



Learn • Code • Create

Code Today
Create
Tomorrow!
😊

PYTHON for KIDS



A Fun and Easy Guide to Learn Python
Step-by-Step!



Easy Lessons
for Beginners



Real Examples
& Practice



Fun Projects
to Build



Quizzes &
Challenges



Step-by-Step
Learning



Perfect for
8-15 Years

```
1 # My First Python Program
2 print("Hello, Kids!")
3 name = input("What is your name? ")
4 age = int(input("How old are you? "))
5 print("Nice to meet you, " + name + "!")
6 print("You are " + str(age) + " years old.")
7 print("Let's code something amazing today! 🚀")
```

Think Like
a **Coder**,
Have Fun,
Build the
Future!



10
Modules

50
Lessons

150+
Topics

10+
Projects

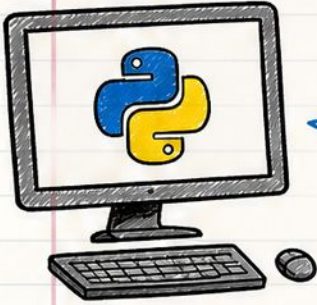
?
Quizzes
& More!



www.codescompiler.com



What is Programming?



Programming is the process of **writing instructions** that tell a computer what to do. These instructions, written in a programming language (like Python), help computers **solve problems** and **perform tasks**.

IDEA

CODE

OUTPUT

1 What is Coding?

Coding is the act of **writing code** using a programming language.

It's how we communicate with computers to build software, websites, apps and more!

```
# Hello, World in Python
print("Hello, World!")
```

This is a simple Python code that prints a message on the screen.

2 Why Learn Programming?



Boosts Problem-Solving Skills

Helps you think logically and solve real-world problems.



Great Career Opportunities

High demand in tech, data science, AI, web development and more.



Build Amazing Things

Create websites, apps, games, automate tasks and bring ideas to life!



Future-Proof Your Skills

Programming is a superpower that stays useful in every future.

3 Real-Life Uses of Python

1 Web Development



Python is used to build powerful websites and web applications.

Example: Django, Flask

2 Data Science & Analytics



Python helps analyze data, visualize trends and make smart decisions.

Example: Pandas, NumPy, Matplotlib

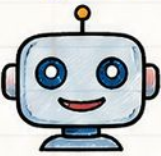
3 Automation



Automate boring and repetitive tasks like file handling, emails, etc.

Example: Scripts, Bots

4 Artificial Intelligence



Python powers AI, machine learning and deep learning applications.

Example: TensorFlow, PyTorch

5 Data Visualization



Create stunning charts, dashboards and reports with ease.

Example: Seaborn, Plotly

6 Game Development



Python can be used to build 2D games and interactive applications.

Example: Pygame



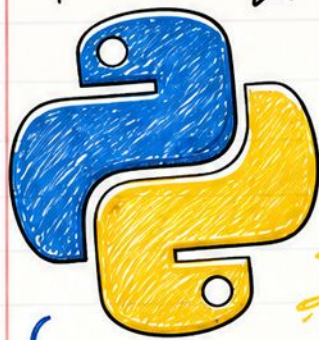
Start small. Keep practicing. Build projects.

The code you write today can **change your future!**

EVERY EXPERT
WAS ONCE A
BEGINNER. KEEP
CODING!



www.codescompiler.com



GETTING STARTED WITH PYTHON

Python is easy to learn, powerful, and fun to code!



Let's begin your Python journey! ☆

1. Installing Python

Python is available for Windows, macOS and Linux. Follow these simple steps:

- 1 Go to the official website:
<https://www.python.org/downloads/>
- 2 Download the latest Python 3.x version for your operating system.
- 3 Run the installer.
Important: ☒ Check "Add Python to PATH" before clicking Install Now.
- 4 Click Install Now and wait for completion.
- 5 Verify installation:

Open Command Prompt / Terminal and type:
`python --version`
You should see the Python version installed.



Why Python?

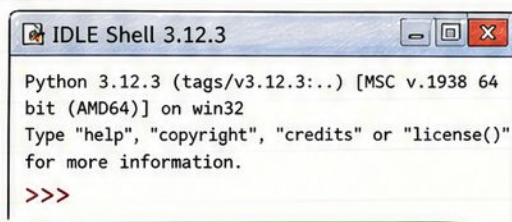
- ☒ Easy to learn and read
- ☒ Open source and free
- ☒ Huge community support
- ☒ Used in Web, Data Science, AI, Automation, and more!

Fun Fact

The name "Python" comes from the British comedy group "Monty Python".



2. Python IDLE



IDLE is Python's built-in development environment. It is great for beginners!

IDLE (Integrated Development and Learning Environment) comes bundled with Python.

You can use IDLE to:

- Write and run Python programs
- Interact with the Python shell
- Test code quickly
- Learn and explore



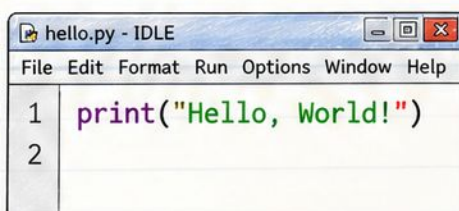
How to open IDLE?

- Windows: Search "IDLE" in the Start Menu.
- macOS: Open Terminal and type: `idle3`
- Linux: Usually available in applications menu as "IDLE (Python 3)".

3. Writing Your First Program

Let's write the classic first program: "Hello, World!"

- 1 Open IDLE.
- 2 Go to File > New File.
- 3 Type the following code:



- 4 Save the file as `hello.py`
- 5 Click Run > Run Module (or press F5)
- 6 See the output in the Shell window:



Congratulations!

You just wrote and ran your first Python program. 😊



Tips for Beginners

- ★ Practice regularly.
- ★ Don't worry about mistakes—keep learning!
- ★ Explore and have fun with code.

Python is a simple, powerful and beginner-friendly language.



Hello, Python!

Let's write our first Python program! 😊



Every great journey begins with a single step. In Python, it starts with printing "Hello, Python!" to the screen. Let's explore how! ❤️

1 The print() Function

The `print()` function is used to display output on the screen.

SYNTAX

`print(value)`

↑ ↑
function name value to be displayed



The value can be a string (text), number, or any other data.

2 Displaying Messages

Use `print()` to show messages, text, or results.

EXAMPLES

```
print("Hello, Python!")      # simple message
print('Welcome to Python!') # single quotes
print(123)                    # numbers
print("Python is fun!")      # another message
```

OUTPUT

Hello, Python!
Welcome to Python!
123
Python is fun!



3 Running Programs

Follow these simple steps to run your Python program.

1. Write Code



Open a code editor (IDLE, VS Code, PyCharm, or any editor). Write your Python code and save the file.

2. Save File



Save your file with a .py extension.
Example: hello.py

3. Run Program



Run the file using the Run button in your editor or from the terminal.

4. See Output



The output will be displayed in the console or terminal.

Ways to Run

- Using IDE: Click the Run button (▶) in your editor.
- Using Terminal: Open terminal/command prompt and type:
`python hello.py`

QUICK TIPS

- ✓ Python is case-sensitive.
- ✓ Every statement is written on a new line.
- ✓ Use quotes (" " or ' ') for text.
- ✓ Practice by changing the message and running again!

You just printed your first message in Python.

Great job!



Keep coding, keep growing! ❤️



www.codescompiler.com



In Python, errors are messages that help us find problems in our code.

UNDERSTANDING ERRORS

Errors are normal. Learning from them makes you a better Python programmer! ❤️



1 SYNTAX ERRORS

Syntax errors occur when Python cannot understand the code because it breaks the rules of the language.

EXAMPLE

```
print("Hello World")
```

SyntaxError: unexpected EOF while parsing

Why it happens?

- Missing parentheses, brackets or colons
- Incorrect indentation
- Forgetting quotes around strings

TIP

Read the error message carefully. It shows the line and the problem!

2 TYPING MISTAKES

These are not syntax errors. The code runs, but gives wrong results or raises an error because of incorrect typing.

EXAMPLES

- ❌ `print("Hello)` (missing closing quote)
Error: EOL while scanning string literal
- ❌ `pritrn("Hello")` (misspelled function name)
Error: NameError: name 'pritrn' is not defined
- ❌ `my_var = 10`
`print(my-var)` (using - instead of _)
Error: NameError: name 'my' is not defined
- ❌ `age = input("Enter age: ")`
`print(age + 5)` (string + int)
Error: TypeError: can only concatenate str (not "int") to str

REMEMBER

Small typos can cause big problems! Always check spelling, quotes, and variable names.

3 FIXING COMMON ERRORS

Follow a systematic approach to find and fix errors quickly.

1

READ THE ERROR MESSAGE



Python tells you what went wrong and where. Read it carefully!

Example:

SyntaxError: invalid syntax (line 3)

2

CHECK THE LINE NUMBER



Go to the line number mentioned in the error message and inspect your code there.

3

FIND THE CAUSE



Look for missing symbols, typos, wrong indentation, or incorrect logic.

4

FIX THE CODE



Make the necessary correction and save your file.

5

RUN AGAIN & TEST



Run the code again. If no error appears, test with different inputs!

GOOD PRACTICES

- ✓ Write code slowly and carefully.
- ✓ Use meaningful variable names.
- ✓ Indent your code properly.
- ✓ Test your code in small parts.
- ✓ Don't ignore warnings and tracebacks!



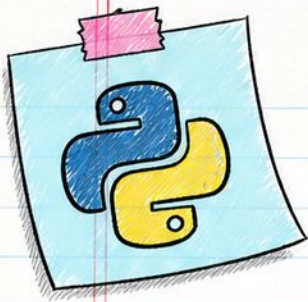
KEEP PRACTICING!

Everyone makes mistakes. Great programmers are great error fixers!



Keep coding. Keep learning. You've got this! 🚀

www.codescompiler.com



MINI PROJECT



Small Projects
Big Learning!

Build 3 simple Python programs!



Mini projects help you practice Python basics like `input()`, `print()`, variables and string formatting.



1

GREETING PROGRAM

This program takes the user's name and greets them with a friendly message.

Hello!



CODE

```
name = input("What is your name? ")  
print("Hello, " + name + "! 😊")
```

OUTPUT

What is your name? Alex
Hello, Alex! 😊

2

FAVORITE COLOR PROGRAM

This program asks the user for their favorite color and shows a fun response.



CODE

```
color = input("What is your favorite color? ")  
print("Wow! " + color + " is a great color!")  
print("Keep shining in your " + color + " world! ✨")
```

OUTPUT

What is your favorite color? Blue
Wow! Blue is a great color!
Keep shining in your Blue world! ✨

You can change the answers to make it yours! ★

3

FUN MESSAGE GENERATOR

This program creates a funny or cool message using two inputs from the user.

Let's make some magic!



CODE

```
adjective = input("Enter an adjective (e.g. amazing): ")  
thing = input("Enter a noun (e.g. coder): ")  
print("You are a " + adjective + " " + thing + "!")  
print("Keep being awesome! 🚀")
```

OUTPUT

Enter an adjective (e.g. amazing): creative
Enter a noun (e.g. coder): developer
You are a creative developer!
Keep being awesome! 🚀

KEY CONCEPTS USED

- ✓ `input()` - to take user input
- ✓ `print()` - to display output
- ✓ Variables - to store data
- ✓ String Concatenation - to combine text
- ✓ Creativity - to make your programs unique! 🌈



TIPS

- ★ Try running the programs with different inputs.
- ★ Add more messages and emojis.
- ★ Share your programs with friends!
- ★ Practice daily and have fun! 🎉



Keep Coding. Keep Learning. Keep Building! ❤️



What are Variables?

Variables are like labeled boxes that store data in Python.



A variable is a name given to a memory location where we store information that can be used and changed in our program.

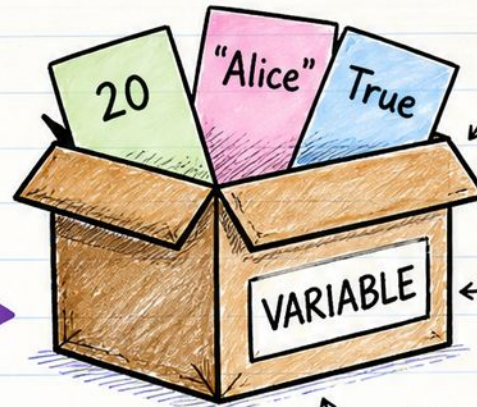


1 STORING INFORMATION

Variable let us store different types of data like numbers, text, or even True/False values.

Example:

```
age = 20
name = "Alice"
is_student = True
```



We put a value inside the box.

The box has a name.

We can use this value later in our program.

2 NAMING VARIABLES

In Python, variable names must follow certain rules.



- Start with a letter or underscore (_)
- Can contain letters, numbers, and underscores
- Case-sensitive (age, Age and AGE are different)
- Should be meaningful and descriptive



Invalid Examples



- 1age (cannot start with a number)
- age@ (special characters not allowed)
- my name (no spaces allowed)
- for (Python keywords cannot be used)



Good Examples: age, student_name, totalMarks, _count1

3 UPDATING VALUES

We can change the value stored in a variable whenever needed.

Example:

```
score = 10          # Initial value
print(score)        # Output: 10
score = score + 5    # Update value
print(score)        # Output: 15
```



The variable name stays the same, but the value inside the box changes!



Variables make our programs flexible and powerful!



www.codescompiler.com





DATA TYPES

In Python

Everything in Python is an **OBJECT**, and every object has a **DATA TYPE**.

A **data type** defines the kind of value a variable can hold and the operations we can perform on it.

1

TEXT (STRINGS)

Strings are used to represent text.

They are written inside single quotes `' '` or double quotes `" "`.

EXAMPLES

```
name = "Alice"
greeting = 'Hello, Python!'
empty = ""
```

KEY POINTS

- ✓ Ordered collection of characters
- ✓ Supports indexing and slicing
- ✓ Immutable (cannot be changed)

COMMON OPERATIONS

```
len(name)      # Length of string
name[0]         # First character
name[-1]        # Last character
name.upper()    # Convert to upper case
"Py" + "thon"   # Concatenation
```

2

NUMBERS (INTEGERS)

Integers are **whole** numbers without any decimal part. They can be positive, negative, or zero.

EXAMPLES

```
age = 25
temperature = -10
count = 0
big_number = 1000000
```

KEY POINTS

- ✓ No decimal part
- ✓ Can be positive, negative or zero
- ✓ No limit to size in Python

COMMON OPERATIONS

```
a = 10, b = 3
a + b      # Addition (13)
a - b      # Subtraction (7)
a * b      # Multiplication (30)
a // b     # Floor Division (3)
a % b      # Modulus (1)
a ** b     # Exponent (1000)
```

3

DECIMAL NUMBERS (FLOATS)

Floats (floating point numbers) are numbers with a **decimal point**.

EXAMPLES

```
pi = 3.14159
price = 19.99
temperature = -2.5
scientific = 1.2e3
           (1.2 x 103 = 1200.0)
```

KEY POINTS

- ✓ Has a decimal point
- ✓ More precision than integers
- ✓ Used for real-world measurements

COMMON OPERATIONS

```
x = 5.5, y = 2.0
x + y      # 7.7
x - y      # 3.5
x * y      # 11.0
x / y      # 2.75
round(x, 1) # 5.5 (rounded to 1 decimal)
```

Python is **dynamically typed**, which means you don't need to declare the type of a variable. Python automatically determines the type based on the value you assign.

Keep Practicing, Keep Coding! ❤️

www.codescompiler.com

WORKING WITH STRINGS

In Python, a **string** is a sequence of characters enclosed in quotes.

Strings help us work with text data. ❤️

1 CREATING TEXT

We can create strings by placing characters inside quotes.



```
>>> name = "Alice"
>>> message = 'Hello, World!'
>>> multiline = """This is
a string that spans
multiple lines."""
```

Examples

- "Hello" # double quotes
- 'Hello' # single quotes
- "Python is fun!" # a sentence
- """Multiple lines string""" # multiline string



2 COMBINING TEXT

We can combine (concatenate) strings in different ways.

```
>>> first = "Hello"
>>> last = "World"
>>> combined = first + " " + last
>>> combined
'Hello World'

>>> message = "Python " * 3
>>> message
'Python Python Python '
```

Ways to Combine:

- +** Using + operator
Joins strings together.
"Hi" + " there" -> "Hi there"
- *** Using * operator
Repeats a string.
"Py" * 4 -> "PyPyPyPy"
- f** Using f-strings (Python 3.6+)
Embed variables in strings easily.
name = "Sam"
greeting = f"Hello, {name}!" -> "Hello, Sam!"

Strings can be joined, repeated & formatted!

3 STRING LENGTH

We use the len() function to find the number of characters in a string.

```
>>> text = "Python Programming"
>>> len(text)
18
>>> len("")
0
>>> len("Hello!")
6
```



Notes

- ★ Spaces are counted.
- ★ Empty string has length 0.
- ★ Indexing in strings starts from 0.

```
text = "Python"
0 1 2 3 4 5
P y t h o n
```

Quick Example

```
>>> name = "Code"
>>> len(name)
4
>>> name + "s"
'Codes'
```



TIPS

- Strings are immutable (cannot be changed).
- Use meaningful variable names for text data.
- Explore more string methods: .upper(), .lower(), .strip(), .replace() and more!



Python programs can talk to users using `input()`. It allows our program to ask questions and get answers! ❤️

USER INPUT



User input is the information we get from the person using our program. Python provides the `input()` function to make this easy.

You ask.
User answers.
Program uses it!

1 USING `input()`

The `input()` function shows a prompt (question) to the user and waits for them to type an answer.

```
name = input("What is your name? ")  
print("Hello, " + name + "!")
```

Example Output

What is your name? Alex
Hello, Alex!

Key Points

- ★ `input()` always returns a string.
- ★ Use quotes inside `input()` to show the prompt to the user.
- ★ The program waits until the user presses **Enter**.

2 ASKING QUESTIONS

We can ask anything! `input()` lets us collect different kinds of information.

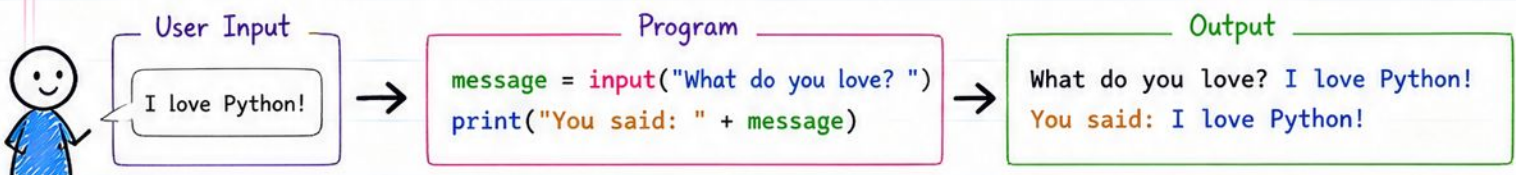
- | | | |
|------------------|---|---|
| ✓ Name |  | <code>name = input("What is your name? ")</code> |
| ✓ Age |  | <code>age = input("How old are you? ")</code> |
| ✓ Favorite Color |  | <code>color = input("What is your favorite color? ")</code> |
| ✓ City |  | <code>city = input("Which city do you live in? ")</code> |

Prompt Tips

- Be clear and friendly.
- End with a space for better user experience.
- Example: "Enter your age: "

3 SAVING ANSWERS

The answer typed by the user is saved in a **variable**. We can use it in our program.



★ Important: Data Types

`input()` returns a **string**. If you need a number, convert it using `int()` or `float()`.

```
age = int(input("Enter your age: ")) # integer  
price = float(input("Enter price: ")) # decimal
```



Remember:

Great programs listen to their users! 😊

Ask questions. Get answers.
Make your programs interactive!

Practice more,
Build awesome! 😊

MINI PROJECT

Build simple. Think big. Code your ideas!

LEARN
BUILD
SHARE

In this mini project collection, you will create three beginner-friendly programs using Python. These projects help you practice `input()`, `print()`, variables, and string formatting.

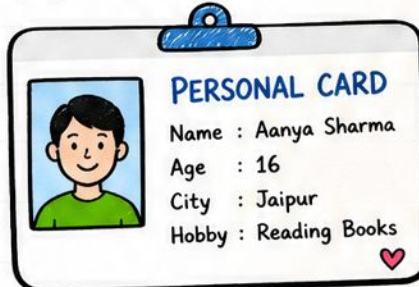

1

PERSONAL INFORMATION CARD

Create a program that takes your personal details as input and displays them in a well-formatted card.

Take input for:

- ✓ Full Name
- ✓ Age
- ✓ City
- ✓ Hobby



SAMPLE CODE (HINT)

```
name = input("Enter your name: ")
age = input("Enter your age: ")
city = input("Enter your city: ")
hobby = input("Enter your hobby: ")
```

```
print("\n--- PERSONAL INFORMATION CARD ---")
print("Name :", name)
print("Age :", age)
print("City :", city)
print("Hobby :", hobby)
```

2

FAVORITE THINGS SURVEY

Ask the user about their favorite things and store the answers. Then show a summary of the survey.

Ask about:

- ★ Favorite Color
- ★ Favorite Food
- ★ Favorite Movie
- ★ Favorite Subject

SURVEY RESULT

Color : Blue
Food : Pizza
Movie : Inception
Subject : Computer Science

SAMPLE CODE (HINT)

```
color = input("Favorite color: ")
food = input("Favorite food: ")
movie = input("Favorite movie: ")
subject = input("Favorite subject: ")
```

```
print("\n--- SURVEY RESULT ---")
print("Color :", color)
print("Food :", food)
print("Movie :", movie)
print("Subject :", subject)
```

3

INTRODUCTION GENERATOR

Create a program that generates a friendly introduction using details entered by the user.

Make your
own intro
in style! 😊

Hello! 🙌

My name is Aanya Sharma.
I am 16 years old.
I live in Jaipur.
My hobby is Reading Books.
Nice to meet you! 😊

SAMPLE CODE (HINT)

```
name = input("Your name: ")
age = input("Your age: ")
city = input("Your city: ")
hobby = input("Your hobby: ")

print(f"Hello! 🙌")
print(f"My name is {name}.")
print(f"I am {age} years old.")
print(f"I live in {city}.")
print(f"My hobby is {hobby}.")
print("Nice to meet you! 😊")
```

TIPS

- ★ Use meaningful variable names.
- ★ Use `\n` for new lines.
- ★ Make your output clean and user-friendly.
- ★ Experiment and have fun!

WHAT YOU PRACTICE

- ✓ Input from user
- ✓ Variables & Data types
- ✓ String formatting
- ✓ `print()` function
- ✓ Creativity & Logic

MORE IDEAS?

- Daily Quote Generator
- Simple Calculator
- To-Do List (Basic)
- Birthday Wish Generator
- ... and many more!

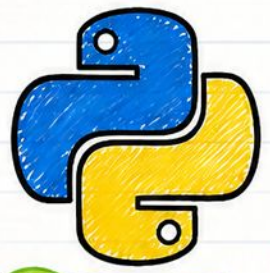
Code.
Learn.
Repeat.

Python can do more than just code - it can do math too! 😊

PYTHON

as a

CALCULATOR



Python makes calculations easy and fun! Let's do some math!



Python supports all basic arithmetic operations. Let's see how we can use it as a **calculator**!



1

ADDITION

Adds two or more numbers and gives the total.

Operator: **+**

EXAMPLE

```
a = 25
b = 17
result = a + b
print(result)
```

OUTPUT

42

So, **25 + 17 = 42**



2

SUBTRACTION

Subtracts one number from another and gives the difference.

Operator: **-**

EXAMPLE

```
a = 50
b = 18
result = a - b
print(result)
```

OUTPUT

32

So, **50 - 18 = 32**



3

MULTIPLICATION

Multiplies two numbers and gives the product.

Operator: *****

EXAMPLE

```
a = 12
b = 8
result = a * b
print(result)
```

OUTPUT

96

So, **12 * 8 = 96**



QUICK TIPS

- Numbers can be whole numbers or decimals.
- You can store calculations in variables.
- Python follows BODMAS (PEMDAS) rules.
- Use parentheses () to control the order.
- Try combining operations!

Example: **(5 + 3) * 2 = 16** ✓

TRY IT YOURSELF!

What will be the output? ?

```
a = 7
b = 3
result = (a + b) * 4 - 5
print(result)
```

OUTPUT

—



Python isn't just a programming language, it's your smart calculator! 🐍💡

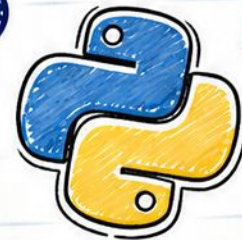


MORE MATH OPERATIONS



These operations help you solve real-world problems with ease! 😊

Python provides powerful operators to perform **more advanced mathematical** operations.



1 DIVISION

Used to divide one number by another.

In Python 3, the `/` operator always returns a **float** (decimal) result.



```
a = 20
b = 4
result = a / b
print(result)
# Output: 5.0
```

MORE TO KNOW:

- `/` → True Division (returns float)
 - `//` → Floor Division (returns integer part)
- $20 / 4 \rightarrow 5.0$
 $20 // 4 \rightarrow 5$

2 MODULUS

Returns the remainder after dividing one number by another.

Very useful in tasks like:

- ✓ Checking even or odd
- ✓ Cyclic operations
- ✓ Index wrapping



```
a = 20
b = 3
result = a % b
print(result)
# Output: 2
```

EXAMPLES:

- $10 \% 3 \rightarrow 1$
- $15 \% 4 \rightarrow 3$
- $7 \% 2 \rightarrow 1$ (odd)
- $8 \% 2 \rightarrow 0$ (even)



Tip: Use `% 2` to check if a number is **even** (0) or **odd** (1).

3 EXPONENTS

Used to raise a number (base) to the power of another (exponent).

In Python, use the `**` operator for exponents.

$$a^b \rightarrow a ** b$$

```
a = 2
b = 5
result = a ** b
print(result)
# Output: 32
```

EXAMPLES:

- $2 ** 3 \rightarrow 8$ (2^3)
- $5 ** 2 \rightarrow 25$ (5^2)
- $10 ** 0 \rightarrow 1$ (any number to power 0 is 1)
- $3 ** 4 \rightarrow 81$



QUICK RECAP



Division
Divide numbers.

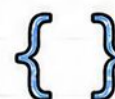


Modulus
Remainder after division.



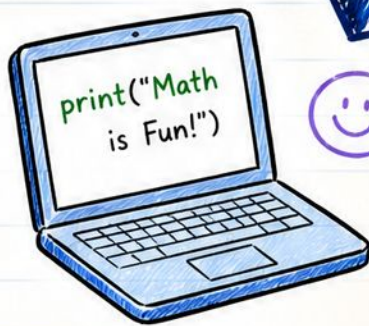
Exponents
Raise to a power.

Practice makes perfect! Keep coding! 💻



VARIABLES

in MATH



In Python, variables help us store numbers, perform calculations, and update values easily.

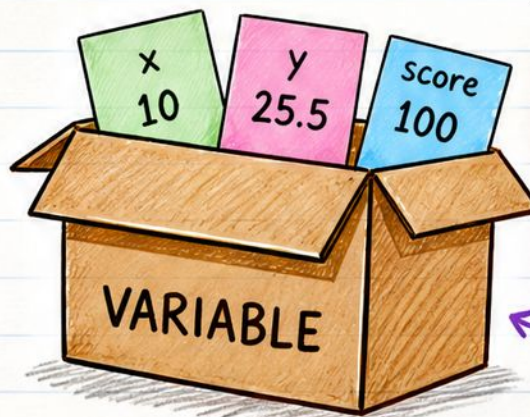
A variable is a **named container** that stores a value you can use and **change**.



1 STORING NUMBERS

We can **store numbers** in variables and use them anytime.

```
x = 10      # x stores 10
y = 25.5    # y stores 25.5
score = 100  # score stores 100
```



Think of a variable as a labeled box that holds a value.

2 CALCULATING RESULTS

We can use variables in **math operations** to calculate new results.

```
a = 12      # a stores 12
b = 8        # b stores 8
total = a + b # 12 + 8 = 20
product = a * b # 12 * 8 = 96
difference = a - b # 12 - 8 = 4
average = total / 2 # 20 / 2 = 10.0
```

Example Result

```
total    = 20
product  = 96
difference = 4
average  = 10.0
```



3 UPDATING SCORES

We can **change the value** of a variable as our program runs.

```
score = 50      # initial score
score = score + 10 # add 10 points
score = score - 5  # lose 5 points
score = score + 20 # add 20 points
print(score)      # final score: 75
```



Variables are flexible—update them anytime to keep track of changes!



Good variable names make your code easy to read!

✓ player_score ✓ total_marks ✗ x1

Quick Tips

- ★ Use meaningful names
- ★ Variables store values
- ★ Values can change
- ★ Used in calculations & beyond!

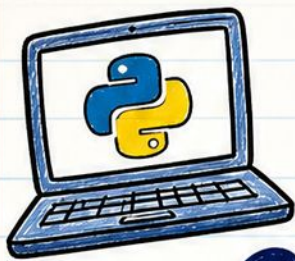


Learn. Code. Compile. Succeed!



www.codescompiler.com





FUN CALCULATIONS

Small Programs,
Big Learning! ♥

Simple Python Projects for Everyday Use ♥



1 AGE CALCULATOR

Calculate your exact age in years, months and days!



How it works:

1. Enter your Date of Birth
2. Enter today's date
3. Get your age in Years, Months & Days

```
from datetime import date
```

```
dob = date(2005, 7, 21)  
today = date.today()  
age = today - dob
```

```
years = age.days // 365  
months = (age.days % 365) // 30  
days = (age.days % 365) % 30
```

Example Output

You are
18 years,
9 months and
12 days old. 😊

2 DAYS CALCULATOR

Find the total number of days between two dates!



How it works:

1. Enter start date
2. Enter end date
3. Get total days between the dates

```
from datetime import date
```

```
start = date(2024, 1, 1)  
end = date(2024, 12, 31)  
delta = end - start
```

```
total_days = delta.days
```

Example Output

Total days
between the
dates: 365

3 POCKET MONEY TRACKER

Track your income, expenses and savings like a pro!



Features:

- ★ Add Income
- ★ Add Expense
- ★ View Balance
- ★ Save Smart 💰

```
balance = 0
```

```
while True:
```

```
    print("\n1. Add Income")
```

```
    print("2. Add Expense")
```

```
    print("3. View Balance")
```

```
    print("4. Exit")
```

```
    choice = input("Choose an option: ")
```

```
    if choice == '1':  
        amount = float(input("Amount: "))  
        balance += amount  
        print("Income added!")
```

```
    elif choice == '2':  
        amount = float(input("Amount: "))  
        balance -= amount  
        print("Expense added!")
```

```
    elif choice == '3':  
        print(f"Your Balance: ₹{balance:.2f}")
```

```
    elif choice == '4':  
        break
```

```
    else:  
        print("Invalid option!")
```

Sample Run

1. Add Income
2. Add Expense
3. View Balance
4. Exit

Choose an option: 1
Amount: 500

Income added!

Your Balance: ₹500.00

Choose an option: 2
Amount: 120

Expense added!

Your Balance: ₹380.00

Practice Today,
Build Tomorrow!



www.codescompiler.com





Python

MODULE 3

MINI PROJECT

Practice.
Build.
Improve!



Apply what you've learned in Python by building these mini projects. *Simple ideas, real learning!*



PROJECT IDEAS

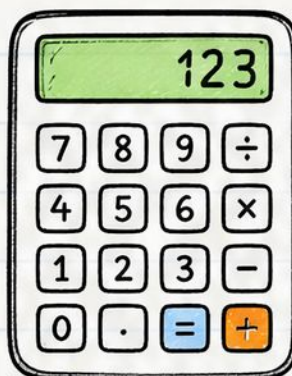
1

BASIC CALCULATOR

Create a calculator that can perform basic arithmetic operations.

Operations:

Addition (+) Subtraction (-)
Multiplication (*) Division (/)



Sample Run:

Enter first number: 12
Enter operator (+, -, *, /): +
Enter second number: 8
Result: 20

2

SHOPPING TOTAL CALCULATOR

Build a program that calculates the total cost of items in a cart and applies discount if any.

Features:

- ✓ Add multiple items
- ✓ Apply discount
- ✓ Calculate total
- ✓ Show final amount



Sample Run:

Enter number of items: 3
Enter price of item 1: 120
Enter price of item 2: 85
Enter price of item 3: 45
Enter discount (%): 10
Total Payable: 225.00

3

SCORE COUNTER

Create a score counter for a game or quiz. Track points and display the final score.

You can extend:

- Reset scores
- Add multiple players
- Find highest score



Sample Run:

Player 1 score: 15
Player 2 score: 22
Player 1 score: 10
Player 2 score: 18
Final Score →
Player 2 Wins! (40 points)

TIPS

1 Break the problem into small steps.



2 Use input(), variables, operators and conditionals.



3 Test with different values and improve your program.



Keep Coding. Keep Growing! ❤️



www.codescompiler.com





INTRODUCTION TO CONDITIONS

Conditions help our programs make decisions!



A condition is a statement that can be either **True** or **False**. Python uses conditions to decide what to do next.

1 TRUE AND FALSE

Every condition evaluates to one of these two values:

TRUE



The condition is satisfied.
(Yes, it is!)

FALSE



The condition is not satisfied.
(No, it isn't!)

Example:

```
5 > 3           # True
10 == 10        # True
2 < 1           # False
'apple' == 'apple' # True
'hi' != 'hello'  # True
```



2 COMPARING VALUES

Conditions are usually built by comparing values using comparison operators.

Operator	Meaning	Example	Result
<code>==</code>	Equal to	<code>5 == 5</code>	True
<code>!=</code>	Not equal to	<code>5 != 3</code>	True
<code>></code>	Greater than	<code>7 > 4</code>	True
<code><</code>	Less than	<code>2 < 3</code>	True
<code>>=</code>	Greater than or equal to	<code>6 >= 6</code>	True
<code><=</code>	Less than or equal to	<code>4 <= 2</code>	False

3 WHY CONDITIONS MATTER

Conditions let your program think and make decisions. They are the foundation of:



Decision Making
Choose different paths in your program.



Smart Behavior
Respond to user input, data, or situations.



Real-World Logic
Used in games, apps, calculators, and more!



Mini Example



```
age = 18
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

The condition `age >= 18` is checked. It's **True**, so the first block runs.



Remember:

Conditions always evaluate to **True** or **False**. They help your program make smart choices!





Using if

Make smart decisions in your code!

What is if?

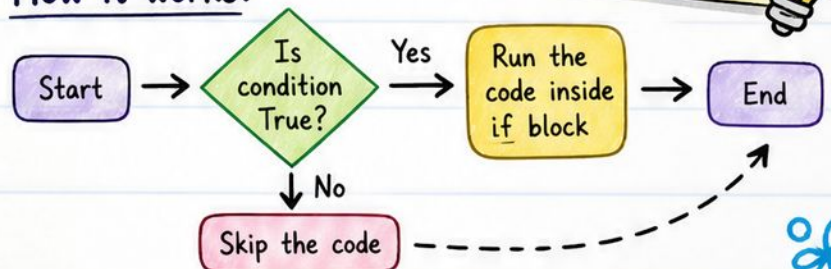
The if statement lets your program run certain code only when a condition is True.



General Structure:

```
if condition:
    # code runs if condition is True
# code here is skipped if condition is False
```

How it works:



1 Simple Decisions

Use if to run code when a condition is True.



Example:

```
age = 18
if age >= 18:
    print("You are an adult!")
print("End of program")
```

Output:

You are an adult!
End of program

Key Point



If the condition is True, the indented code runs.

2 Checking Numbers

Use comparison operators to check numbers.

Common Operators

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- == Equal to
- != Not equal to

Example:

```
score = 85
if score >= 60:
    print("You passed!")
else:
    print("You failed.")
```

Output:

You passed!

Tip



You can add else to run code when the condition is False.

3 Checking Names

Use if to check text values.

Strings are text inside quotes.

66 99

Example:

```
name = "Alice"
if name == "Alice":
    print("Hello Alice!")
else:
    print("Hello friend!")
```

Output:

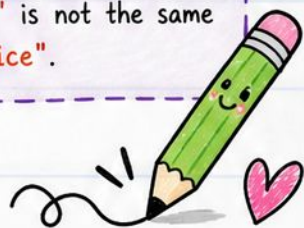
Hello Alice!

Note



String comparison is case-sensitive!
"Alice" is not the same as "alice".

Keep practicing and build amazing programs!



Python helps us make smart decisions using **if-else** statements.

Using if-else

Think. Decide. Act!

Make decisions in your code!

What is if-else?

The **if-else** statement allows your program to choose between **TWO** possible outcomes based on a condition.

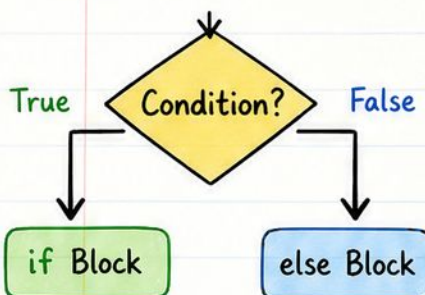
Syntax

```
if condition:  
    # code to run if condition is True  
else:  
    # code to run if condition is False
```

Key Concepts

1 Two Possible Outcomes

if leads to one block of code.
else leads to another.



Only one block runs, not both!

2 Positive & Negative Results

Use **if-else** to handle positive (**True**) and negative (**False**) results.



Positive (**True**)

Condition is met.
Run **if** block.



Negative (**False**)

Condition is not met.
Run **else** block.

3 User Choices

if-else helps your program respond to the choices or inputs from users.

Enter your choice (1 or 2):

- 1 Show Greeting
- 2 Show Farewell

The program does different actions based on what the user selects!

Example

Let's check if a number is **positive** or **negative**.

```
num = int(input("Enter a number: "))  
if num > 0:  
    print("The number is positive.")  
else:  
    print("The number is negative or zero.")
```

Try it!

- ★ Enter 10 → The number is positive.
- ★ Enter -5 → The number is negative or zero.
- ★ Enter 0 → The number is negative or zero.

Pro Tip

Indentation is important in Python! Make sure your **if** and **else** blocks are indented properly.

Keep practicing. Keep coding. Keep deciding! ❤️



Using elif

Smarter decisions with multiple conditions!



The **elif** statement lets you check multiple conditions one by one.

If the first condition is **False**, Python moves to the next **elif** condition.

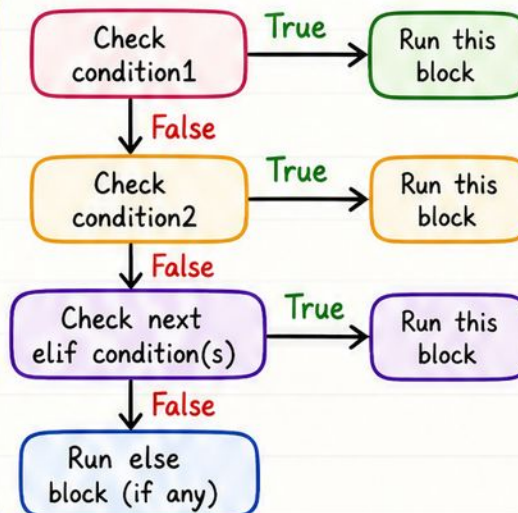


SYNTAX

```
if condition1:
    # block of code
elif condition2:
    # block of code
elif condition3:
    # block of code
else:
    # block of code
```



HOW IT WORKS



KEY POINTS

- ✓ Conditions are checked in **order**.
- ✓ Only the **first True** condition runs.
- ✓ Use **else** for the case when all conditions are **False**.
- ✓ **elif** is optional - you can have one or many.

REAL-WORLD EXAMPLES

1

GRADING SYSTEM

Assign grades based on marks.



```
marks = int(input("Enter marks: "))
if marks >= 90:
    grade = 'A'
elif marks >= 75:
    grade = 'B'
elif marks >= 60:
    grade = 'C'
elif marks >= 40:
    grade = 'D'
else:
    grade = 'F'
print("Grade:", grade)
```

RANGE

90 - 100 → A
75 - 89 → B
60 - 74 → C
40 - 59 → D
0 - 39 → F



2

WEATHER DECISIONS

Give suggestions based on weather.



```
temp = int(input("Enter temperature: "))
if temp >= 30:
    advice = 'Stay hydrated! ☀️'
elif temp >= 20:
    advice = 'Perfect for outing! 😊'
elif temp >= 10:
    advice = 'Wear a light jacket. 🧥'
elif temp >= 0:
    advice = 'Wear something warm. 🧣'
else:
    advice = 'It's freezing! ❄️'
print(advice)
```

EXAMPLES

- 32°C → Stay hydrated!
- 25°C → Perfect for outing!
- 12°C → Light jacket
- -2°C → It's freezing!



QUICK TIP



Use **elif** when you have multiple choices. It keeps your code clean, readable and efficient!



Think of it as:
First match wins!



MINI PROJECT

Module 4 - Python Programming

Mini Projects help you apply Python concepts, strengthen logic and build real-world skills!

PROJECT IDEAS

1

AGE CHECKER



A simple program that checks a person's age and tells whether they are a child, teenager or adult.

KEY CONCEPTS

- Input()
- If-elif-else
- Comparison Operators

```
age = int(input("Enter your age: "))
if age < 13:
    print("You are a Child.")
elif age < 20:
    print("You are a Teenager.")
else:
    print("You are an Adult.")
```

2

GRADE CALCULATOR



Takes marks as input and calculates percentage and displays the grade.

KEY CONCEPTS

- Input()
- Arithmetic Operators
- If-elif-else
- Logical Thinking

```
m1 = float(input("Maths: "))
m2 = float(input("Science: "))
m3 = float(input("English: "))
percent = (m1+m2+m3)/3
if percent >= 90:
    grade = 'A'
elif percent >= 75:
    grade = 'B'
elif percent >= 60:
    grade = 'C'
else:
    grade = 'D'
print(f"Percentage: {percent:.2f}%")
print(f"Grade: {grade}")
```

3

SMART QUIZ



A quiz program that asks questions, checks answers, and shows the final score.

KEY CONCEPTS

- Lists
- Loops (for)
- Conditionals
- Score Tracking

```
questions = [("2+2=?", "4"), ("5*3=?", "15")]
score = 0
for q, ans in questions:
    user = input(q + " ")
    if user == ans:
        print("Correct! 🎉")
        score += 1
    else:
        print("Oops! ❌")
print(f"Your Score: {score}/{len(questions)}")
```

TIPS -

- ★ Break the problem into smaller steps.
- ★ Use meaningful variable names.
- ★ Test your code with different inputs.
- ★ Add your own features and make it creative!

Practice today,
Build tomorrow,
Code your future!



www.codescompiler.com



WHY USE LOOPS?



Write Less Code, Do More!

Loops allow us to repeat a block of code multiple times, making our programs **smarter**, **shorter** and more **efficient**.



1 REPEATING TASKS

Loops let us run the same code over and over again, without writing it again and again.

Without Loops (**Tedious!** 😞)

```
print("Hello!")
print("Hello!")
print("Hello!")
print("Hello!")
print("Hello!")
```

VS.

With Loops (**Easy!** 😎)

```
for i in range(5):
    print("Hello!")
```

Think of it like:

Telling someone, "Do this 5 times" instead of saying "Do this" 5 times!

2 SAVING TIME

Loops save time, reduce errors, and make code easier to read and maintain.



More Code
= More Time

- ✗ More typing
- ✗ More chances of mistakes
- ✗ Harder to update



Fewer Lines
= Faster Development

- ✓ Less typing
- ✓ Fewer mistakes
- ✓ Easy to update

3 LOOP EXAMPLES

Python has two main types of loops: **for loop** and **while loop**.

FOR LOOP

Used when you know how many times you want to repeat.

```
for i in range(1, 6):
    print("Count:", i)
```

Output: 1 2 3 4 5

Example: Print numbers from 1 to 5

1 2 3 4 5

WHILE LOOP

Used when you don't know how many times in advance. It runs while a condition is True.

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

Output: 1 2 3 4 5

Example: Print numbers from 1 to 5

1 2 3 4 5

Both achieve the same result! 😊

Use **for** when:

- ✓ You have a sequence (numbers, list, string, etc.)
- ✓ You know the number of repetitions

Use **while** when:

- ✓ You don't know how many times
- ✓ You want to repeat until a condition is met!

QUICK RECAP



Loops repeat tasks



Loops save time



Loops reduce errors



Loops make code clean and efficient



Python makes repetition easy and powerful with **for loops**!

FOR LOOPS



Repeat. Iterate. Automate.

A for loop is used to iterate over a **sequence** (like a **list**, **string**, **tuple**, **range**, etc.) and execute a block of code for each item.

SYNTAX

for variable **in** sequence:

indented block

code to execute

HOW IT WORKS

- 1 The loop takes the first item from the sequence.
- 2 Assigns it to the loop variable.
- 3 Executes the indented block of code.
- 4 Repeats until all items are processed.

1 COUNTING NUMBERS

Use for loops with **range()** to repeat a block of code a specific number of times.

```
for i in range(1, 6):  
    print(i)
```

OUTPUT

1
2
3
4
5

Counts from 1 to 5!

2 PRINTING MESSAGES

Iterate over items and perform actions like printing messages.

```
messages = ["Hello", "Python", "Learner!"]  
for msg in messages:  
    print(msg)
```

OUTPUT

Hello
Python
Learner!

Great for personalized outputs!

3 LOOP VARIABLES

The loop variable holds the current item in each iteration.

```
fruits = ["Apple", "Banana", "Cherry"]  
for fruit in fruits:  
    print("I like", fruit)
```

OUTPUT

I like Apple
I like Banana
I like Cherry

The variable 'fruit' changes each time!

MORE EXAMPLES

ITERATE OVER A STRING

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

Output: P y t h o n

SUM OF NUMBERS

```
total = 0  
for n in range(1, 6):  
    total += n  
print("Sum:", total)
```

Output: Sum: 15

ITERATE OVER A LIST

```
numbers = [10, 20, 30]  
for num in numbers:  
    print(num * 2)
```

Output: 20 40 60

For loops help you write efficient, clean, and readable code. Practice more, iterate better!

www.codescompiler.com



Keep going!

While Loops



Repeat actions as long as a condition is **True**.

What is a While Loop?

A **while** loop in Python repeatedly executes a block of code as long as the specified condition is **True**.

When the condition becomes **False**, the loop stops.

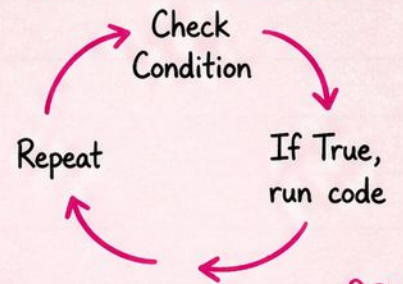


Syntax

```
while condition:  
    # block of code
```

Indentation is important in Python!

Think of it like...



KEY CONCEPTS

1 Repeating Until

The loop continues until the condition becomes **False**.

```
count = 1  
while count <= 5:  
    print("Count:", count)  
    count += 1  
print("Done!")
```

Output:

```
Count: 1  
Count: 2  
Count: 3  
Count: 4  
Count: 5  
Done!
```



Stops when count becomes 6 (condition is **False**)

2 Conditions in Loops

Any valid condition can control a **while** loop.

Examples:

Using comparison:

```
x = 10  
while x > 0:  
    print(x)  
    x -= 1
```

Using user input:

```
user = ""  
while user != "quit":  
    user = input("Enter 'quit' to stop: ")  
    print("You typed:", user)
```

Using conditions with and, or, not:

```
i = 0  
while i < 10 and i != 5:  
    print(i)  
    i += 1
```



The condition is checked **before** every iteration.

3 Infinite Loops

If the condition never becomes **False**, the loop runs **forever**!

```
while True:  
    print("This will never stop!")  
    # Be careful!
```



Infinite loops can freeze your program. Use with caution!

Common Ways to Stop

- Press **Ctrl + C** in the terminal
- Close the program
- Add a **break** condition



Help! I can't stop!

Best Practices

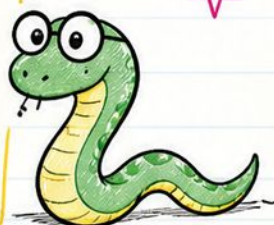
- ✓ Make sure the condition will eventually become **False**.
- ✓ Update your variables inside the loop.
- ✓ Avoid infinite loops unless absolutely necessary.
- ✓ Use **break** to exit the loop early when needed.



Did you know?



while True is often used in menus, game loops, and server programs where the program needs to keep running and wait for user action.



Keep Practicing. Keep Coding. Keep Growing!



➡ www.codescompiler.com ⬅



LOOP CONTROL

Control the flow inside loops to make your code smarter!

Loops help us repeat tasks. Loop control lets us decide what happens next!

What is Loop Control?

Loop control statements change the normal flow of a loop. They help us skip iterations, exit loops early, or control complex looping situations.

Common Loop Control Statements

break

Exits the loop completely.



continue

Skips the rest of the current iteration and moves to the next iteration.



Nested Loops

A loop inside another loop. Used to work with multi-dimensional data like matrices, patterns, etc.



1. break Statement

Terminates the loop as soon as it is encountered.

```
for i in range(1, 6):  
    if i == 3:  
        break  
    print(i)
```

Output:

1
2

When i becomes 3, break is triggered and the loop stops.

2. continue Statement

Skips the rest of the current iteration and continues with the next iteration.

```
for i in range(1, 6):  
    if i == 3:  
        continue  
    print(i)
```

Output:

1
2
4
5

When i is 3, continue skips `print(i)` and moves to the next value.

3. Nested Loops

A loop inside another loop. The inner loop completes all its iterations for each iteration of the outer loop.

```
for i in range(1, 4):  
    for j in range(1, 4):  
        print(f"{i} {j}", end=" ")  
    print()
```

Output:

1 1 1 2 1 3
2 1 2 2 2 3
3 1 3 2 3 3

Use Cases

- ✓ Printing patterns (Matrix, Pyramid)
- ✓ Working with tables
- ✓ Game boards
- ✓ Graph traversals



Quick Tips

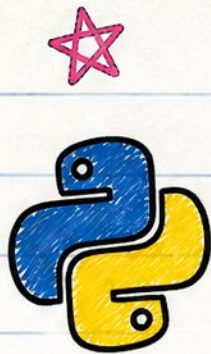
- Use break when you found what you need.
- Use continue to skip unwanted values.
- Keep nested loops clean and readable.
- Too many nested loops? Consider refactoring!

Remember

break and continue only affect the innermost loop in nested loops.

Visual Analogy

break = Exit the building
continue = Skip this room and go to the next one
Nested Loops = Rooms inside a building



PYTHON PROGRAMMING

MODULE 5

MINI PROJECT



In this mini project, we will build three simple Python programs. These projects will help you practice loops, conditionals, and user input.



1

NUMBER COUNTER



This program counts numbers from a starting number to an ending number with a step.

Example:

Start: 1, End: 10, Step: 1

Output:

1 2 3 4 5 6 7 8 9 10

Great for understanding loops!



2

COUNTDOWN TIMER

This program takes a number of seconds from the user and counts down to 0.



Example:

Enter seconds: 5

Output:

5
4
3
2
1
0

Time's up! 🕒

Useful for timers, games, and reminders!

3

MULTIPLICATION TABLE GENERATOR

This program generates the multiplication table of a number entered by the user.

Example:

Enter number: 5

Output:

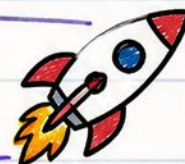
5 x 1 = 5 5 x 2 = 10 5 x 3 = 15 5 x 4 = 20 5 x 5 = 25
5 x 6 = 30 5 x 7 = 35 5 x 8 = 40 5 x 9 = 45 5 x 10 = 50

What you'll practice:

- ☒ Loops (for)
- ☒ User Input
- ☒ Conditionals (if)
- ☒ Printing Output
- ☒ Problem Solving

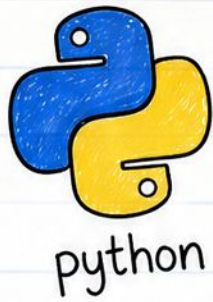


These mini projects are perfect for building your Python skills. Try modifying them and add your own creative ideas!



www.codescompiler.com





Introduction to LISTS

A list is a collection of items in a particular **order**.



Lists are one of the most **versatile** data structures in Python. They can store items of any data type and are **mutable** (changeable).

1 CREATING LISTS

You can create a list by placing items inside square brackets `[]`, separated by commas.

Examples:

```
>>> fruits = ["apple", "banana", "cherry"]
>>> numbers = [1, 2, 3, 4, 5]
>>> mixed = ["hello", 10, 3.14, True]
>>> empty_list = []
```

Notes

- ✓ Items in a list can be of **any type**.
- ✓ Lists are **ordered** (items have a position).
- ✓ Lists are **changeable** (you can add, remove, or modify items).



2 ACCESSING ITEMS

List items are indexed, starting from **0**.

```
fruits = ["apple", "banana", "cherry", "date"]
```

Index: 0 1 2 3

Examples:

```
>>> fruits = ["apple", "banana", "cherry", "date"]
>>> fruits[0]    # 'apple' (first item)
>>> fruits[1]    # 'banana' (second item)
>>> fruits[3]    # 'date' (fourth item)
>>> fruits[-1]   # 'date' (last item)
>>> fruits[-2]   # 'cherry' (second last item)
```

Use **negative indexes** to access items from the end of the list. -1 refers to the last item, -2 to the second last, and so on.



3 LIST LENGTH

You can find out how many items are in a list using the **len()** function.

Examples:

```
>>> fruits = ["apple", "banana", "cherry"]
>>> len(fruits)
3
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> len(numbers)
6
>>> empty_list = []
>>> len(empty_list)
0
```

Key Takeaways

- ★ Lists store multiple items in order.
- ★ Use indexes to access items.
- ★ Use **len()** to find the number of items.
- ★ Lists are mutable – you can change them anytime!

Practice makes perfect!





MODIFYING LISTS



Lists are ordered, changeable, and allow duplicates.

Lists in Python are mutable, which means we can change them after they are created.

Example List

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

INDEX REMINDER

Indexing starts at 0.

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

0 1 2

1

ADDING ITEMS

Add one or more items to a list.

- A** `append()` - adds an item at the end.



```
fruits = ["apple", "banana"]
fruits.append("cherry")
# Output: ["apple", "banana", "cherry"]
```

- B** `insert()` - adds an item at a specific position.



```
fruits = ["apple", "cherry"]
fruits.insert(1, "banana")
# Output: ["apple", "banana", "cherry"]
```

- C** `extend()` - adds multiple items from another list.



```
fruits = ["apple"]
fruits.extend(["banana", "cherry"])
# Output: ["apple", "banana", "cherry"]
```

TIP

Use `append()` for single items, `extend()` to add multiple items from another list.

2

REMOVING ITEMS

Remove items from a list.



- A** `remove()` - removes the first occurrence of a value.

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
# Output: ["apple", "cherry"]
```

- B** `pop()` - removes and returns an item at a given index.

```
fruits = ["apple", "banana", "cherry"]
item = fruits.pop(1)
# item: 'banana'
# Output: ["apple", "cherry"]
```

- C** `del` - deletes an item at a given index.

```
fruits = ["apple", "banana", "cherry"]
del fruits[0]
# Output: ["banana", "cherry"]
```

- D** `clear()` - removes all items from the list.

```
fruits = ["apple", "banana"]
fruits.clear()
# Output: []
```

Choose the right method based on what you need!

3

UPDATING ITEMS

Change the value of existing items.

- A** Update using index



```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"
# Output: ["apple", "blueberry", "cherry"]
```

- B** Update using negative index



```
fruits = ["apple", "banana", "cherry"]
fruits[-1] = "grape"
# Output: ["apple", "banana", "grape"]
```

- C** Update a range (slicing)

```
fruits = ["apple", "banana", "cherry", "date"]
fruits[1:3] = ["kiwi", "mango"]
# Output: ["apple", "kiwi", "mango", "date"]
```

QUICK NOTES

- ✓ Lists are mutable.
- ✓ Changes made to a list affect the original list.
- ✓ Always check index positions to avoid mistakes!

LET'S SEE IT ALL TOGETHER!



```
fruits = ["apple", "banana"]
fruits.append("cherry")
fruits.insert(1, "blueberry")
fruits[2] = "kiwi"
fruits.remove("banana")
fruits.pop()

# start
# ["apple", "banana", "cherry"]
# ["apple", "blueberry", "banana", "cherry"]
# ["apple", "blueberry", "kiwi", "banana", "cherry"]
# ["apple", "blueberry", "kiwi", "cherry"]
# removes 'cherry' -> ["apple", "blueberry", "kiwi"]
```



Lists store multiple items in Python. ★

LOOPING THROUGH LISTS

Use loops to work with each item one by one!



Loops make it easy to access, process and analyze every item in a list. ♥



1 DISPLAYING ITEMS

Use a loop to print each item in the list.



Great for showing all items clearly!

EXAMPLE

```
fruits = ["Apple", "Banana", "Mango", "Orange"]  
  
for fruit in fruits:  
    print(fruit)
```

OUTPUT

Apple
Banana
Mango
Orange



2 COUNTING ELEMENTS

Loop through the list and count how many items it has.

Useful when you don't want to use len() function!



EXAMPLE

```
numbers = [10, 20, 30, 40, 50]  
count = 0  
  
for num in numbers:  
    count = count + 1  
  
print("Total elements:", count)
```

OUTPUT

Total elements: 5

The loop visits each item once and increases the count.



3 SEARCHING LISTS

Use a loop to search for an item in the list.



Helps to check existence or find an item's position!

EXAMPLE

```
items = ["pen", "pencil", "book", "eraser"]  
target = "book"  
found = False  
  
for item in items:  
    if item == target:  
        found = True  
        break  
  
if found:  
    print("Item found in the list!")  
else:  
    print("Item not found.")
```

OUTPUT (when target = "book")

Item found in the list!

HOW IT WORKS

- ✓ The loop checks each item.
- ✓ If it matches the target, we set found = True.
- ✓ break stops the loop early (optional but efficient!).



QUICK TIPS



Use meaningful variable names for better code.



Use break to stop the loop early.



Use else with loops in searches (advanced use).



Practice with different lists to get better!

Practice makes perfect! ♥



USEFUL LIST METHODS

Lists are ordered, changeable collections. These methods make working with lists easy and powerful!

★ Python provides built-in methods that help us manipulate lists efficiently. Let's explore some of the most useful ones!

1 SORTING LISTS

Sort the elements of a list in ascending order (alphabetical or numerical).

- Using `sort()` - sorts the list in place (modifies the original list)

```
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort()
print(numbers)

# Output: [1, 1, 3, 4, 5, 9]
```

- Using `sorted()` - returns a new sorted list (original list remains same)

```
names = ["Charlie", "Alice", "Bob"]
sorted_names = sorted(names)
print(sorted_names)
print(names)

# Output: ['Alice', 'Bob', 'Charlie']
# Output: ['Charlie', 'Alice', 'Bob']
```

MORE OPTIONS

- `sort(reverse=True)`
→ Descending order
- `sorted(reverse=True)`
→ Returns new list in descending order
- `sort(key=str.lower)`
→ Case-insensitive sorting for strings

2 REVERSING LISTS

Reverse the order of elements in a list.

- Using `reverse()` - reverses the list in place (modifies the original list)

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
print(fruits)

# Output: ['cherry', 'banana', 'apple']
```

- Using `reversed()` - returns an iterator. Convert it to a list to see the result.

```
nums = [10, 20, 30, 40]
rev_nums = list(reversed(nums))
print(rev_nums)
print(nums)

# Output: [40, 30, 20, 10]
# Output: [10, 20, 30, 40]
```

Think of it as flipping the list!



3 JOINING LISTS

Combine two or more lists into a single list.

- Using `+` operator - creates a new list

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2
print(combined)

# Output: [1, 2, 3, 4, 5, 6]
```

- Using `extend()` - adds elements of one list to the end of another (modifies the original list)

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)

# Output: [1, 2, 3, 4, 5, 6]
```

Can join more than two lists too!

```
a = ["A", "B"]
b = ["C"]
c = ["D", "E"]
result = a + b + c
print(result)

# Output: ['A', 'B', 'C', 'D', 'E']
```

QUICK TIPS

- ✓ `sort()` and `reverse()` change the original list.
- ✓ `sorted()` and `reversed()` return new results.
- ✓ `extend()` is more efficient than `+` for large lists.
- ✓ Lists can store mixed data types.



MODULE 6

MINI PROJECT

Practice.
Build.
Learn.
Repeat!



Python Programming



In this mini project, you will build small, useful programs using **Python lists**. These projects will help you practice list operations in a real-life way!



1 FAVORITE MOVIES LIST



Create a program to store and manage your favorite movies.

- ✓ Add movies to the list
- ✓ View all movies
- ✓ Remove a movie
- ✓ Search for a movie
- ✓ Count total movies

Example

```
Movies = ["Inception", "Interstellar",  
          "The Dark Knight"]
```

- Add: "The Matrix"
- Remove: "Interstellar"
- Search: "Inception" → Found!
- Total Movies: 3



2 CLASSROOM ATTENDANCE LIST



Create a program to manage daily classroom attendance.

- ✓ Add student names to the list
- ✓ Mark present / absent
- ✓ View present students
- ✓ View absent students
- ✓ Total students and attendance count

Example

```
Students = ["Aarav", "Diya", "Kabir", "Meera"]  
Present = ["Aarav", "Meera"]
```

- Mark "Diya" as Present
- Present Students: Aarav, Diya, Meera
- Absent Students: Kabir
- Total Students: 4



3 SHOPPING LIST MANAGER



Create a program to manage your shopping list.

- ✓ Add items to the list
- ✓ View all items
- ✓ Remove items
- ✓ Mark items as purchased
- ✓ Show pending items

Example

```
ShoppingList = ["Milk", "Bread", "Eggs"]  
Purchased = ["Milk"]
```

- Add: "Butter"
- Remove: "Bread"
- Purchased: Milk
- Pending: Eggs, Butter



KEY CONCEPTS YOU WILL USE

- ★ Lists
- ★ Indexing & Slicing
- ★ Add / Remove Items
- ★ Search in List
- ★ Count & Length
- ★ Loops & Conditionals



TIP

Break your project into small steps. Test each feature. Have fun and make your project better every day!



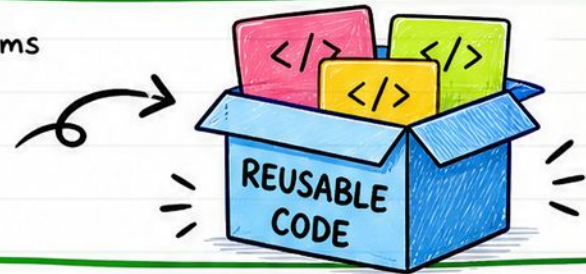
⇒ www.codescompiler.com ⇐



WHAT ARE FUNCTIONS?



A **function** is a block of code that performs a specific task. You can call (use) it whenever you need, without writing the same code again and again.



SYNTAX (IN PYTHON)

```
def function_name(parameters):  
    # Code to be executed  
    return value # optional
```

- **def** → keyword to define a function
- **function_name** → any valid name
- **parameters** → optional input(s)
- **return** → sends a result back (optional)

1 REUSABLE CODE

Write once, use many times!
Functions help you avoid repeating the same code.

✗ Without Function

```
print("Hello!")  
print("Hello!")  
print("Hello!")  
print("Hello!")
```

✓ With Function

```
def greet():  
    print("Hello!")  
  
greet()  
greet()  
greet()  
greet()
```

Cleaner, shorter, better!

2 FUNCTION BENEFITS



Reusability

Use the same code multiple times without rewriting.



Modularity

Break big programs into small, manageable parts.



Readability

Makes code easier to read and understand.



Easy Maintenance

Update the code in one place, and it works everywhere.



Scalability

Helps build and manage large programs efficiently.

3 REAL-LIFE EXAMPLES



Making Coffee

A recipe (function) you can use every time to make the same great coffee!



Calculator Operations

Add, subtract, multiply, divide - same operations used again and again.



ATM Transactions

Functions are used to check balance, deposit money, withdraw cash, etc.



GPS Navigation

Functions help calculate routes, distance, and time repeatedly.



Sending Emails

The same process (function) is used to send different emails.

QUICK EXAMPLE



Let's see a simple function!

```
def add(a, b):  
    """Returns the sum of a and b"""  
    return a + b  
  
result = add(5, 7)  
print("Sum is:", result) # Output: Sum is: 12
```

What's happening?

- ✓ We created a function 'add'.
- ✓ It takes two numbers.
- ✓ It returns their sum.
- ✓ We called it and got the result!

Keep Practicing. Keep Coding. Keep Growing!

www.codescompiler.com



★ CREATING FUNCTIONS

Write once, use many times!

Functions help us break down our code into small, reusable blocks.



A function is a block of code that performs a **specific task**. It helps make our programs **organized**, **reusable** and **easy to understand**.



1 Using def

We use the keyword **def** to create a function.

```
def greet(name):  
    print(f"Hello, {name}!")  
    print("Welcome to Python!")
```

keyword to define a function

function name (we choose this)

parameter (input)

★ Key Points

- **def** tells Python we are creating a function.
- **greet** is the name of the function.
- **name** is a **parameter** (it accepts input).
- Indented code is the **body** of the function.

2 Calling Functions

We **call** (or invoke) a function using its name followed by parentheses **()**.

```
# function definition  
def greet(name):  
    print(f"Hello, {name}!")  
    print("Welcome to Python!")  
  
# calling the function  
greet("Alice")  
greet("Bob")
```

These calls will run the function.

Output

```
Hello, Alice!  
Welcome to Python!  
Hello, Bob!  
Welcome to Python!
```

3 Function Names

Good function names make code easy to read and understand.

Rules for Function Names

- ✓ Must start with a letter or underscore (_).
- ✓ Can contain letters, numbers and underscores.
- ✓ Case-sensitive (greet and Greet are different).
- ✓ Should be descriptive and meaningful.
- ✓ Cannot use Python keywords.

Good Examples ✓

```
calculate_area()  
get_user_input()  
print_summary()  
is_palindrome()
```



Bad Examples ✗

```
1stFunction()  
print data()  
def()  
total-sum()
```



TIP

Use clear names that describe what your function does. It helps you and others understand your code better!

More Example

```
# Function to add two numbers  
def add(a, b):  
    result = a + b  
    return result
```

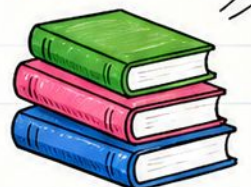
```
# calling the function  
sum1 = add(5, 7)  
sum2 = add(10, 20)  
print("Sum 1:", sum1)  
print("Sum 2:", sum2)
```

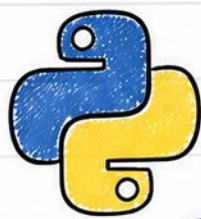
Remember:

Define once,
Use anytime!

Practice more. Build more. Become a better Pythonista!

www.codescompiler.com





FUNCTION PARAMETERS

Parameters make functions **flexible** and **reusable**!



Parameters are variables listed inside the parentheses in a function definition. They allow the function to work with different inputs.

SYNTAX

```
def function_name(param1, param2, ...):  
    # function body  
    return value
```

WHY USE PARAMETERS?

- ✓ Make functions dynamic
- ✓ Avoid repeating code
- ✓ Work with different data
- ✓ Improve code reusability



1 PASSING VALUES

We pass values (**arguments**) to a function when we call it.

```
def greet(name):  
    print("Hello, " + name + "!")  
  
greet("Alice") # Alice is passed  
               as an argument
```

OUTPUT

Hello, Alice!



The value 'Alice' is passed to the parameter 'name'.

2 MULTIPLE PARAMETERS

A function can have more than one parameter.

```
def add(a, b):  
    sum = a + b  
    return sum  
  
result = add(5, 7)  
print("Sum is:", result)
```

OUTPUT

Sum is: 12

Values 5 and 7 are passed to parameters **a** and **b** in the same order.

3 USER INPUT IN FUNCTIONS

We can take user input and pass it as arguments to a function.

```
def square(num):  
    return num * num  
  
n = int(input("Enter a number: "))  
result = square(n)  
print("Square is:", result)
```

EXAMPLE RUN

Enter a number: 6
Square is: 36



User enters 6, which is passed to 'num' parameter.

QUICK NOTES

- ★ Parameters are placeholders in the function definition.
- ★ Arguments are the actual values passed during the function call.
- ★ The number of arguments must match the number of parameters.
- ★ Parameters make our functions powerful and flexible!





RETURNING VALUES

In Python, functions can send data back using the **return** statement.



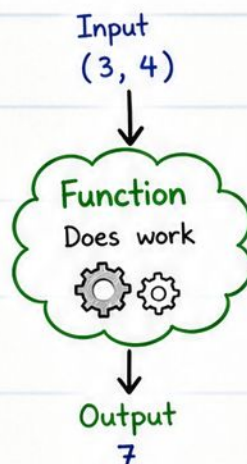
A function doesn't just do work — it can give back a result. That result can be stored, used, and reused anywhere in your program!

1 USING return

The **return** statement sends a value back from a function to the caller.

```
def add(a, b):  
    result = a + b    # calculate sum  
    return result     # send result back
```

```
>>> add(3, 4)  
7
```



What is return?

- **return** sends a value out of the function.
- The function stops running at **return**.



2 STORING RESULTS

You can store the returned value in a variable for later use.

```
def multiply(x, y):  
    return x * y
```

```
product = multiply(5, 6) # store the result  
print(product)          # 30
```

multiply(5, 6)



Stored in variable "product"

Think of it like:

The function gives you a gift (return value), and you put it in a box (variable) to use whenever you need it!



3 REUSING OUTPUTS

Returned values can be reused in calculations, passed to other functions, or displayed.

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

```
a = square(4)    # 16  
b = square(a)    # square of 16 -> 256  
print(b)         # 256
```

Ways to reuse:



In calculations
`result = square(3) + 10`



As arguments to other functions
`print(square(2))`



In output / display
`print(square(5))`

Why return?

- ✓ Makes functions more flexible and powerful.
- ✓ Keeps your code clean and organized.
- ✓ Allows functions to work together.



Example in Real Life

A chef (function) cooks a dish and hands it to you (return value). You can eat it now, save it for later, or use it in another recipe!



PRO TIP

If a function doesn't have a return statement, it returns **None** by default.

```
def hello():  
    print("Hi!")  
  
>>> hello()  
Hi!  
>>> print(hello())  
None
```



no return
=> None

Keep returning, keep learning, keep building! 🚀





MINI PROJECT

Build small.
Learn Big.
Code Smart.



In this mini project, we will create 3 useful Python functions and use them in a simple program.



1

GREETING FUNCTION

This function takes a user's name and returns a friendly greeting.

Hello!



```
def greet(name):
    return f"Hello, {name}!"
```

Example

```
msg = greet("Alice")
print(msg) # Hello, Alice!
```

2

CALCULATOR FUNCTION

This function performs basic arithmetic operations.
(add, subtract, multiply, divide)



```
def calculate(a, b, op):
    if op == "+":
        return a + b
    elif op == "-":
        return a - b
    elif op == "*":
        return a * b
    elif op == "/":
        return a / b
    else:
        return "Invalid operation"
```

3 + 5 = 8

10 - 4 = 6

3 x 7 = 21

8 / 2 = 4

3

NAME FORMATTER

This function takes a first and last name and returns a properly formatted name.

Hello!
My name is

```
def format_name(first, last):
    first = first.strip().title()
    last = last.strip().title()
    return f"{first} {last}"
```

Example

```
full_name = format_name("john", "doe")
print(full_name) # John Doe
```

"john", "doe"

John Doe

PUTTING IT TOGETHER

Use these functions in a simple menu-driven program where the user can:

☒ Get a greeting ☒ Do calculations ☒ Format names

This helps practice functions and user input!

Small projects today,
Big skills tomorrow!





WHAT ARE DICTIONARIES?



A dictionary in Python is an unordered collection of data in the form of **key-value pairs**. It is written using curly braces `{ }` with each key and value separated by a colon `:`.

Mutable
Unordered
Indexed by
keys

1. KEY-VALUE PAIRS

A dictionary stores data as **key-value pairs**.

Key → : → Value

Example:

```
student = { 'name': 'Alice',  
            'age': 20,  
            'course': 'Python' }
```

Key	Value
'name'	'Alice'
'age'	20
'course'	'Python'

★ Keys must be **unique** and **immutable** (like strings, numbers, or tuples). Values can be of any data type.

3. COMMON OPERATIONS

★ Access value	<code>person['name']</code>	# 'John'
★ Add/Update	<code>person['age'] = 26</code>	# Update age
★ Add new pair	<code>person['course'] = 'Python'</code>	# Add new
★ Remove a pair	<code>del person['city']</code>	# Removes 'city'
★ Check key	<code>'name' in person</code>	# True
★ Get value	<code>person.get('name')</code>	# 'John'
★ Keys / Values	<code>person.keys(), person.values()</code>	



Dictionaries are super useful for storing and organizing related data! ❤️

2. CREATING DICTIONARIES

You can create dictionaries in Python in several ways.

a) Using curly braces `{ }`

```
person = { 'name': 'John', 'age': 25,  
          'city': 'New York' }
```

b) Using `dict()` constructor

```
person = dict(name='John', age=25, city='New York')
```

c) From sequence of key-value pairs

```
pairs = [('name', 'John'), ('age', 25), ('city', 'New York')]  
person = dict(pairs)
```



4. REAL-LIFE EXAMPLES

Dictionaries help us model real-world data in a meaningful way.



1. Student Record

```
{ 'id': 101, 'name': 'Neha',  
  'grade': 'A', 'age': 21 }
```



2. Product Information

```
{ 'id': 'P100', 'name': 'Laptop',  
  'price': 55000, 'brand': 'Dell' }
```



3. Weather Data

```
{ 'city': 'Delhi', 'temp': 32,  
  'humidity': 60, 'condition': 'Sunny' }
```



From apps and websites to data analysis, dictionaries are used everywhere! ✓✍️



Practice more. Code more. Get better!





ACCESSING DATA



Working with Dictionaries in Python

Dictionaries store data in **key-value pairs**.

Here's how we access and work with that data!

Dictionary
= {key: value}

Unique keys
Access fast!

1 READING VALUES

Access the value of a specific key using **square brackets []**.

Example:

```
student = {  
    "name": "Alice",  
    "age": 20,  
    "course": "Python"  
}  
  
name = student["name"]  
age = student["age"]  
  
# name → "Alice"  
# age → 20
```

Output
Alice
20

Other useful methods:

- dict.get(key) → returns value or **None** if key doesn't exist
- dict.get(key, default) → returns **default** if key not found

```
student.get("course")      # 'Python'  
student.get("grade")      # None  
student.get("grade", "N/A") # 'N/A'
```

2 FINDING KEYS

Check if a **key exists** in the dictionary.

```
student = {"name": "Alice", "age": 20}  
  
"name" in student  # True  
"grade" in student # False
```

Useful Methods:

- dict.keys() → all keys
- dict.values() → all values
- dict.items() → all key-value pairs

```
student.keys()      # dict_keys(['name', 'age'])  
student.values()    # dict_values(['Alice', 20])  
student.items()     # dict_items([('name', 'Alice'),  
                                ('age', 20)])
```

3 UPDATING INFORMATION

Change the value of an **existing key** or **add a new key-value pair**.

```
student = {"name": "Alice", "age": 20}  
student["age"] = 21          # Update  
student["course"] = "Python" # Add new  
del student["name"]          # Remove  
  
# Updated dictionary:  
# {'age': 21, 'course': 'Python'}
```

You can:

Update

Change an existing value



Add

Insert a new key-value pair



Remove

Delete a key-value pair



Dictionaries are powerful, flexible and **SUPER** useful!

PRO
TIP



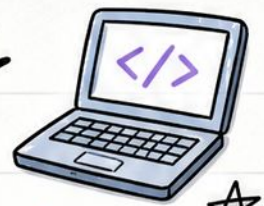
Always use `.get()` when the key might not exist to avoid **KeyError**
`value = student.get("grade", "N/A")` # Safe and clean!

PRACTICE
MAKES
PERFECT!





ADDING AND REMOVING DATA



Work with data dynamically using Python Dictionaries!

Dictionaries in Python are mutable, which means we can **add** new key-value pairs or **remove** existing ones anytime.

```
student = {
    'name': 'Alice',
    'age': 20,
    'grade': 'A'
}
```

← Our starting dictionary

1

NEW ENTRIES

Add new key-value pairs to a dictionary using assignment.

```
>>> student = {'name': 'Alice', 'age': 20}
>>> student['grade'] = 'A'      # Add new entry
>>> student['city'] = 'New York' # Add another
>>> student
{'name': 'Alice', 'age': 20, 'grade': 'A', 'city': 'New York'}
```

★ If the key already exists, its value will be **updated**.

```
>>> student['age'] = 21 # Update existing key
>>> student['age']
21
```

3

DICTIONARY METHODS

Useful methods related to adding and removing data.

Method	Description	Example
<code>update()</code>	Adds key-value pairs from another dictionary.	<code>student.update({'age': 22, 'course': 'Python'})</code>
<code>setdefault(key, [default])</code>	Returns the value of key. If key doesn't exist, inserts it with default.	<code>student.setdefault('age', 18)</code>
<code>keys()</code>	Returns all keys.	<code>student.keys()</code>
<code>values()</code>	Returns all values.	<code>student.values()</code>
<code>items()</code>	Returns all key-value pairs.	<code>student.items()</code>

2

DELETING ENTRIES

Remove key-value pairs from a dictionary using different methods.

★ **del statement** – removes the key and its value.

```
>>> del student['grade']
>>> student
{'name': 'Alice', 'age': 21, 'city': 'New York'}
```



★ **pop(key)** – removes the key and returns its value.

```
>>> age = student.pop('age')
>>> age
21
>>> student
{'name': 'Alice', 'city': 'New York'}
```

★ **popitem()** – removes and returns the last inserted key-value pair (Python 3.7+).

```
>>> item = student.popitem()
>>> item
('city', 'New York')
>>> student
{'name': 'Alice'}
```

★ **clear()** – removes all items from the dictionary.

```
>>> student.clear()
>>> student
{}
```



QUICK TIP

Always check if a key exists before deleting it to avoid errors.

```
if 'age' in student:
    del student['age']
```



KEY TAKEAWAY

Adding and removing data makes dictionaries powerful and flexible for real-world applications like user data, configurations, and more!



In Python,
a dictionary is
an unordered
collection of
key-value
pairs.

LOOPING THROUGH DICTIONARIES



Why loop?

Looping helps us
access and process
each key, each value,
or each key-value
pair in a dictionary.

KEYS • VALUES • KEY-VALUE PAIRS



KEYS

Unique identifiers
used to access
values.



VALUES

The data
associated with
each key.

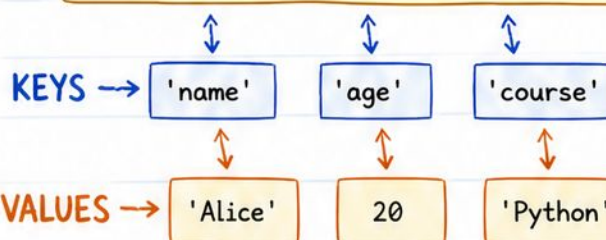


KEY-VALUE PAIRS

Each item in a
dictionary is a pair
of key and value.

EXAMPLE DICTIONARY

```
student = { 'name': 'Alice', 'age': 20, 'course': 'Python' }
```



KEY-VALUE PAIRS

```
'name' : 'Alice'  
'age' : 20  
'course' : 'Python'
```

WAYS TO LOOP THROUGH A DICTIONARY

1 Loop through **KEYS**

Use the dictionary itself in
the loop.

```
student = { 'name': 'Alice',  
            'age': 20,  
            'course': 'Python' }
```

```
for key in student:  
    print(key)
```

Output:

```
name  
age  
course
```



2 Loop through **VALUES**

Use the `.values()` method.

```
student = { 'name': 'Alice',  
            'age': 20,  
            'course': 'Python' }
```

```
for value in student.values():  
    print(value)
```

Output:

```
Alice  
20  
Python
```



3 Loop through **KEY-VALUE PAIRS**

Use the `.items()` method.

```
student = { 'name': 'Alice',  
            'age': 20,  
            'course': 'Python' }
```

```
for key, value in student.items():  
    print(key, '→', value)
```

Output:

```
name → Alice  
age → 20  
course → Python
```



QUICK TIPS

- ✓ Dictionaries are unordered (Python 3.7+ maintains insertion order, but don't rely on it).
- ✓ Keys must be unique and immutable (like strings, numbers, tuples).
- ✓ Use `.keys()`, `.values()`, or `.items()` for explicit access.

REMEMBER

Dictionary = { **Key** : **Value** }

Loop = Visit each item one by one

Process = Do something useful!



REAL-WORLD EXAMPLE

Looping through dictionaries is
useful when working with data
like user info, settings, JSON
responses, configuration files,
and more!





MINI PROJECT



Mini Projects help you apply Python concepts to build real-world, useful applications.

LEARN
CODE
BUILD
GROW

PROJECT IDEAS

1 STUDENT RECORD SYSTEM

A system to store and manage student details like name, roll number, marks, and grade.

FEATURES:

- Add new student
- View all students
- Search by roll number
- Update student details
- Delete student record
- Calculate grade

Roll No.	Name	Marks	Grade
101	Anaya	89	A
102	Rohan	76	B
103	Isha	93	A+



Concepts Used:

Lists, Dictionaries, Functions, Input/Output, Loops

2 PHONE BOOK

A simple contact management system to store and search phone contacts.

FEATURES:

- Add contact
- View all contacts
- Search contact
- Update contact
- Delete contact
- Sort contacts

CONTACTS	
	Aarav 9876543210
	Meera 9123456780
	Kabir 9988776655

Concepts Used:

Dictionaries, Functions, File Handling (Optional), Searching

3 ANIMAL INFORMATION DATABASE

Store and manage information about animals in a database.

FEATURES:

- Add animal record
- View all animals
- Search by name/type
- Update information
- Delete record
- Filter by habitat/type



Concepts Used:

Lists, Dictionaries, Functions, Loops, Conditional Statements, File Handling

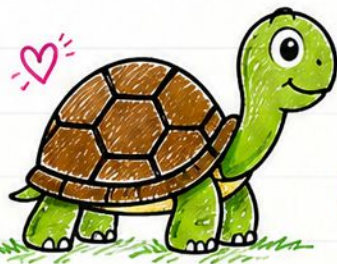


Name	Type	Habitat	Diet
Lion	Mammal	Savannah	Carnivore
Elephant	Mammal	Forest	Herbivore
Penguin	Bird	Antarctica	Carnivore

PRACTICE
BUILD
IMPROVE

★ Think | Code | Debug | Improve
That's the Python Way! 😊

SMALL PROJECTS,
BIG LEARNING!



INTRODUCTION TO TURTLE GRAPHICS

Turtle Graphics is a built-in Python module that lets us draw shapes and patterns using a small turtle (🐢) on the screen.

We give commands to the turtle and it **moves** around the screen, leaving a **trace** behind!

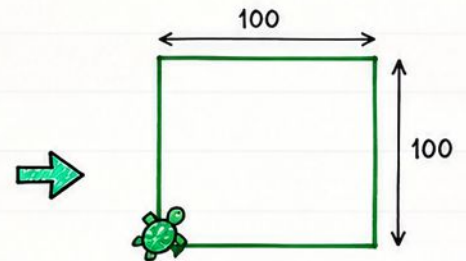
1 MOVING THE TURTLE

We control the turtle's movement using simple commands.

forward(d) → Move forward by d units
backward(d) → Move backward by d units
left(a) → Turn left by a degrees
right(a) → Turn right by a degrees
goto(x, y) → Move to position (x, y)
setx(x) → Set turtle's x position
sety(y) → Set turtle's y position
home() → Go to home position (0, 0)
circle(r) → Draw a circle with radius r

Example

```
import turtle
t = turtle.Turtle()
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
```



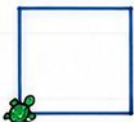
Angles are in **degrees**.
90° makes a perfect turn!



2 DRAWING SHAPES

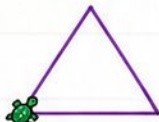
By combining movement and turns, we can draw many shapes and patterns.

Square



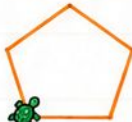
```
for i in range(4):
    t.forward(100)
    t.right(90)
```

Triangle



```
for i in range(3):
    t.forward(100)
    t.right(120)
```

Pentagon



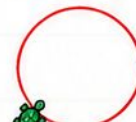
```
for i in range(5):
    t.forward(100)
    t.right(72)
```

Hexagon



```
for i in range(6):
    t.forward(100)
    t.right(60)
```

Circle



```
t.circle(100)
```

Tip! ★

For polygons with n sides, use:
 $\text{angle} = 360 / n$

3 COLORS AND PENS

We can make our drawings colorful and creative using pen and **color** commands.

color(c) → Set both pen color and fill color
pencolor(c) → Set pen (outline) color
fillcolor(c) → Set fill color
pensize(w) → Set pen thickness
penup() → Lift the pen (no drawing)
pendown() → Put the pen down (start drawing)
begin_fill() → Start filling a shape
end_fill() → End filling the shape

Example: Colorful Star ★

```
import turtle
t = turtle.Turtle()
t.pencolor('violet')
t.pensize(3)
t.fillcolor('yellow')
t.begin_fill()
for i in range(5):
    t.forward(200)
    t.right(144)
t.end_fill()
```



Use **penup()** to move without drawing!

Fun Fact

Turtle was created in the 1960s to teach programming in a fun and visual way! 😊

MORE USEFUL COMMANDS

clear()

Clears the screen



reset()

Resets the turtle and clears screen



speed(n)

Control turtle speed (1=slow, 10=fastest)



hideturtle()

Make the turtle invisible



showturtle()

Make the turtle visible



Practice
Makes
Perfect!



Fun Drawings

with Python

What is this?

Python isn't just about numbers and text—we can create fun patterns and shapes using simple code! Let's draw some awesome things.



We'll use printed characters like `*` or `#` in nested loops (the **outer loop** for rows, the **inner loop** for columns).



1 SQUARES

Let's draw a solid square using `*`

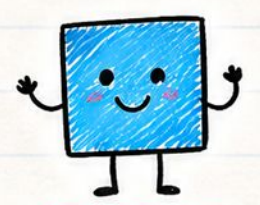
Logic:
Print `*` n times in each of n rows.

CODE (Python)

```
n = 5 # size of the square
for i in range(n):
    for j in range(n):
        print('*', end='')
    print()
```

OUTPUT

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```



Change n to make bigger squares!

2 STARS

Let's draw a right-angled triangle star pattern.

Logic:
Print `*` from 1 to n increasing each row.

CODE (Python)

```
n = 5
for i in range(1, n+1):
    for j in range(i):
        print('*', end='')
    print()
```

OUTPUT

```
*
* *
* * *
* * * *
* * * * *
```

Try reversing the loops to make other patterns!

3 HOUSES

Let's draw a simple house using `*`

Logic:
1. Print the roof (triangle)
2. Print the body (rectangle)
3. Add a door

CODE (Python)

```
n = 5 # base width of house
# Roof
for i in range(1, n+1):
    print(' ' * (n - i) + '*' * (2*i - 1))
# Body
for i in range(n):
    print('*' * n)
# Door
print(' * ')
print(' * ')

```

OUTPUT

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * *
* * * * *
 * * * * *
  * * * * *
   * * * * *
    * * * * *
     * * * * *
      * * * * *
```



You can get creative and draw almost anything!

TIPS

• Use `end=""` to print on the same line.

• Use `print()` to move to the next line.

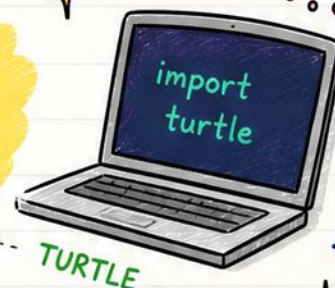
• Experiment with loops, spaces and different characters!





Bringing static drawings to life!

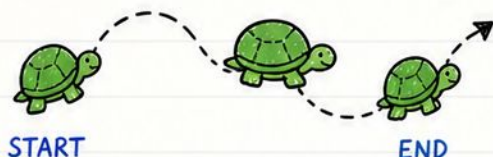
ANIMATION BASICS



In Python, we can create animations by updating drawings over time.

1 MOVING OBJECTS

Change the position of an object little by little to create motion.



EXAMPLE (TURTLE)

```
import turtle
import time

t = turtle.Turtle()
t.shape("turtle")

# Move forward in steps
for i in range(100):
    t.forward(5)
    time.sleep(0.05) # small delay
```

create turtle object

repeat 100 times

move a little

wait for smooth motion

KEY IDEA

Small changes
+ short delays
= smooth animation!

2 TIMING CONCEPTS

Timing controls how fast the animation plays and how smooth it looks.



DELAY

Use `time.sleep(seconds)` to pause between frames.



FRAME RATE

More frames per second (smaller delay) = smoother animation.



CONSISTENCY

Keep movement + delay consistent for natural looking motion.

DELAY COMPARISON

`time.sleep(0.5)`
SLOW



`time.sleep(0.1)`
MEDIUM



`time.sleep(0.01)`
FAST



TIP: Adjust the delay to get the perfect speed!

3 INTERACTIVE DRAWING

Let the user control the for fun and creativity!

CONTROLS EXAMPLE

↑ : Move Up
↓ : Move Down
← : Move Left
→ : Move Right
q : Quit

EXAMPLE: CONTROL A TURTLE

```
import turtle
screen = turtle.Screen()
t = turtle.Turtle()
t.shape("circle")

def go_up(): t.sety(t.ycor() + 20)
def go_down(): t.sety(t.ycor() - 20)
def go_left(): t.setx(t.xcor() - 20)
def go_right(): t.setx(t.xcor() + 20)

screen.listen()
screen.onkey(go_up, "Up")
screen.onkey(go_down, "Down")
screen.onkey(go_left, "Left")
screen.onkey(go_right, "Right")
screen.onkey(screen.bye, "q")

screen.mainloop()
```

You control the action!

Make games, stories, and simulations!

Endless creative possibilities!

QUICK RECAP

- Move objects in small steps
- Use timing (delay) for smooth motion
- Add interaction to make it exciting!



Keep practicing and bring your ideas to life!





Simple GAME LOGIC

Think
Plan
Code
Play!

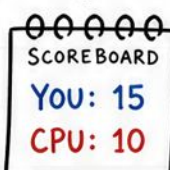
Game logic is the brain behind any game.
It decides what happens, keeps track of progress,
and tells us who wins!



KEY BUILDING BLOCKS

1 SCORES

Scores track how well the player is doing.



- ★ Increase when the player wins a round.
- ★ Keep score between rounds.
- ★ Determine the final winner.

```
player_score = 0  
cpu_score = 0
```

```
player_score += 1 # Add point  
print("Player: "; player_score)
```

2 USER CHOICES

The game gets input from the player.



Rock
Paper
Scissors

- ★ Player makes a choice.
- ★ The game validates the input.
- ★ Used to decide the outcome.

```
choice = input("Rock, Paper  
or Scissors? ")  
choice = choice.lower()  
# Validate choice  
if choice not in ["rock", "paper",  
"scissors"]:  
    print("Invalid choice!")
```

3 WINNING CONDITIONS

Rules that decide who wins each round (and the game!).



- ★ Define the rules of the game.
- ★ Compare player vs opponent.
- ★ Decide win, lose or draw.

```
def check_winner(player, cpu):  
    if player == cpu:  
        return "Draw"  
    elif (player == "rock" and cpu == "scissors") or \\  
          (player == "paper" and cpu == "rock") or \\  
          (player == "scissors" and cpu == "paper"):  
        return "Player Wins"  
    else:  
        return "CPU Wins"
```

HOW IT ALL WORKS TOGETHER

1

PLAYER
CHOICE



Player picks
an option.

2

GAME
LOGIC



Game processes
the choice and
compares.

3

UPDATE
SCORES



Add points based
on the result.

4

CHECK
WIN CONDITION



Check if someone
has reached the
target score.

5

SHOW
RESULT

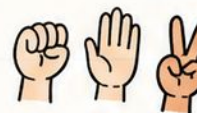


Announce result
and continue or
end the game!

EXAMPLE: ROCK, PAPER, SCISSORS

Rules:

- Rock beats Scissors
- Scissors beats Paper
- Paper beats Rock
- Same choice = Draw



VS

Sample Round:

You: Rock
CPU: Scissors
Result: You Win!

Score:
You: 3 CPU: 2



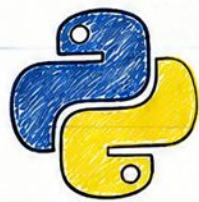
TIPS

- ✓ Keep your code modular (use functions).
- ✓ Test edge cases (invalid input, ties, etc.).
- ✓ Use clear variable names.
- ✓ Have fun and experiment!



Great games start with simple logic. Build it step by step and level up your skills!





PYTHON PROGRAMMING



MODULE 9



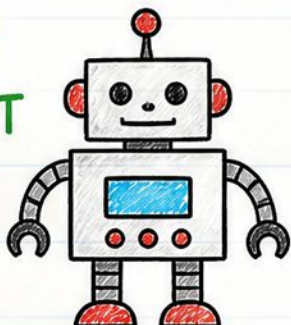
MINI PROJECT



Apply your Python skills by building these fun and creative mini projects!

WHAT YOU WILL BUILD:

1 DRAW A ROBOT



DESCRIPTION:

Use Python's turtle module to draw a cool robot with shapes, colors and creativity!



KEY CONCEPTS:

- ✓ Turtle Graphics
- ✓ Shapes & Lines
- ✓ Colors
- ✓ Position & Movement

2 COLORFUL PATTERN GENERATOR



DESCRIPTION:

Create amazing patterns using loops, angles and colors. Make your own art generator!



KEY CONCEPTS:

- ✓ Loops (for)
- ✓ Angles & Shapes
- ✓ Color Combinations
- ✓ Creativity & Logic

3 MINI GUESSING GAME



DESCRIPTION:

Build a number guessing game where the computer picks a number and you try to guess it!

KEY CONCEPTS:

- ✓ Random Module
- ✓ Input & Output
- ✓ Conditions (if/else)
- ✓ Loops (while)

Learn.
Code.
Create.
Repeat!



These mini projects will help you strengthen your Python concepts and build your confidence!



Keep Practicing. Keep Building. Keep Growing!



www.codescompiler.com



Plan Today,
Code Better
Tomorrow!

PROJECT PLANNING

Good planning = Successful Projects!



What is Project Planning?

Project planning is the first and most important step in building any Python program.

A well-planned project saves time, reduces errors and makes your code easy to understand and maintain.

1

UNDERSTANDING REQUIREMENTS

Understand what the program should do. Define the problem clearly before writing code.

KEY STEPS

- Read and analyze the requirements
- Identify inputs, outputs and constraints
- Break the problem into smaller parts
- List all features and goals
- Ask questions and clarify doubts

Example

Build a program to manage a student's marks.

Input: Name, Subject, Marks

Output: Total, Average, Grade

2

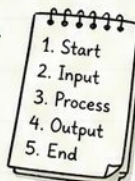
DESIGNING PROGRAMS

Design the logic and structure of your program before coding. Think before you type!

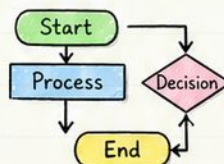
TOOLS & TECHNIQUES

Algorithm / Pseudocode

Write step-by-step instructions in simple English.



Flowchart



Visualize the flow of your program.

Examples & Test Cases

Think of sample inputs and expected outputs to test your logic.



3

ORGANIZING CODE

Organize your code to make it clean, readable and easy to maintain.

BEST PRACTICES

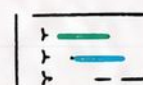
- Use meaningful variable and function names

total_marks ✓
tm ✗

- Break code into functions



- Follow consistent indentation & style



- Add comments where needed

Calculate average marks

- Keep related code together



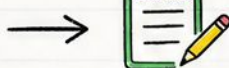
Tip

Clean code is not just for others, it helps you in the future!

THE PROJECT PLANNING FLOW



Understand Requirements



Design the Solution



Organize Code



Test & Debug



Successful Project!

QUICK CHECKLIST

- ✓ I understand the problem
- ✓ I have a plan / algorithm
- ✓ I have test cases
- ✓ I know how to organize my code
- ✓ I am ready to code!



Great code starts with great planning!



Remember in Python:

- Python values readability.
- Simple, clean and modular code is the key.
- Plan smart, code easy, debug less!





PYTHON

QUIZ GAME

PROJECT

A fun, interactive command-line quiz game built with Python!



1 QUESTIONS AND ANSWERS

Store a list of questions with multiple choice options and the correct answer.

EXAMPLE

Q: What is the capital of France?

- A) Berlin
- B) Madrid
- C) Paris
- D) Rome

Correct Answer: C) Paris

Python Example (Questions List)

```

questions = [
    {
        "question": "What is the capital of France?",
        "options": ["A) Berlin", "B) Madrid",
                   "C) Paris", "D) Rome"],
        "answer": "C"
    },
    # Add more questions here...
]

```

2 SCORING SYSTEM

Award points for correct answers and keep track of the total score.

HOW IT WORKS

- Start score at 0.
- +1 point for each correct answer.
- 0 points for incorrect answers.
- Show total score at the end.

```

score = 0
for q in questions:
    # (ask question)
    if user_answer == q["answer"]:
        score += 1
print(f"\nYour Score: {score}/{len(questions)}")

```

Example: You scored 7 out of 10!

3 USER FEEDBACK

Give immediate feedback for every answer to keep the game engaging.



Correct Answer

Great job! +1 point



Wrong Answer

Oops! The correct answer is C) Paris.



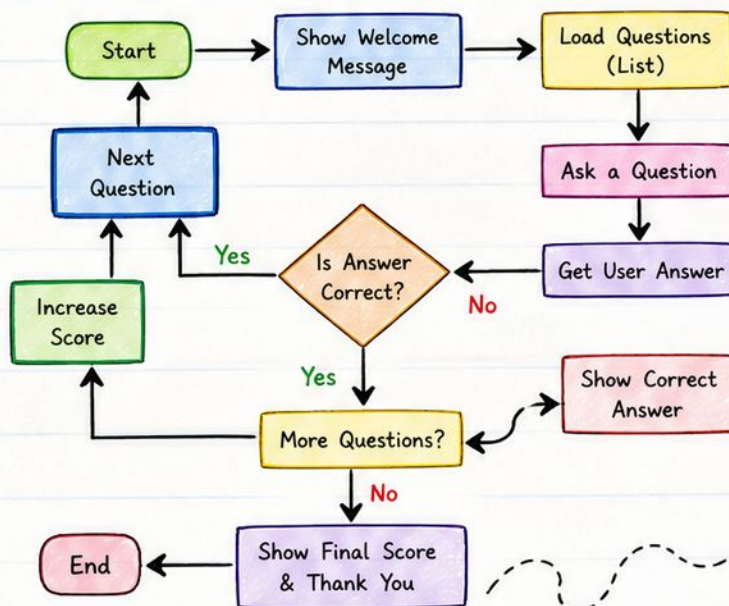
Final Summary

Thanks for playing! Keep learning!



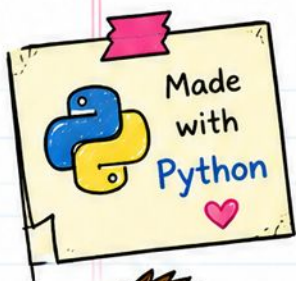
You can add difficulty levels, timers, or random question order to make it even more exciting!

GAME FLOW



CODE. PLAY. LEARN. REPEAT.

Have Fun Coding!



PYTHON PROJECT

ADVENTURE STORY GAME



Create your own interactive text-based adventure where every choice matters!



★ WHAT IS AN ADVENTURE STORY GAME?

It's a text-based game where the player reads the story and makes choices. Each choice leads to different paths and different endings.



KEY CONCEPTS

1 CHOICES AND PATHS

The player chooses an option from the given choices. Each choice leads to a new part of the story.

Example:

Do you go left or right?
→ Left path leads to the forest.
→ Right path leads to the village.



2 CONDITIONS

Conditions decide what happens next based on the player's input. We use **if**, **elif**, **else** statements to control the flow.

Example:

```
if choice == 'left':  
    # go to forest path  
elif choice == 'right':  
    # go to village path  
else:  
    # invalid choice
```



3 MULTIPLE ENDINGS

Different choices lead to different endings. This makes the game exciting and replayable!

Examples:

- ★ You find the treasure and win!
- ★ You get lost in the dark forest.
- ★ You are caught by the guards.
- ★ You become a legend!



HOW IT WORKS (EXAMPLE)

START

You wake up in a dark cave.
You see two exits.

1. Go left
2. Go right



1

2

You walk through a spooky forest.

You see a treasure chest!

1. Open it
2. Ignore it



1

2

You reach a village.
The guard stops you.

1. Talk to the guard
2. Attack the guard



1

2

ENDING

You found gold and become rich!



ENDING

A trap triggers!
You get caught by monsters.



ENDING

The guard trusts you. You are safe and rest.



ENDING

You fight and lose. Game Over.



CODE SNIPPET

```
print("You are in a dark cave.")  
print("Do you go left or right?")  
choice = input("Type 'left' or 'right': ").lower()  
  
if choice == 'left':  
    print("You walk through the forest.")  
    print("You found a treasure! You Win! 🏆")  
elif choice == 'right':  
    print("You reach a village.")  
    print("The guard catches you. Game Over. 😞")  
else:  
    print("Invalid choice. Try again!")
```

Using conditions to control the story!



TIPS

- ✓ Plan your story and pathways first.
- ✓ Use clear and engaging descriptions.
- ✓ Test all choices to avoid dead ends.
- ✓ Add more humor, sound (text), or ASCII art to make it awesome!



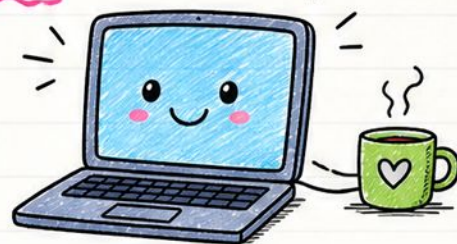
Your story. Your choices. Your adventure.
Start coding and let the adventure begin!



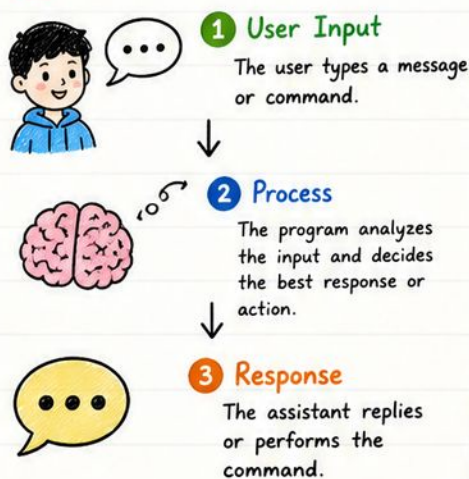
PERSONAL ASSISTANT PROGRAM

Your smart helper, built with Python!

A Personal Assistant Program is a simple Python application that can interact with users by **greeting** them, **answering questions** and executing **simple commands**.



HOW IT WORKS



KEY FEATURES

- Greeting Users**
Welcomes the user with a friendly message.
- Answering Questions**
Provides answers to common questions.
- Simple Commands**
Performs basic tasks like telling time, date, opening websites, and more.

TECH USED

- ✓ Python 3
- ✓ if-else logic
- ✓ Dictionaries
- ✓ Built-in Modules (datetime, webbrowser, random)

Great beginner project to improve logic and Python skills!

KEY CAPABILITIES

1 GREETING USERS

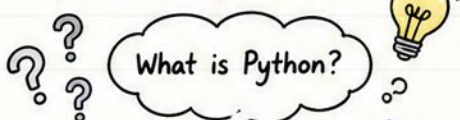
Assistant greets the user warmly.



Example:
User: hi
Assistant: Hello! How can I help you today?

2 ANSWERING QUESTIONS

Answers general knowledge or predefined questions.



Example:
User: what is python?
Assistant: Python is a high-level programming language.

3 SIMPLE COMMANDS

Executes simple tasks to make life easier.



Example:
User: what time is it?
Assistant: The current time is 10:30 AM.

SAMPLE CODE (SNIPPET)

This is just a basic version. You can add more features and make it smarter!

```

import datetime
import webbrowser

def assistant():
    print("Hello! I am your personal assistant.")
    while True:
        user = input("You: ").lower()
        if "hello" in user or "hi" in user:
            print("Assistant: Hello! How can I help you?")
        elif "time" in user:
            now = datetime.datetime.now().strftime("%I:%M %p")
            print(f"Assistant: The current time is {now}")
        elif "google" in user:
            webbrowser.open("https://www.google.com")
            print("Assistant: Opening Google for you!")
        elif "bye" in user or "exit" in user:
            print("Assistant: Goodbye! Have a great day!")
            break
        else:
            print("Assistant: Sorry, I can only do simple tasks for now.")
    assistant()
  
```

IDEAS TO EXPAND

- ★ Add voice input and voice output (using speech_recognition and pyttsx3)
- ★ Integrate with APIs (weather, news, etc.)
- ★ Add reminders and note-taking
- ★ Build a GUI for your assistant

You've learned.
You've practiced.
Now, it's time to
CREATE!

FINAL CAPSTONE PROJECT

The grand finale of your Python journey!

Bring everything you've learned together and
build something amazing from start to finish!

This is your
chance to:

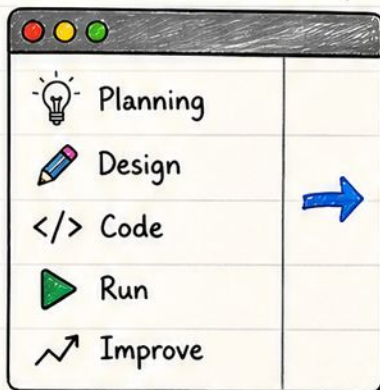
- ✓ Solve real problems
- ✓ Think like a developer
- ✓ Build with confidence
- ✓ Showcase your skills!



1

COMPLETE PYTHON APPLICATION

Build a full project from start to finish!



Key Elements

- User Input
- Data Handling
- Logic & Functions
- Libraries & Modules
- File I/O (Save & Load)
- User Interface (CLI or GUI)

Project Ideas

- ★ Task Manager App
- ★ Student Grade Tracker
- ★ Budget Planner
- ★ Weather Dashboard
- ★ Quiz Game
- ★ Password Manager

... or your own idea!



★ **Tip:** Break your project into small steps. Plan, code, test, and improve each part along the way!

2

TESTING PROGRAMS

Great projects are built, tested, and refined!

Why Test?

- ✓ Find and fix bugs
- ✓ Improve reliability
- ✓ Ensure correct output
- ✓ Build better code
- ✓ Gain confidence



Testing Checklist

- ✓ Does the program run without errors?
- ✓ Does it handle edge cases?
- ✓ Is the output correct?
- ✓ Have you tested all features?
- ✓ Is the user experience smooth?

Tools You Can Use

unittest assert pdb (Debugger) print() Logging

“Test your code today, save yourself from bugs tomorrow!”

3

SHARING PROJECTS

Share your work and inspire others!



Push to GitHub

Store your code, track changes, and build your portfolio.



Add a README

Explain your project, features, how to run it, and screenshots.



Share Your Project

Share the link with friends, mentors, or on social media.



Get Feedback

Feedback helps you improve and grow as a developer.

Great work!



Every project you build is a step toward becoming a better developer!



Build. Test. Share. Repeat.

You've got this! 🏆

Your idea.
Your code.
Your future.

www.codescompiler.com