Python Programming Guide - UNIT I: Introduction to Python

Python Programming Guide - UNIT I: Introduction to Python

1. Introduction to Python

Python is a high-level, interpreted, dynamically-typed programming language designed with simplicity and readability in mind. It emphasizes code readability through its clean syntax, making it an excellent choice for both beginners and experienced developers.

Why Python?

The name “Python” comes from the British comedy series “Monty Python’s Flying Circus,” reflecting the creator’s desire to make programming fun and approachable. Despite its playful name, Python is a serious and powerful language used in production environments worldwide.

Key Characteristics:

Simple and Readable Syntax: Python code reads almost like English, with clear structure that reduces the learning curve.

```python
# Example: Simple greeting program
name = "Alice"
age = 25
print(f"Hello, {name}! You are {age} years old.")
```

Output: `Hello, Alice! You are 25 years old.`

---

2. History and Features of Python

Historical Development

1989: Guido van Rossum begins Python development at CWI (Centrum Wiskunde & Informatica) in Amsterdam.

1991: Python 0.9.0 is released with basic features including classes and exception handling.

2000: Python 2.0 introduces list comprehensions, garbage collection, and Unicode support.

2008: Python 3.0 (Py3k) is released with significant changes, breaking backward compatibility to improve the language.

2020: Python 2.7 reaches end-of-life; Python 3 becomes the standard.

2023: Python 3.12 brings performance improvements and enhanced error messages.

Key Features of Python

Interpreted Language: Python code is executed line-by-line by an interpreter, no compilation needed.

```
# This runs immediately without compilation
print("Python executes this directly")
```

Dynamically Typed: Variables don't require explicit type declaration; types are determined at runtime.

```
x = 10           # x is an integer
x = "Hello"      # x is now a string
x = 3.14         # x is now a float
print(type(x))   # Output: <class 'float'>
```

Object-Oriented: Everything in Python is an object, supporting OOP principles.

```
class Dog:
    def bark(self):
        return "Woof!"

dog = Dog()
print(dog.bark())  # Output: Woof!
```

Rich Standard Library: Python comes with extensive built-in modules for various tasks.

```
import math
import datetime
import os

print(math.sqrt(16))           # Output: 4.0
print(datetime.datetime.now()) # Output: 2026-02-03 [timestamp]
print(os.getcwd())             # Current working directory
```

Platform Independent: Python code runs on Windows, macOS, Linux, and more without modification.

Supports Multiple Programming Paradigms: Procedural, object-oriented, and functional programming.

```
# Procedural approach
def add_procedural(a, b):
    return a + b

# Functional approach with lambda
add_functional = lambda x, y: x + y
```

```python
# OOP approach
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b

print(add_procedural(5, 3))       # Output: 8
print(add_functional(5, 3))       # Output: 8
print(Calculator.add(5, 3))       # Output: 8
```

---

3. Applications of Python

Web Development

Django and Flask frameworks enable rapid web application development.

```python
# Simple Flask web application
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Web World!"

if __name__ == '__main__':
    app.run(debug=True)
```

Data Science and Machine Learning

NumPy, Pandas, Scikit-learn, TensorFlow, PyTorch are industry-standard libraries.

```python
import pandas as pd
import numpy as np

# Data analysis example
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Scores': [85, 92, 78]
}
df = pd.DataFrame(data)
print(df)
print(f"Average Score: {df['Scores'].mean()}")
```

Artificial Intelligence

Deep learning frameworks like TensorFlow and PyTorch are built on Python.

```python
# Neural network example (conceptual)
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=
(784,)),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

Scientific Computing

SciPy, Matplotlib, SymPy support complex scientific calculations and visualizations.

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

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title('Sine Wave')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.show()
```

Automation and Scripting

Python excels at automating repetitive tasks.

```
import os
import shutil

# Example: Organizing files by extension
def organize_files(directory):
    for filename in os.listdir(directory):
        if filename.endswith('.txt'):
            shutil.move(filename, 'text_files/')
        elif filename.endswith('.pdf'):
            shutil.move(filename, 'pdf_files/')

organize_files('/my/messy/folder')
```

Cybersecurity

Tools like Scapy, Paramiko, and requests enable security testing and automation.

```
import requests

# Simple security scan
response = requests.get('https://api.example.com/status')
print(f"Status Code: {response.status_code}")
print(f"Response: {response.json()}")
```

Game Development

Pygame library supports game creation.

```
import pygame

pygame.init()
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("My Game")
```

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

pygame.quit()
```

---

4. Python Installation and Environment Setup

Windows Installation

Step 1: Visit python.org and download the latest stable version.

Step 2: Run the installer and check "Add Python to PATH" (important!).

Step 3: Verify installation by opening Command Prompt and typing:

```
python --version
```

macOS Installation

Using Homebrew (recommended):

```
brew install python3
python3 --version
```

Or download from python.org.

Linux Installation

For Ubuntu/Debian:

```
sudo apt-get update
sudo apt-get install python3
python3 --version
```

For Fedora/RHEL:

```
sudo yum install python3
python3 --version
```

Setting Up Virtual Environments

Virtual environments isolate project dependencies.

```
# Create a virtual environment
python -m venv my_project_env

# Activate (Windows)
my_project_env\Scripts\activate

# Activate (macOS/Linux)
source my_project_env/bin/activate
```

```
# Install packages
pip install flask numpy pandas

# Deactivate
deactivate
```

Package Management with pip

```
# Install a package
pip install requests

# Install specific version
pip install django==4.2.0

# List installed packages
pip list

# Create requirements file
pip freeze > requirements.txt

# Install from requirements file
pip install -r requirements.txt

# Upgrade a package
pip install --upgrade requests
```

---

5. Python Interpreter

Python provides two modes for code execution:

Interactive Mode

Interactive mode allows line-by-line code execution and instant feedback, ideal for learning and testing small code snippets.

Starting Interactive Mode:

```
python
# or
python3
```

Interactive Session Example:

```
>>> print("Welcome to Python!")
Welcome to Python!

>>> x = 10
>>> y = 20
>>> print(x + y)
30

>>> import math
>>> math.factorial(5)
120

>>> # Calculations
>>> (10 + 5) * 2
```

```
30

>>> # String operations
>>> "Hello" + " " + "World"
'Hello World'

>>> exit()
```

Advantages: - Immediate feedback for each command - Excellent for learning and experimentation - Quick testing of code snippets - Interactive debugging

Disadvantages: - Not suitable for large programs - Session ends upon exit (code not saved) - Difficult to manage complex logic

Script Mode

Script mode involves writing code in a file (.py) and executing the entire file, suitable for complete programs.

Creating a Python Script (`hello_world.py`):

```
# This is a Python script
print("Script Mode Example")
print("=" * 40)

# Taking user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Calculations
birth_year = 2026 - age

# Output
print(f"\nHello, {name}!")
print(f"You were born around {birth_year}")
print(f"Age verification: {age} years old")
```

Running the Script:

```
python hello_world.py
# or
python3 hello_world.py
```

Sample Output:

```
Script Mode Example
========================================
Enter your name: Alice
Enter your age: 25

Hello, Alice!
You were born around 2001
Age verification: 25 years old
```

Advantages: - Code is saved and reusable - Suitable for complex programs - Can be executed multiple times - Easy to share and collaborate

More Complex Script Example (`student_database.py`):

```python
# Student Grade Management System
students = {}

def add_student(name, grade):
    students[name] = grade
    print(f"Added {name} with grade {grade}")

def display_students():
    print("\nStudent Database:")
    for name, grade in students.items():
        print(f"  {name}: {grade}")

def get_average():
    if not students:
        return 0
    return sum(students.values()) / len(students)

# Main program
add_student("Alice", 85)
add_student("Bob", 92)
add_student("Charlie", 78)

display_students()
print(f"\nAverage Grade: {get_average():.2f}")
```

---

6. Structure of a Python Program

A well-organized Python program typically follows this structure:

Complete Program Structure Example

```python
"""
Module: Employee Management System
Description: Manages employee records and calculations
Author: Your Name
Date: 2026-02-03
"""

# SECTION 1: Imports
import json
from datetime import datetime

# SECTION 2: Constants
COMPANY_NAME = "TechCorp"
MAX_EMPLOYEES = 100
HOURLY_RATE = 25.00

# SECTION 3: Global Variables
employees = []

# SECTION 4: Function Definitions
def add_employee(name, department):
    """
    Add a new employee to the system.

    Args:
        name (str): Employee's full name
        department (str): Department assignment
```

```
            Returns:
                bool: True if successful, False otherwise
            """
            if len(employees) >= MAX_EMPLOYEES:
                print(f"Cannot add employee. Maximum limit ({MAX_EMPLOYEES})
reached.")
                return False

            employee = {
                'name': name,
                'department': department,
                'joined_date': datetime.now().strftime("%Y-%m-%d")
            }
            employees.append(employee)
            return True

        def display_employees():
            """Display all employees in the system."""
            if not employees:
                print("No employees found.")
                return

            print(f"\n{COMPANY_NAME} - Employee List")
            print("=" * 50)
            for idx, emp in enumerate(employees, 1):
                print(f"{idx}. {emp['name']} | {emp['department']} | Joined:
{emp['joined_date']}")

        # SECTION 5: Main Program
        def main():
            """Main function to run the program."""
            print(f"Welcome to {COMPANY_NAME} Management System\n")

            # Add some employees
            add_employee("Alice Johnson", "Engineering")
            add_employee("Bob Smith", "Sales")
            add_employee("Carol White", "HR")

            # Display employees
            display_employees()

            print(f"\nTotal Employees: {len(employees)}/{MAX_EMPLOYEES}")

        # SECTION 6: Entry Point
        if __name__ == "__main__":
            main()
```

Output:

```
Welcome to TechCorp Management System

TechCorp - Employee List
==================================================
1. Alice Johnson | Engineering | Joined: 2026-02-03
2. Bob Smith | Sales | Joined: 2026-02-03
3. Carol White | HR | Joined: 2026-02-03

Total Employees: 3/100
```

Structure Breakdown:

Docstring: Module-level documentation at the beginning.

Imports: All necessary modules at the top.

Constants: Global constants in UPPER_CASE.

Global Variables: Shared variables (use sparingly).

Function Definitions: Reusable code blocks.

Main Function: Entry point logic.

Entry Point: `if __name__ == "__main__":` block.

---

7. Keywords and Identifiers

Python Keywords

Keywords are reserved words with special meaning in Python. They cannot be used as variable names.

Complete List of Python Keywords (35 total):

```
# Keywords (cannot be used as variable names)
False, True, None          # Constants
and, or, not              # Logical operators
if, elif, else            # Conditional statements
for, while                # Looping statements
break, continue, pass     # Loop control
def, return               # Function definition
class                     # Class definition
import, from, as          # Module importing
try, except, else, finally # Exception handling
raise                     # Exception raising
with                      # Context managers
lambda                    # Anonymous functions
yield                     # Generator functions
assert                    # Debugging assertions
del                       # Deletion statement
global, nonlocal          # Scope modifiers
is, in                    # Membership and identity
```

Keyword Usage Examples:

```
# Reserved keywords in action
if True:
    print("This is allowed")

# Invalid usage (causes SyntaxError):
# if = 10  # Error: "if" is a keyword

# Check if something is a keyword
import keyword
print(keyword.iskeyword("if"))      # Output: True
print(keyword.iskeyword("myvar"))   # Output: False
print(keyword.kwlist)               # Prints all keywords
```

Identifiers

Identifiers are names given to variables, functions, classes, and modules.

Rules for Identifiers:

1. Must start with a letter (a-z, A-Z) or underscore (_)
2. Can contain letters, digits (0-9), and underscores
3. Case-sensitive (age, Age, AGE are different)
4. Cannot contain spaces or special characters
5. Cannot be Python keywords

Valid Identifiers:

```python
# Valid identifier names
name = "John"
_private_var = 100
var123 = 45.6
myVariable = "Hello"
MY_CONSTANT = 3.14
__dunder__ = True
class_name = "Python"
student_id = 12345
```

Invalid Identifiers:

```python
# Invalid - starts with digit
123name = "Invalid"  # SyntaxError

# Invalid - contains space
my name = "Invalid"  # SyntaxError

# Invalid - contains special character
my-var = 10  # SyntaxError

# Invalid - is a keyword
class = "Invalid"  # SyntaxError

# Invalid - contains special character
my$var = 20  # SyntaxError
```

Naming Conventions (Python Style Guide - PEP 8):

```python
# Variables and functions: lowercase with underscores (snake_case)
student_name = "Alice"
calculate_average = lambda x, y: (x + y) / 2

# Constants: UPPERCASE with underscores
MAX_AGE = 100
PI_VALUE = 3.14159

# Classes: PascalCase
class StudentRecord:
    pass

class DataProcessor:
    pass

# Private variables: leading underscore
_private_variable = 10
```

```python
__name_mangled = 20

# Avoid ambiguous names
# Bad
o = []  # Looks like zero
I = "something"  # Looks like one
l = []  # Looks like one

# Good
my_list = []
my_string = "something"
items = []
```

Identifier Scope Example:

```python
# Global identifier
global_var = 100

def my_function():
    # Local identifier
    local_var = 50
    print(f"Local: {local_var}")
    print(f"Global: {global_var}")

my_function()  # Output: Local: 50, Global: 100

# print(local_var)  # NameError: local_var not defined globally
```

---

8. Variables and Constants

Variables

Variables are containers for storing data values. In Python, variables are created when you assign a value to them.

Creating and Using Variables:

```python
# Simple variable assignment
name = "Alice"
age = 25
height = 5.6
is_student = True

print(name)       # Output: Alice
print(age)        # Output: 25
print(type(age))  # Output: <class 'int'>
```

Variable Naming Best Practices:

```python
# Descriptive names
student_name = "Bob"
student_age = 20
is_graduated = False

# Not recommended (unclear)
s = "Bob"
a = 20
g = False
```

Dynamic Typing:

```
# Variables can change type
x = 10              # int
print(type(x))      # Output: <class 'int'>

x = "Hello"         # str
print(type(x))      # Output: <class 'str'>

x = 3.14            # float
print(type(x))      # Output: <class 'float'>

x = [1, 2, 3]       # list
print(type(x))      # Output: <class 'list'>
```

Multiple Assignment:

```
# Unpacking assignment
x, y, z = 10, 20, 30
print(x, y, z)  # Output: 10 20 30

# Swapping variables
a, b = 5, 10
a, b = b, a
print(a, b)  # Output: 10 5

# Multiple values from list
values = [100, 200, 300]
first, second, third = values
print(first, second, third)  # Output: 100 200 300
```

Variable Scope:

```
# Global variable
global_var = 50

def my_function():
    # Local variable
    local_var = 10
    print(f"Inside function - Local: {local_var}, Global: {global_var}")

my_function()
# Output: Inside function - Local: 10, Global: 50

print(f"Outside function - Global: {global_var}")
# print(local_var)  # NameError: local_var is not defined outside function
```

Constants

Constants are identifiers that should not be changed after initialization. Python doesn’t enforce immutability, but by convention, UPPERCASE names indicate constants.

Defining Constants:

```
# Mathematical constants
PI = 3.14159
E = 2.71828
GOLDEN_RATIO = 1.61803
```

```python
# Configuration constants
MAX_RETRIES = 3
TIMEOUT_SECONDS = 30
DATABASE_URL = "postgresql://localhost:5432/mydb"

# Status constants
STATUS_ACTIVE = "ACTIVE"
STATUS_INACTIVE = "INACTIVE"
STATUS_PENDING = "PENDING"

print(f"Pi value: {PI}")
print(f"Max retries: {MAX_RETRIES}")
print(f"Database: {DATABASE_URL}")
```

Real-World Example: Configuration Constants:

```python
# config.py - Configuration file
import os

# API Configuration
API_KEY = "your-api-key-here"
API_URL = "https://api.example.com"
API_TIMEOUT = 30

# Database Configuration
DB_HOST = "localhost"
DB_PORT = 5432
DB_NAME = "production_db"
DB_USER = "admin"
DB_PASSWORD = os.getenv("DB_PASSWORD", "default_password")

# Application Configuration
APP_NAME = "MyApp"
APP_VERSION = "1.0.0"
DEBUG_MODE = False
MAX_CONNECTIONS = 100

# Security Constants
PASSWORD_MIN_LENGTH = 8
SESSION_TIMEOUT_MINUTES = 30
MAX_LOGIN_ATTEMPTS = 5
```

Using Constants in Functions:

```python
# Constants for a calculator
PI = 3.14159
GRAVITY = 9.8

def calculate_circle_area(radius):
    """Calculate the area of a circle."""
    return PI * radius ** 2

def calculate_fall_distance(time):
    """Calculate distance fallen under gravity."""
    return 0.5 * GRAVITY * time ** 2

# Using constants
print(f"Area of circle (r=5): {calculate_circle_area(5):.2f}")
print(f"Distance fallen in 3 seconds:
{calculate_fall_distance(3):.2f} meters")
```

Output:

```
Area of circle (r=5): 78.54
Distance fallen in 3 seconds: 44.10 meters
```

Complete Example: Banking System with Variables and Constants

```python
"""
Banking System - Demonstrates variables and constants
"""

# CONSTANTS
INTEREST_RATE = 0.04  # 4% annual interest
MINIMUM_BALANCE = 100
TRANSACTION_FEE = 5.00
BANK_NAME = "Secure Bank"

# Main banking operations
class BankAccount:
    def __init__(self, account_holder, initial_balance):
        # Instance variables
        self.account_holder = account_holder
        self.balance = initial_balance
        self.transactions = []

    def deposit(self, amount):
        """Deposit money into account."""
        if amount > 0:
            self.balance += amount
            self.transactions.append(f"Deposit: +${amount}")
            return True
        return False

    def withdraw(self, amount):
        """Withdraw money from account."""
        if self.balance - amount >= MINIMUM_BALANCE and amount > 0:
            self.balance -= amount
            self.balance -= TRANSACTION_FEE  # Deduct fee
            self.transactions.append(f"Withdrawal: -${amount} (Fee:
${TRANSACTION_FEE})")
            return True
        return False

    def apply_interest(self):
        """Apply annual interest."""
        interest = self.balance * INTEREST_RATE
        self.balance += interest
        self.transactions.append(f"Interest: +${interest:.2f}")

    def display_account(self):
        """Display account details."""
        print(f"\n{BANK_NAME} - Account Statement")
        print("=" * 50)
        print(f"Account Holder: {self.account_holder}")
        print(f"Current Balance: ${self.balance:.2f}")
        print(f"\nTransaction History:")
        for transaction in self.transactions:
            print(f"  - {transaction}")
# Create and use account
account = BankAccount("John Doe", 1000)
account.deposit(500)
account.withdraw(200)
account.apply_interest()
account.display_account()
```

Output:

```
Secure Bank - Account Statement
==================================================
Account Holder: John Doe
Current Balance: $1247.80

Transaction History:
  - Deposit: +$500
  - Withdrawal: -$200 (Fee: $5.00)
  - Interest: +$52.80
```

Summary of Unit I

This unit covers the fundamentals of Python programming:

- **Introduction**: Python is a readable, versatile language for various applications
- **History and Features**: From its inception in 1989 to its current dominance in data science
- **Applications**: Web development, AI/ML, scientific computing, automation, and more
- **Installation**: Multiple methods across Windows, macOS, and Linux
- **Interpreter Modes**: Interactive mode for testing, script mode for applications
- **Program Structure**: Organized code with imports, functions, and main block
- **Keywords and Identifiers**: Reserved words and proper naming conventions
- **Variables and Constants**: Dynamic typing and configuration management

These foundational concepts prepare you for advanced topics in Units II-V.

Practice Exercises for Unit I

1. Install Python and verify the installation
2. Write a simple program in interactive mode and script mode
3. Create variables of different types and verify their types
4. Write a program with at least 3 functions that uses constants
5. Create a program that demonstrates proper naming conventions

---

Next Unit: Unit II - Data Types and Operators