Python Programming: Beginner to Advanced Full Course

A comprehensive journey through Python — from writing your first line of code to mastering advanced programming concepts.

Why Learn Python?

🌐 Versatile

Web dev, data science, AI, and automation — Python does it all.

🚀 In-Demand

One of the top languages employers seek in 2026's tech landscape.

🎓 Beginner-Friendly

Clean, readable syntax that scales to powerful advanced features.

💼 Career Growth

Opens doors to high-growth roles in engineering, ML, and beyond.

MODULE 1

Python Basics: Foundations

01

Setup

Install Python and configure your development environment (VS Code, PyCharm).

02

Core Concepts

Variables, data types — `int`, `float`, `str`, `bool` — and expressions.

03

I/O & Comments

Use `print()` and `input()` for interaction; document with # comments.

04

First Program

Write and run your very first Python script — a satisfying milestone!

Code Examples & Variables

```
# Basic Variable Declarations
age = 25              # int variable: Stores whole numbers.
name = "Alice"        # string variable: Stores text.
height = 5.8          # float variable: Stores decimal numbers.
is_student = True     # boolean variable: Stores True or False.

# Print() and Input() Examples
print("Hello, Python!")                  # Prints the given text to the console.
user_name = input("What's your name? ")  # Prompts user for input and stores it.
print(f"Nice to meet you, {user_name}!") # Example of using the input from the user

# Basic Arithmetic Operations
num1 = 10
num2 = 5
sum_result = num1 + num2          # Addition: Adds two numbers.
difference_result = num1 - num2   # Subtraction: Finds the difference between two numbers.
product_result = num1 * num2      # Multiplication: Multiplies two numbers.
```

MODULE 2

Control Flow & Data Structures

Conditionals

Use `if`, `elif`, and `else` to make decisions and branch your program logic.

Loops

`for` and `while` loops with `break` and `continue` for fine-grained iteration control.

Lists & Tuples

Ordered collections for storing and iterating over sequences of data.

Sets & Dicts

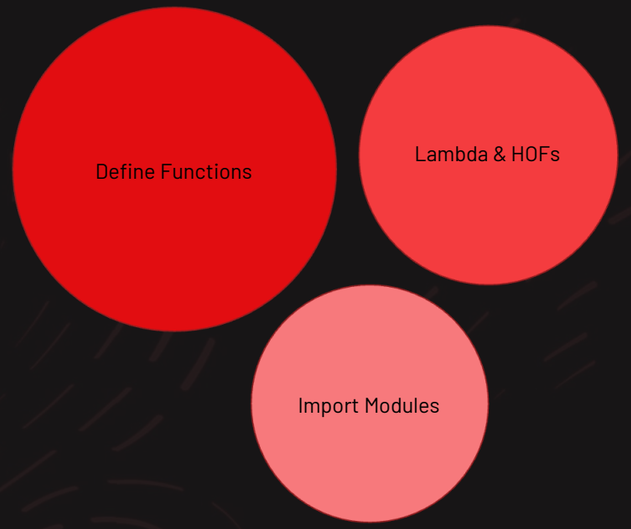
Powerful structures for unique data, fast lookups, and key-value mapping.

Code Examples & Variables

```
# Conditional Examples
age = 18  # Integer variable for age

if age >= 18:  # Check if age is 18 or greater
    print("You are an adult")  # Executed if the condition is True
else:  # If the first condition is False
    print("You are a minor")  # Executed if the condition is False

# Loop Examples
print("\nFor loop with range:")
for i in range(5):  # Loop 5 times, i will be 0, 1, 2, 3, 4
    print(i)  # Print the current value of i

print("\nFor loop through a list:")
numbers = [1, 2, 3, 4, 5]  # A list of numbers
for num in numbers:  # Iterate through each item in the 'numbers' list
    print(num * 2)  # Print each number multiplied by 2

# Data Structure Examples
my_list = [10, 20, 30]  # A list: an ordered, changeable collection of items
print(f"\nMy list: {my_list}")

my_dict = {"name": "Bob", "age": 30}  # A dictionary: an unordered, changeable collection of key-value pairs
print(f"My dictionary: {my_dict}")
print(f"Bob's age: {my_dict['age']}") # Accessing a value by its key

my_set = {1, 2, 3}  # A set: an unordered collection of unique items
print(f"My set: {my_set}")
my_set.add(4) # Add an element to the set
```

MODULE 3

Functions & Modules



Building Blocks of Reusable Code

Functions let you encapsulate logic, accept parameters, and return values — making your programs modular and maintainable.

- Define with `def`, return values cleanly
- Understand local vs. global scope
- Write `docstrings` for self-documenting code
- Compact logic with `lambda` expressions
- Leverage Python's standard library and create custom modules

Code Examples & Variables

```
# Function Definition
def greet(name):  # Define a function 'greet' that takes one parameter 'name'
    return f"Hello, {name}!"  # Returns a greeting string
result = greet("Alice")  # Call the function with "Alice" and store the returned value in 'result'
# print(result)  # Output: Hello, Alice!

# Function with Multiple Parameters
def add(a, b):  # Define a function 'add' that takes two parameters 'a' and 'b'
    return a + b  # Returns the sum of 'a' and 'b'
sum_result = add(5, 3)  # Call the function with 5 and 3, store the sum in 'sum_result'
# print(sum_result)  # Output: 8

# Lambda Example
square = lambda x: x ** 2  # Define a lambda (anonymous) function 'square' that takes 'x' and returns 'x'
squared
result_lambda = square(4)  # Call the lambda function with 4 and store the result in 'result_lambda'
# print(result_lambda)  # Output: 16

# Module Import
import math  # Import the built-in 'math' module to access mathematical functions and constants
area = math.pi * 5 ** 2  # Access 'pi' from the 'math' module and calculate the area of a circle with radius 5
# print(area) # Output: 78.53981633974483
```

MODULE 4

Object-Oriented Programming

Classes & Objects

Define blueprints with `class`; create instances with attributes and methods.

Encapsulation

Bundle data and behavior together; control access with public and private members.

Inheritance

Extend existing classes to share and override functionality across hierarchies.

Polymorphism

Write flexible code that works across different object types using a unified interface.

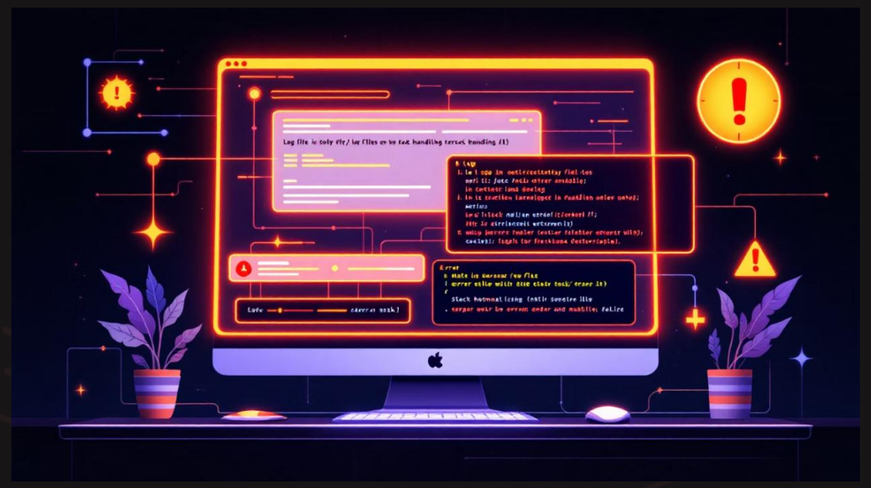
Real-World Project: Model a simple banking system — accounts, deposits, withdrawals, and transaction history using OOP principles.

Code Examples & Variables

```
# Class Definition
class BankAccount:  # Define a class named 'BankAccount'
    def __init__(self, owner, balance):  # Constructor method to initialize object attributes
        self.owner = owner  # Attribute: stores the account owner's name
        self.balance = balance  # Attribute: stores the account balance
    def deposit(self, amount):  # Method: defines behavior for depositing money
        self.balance += amount  # Update the balance by adding the deposit amount

# Object Creation and Usage
account = BankAccount("Alice", 1000)  # Create an object (instance) of BankAccount with owner "Alice" and
balance 1000
account.deposit(500)  # Call the 'deposit' method on the 'account' object, adding 500 to its balance
# print(account.owner)    # Output: Alice
# print(account.balance)  # Output: 1500

# Inheritance Example
class SavingsAccount(BankAccount):  # Define 'SavingsAccount' that inherits from 'BankAccount'
    def __init__(self, owner, balance, interest_rate):  # Constructor for SavingsAccount
        super().__init__(owner, balance)  # Call the parent class's constructor to initialize owner and
balance
        self.interest_rate = interest_rate  # New attribute specific to SavingsAccount: stores the interest
rate
```

MODULE 5

Files & Exception Handling



File I/O

Read and write text and CSV files using Python's built-in `open()` function.

Try / Except

Catch and handle exceptions gracefully to prevent unexpected program crashes.

Context Managers

Use `with` statements to ensure files are always properly closed after use.

Project

Build a **log file analyzer** that parses errors and generates a summary report.

Code Examples & Variables

```
# File Reading Example
# 'with open(...)' ensures the file is automatically closed
# 'r' mode is for reading
with open("data.txt", "r") as file:
    content = file.read() # Reads the entire file content into a single string
    print("Full content:", content)
    # After .read(), the file pointer is at the end. To read lines, you might reopen or seek(0).
    file.seek(0) # Reset file pointer to the beginning to read lines
    lines = file.readlines() # Reads all lines into a list of strings
    print("Lines:", lines)

# File Writing Example
# 'w' mode is for writing (creates file if not exists, overwrites if it does)
with open("output.txt", "w") as file:
    file.write("Hello, World!") # Writes the string to the file
    print("Written 'Hello, World!' to output.txt")

# Exception Handling Example
try:
    number = int(input("Enter a number: ")) # Attempts to convert input to an integer
    result = 10 / number # Performs division, might raise ZeroDivisionError
except ValueError: # Catches errors when input cannot be converted to int
    print("Invalid input: Please enter a valid integer.")
except ZeroDivisionError: # Catches errors when dividing by zero
```

MODULE 6

Advanced Python Features

Generators & Comprehensions

Write memory-efficient code with `yield`, list comprehensions, and generator expressions.

Decorators

Extend function behavior elegantly without modifying the original code — a powerful Pythonic pattern.

Concurrency

Boost performance with threading for I/O-bound tasks and `asyncio` for async workflows.

Virtual Environments

Isolate dependencies with `venv` and manage packages efficiently using `pip`.

Code Examples & Variables

```
# List Comprehension Examples
numbers = [1, 2, 3, 4, 5]

# Create a new list where each element is the square of the original numbers
# This is a concise way to create lists based on existing iterables.
squares = [x**2 for x in numbers]
print("Squares:", squares) # Output: [1, 4, 9, 16, 25]

# Create a new list containing only the even numbers from the original list
# A conditional statement (if x % 2 == 0) can be added to filter elements.
evens = [x for x in numbers if x % 2 == 0]
print("Evens:", evens) # Output: [2, 4]

# Generator Example
# Generators are functions that can be paused and resumed, producing a sequence of values over time.
# They are memory-efficient as they generate values on the fly rather than storing them all in memory.
def count_up(n):
    i = 0
    while i < n:
        # 'yield' keyword makes this function a generator.
        # It pauses the function's execution and returns the yielded value.
        # When called again, it resumes from where it left off.
        yield i
        i += 1

print("\nCounting up using a generator:")
for num in count_up(5): # Iterating over the generator object
    print(num) # Output: 0, 1, 2, 3, 4

# Decorator Example
# Decorators provide a way to modify or enhance a function's behavior
# without directly changing its source code. They are functions that take
# another function as an argument and return a new function.
def my_decorator(func):
    # The 'wrapper' function is what replaces the original function
```

MODULE 7

Python for Data Science & Web

Key Libraries & Frameworks

NumPy
Fast numerical arrays and math

Pandas
Data manipulation and analysis

Matplotlib
Charts and data visualization

Flask/Django
Build powerful web applications

Specialize your Python career in **AI & Machine Learning**, **data analytics**, **automation scripting**, or **backend web development**.

Code Examples & Variables

```
# NumPy Example: Numerical Operations
# NumPy is fundamental for numerical computing in Python, providing support for large, multi-dimensional
arrays and matrices.
import numpy as np

# Create a NumPy array from a Python list
arr = np.array([1, 2, 3, 4, 5])
print("NumPy Array:", arr) # Output: [1 2 3 4 5]

# Calculate the mean of the array elements
mean = np.mean(arr)
```

Next Steps & Your Python Journey

1

Practice Daily

Tackle coding challenges on LeetCode, HackerRank, and build real personal projects.

2

Learn from the Best

Explore GeeksforGeeks, Real Python, and Udacity for structured, high-quality courses.

3

Join the Community

Connect with Python developers on Reddit, Discord, and contribute to open-source projects on GitHub.

4

Keep Advancing

Pursue advanced frameworks, earn certifications, and specialize in your chosen domain.

Your Python journey starts now — code every day, build things that matter, and never stop learning. 🐍🚀