BITQUANTA

Python Programming Course Notes

Learn Python from Scratch to Project Level

(Real-world Projects & Examples)

AUTHOR

Created by: Muhammad Ishfaq Khan

Email: ishfaqkhan.dev@gmail.com

YouTube Channel: @Bitquanta

GitHub: @ishfaqkhan-dev

Table of Contents

- break, continue, pass
- range() and enumerate() functions

---

5. Data Structures

- Lists and List Comprehension
- Tuples and Unpacking
- Sets and Set Operations
- Dictionaries and Nested Dictionaries

---

6. Functions

- Defining functions
- Parameters and Return Values
- *args and **kwargs
- Lambda Functions
- Built-in Functions

---

7. Error and Exception Handling

- try, except, finally
- Raising Exceptions
- Custom Error Messages

---

8. File Handling

- Reading and Writing Files
- Working with CSV files
- Using the with statement

---

9. Object-Oriented Programming (OOP)

- Classes and Objects
- Constructors (__init__ method)
- Instance vs Class Variables
- Inheritance
- Polymorphism
- Encapsulation

---

10. Modules and Packages

- Importing built-in modules
- Creating custom modules
- __name__ == __main__ Explained

---

11. Final Projects

- Calculator App
- To-Do List (CLI App)
- Contact Book
- Number Guessing Game
- Final Mini Project

12. Wrap-Up and Practice

- Best Learning Resources
- Practice Challenges

Chapter # 1
Getting Started with Python

1.1 What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and released in 1991.

Key Features:

- Easy to learn and write
- Interpreted (no need to compile)
- Dynamically typed (no need to declare variable types)
- Supports multiple programming paradigms (procedural, object-oriented, functional)
- Large standard library and active community

Used For:

- Web development (Django, Flask)
- Data science and machine learning (Pandas, NumPy, TensorFlow)
- Automation and scripting
- Game development
- Desktop applications

1.2 Installing Python

To write and run Python code, you need to install Python on your system.

Steps:

1. Visit the official website: https://python.org
2. Download the latest version (recommended: Python 3.10 or above)
3. Run the installer:
   - **Windows:** Check the box "Add Python to PATH" before clicking Install
   - **macOS/Linux:** Use default package managers (like brew or apt)

Verify Installation:

- Open terminal or command prompt
- Type: python --version
- If Python is installed, it will show the installed version (e.g. Python 3.11.4)

1.3 Setting Up VS Code

VS Code is a lightweight and powerful code editor that supports Python development.

Steps to set up:

1. Download and install VS Code: https://code.visualstudio.com
2. Open VS Code and go to **Extensions** (Ctrl + Shift + X)
3. Search for “Python” and install the official Python extension by Microsoft
4. Restart VS Code
5. Open a folder where you want to write your code
6. Create a new file with .py extension (e.g. hello.py)

Optional:

- Install the "Code Runner" extension to run code quickly inside the editor

---

1.4 Writing Your First Program

Let’s write and run your very first Python program.

Steps:

1. Open VS Code
2. Create a new file: hello.py
3. Write the following code:

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

4. Save the file
5. Run the code:

- If using **terminal**, type: python hello.py
- If using **Code Runner**, click the Run icon (►)

Output:

```
Hello, World!
```

This is the basic structure of any Python program. You just told the computer to print text on the screen.

Chapter # 2
Python Basics

2.1 Variables and Data Types

What is a Variable?

A variable is like a container or a label that stores some value in memory. You can give it a name and assign a value to it.

Example:

```
x = 10
name = "Ali"
```

Here, x is storing a number (10) and name is storing text ("Ali").

What is a Data Type?

Data type tells Python **what kind of data** is stored in a variable. For example, whether it's a number, a word, or True/False.

Common Data Types in Python:

- int → Whole numbers (e.g. 5, -1, 100)
- float → Decimal numbers (e.g. 3.14, -0.5)
- str → Text data (e.g. "hello", '123')
- bool → True or False values

Example:

```
a = 5           # int
b = 3.5         # float
c = "hello"     # str
d = True        # bool
```

2.2 Type Conversion

Sometimes, you may need to change one type into another. This is called **type casting** or **type conversion**.

Example:

```
x = "123"        # This is a string
y = int(x)       # Now it becomes an integer
```

Common type conversion functions:

- int() → converts to integer
- float() → converts to decimal
- str() → converts to string
- bool() → converts to True/False

Example:

```
age = input("Enter your age: ")
age = int(age)
```

Here, input() gives a string, and we convert it to an integer.

2.3 Input and Output

Output using print()

Used to display text or values on the screen.

Example:

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

You can also print multiple things:

```
name = "Ali"
print("My name is", name)
```

Input using input()

Used to take input from the user. It always returns a string.

Example:

```
city = input("Enter your city: ")
print("You live in", city)
```

If you want to take a number as input:

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

2.4 Operators

Operators are symbols that perform operations on values or variables.

Arithmetic Operators

Used for basic math.

Operator	Meaning	Example
+	Addition	2 + 3 → 5
-	Subtraction	5 - 2 → 3
*	Multiplication	4 * 3 → 12
/	Division	10 / 2 → 5.0
//	Floor Division	7 // 2 → 3
%	Remainder	7 % 2 → 1
**	Power	2 ** 3 → 8

Example:

```
a = 10
b = 3
```

```
print(a % b)      # Output: 1
```

Comparison Operators

Used to compare two values. They return True or False.

Example:

```
x = 5
print(x == 5)   # True
print(x != 3)   # True
```

Logical Operators

Used to combine conditions.

- and → True only if both are True
- or → True if at least one is True
- not → Reverses the result

Example:

```
x = 5
print(x > 3 and x < 10)   # True
```

2.5 String Operations

A string is a sequence of characters (letters, numbers, symbols). Strings are written inside quotes.

Example:

```
text = "Python"
```

String Functions:

```
print(len(text))          # Length of string → 6
print(text.upper())       # 'PYTHON'
print(text.lower())       # 'python'
print(text[0])            # 'P' (1st character)
print(text[-1])           # 'n' (last character)
```

String Concatenation

Combining two or more strings.

Example:

```
first = "Bit"
last = "Quanta"
full = first + last
print(full)    # BitQuanta
```

String Formatting

Used to insert variables inside a string in a clean way.

Example:

```
name = "Ali"
age = 20
print(f"My name is {name} and I am {age} years old.")
```

Here, f before the string means it's a formatted string (f-string). It automatically puts variable values inside {}.

Chapter # 3
Control Flow

3.1 if, else, elif Statements

What is Control Flow?

Control flow allows your program to **make decisions.** Instead of running all lines one by one, Python can decide which lines to run based on conditions (True or False).

if Statement

The if statement is used to run a block of code **only if a certain condition is true**.

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

if...else Statement

else runs when the if condition is **not true**.

```
age = 15
if age >= 18:
    print("Eligible to vote.")
else:
    print("Not eligible to vote.")
```

if...elif...else

Use elif (else if) when you want to check **multiple conditions.**

```
marks = 75
if marks >= 90:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

Indentation

In Python, **indentation is very important.** It shows which lines are inside the if-block. Usually 4 spaces or 1 tab is used.

3.2 Nested Conditions

You can place one if inside another. This is called a **nested condition.**

```
age = 20
citizen = True
if age >= 18:
    if citizen:
        print("You can vote.")
    else:
        print("Only citizens can vote.")
```

```
else:
    print("You must be 18 or older to vote.")
```

Use nested if only when needed. Too much nesting can make code hard to read.

3.3 Match-Case (Python 3.10+)

match is similar to switch in other languages. It checks the value and matches it with case.

This feature works in **Python 3.10 and above** only.

```
day = "Monday"
match day:
    case "Monday":
        print("Start of the week.")
    case "Friday":
        print("Almost weekend!")
    case _:
        print("Just another day.")
```

- case _: works like default in other languages (when no case matches).
- match-case is useful when you want to check many fixed values.

Chapter # 4
Loops and Iteration

4.1 What is a Loop?

A loop is used when you want to **repeat a block of code multiple times**. Instead of writing the same code again and again, we use loops.

Python has two main types of loops:

- for loop
- while loop

4.2 for Loop

A for loop is used when you want to repeat something **a specific number of times**, or when you want to **go through items in a list, string, etc.**

Example: Looping through numbers

```
for i in range(5):
    print(i)
```

Output:

```
0
1
```

```
2
3
4
```

The range(5) means 0 to 4 (not including 5).

Example: Looping through a string

```
for letter in "Python":
    print(letter)
```

4.3 while Loop

A while loop runs **as long as the condition is true.**

Example:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

Be careful: If the condition never becomes false, the loop will run forever. This is called an **infinite loop**.

4.4 break, continue, pass

break

Used to **exit the loop early**, even if the condition is still true.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Output:

```
0 1 2 3 4
```

---

continue

Used to **skip the current iteration** and continue with the next one.

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

Output:

```
0 1 3 4
```

(2 is skipped)

pass

Used as a placeholder when you want to write nothing (no code) temporarily.

```
for i in range(3):
    pass
```

This runs but does nothing. Useful in loops or functions that you plan to complete later.

4.5 range() and enumerate()

range(start, stop, step)

Generates a sequence of numbers.

```
for i in range(1, 10, 2):
    print(i)
```

Output:

```
1 3 5 7 9
```

enumerate()

Used when you want both **index and value** in a loop.

```
names = ["Ali", "Sara", "John"]

for index, name in enumerate(names):

    print(index, name)
```

Output:

```
0 Ali
1 Sara
2 John
```

Chapter # 5
Data Structures

Data structures are used to **store and organize data** in Python. They help us manage multiple values under a single variable.

5.1 Lists

What is a List?

A list is a collection of **ordered, changeable items.** It can store different types of values like numbers, strings, etc.

```
fruits = ["apple", "banana", "mango"]
```

Accessing List Items

```
print(fruits[0])    # apple
print(fruits[-1])   # mango (last item)
```

Changing Items

```
fruits[1] = "orange"
```

List Methods

```
fruits.append("grape")      # Add at end
fruits.insert(1, "kiwi")    # Add at position 1
fruits.remove("apple")      # Remove item
fruits.pop()                # Remove last item
print(len(fruits))          # Length of list
```

5.2 List Comprehension

A short way to create a new list using a loop in one line.

```
squares = [x*x for x in range(5)]
print(squares)     # [0, 1, 4, 9, 16]
```

You can also add conditions:

```
even = [x for x in range(10) if x % 2 == 0]
```

5.3 Tuples

What is a Tuple?

A tuple is like a list, but it is **unchangeable (immutable)**. Once created, its items cannot be modified.

```
colors = ("red", "green", "blue")
```

Accessing Tuple Items

```
print(colors[1])   # green
```

If you try to change an item:

```
colors[0] = "yellow"    # Error! Tuples can't be
changed
```

Tuples are used when you want your data to be **safe from accidental changes.**

5.4 Unpacking Tuples

You can assign values from a tuple directly into variables.

```
name, age, country = ("Ali", 25, "Pakistan")
```

Now:

- name → "Ali"
- age → 25
- country → "Pakistan"

5.5 Sets

What is a Set?

A set is an **unordered collection of unique items.** Duplicates are not allowed.

```
numbers = {1, 2, 3, 3, 4}

print(numbers)    # Output: {1, 2, 3, 4}
```

Common Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))         # {1, 2, 3, 4, 5}
print(a.intersection(b))  # {3}
print(a.difference(b))    # {1, 2}
```

Sets are useful when you need to **remove duplicates** or perform math-like set operations.

5.6 Dictionaries

What is a Dictionary?

A dictionary is a collection of **key-value pairs**. Each item has a key and a value.

```
student = {
    "name": "Ali",
    "age": 20,
    "grade": "A"
}
```

Accessing Values

```
print(student["name"])     # Ali
```

Changing and Adding Values

```
student["age"] = 21
student["city"] = "Lahore"
```

Dictionary Methods

```
student.keys()      # All keys
student.values()    # All values
student.items()     # All key-value pairs
```

5.7 Nested Dictionaries

You can store dictionaries inside dictionaries.

```
students = {
    "student1": {"name": "Ali", "age": 20},
    "student2": {"name": "Sara", "age": 22}
}
```

Accessing nested values:

```
print(students["student1"]["name"])    # Ali
```

Chapter # 6
Functions

Functions help us **organize code into reusable blocks**. They make the code clean, readable, and easy to manage.

6.1 What is a Function?

A function is a block of code that **performs a specific task**. You can **call** it whenever you want.

Why Use Functions?

- Avoid repeating the same code.
- Organize large programs into smaller parts.
- Make your code reusable and readable.

6.2 Defining a Function

You define a function using the def keyword.

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

To call this function:

```
greet()
```

6.3 Parameters and Arguments

You can pass values into functions.

```
def greet(name):
    print("Hello", name)

greet("Ali")
```

Here:

- name is a **parameter** (defined inside function).
- "Ali" is an **argument** (value passed when calling).

6.4 Returning Values

Functions can **return results** using the return keyword.

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

result = add(5, 3)
print(result)    # Output: 8
```

6.5 Default Parameter Values

You can set default values for parameters.

```
def greet(name="User"):
    print("Hello", name)

greet()          # Hello User
greet("Sara")    # Hello Sara
```

6.6 Keyword Arguments

You can call a function using parameter names.

```
def student(name, age):
    print(name, "is", age, "years old.")

student(age=20, name="Ali")
```

6.7 Variable Number of Arguments

*args (Non-keyword arguments)

Allows passing multiple values.

```
def add_all(*numbers):
    total = sum(numbers)
    print("Total:", total)

add_all(1, 2, 3, 4)    # Total: 10
```

**kwargs (Keyword arguments)

Allows passing multiple key-value pairs.

```
def print_info(**info):
    for key, value in info.items():
        print(key, ":", value)

print_info(name="Ali", age=21, city="Lahore")
```

6.8 Scope of Variables

- **Local variable**: Defined inside function.
- **Global variable**: Defined outside any function.

```python
x = 10  # Global

def show():
    x = 5  # Local
    print(x)

show()       # 5
print(x)     # 10
```

To use global variable inside a function:

```python
def change():
    global x
    x = 20
```

6.9 Lambda Functions (Anonymous Functions)

Short one-line functions.

```python
square = lambda x: x * x
print(square(5))    # Output: 25
```

Useful for simple, quick tasks.

6.10 Docstrings

Used to add descriptions to functions.

```python
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__doc__)
```

6.11 Built-in Functions

Python provides many built-in functions that are always available.

Some commonly used built-in functions:

Function	Description
print()	Displays output
input()	Takes user input
len()	Returns length of a sequence
type()	Returns the data type of a value
range()	Generates a range of numbers
sum()	Returns the sum of values
max()	Returns the largest value
min()	Returns the smallest value
sorted()	Returns a sorted version of a list
int(), float(), str()	Type conversions

Examples:

```
print(len("Python"))       # 6
print(type(5.5))           # <class 'float'>
print(sum([1, 2, 3]))      # 6
print(sorted([3, 1, 2]))   # [1, 2, 3]
```

Chapter # 7
Error and Exception Handling

Handling errors in Python helps us prevent the program from crashing. With exception handling, we can manage unexpected situations like dividing by zero, file not found, etc.

7.1 What is an Error?

There are two main types of errors:

- **Syntax Error**: Mistake in code structure. Python will not run the code.
- **Runtime Error (Exception)**: Code runs, but something goes wrong during execution.

Example:

```
# Syntax Error
print("Hello"    # Missing closing parenthesis

# Runtime Error
a = 10 / 0       # ZeroDivisionError
```

7.2 try and except Blocks

Use try block to write risky code. Use except to handle the error.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Please enter a valid number.")
```

7.3 finally Block

The finally block always runs, no matter if there is an error or not.

```
try:
    f = open("data.txt")
    print(f.read())
except:
    print("Something went wrong.")
finally:
    print("Execution finished.")
```

7.4 else Block

The else block runs only if no exception occurs.

```
try:
    num = int(input("Enter a number: "))
except:
    print("Invalid input.")
else:
    print("You entered:", num)
```

7.5 Raising Exceptions

You can raise your own errors using raise.

```
age = int(input("Enter age: "))
if age < 0:
    raise ValueError("Age cannot be negative.")
```

7.6 Custom Error Messages

You can customize the message shown when an error occurs.

```
try:
    x = int(input("Enter a number: "))
    if x < 0:
        raise Exception("Negative numbers not
allowed.")
except Exception as e:
    print("Error:", e)
```

7.7 Common Python Exceptions

Some common built-in exceptions:

- ZeroDivisionError: Dividing by zero.
- ValueError: Invalid value.
- FileNotFoundError: File not found.
- TypeError: Wrong data type used.
- IndexError: Accessing an invalid index in list.

Chapter # 8
File Handling

File handling allows your program to read from and write to files. It is useful for storing data permanently.

8.1 Why Use File Handling?

- Save data across sessions.
- Read data from text, CSV, or other files.
- Perform tasks like logging, configuration, and exporting results.

8.2 Opening and Reading a File

Use the open() function to open a file. By default, it opens in read mode ("r").

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

8.3 Writing to a File

Use write mode ("w") to write to a file. If the file doesn’t exist, it creates one.

```
file = open("output.txt", "w")
file.write("This is some text.")
file.close()
```

Note: Writing in "w" mode erases existing content.

8.4 Appending to a File

Use append mode ("a") to add data without removing existing content.

```
file = open("output.txt", "a")
file.write("\nThis is a new line.")
file.close()
```

8.5 Reading Line by Line

```
file = open("example.txt", "r")
for line in file:
    print(line.strip())
file.close()
```

8.6 Using the with Statement

The with statement automatically closes the file, even if an error occurs.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

8.7 Writing and Reading Files Together

Use mode "r+" to read and write.

```
with open("data.txt", "r+") as file:
    data = file.read()
    file.write("\nNew content added.")
```

8.8 File Modes Summary

Mode	Description
"r"	Read only
"w"	Write only (overwrite)
"a"	Append
"r+"	Read and write
"x"	Create a new file (error if exists)

8.9 Working with CSV Files

CSV files are used to store tabular data.

```
import csv

# Writing CSV
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Ali", 20])
    writer.writerow(["Sara", 22])
# Reading CSV
```

```
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Chapter # 9
Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a method of structuring a program using objects and classes. It allows you to build applications that are modular, reusable, and easier to manage. This is especially useful for large and complex programs.

9.1 What are Classes and Objects?

- A **class** is a blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that the object will have.
- An **object** is an actual instance created from a class. Multiple objects can be created from the same class.

Example:

```
class Car:
    def drive(self):
        print("The car is moving.")

my_car = Car()     # Creating an object
my_car.drive()     # Output: The car is moving.
```

9.2 The __init__() Method (Constructor)

- The __init__ method is a special method that is automatically called when an object is created.
- It is used to initialize the attributes (variables) of the object.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name    # Assign value to
object attribute
        self.age = age

    def show_info(self):
        print("Name:", self.name)
        print("Age:", self.age)

p1 = Person("Ali", 25)
p1.show_info()
```

Explanation:

- self refers to the current object.

- name and age are parameters.
- self.name and self.age are instance variables attached to the object.

9.3 Instance Variables vs Class Variables

- **Instance variables** are specific to each object. Each object can have different values.
- **Class variables** are shared among all objects of the class.

Example:

```
class Student:
    school = "ABC School"   # Class variable

    def __init__(self, name):
        self.name = name    # Instance variable

s1 = Student("Sara")
s2 = Student("Ahmed")

print(s1.name)         # Output: Sara
print(s2.name)         # Output: Ahmed
print(s1.school)       # Output: ABC School
```

9.4 Inheritance

- Inheritance allows a class to use the properties and methods of another class.

- The class that inherits is called the **child class**, and the class being inherited from is the **parent class**.
- It promotes **code reuse** and **extension**.

Example:

```
class Animal:
    def speak(self):
        print("Animal sound")

class Dog(Animal):     # Dog is inheriting from
Animal
    def bark(self):
        print("Bark")

d = Dog()
d.speak()       # Output: Animal sound
d.bark()        # Output: Bark
```

9.5 Polymorphism

- Polymorphism means having many forms. It allows methods to behave differently based on the object calling them.
- The same method name can exist in multiple classes and perform different tasks.

Example:

```
class Bird:
    def make_sound(self):
        print("Tweet")
```

```

class Cat:
    def make_sound(self):
        print("Meow")

for animal in (Bird(), Cat()):
    animal.make_sound()
```

Output:

```
Tweet
Meow
```

Each object runs its own version of make_sound().

9.6 Encapsulation

- Encapsulation is the process of hiding internal details and exposing only what is necessary.
- You can make variables **private** by prefixing them with double underscores (__).
- You can then control access using methods.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance   # Private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())    # Output: 1500
```

You cannot access __balance directly from outside. This helps protect data from being accidentally changed.

9.7 Summary Table

Concept	**Explanation**
Class	Blueprint for creating objects
Object	Instance of a class
__init__()	Constructor method called when object is created
Instance Variable	Belongs to one specific object
Class Variable	Shared across all objects
Inheritance	Child class uses code from parent class
Polymorphism	Same method name behaves differently
Encapsulation	Hiding internal details using private variables

Chapter # 10
Modules and Packages

Modules and packages help organize Python code into separate files so that it becomes easier to reuse and maintain. Instead of writing everything in a single file, we can split related code into different modules or group them into packages.

10.1 What is a Module?

A **module** is simply a .py file that contains Python code such as variables, functions, or classes. You can **import** and reuse the code from any module in other files.

Types of Modules:

1. **Built-in Modules** (provided by Python)
2. **User-defined Modules** (created by you)

10.2 Importing Built-in Modules

Python comes with many built-in modules. You can use them by importing them using the import keyword.

Example:

```
import math

print(math.sqrt(16))   # Output: 4.0
print(math.pi)         # Output: 3.141592653589793
```

You can also import only specific parts:

```
from math import sqrt

print(sqrt(25))        # Output: 5.0
```

Or rename the module:

```
import math as m

print(m.pow(2, 3))     # Output: 8.0
```

10.3 Creating Custom Modules

You can write your own module by creating a new (.py) file.

Suppose we create a file called mymath.py:

```
# mymath.py
def add(a, b):
    return a + b

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

Now, you can import it in another Python file:

```
# main.py
import mymath

print(mymath.add(5, 3))        # Output: 8
print(mymath.subtract(10, 4)) # Output: 6
```

10.4 What is a Package?

A **package** is a collection of modules grouped inside a folder. The folder must contain a special file named __init__.py (can be empty). This file tells Python that the folder should be treated as a package.

Structure Example:

```
mypackage/
    __init__.py
    math_tools.py
    string_tools.py
```

Now you can import from the package:

```
from mypackage import math_tools
print(math_tools.add(2, 3))
```

10.5 The __name__ == "__main__" Statement

This is a special Python condition used to control the execution of code when a module is run directly versus when it is imported.

Example:

```
# mymodule.py
def greet():
    print("Hello!")

if __name__ == "__main__":
    greet()
```

Explanation:

- If you run mymodule.py directly, it prints "Hello!".
- If you import mymodule from another file, the greet() function will not run automatically.
- This helps prevent unwanted code from executing during import.

10.6 Summary Table

Concept	**Explanation**
Module	A Python file with reusable code
Built-in Module	Comes with Python (e.g., math, random)
Custom Module	Python file you create with your own code
Package	Folder with multiple modules and __init__.py file
import keyword	Used to load and use modules
__name__ == "__main__"	Ensures certain code runs only when the file is executed directly

Chapter # 11
Final Projects

This chapter includes mini projects that help you practice the concepts learned in previous chapters. These are small, real-world applications you can build using basic Python.

Each project is designed to improve your skills in variables, functions, conditionals, loops, file handling, and more.

11.1 Calculator App (Basic Math Calculator)

Purpose:

Perform basic operations like addition, subtraction, multiplication, and division.

Code Example:

```
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 a / b
    else:
        return "Cannot divide by zero"

print("Select operation:")
print("1. Add\n2. Subtract\n3. Multiply\n4.
Divide")

choice = input("Enter choice (1/2/3/4): ")
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

if choice == '1':
    print("Result:", add(num1, num2))
elif choice == '2':
    print("Result:", subtract(num1, num2))
elif choice == '3':
    print("Result:", multiply(num1, num2))
elif choice == '4':
    print("Result:", divide(num1, num2))
else:
    print("Invalid Input")
```

11.2 To-Do List (CLI App)

Purpose:

Manage daily tasks using a command-line interface (CLI).

Code Example:

```
tasks = []

def add_task(task):
    tasks.append(task)
    print("Task added.")

def show_tasks():
    print("Your To-Do List:")
    for i, task in enumerate(tasks, 1):
        print(f"{i}. {task}")

def delete_task(index):
    if 0 < index <= len(tasks):
        removed = tasks.pop(index - 1)
        print(f"Removed: {removed}")
    else:
        print("Invalid task number.")

while True:
    print("\n1. Add Task\n2. Show Tasks\n3. Delete 
Task\n4. Exit")
    choice = input("Choose an option: (1/2/3/4)")

    if choice == '1':
        task = input("Enter task: ")
        add_task(task)
    elif choice == '2':
        show_tasks()
    elif choice == '3':
        num = int(input("Enter task number to 
delete: "))
```

```
        delete_task(num)
    elif choice == '4':
        break
    else:
        print("Invalid choice")
```

11.3 Contact Book

Purpose:

Store and retrieve contact names and phone numbers.

Code Example:

```
contacts = {}

def add_contact(name, number):
    contacts[name] = number
    print("Contact added.")

def search_contact(name):
    if name in contacts:
        print(f"{name}: {contacts[name]}")
    else:
        print("Contact not found.")

while True:
    print("\n1. Add Contact\n2. Search Contact\n3.
Exit")
    choice = input("Choose an option: ")

    if choice == '1':
        name = input("Enter name: ")
        number = input("Enter number: ")
        add_contact(name, number)
    elif choice == '2':
        name = input("Enter name to search: ")
        search_contact(name)
    elif choice == '3':
        break
    else:
        print("Invalid choice")
```

11.4 Number Guessing Game

Purpose:

Randomly generate a number and let the user guess it.

Code Example:

```
import random

number = random.randint(1, 10)
tries = 0

print("Guess a number between 1 and 10.")

while True:
    guess = int(input("Enter your guess: "))
    tries += 1

    if guess < number:
        print("Too low!")
    elif guess > number:
        print("Too high!")
    else:
        print(f"Correct! You guessed it in {tries}
tries.")
        break
```

11.5 Final Mini Project: Student Report Card Generator

Purpose:

Collect and display marks for a student and calculate total and average.

Code Example:

```
def get_marks():
    subjects = ["Math", "Science", "English"]
    marks = {}
    for subject in subjects:
        score = float(input(f"Enter marks for
{subject}: "))
        marks[subject] = score
    return marks

def calculate_report(marks):
    total = sum(marks.values())
    average = total / len(marks)
    return total, average

def display_report(name, marks, total, average):
    print(f"\nReport Card for {name}")
    for subject, score in marks.items():
        print(f"{subject}: {score}")
    print(f"Total Marks: {total}")
    print(f"Average: {average:.2f}")

# Main Program
name = input("Enter student name: ")
marks = get_marks()
total, average = calculate_report(marks)
display_report(name, marks, total, average)
```

Chapter # 12
Wrap-Up and Practice

This final chapter summarizes your Python learning journey and provides direction for further improvement. It includes the best learning resources and practice challenges to sharpen your skills.

12.1 Best Learning Resources

To continue learning and improving your Python skills, here are some trusted and beginner-friendly platforms:

Websites:

- **W3Schools (https://www.w3schools.com/python/):** Great for basic syntax and interactive examples.

- **Programiz (https://www.programiz.com/python-programming):** Clear explanations and beginner-focused tutorials.
- **GeeksforGeeks (https://www.geeksforgeeks.org/python-programming-language/):** Covers both basic and advanced concepts in detail.

Books:

- *Python Crash Course* by Eric Matthes
  A hands-on, project-based introduction to Python.
- *Automate the Boring Stuff with Python* by Al Sweigart
  Great for learning how to apply Python to real-world problems.

Online Platforms for Practice:

- **HackerRank (https://www.hackerrank.com/domains/tutorials/10-days-of-python):** Practice categorized Python problems with instant feedback.
- **LeetCode (https://leetcode.com/):** Useful for improving logic and solving real coding interview-style problems.

12.2 Practice Challenges

Here are some beginner and intermediate-level coding challenges to help you practice what you have learned.

Beginner Challenges:

1. **Even or Odd Checker:**
   Take a number as input and print whether it's even or odd.
2. **Palindrome Checker:**
   Check if a word or number reads the same backward.
3. **Factorial Calculator:**
   Create a function to calculate the factorial of a number.
4. **Count Vowels in a String:**
   Input a sentence and count how many vowels it contains.
5. **Find the Largest Number:**
   Ask the user for three numbers and print the largest one.

Intermediate Challenges:

1. **Simple ATM Simulation:**
   Create a menu for checking balance, depositing, and withdrawing money.
2. **Number to Words Converter:**
   Convert digits into their word form (e.g., 123 → One Two Three).

3. **Simple Login System:**
   Register a username and password, then allow login using the same.
4. **Quiz App:**
   Build a multiple-choice question quiz with score tracking.
5. **Tic-Tac-Toe Game:**
   Console-based two-player game using a 3x3 grid.