***NoneType***
  - NoneType is a special type with a single value
  - The value is called **None**

```
x = 10  # x is an integer (int)
y = 3.14  # y is a float
z = True  # z is a boolean
a = None  # a has no value (NoneType)
```

Non-Scalar

- Objects having internal structure
- **str** (strings) : is a sequence of characters

```
name = "Python"  # name is a string (non-scalar)
print(name[0])   # Output: 'P' (accessing an internal element)
```

Key Difference

Feature	Scalar	Non-Scalar
Structure	Single, indivisible value	Contains multiple elements
Examples	int, float, bool, NoneType	str, list, tuple, dict, set
Indexing Allowed?	❌ No	✅ Yes (for sequence types like str, list, etc.)

Type Conversion in Python: **Implicit Type Conversion (Automatic)**

- Python automatically converts smaller data types to larger ones to prevent data loss.

```
x = 5     # Integer
y = 2.5   # Float
result = x + y  # x is automatically converted to float
print(result)       # Output: 7.5
print(type(result)) # Output: <class 'float'>
```

- **Why?**
  - int (5) is automatically converted to float (5.0) before addition to 2.5.
  - This ensures precision without explicit conversion.
- **Implicit Conversion**: Python **automatically** converts types when necessary.

Explicit Type Conversion (Type Casting)

- You can manually convert types using **type conversion functions**:
  - int()
  - float()
  - str()
  - bool()
- **Integer to Float**: int → float conversion happens smoothly.
- **Float to Integer**: float → int **truncates** decimals (does not round).

Example 1: Converting float to int

```
num = 3.6
int_num = int(num)   # Truncates decimal part

print(int_num)       # Output: 3
print(type(int_num)) # Output: <class 'int'>
```

- The decimal part **is truncated** (not rounded).

Example 2: Converting int to float

```
a = 10
b = float(a)  # Converts 10 to 10.0

print(b)      # Output: 10.0
print(type(b)) # Output: <class 'float'>
```

Example 3: Converting number to string

```
num = 100
str_num = str(num)   # Converts to "100"

print(str_num)       # Output: '100'
print(type(str_num)) # Output: <class 'str'>
```

Type Conversion and Input () function

```
age = input("How old are you? ")  # Input is taken as a string
print("In 5 years, your age will be", age + 5)  # Error: Trying to add an integer to a string
```

- **Error Explanation**
- **Input() function takes input from user:**
  - input() function **always** returns a string.
  - Trying to add age (a string) to 5 (an integer) causes a **TypeError**.
  - Python does not automatically convert a string to an integer in arithmetic operations.

```
age = int(input("How old are you? "))  # Convert input to integer
print("In 5 years, your age will be", age + 5)  # No error, since both are integers
```

Alternative Correction (Explicit Conversion at Print)

- If you want to keep age as a string and only convert it when needed:

```
age = input("How old are you? ")  # Keeps age as a string
print("In 5 years, your age will be", int(age) + 5)  # Convert only when needed
```

- **Input() returns a string** → Always convert to int or float if doing arithmetic.
- **Mixing string and integer in arithmetic causes errors** (TypeError).
- **Use int() or float() conversion** to avoid errors.
- **Convert at input or during calculation** based on preference.

User-Defined Functions in Python

- A **user-defined function** in Python is a function that you create to perform a specific task. It helps in code reusability and modularity.
- Defining a Function
- In Python, a function is defined using the def keyword:

```
def greet():
    print("Hello, Welcome to Python!")
```

- **Explanation:**
  - def greet(): → Defines a function named greet
  - print("Hello, Welcome to Python!") → Prints a message when the function is called
- **Calling the function:**

```
greet()  # Output: Hello, Welcome to Python!
```

Function with Parameters

- Functions can take **parameters (arguments)** to process input values.

```
def greet_user(name):
print(f"Hello, {name}! Welcome to Python.")
greet_user("Alice") # Output: Hello, Alice!
Welcome to Python.
```

- Here:
  - name is a **parameter** (input variable)
  - "Alice" is an **argument** (actual value passed)

Function with Return Value

- A function can return a value using the return statement.

```
def add(a, b):
  return a + b
result = add(5, 3)
print("Sum:", result)              # Output: Sum: 8
```

- Function with Default Parameters

```
def power(base, exponent=2):
  return base ** exponent

print(power(3))          # Output: 9 (3^2)
print(power(3, 3))       # Output: 27 (3^3)
```

Function with Multiple Arguments (*args)

- To pass multiple arguments dynamically, use *args.

```
def sum_all(*numbers):
    return sum(numbers)

print(sum_all(1, 2, 3, 4, 5))  # Output: 15
```

“””

In the function sum_all(*numbers), the parameter **numbers** is a **tuple** that collects all the arguments passed to the function.

Tuples are discussed in further slides

“””

Module

In Python, a **module** is a file containing Python code (functions, variables, or classes) that can be imported into another program. Modules help in organizing code and reusing functions.

Modules help in code reusability: Functions defined in math_operations.py can be used in any Python script by importing.

Different import methods:

import module_name (requires prefix module_name.function())
from module_name import function_name (direct usage)
import module_name as alias (shorter names for readability)

Modules help in organizing large codebases by keeping related functions in separate files

Step 1: Create a Module (math_operations.py)

```
# math_operations.py
# Defining some
mathematical functions


def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
```

```
def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Error!
Division by zero."
    return a / b

# Defining a constant
PI = 3.14159
```

Step 2: Import and Use the Module

create another Python script (main.py) in the same directory and use the math_operations module.

```
# main.py
# Importing the entire module
import math_operations
# Using functions from the module
sum_result = math_operations.add(10, 5)
sub_result = math_operations.subtract(10, 5)
mul_result = math_operations.multiply(10, 5)
div_result = math_operations.divide(10, 5)
```

Step 2: Import and Use the Module

create another Python script (main.py) in the same directory and use the math_operations module.

```
# Using the constant

pi_value = math_operations.PI

# Printing results

print("Sum:", sum_result)                  # Output: Sum: 15

print("Subtraction:", sub_result)          # Output: Subtraction: 5

print("Multiplication:", mul_result) # Output: Multiplication:
50

print("Division:", div_result)             # Output: Division: 2.0

print("Value of PI:", pi_value)            # Output: Value of PI:
3.14159
```

Step 3: Using Specific Imports

Instead of importing the entire module, you can import only specific functions.

```
# main.py
from math_operations import add, multiply, PI
sum_result = add(7, 3)    # 7 + 3 = 10
mul_result = multiply(7, 3)  # 7 * 3 = 21
print("Sum:", sum_result)        # Output: Sum: 10
print("Multiplication:", mul_result)  # Output:
Multiplication: 21
print("Value of PI:", PI)        # Output: Value of PI:
3.14159
```

Step 4: Using Aliases (as keyword)

```
# main.py

import math_operations as mo  # Giving an alias to the module

result1 = mo.add(8, 2)
result2 = mo.divide(8, 2)

print("Sum using alias:", result1)       # Output: Sum using alias: 10
print("Division using alias:", result2) # Output: Division using alias: 4.0
```

Step 5: Executing the Code

1. Save math_operations.py and main.py in the same directory.
2. Run main.py in the terminal:

```
python main.py
```

3. You will see the expected output.

Practice programs

Questions

1. Write a program to design a Simple Calculator.
2. Write a program for Swapping Two Numbers Without a Temporary Variable
3. Write a program to Convert Celsius to Fahrenheit
4. Write a program to Find the Area and Perimeter of a Rectangle
5. Write a program to Check if a Number is Even or Odd

Answer 1

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

sum_result = num1 + num2
diff_result = num1 - num2
prod_result = num1 * num2
quotient_result = num1 / num2 if num2 != 0 else "Undefined
(division by zero)"

print(f"Sum: {sum_result}")
print(f"Difference: {diff_result}")
print(f"Product: {prod_result}")
print(f"Quotient: {quotient_result}")
```

Answer 2

```
a = int(input("Enter first number (a): "))
b = int(input("Enter second number (b): "))

print(f"Before swapping: a = {a}, b = {b}")

# Swapping without a temporary variable
a, b = b, a

print(f"After swapping: a = {a}, b = {b}")
```

Answer 3

```
celsius = float(input("Enter temperature in Celsius: "))
fahrenheit = (celsius * 9/5) + 32
print(f"{celsius}°C is equal to {fahrenheit}°F")
```

Answer 4

```
length = float(input("Enter length of rectangle: "))
width = float(input("Enter width of rectangle: "))

area = length * width
perimeter = 2 * (length + width)

print(f"Area: {area}")
print(f"Perimeter: {perimeter}")
```

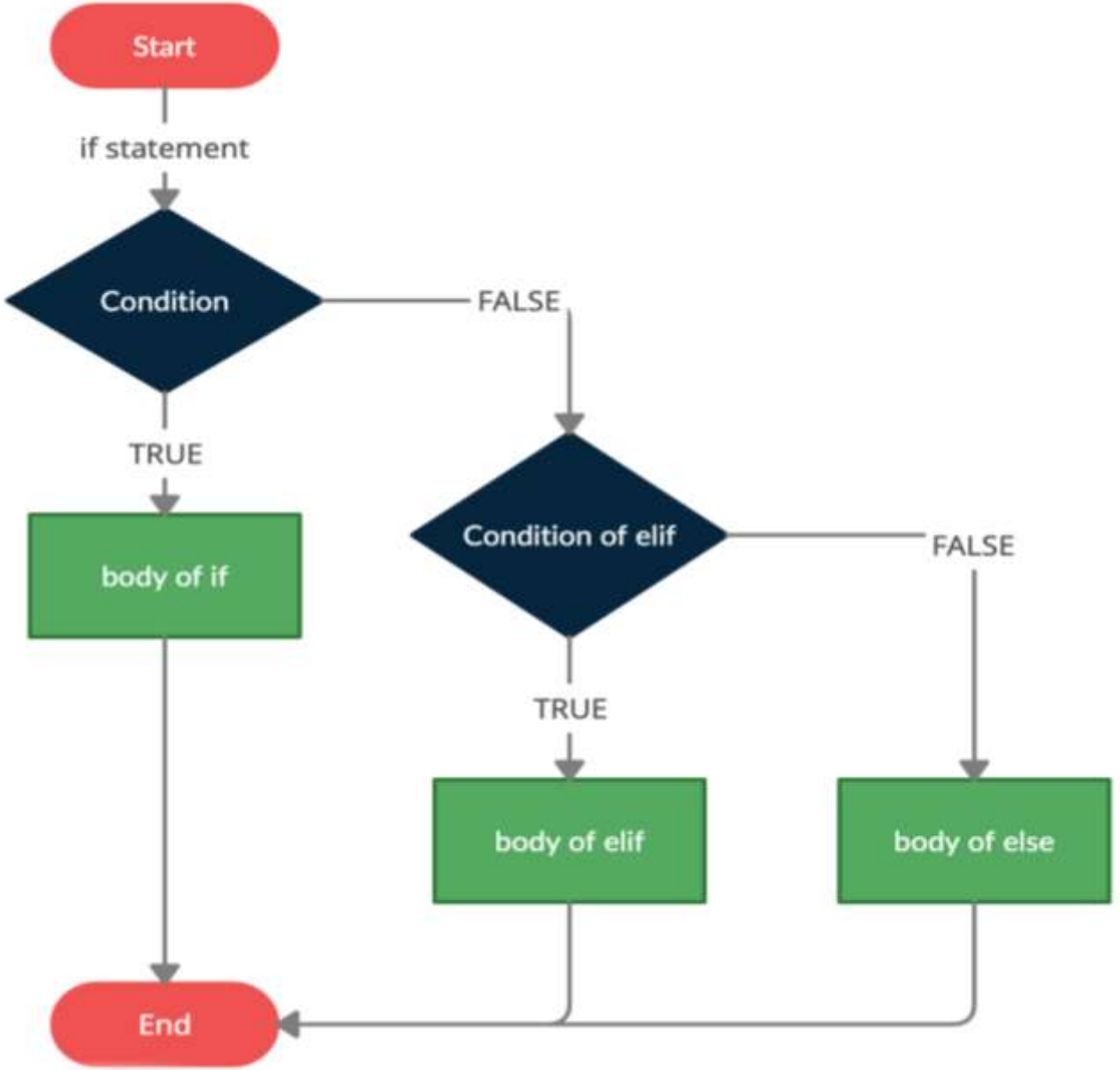
Answer 5

```
num = int(input("Enter a number: "))

if num % 2 == 0:
    print(f"{num} is even.")
else:
    print(f"{num} is odd.")
```

Control Flow in Python

if, elif, and else Statements



Example 1: Basic if Statement

```
temperature = 30
if temperature > 25:
    print("It's a hot day!")
```

Output:

It's a hot day!

The condition temperature > 25 is True, so the block inside the if executes.

Example 2: Using if and else

```
temperature = 20


if temperature > 25:
    print("It's a hot day!")
else:
    print("It's not that hot.")
```

Output:

It's not that hot.

The condition temperature > 25 is False, so the else block executes.

Example 3: Using if, elif, and else

```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

Output:
Grade: B

The first condition score >= 90 is False.The second condition score >= 80 is True, so it prints "Grade: B", and the rest of the elif conditions are skipped.

Logical Operators (and, or, not)

Python allows combining multiple conditions using and, or, and not.

Example 4: and Operator

```
age = 25
income = 50000
if age > 18 and income > 40000:
    print("Eligible for a loan")
```

- output

Eligible for a loan

Example 5: or Operator

- At least one condition must be True for the block to execute.

```
is_raining = False
is_snowing = True

if is_raining or is_snowing:
    print("Take an umbrella.")
```

- Output:

Take an umbrella.

Example 6: not Operator

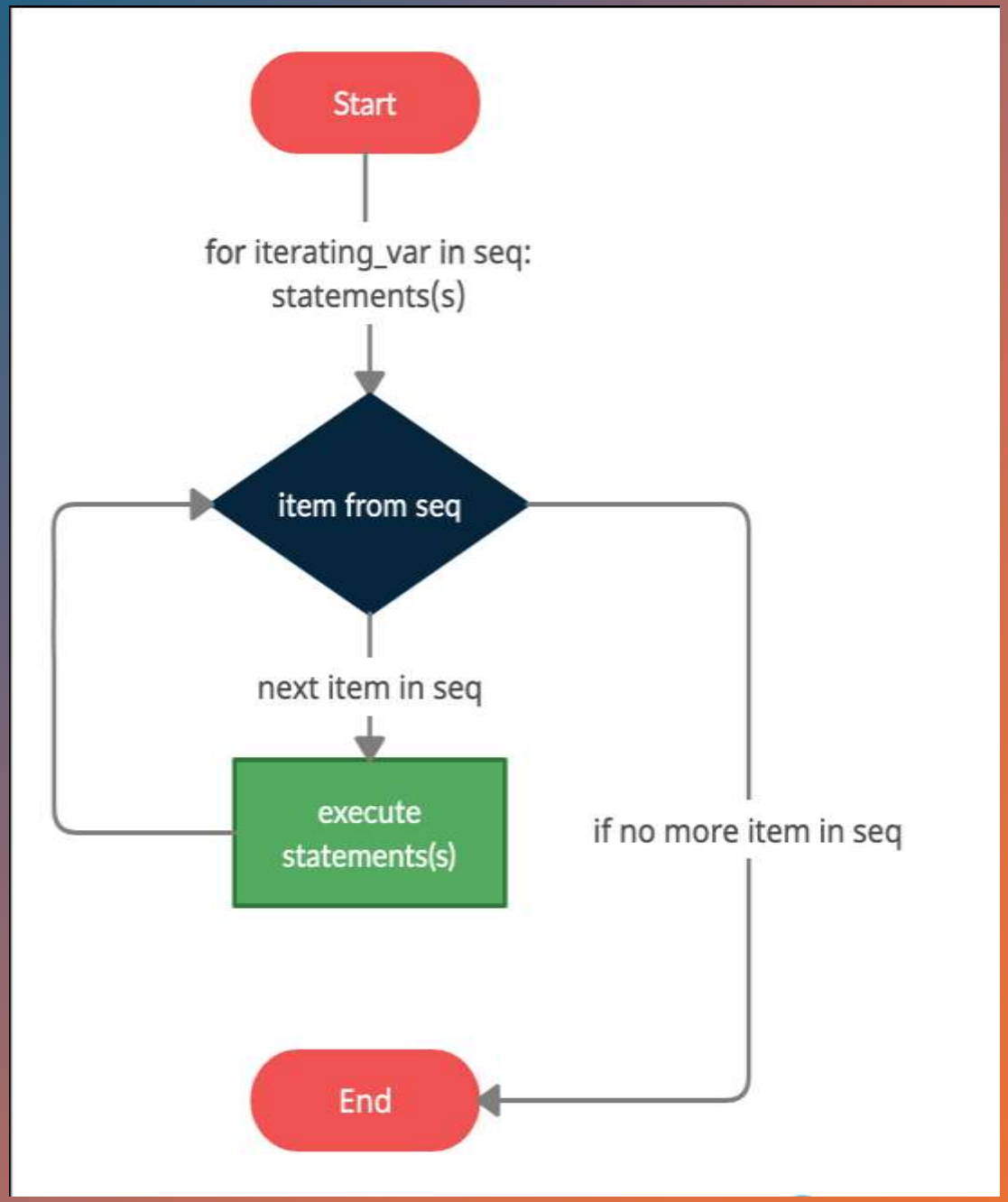
```
logged_in = False

if not logged_in:
    print("Please log in to continue.")
```

- **output**

Please log in to continue.

- *Since logged_in is False, not logged_in becomes True, so the if block runs.*



For Loop

- A for loop in Python is used to iterate over a **collection** (like a list, tuple, dictionary, or string) or an **iterator**.
- It allows us to execute a block of code multiple times, once for each element in the sequence.

Basic Syntax

```
for value in collection:
    # do something with value
Example:
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    print(num)
```

Each value in numbers is assigned to num, and the print(num) statement executes once per iteration.

Using for Loop with range()

- The range() function generates a sequence of numbers

```
for i in range(5):  # Generates numbers from 0 to 4
    print(i)
```

```
0
1
2
3
4
```

Using break in Nested Loops

If there are nested loops, break only exits **the innermost loop**.

```
for i in range(4):                  # Outer loop (i)
    for j in range(4):              # Inner loop (j)
        if j > i:
            break
        print((i, j))
```

Looping Through a List

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

apple
banana
cherry

Using range() with Start and Step

```
for i in range(1, 10, 2): #Starts from 1,increments by 2
    print(i)
```

- Output:

1
3
5
7
9

Iterating Over a String

```
word = "Python"
for char in word:
    print(char)
```

Output:

P
y
t
h
o
n

Nested For Loop (Multiplication Table)

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j, end="\t")
    print()
```

```
1   2   3   4   5
2   3   6   8   10
3   4   9   12  15
4   5   12  16  20
5   6   15  20  25
```

Using else with For Loop

```
for i in range(3):
    print(i)
else:
    print("Loop finished successfully!")
```

```
0
1
2
Loop finished successfully!
```

Breaking a Loop

```
for i in range(5):
    if i == 3:
        print("Breaking at", i)
        break
    print(i)
```

0
1
2
Breaking at 3

Skipping an Iteration (Using continue)

```
for i in range(5):
    if i == 2:
        continue  # Skips when i == 2
    print(i)
```

```
0
1
3
4
```

Real-Time Traffic Light System Simulation

```
import time

traffic_lights = ["Red", "Green", "Yellow"]

for light in traffic_lights:
    print(f"Traffic Light: {light}")
    time.sleep(2)   # Simulating the time interval for each light
```

Detecting Anomalies in Sensor Data

```
sensor_readings = [22, 25, 27, 80, 23, 24]  # 80 is an anomaly

for reading in sensor_readings:
    if reading > 50:  # Assuming threshold for anomaly detection
        print(f"Alert! Unusual reading detected: {reading}")
```

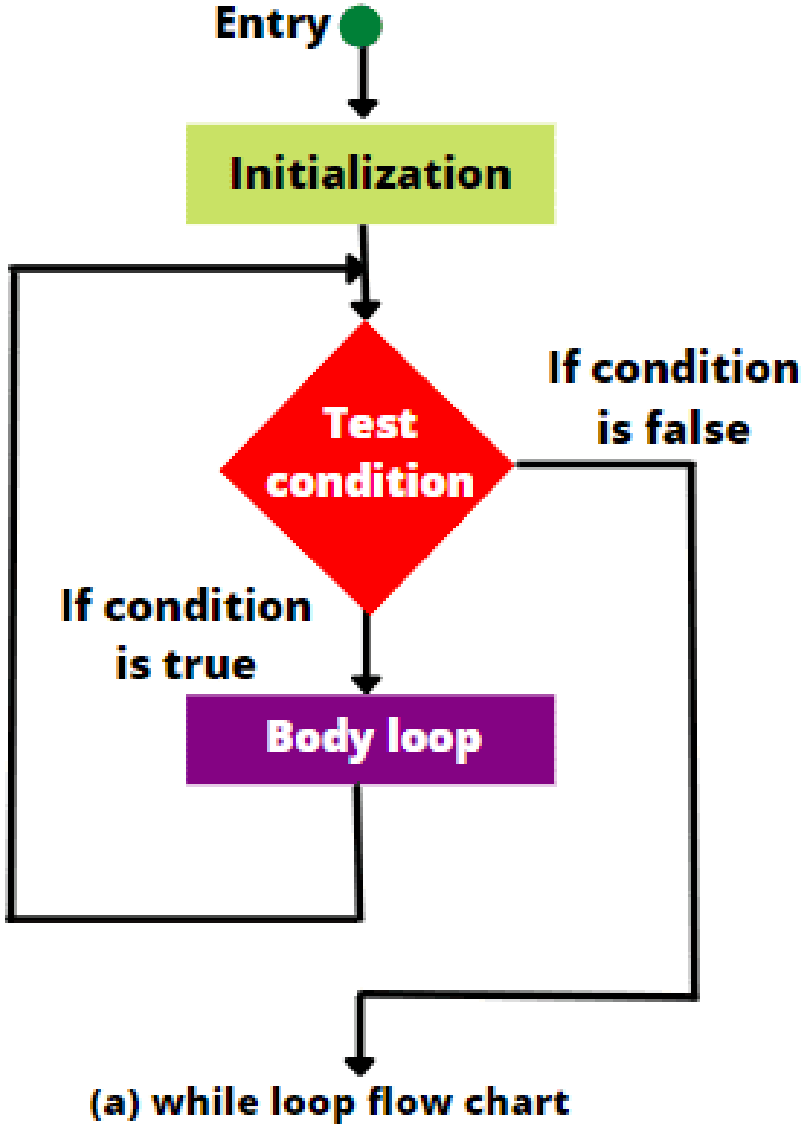
While loop

While loop

- A while loop **repeats a block of code as long as a condition is True**. It is useful when the number of iterations is **not known in advance**.
- Syntax:

```
while condition:
    # Code to execute
```

- The loop **runs** until the condition becomes False.
- 'If the condition never becomes False, the loop runs **indefinitely** (infinite loop).'
- The loop can be **stopped using break**.



(a) while loop flow chart

Example

```
num = -1  # Initialize with a negative number
while num < 0:
    num = int(input("Enter a positive number: "))

print(f"You entered: {num}")
```

Example

```
x = 256
total = 0

while x > 0:    # Loop continues while x is greater than 0
    if total > 500:
        break  # Exit the loop when total exceeds 500
    total += x
    x = x // 2  # Reduce x by half

print("Final total:", total)
```

Explanation

- The loop halves x each time and adds it to total.
- If total exceeds 500, the break statement **stops** the loop.

pass Statement

- The pass statement is a **placeholder** that **does nothing**.
- It is used when a block of code is **required** but no action is needed at the moment.

```
x = 10

if x < 0:
    print("Negative!")
elif x == 0:
    pass  # Placeholder for future code
else:
    print("Positive!")
```

Explanation

- pass allows the code to be syntactically correct **without doing anything**.
- It is **useful for writing stub functions** that will be implemented later.

range() Function

- The range() function generates a sequence of numbers **efficiently**

```
print(list(range(10)))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Explanation**
- Starts at 0 (default).
- Ends **before** 10 (endpoint **not included**).

Using range(start, end, step)

- You can customize:
- **Start** → Beginning value.
- **End** → **Stops before** this value.
- **Step** → The increment (or decrement) between values.

```
print(list(range(0, 20, 2)))  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Starts at 0, increases by 2, and stops **before** 20.

Using range() for Index-Based Iteration

- range() is often used with for loops when we need **index-based access**.
- Example: Accessing Elements by Index

```
seq = [10, 20, 30, 40]

for i in range(len(seq)):  # Iterates over indices
0, 1, 2, 3
    print(f"Index {i}: Value {seq[i]}")
```

Output:
Index 0: Value 10
Index 1: Value 20
Index 2: Value 30
Index 3: Value 40

Summing Multiples of 3 or 5

- Let's sum all numbers **from 0 to 99,999** that are divisible by **3 or 5**.

```
total = 0

for i in range(100000):   # Looping through 0 to 99999
    if i % 3 == 0 or i % 5 == 0:  # Check if i is a multiple
                                    of 3 or 5
        total += i

print("Sum:", total)
```

- range(100000) **does not** create a list of 100,000 numbers in memory.It **generates** numbers **on demand**, saving memory.

Positional vs. Keyword Arguments

```
def introduction(name, age, city="Unknown"):
    return f"My name is {name}, I am {age} years old and
live in {city}."

# Positional Arguments
print(introduction("Alice", 25))

# Keyword Arguments
print(introduction(age=30, name="Bob", city="New York"))
```

Mutable vs. Immutable Objects

Immutable Objects (str, int):

```
x = "hello"
y = x
x = x.upper()  # New string object is created
print(y)  # Output: hello (unchanged)
print(x)  # Output: HELLO (new object)
```

- Strings cannot be modified in place.

Mutable Objects (list):

```
a = [1, 2, 3]
b = a
a.append(4)  # Mutating the original list
print(b)  # Output: [1, 2, 3, 4]
```

- Lists are mutable and change **in place**.

Function Argument Passing

Mutable objects (lists, dictionaries) can be modified inside functions.

Immutable objects (strings, numbers) cannot be changed inside functions.

Function Argument Passing Example

Example 1: Passing a Mutable Object (list)

```
def append_element(some_list, element):
    some_list.append(element)

data = [1, 2, 3]
append_element(data, 4)
print(data)  # Output: [1, 2, 3, 4]
```

- Changes inside the function persist outside.

Example 2: Passing an Immutable Object (int)

```
def increment(n):
    n = n + 1
    print("Inside function:", n)

num = 10
increment(num)
print("Outside function:", num)
# Output: 10 (unchanged)
```

- Changes inside the function do not persist outside.

Summary

Concept	Key Takeaways
Indentation	Python uses spaces instead of {} for blocks.
Objects	Everything in Python is an object.
Comments	Use # for comments, """ """ for docstrings.
Function Calls	f(x, y), obj.method(a, b).
Mutable vs Immutable	Lists are mutable, strings are immutable.
Reference Binding	Variables reference objects, they don’t copy them.