CHAPTER 2 — VARIABLES & NAMING CONVENTIONS

Variables and Identifiers in Python

1. What is a Variable?

A **variable** is a name that refers to a value stored in the computer's memory.
It acts as a container that holds data that can change during program execution.
When you assign a value to a variable, Python automatically creates it — there is no need to declare its type.

Example:

```
name = "Aarav"

age = 16

pi = 3.14

is_active = True
```

Here, name, age, pi, and is_active are variables that store different types of data.

2. What is an Identifier?

An **identifier** is the name given to any programming element — such as a variable, function, class, or module — to identify it uniquely.
Every variable name is an identifier, but not every identifier is necessarily a variable.

Example:

```
student_name = "Riya"

def greet():

  print("Hello!")
```

In this example, student_name and greet are both identifiers.

3. Rules for Naming Variables (and Identifiers)

To maintain clarity and avoid errors, Python follows specific rules for naming variables and identifiers:

Start with a Letter or Underscore:

The first character must be a letter (A–Z, a–z) or an underscore (_).

```
_count = 10    # valid
```

```
name = "Riya"    # valid

1value = 5      # invalid
```

Cannot Start with a Number:
A name cannot begin with digits.

```
2age = 20    # invalid

age2 = 20    # valid
```

No Spaces Allowed:
Spaces are not permitted; use underscores (_) instead.

```
student_name = "Aarav"    # valid

student name = "Aarav"    # invalid
```

Allowed Characters:
Only letters, numbers, and underscores can be used (A–Z, a–z, 0–9, _).

```
total_marks = 98     # valid

price$ = 100        # invalid
```

Cannot Use Python Keywords:
Python reserves certain words for its own use (like if, else, class, True, False). These cannot be used as variable names.

```
if = 5         # invalid

if_value = 5     # valid
```

Case Sensitivity:
Python treats uppercase and lowercase letters as different.

```
name = "Riya"

Name = "Aarav"

print(name)   # Output: Riya
```

Meaningful Names:
Variable names should clearly describe their purpose.

```
student_age = 16     # good practice

x = 16            # unclear
```

Automatic Creation:
Python creates a variable as soon as a value is assigned; no separate declaration is needed.

```
city = "Delhi"
```

Naming Convention — snake_case:
Python recommends using **snake_case** for variable names (words separated by underscores).
Example: student_name, total_price, is_raining

4. Literals

A **literal** is a fixed value written directly in the code. It represents constant data that does not change during execution.

Examples:

```
x = 10       # Integer literal

pi = 3.14    # Float literal

name = "Aarav" # String literal

flag = True   # Boolean literal
```

Types of Literals:

- **Number Literals:** integers and floats
- **String Literals:** text enclosed in single or double quotes
- **Boolean Literals:** True, False
- **Special Literal:** None → represents the absence of a value

5. Identifier vs Literal (Quick View)

Term	Meaning	Example
Identifier	Name you assign to store or reference data	age
Literal	Actual value stored in the variable	16

Example:

```
age = 16
```

```
# age → identifier

# 16  → literal
```

6. Naming Styles in Python

Python mainly uses two naming styles, but one is preferred.

a) snake_case

Words are written in lowercase and separated by underscores.
Example: student_name, total_marks, average_speed
Used for: variable names, functions
Preferred in Python (PEP 8 standard)

b) camelCase

Each new word starts with a capital letter, not an underscore.
Example: studentName, totalMarks
Used in: Java, JavaScript
Not recommended in Python, but still allowed.

2. Data Types in Python

2.1 What Are Data Types?

Every value in Python has a **data type** that tells the computer what kind of data it is and what operations can be performed on it.
Python automatically determines the data type when you assign a value — this is known as **Dynamic Typing**.

In simple terms:

Data types define **what kind of information** a variable stores — a number, text, or a true/false value.

Examples

```
name = "Riya"        # String → sequence of characters

age = 17             # Integer → whole number

height = 5.4         # Float → decimal number

is_student = True    # Boolean → True or False

score = None         # NoneType → represents no value
```

Key Characteristics

- No need to declare types manually → Python infers automatically.
- Data types control **memory size, operations, and behaviour**.
- Python treats everything as an **object** (even numbers and strings).

2.2 Python's Built-In Data Types (Core Categories)

Python has several built-in data types grouped into main categories:

Category	Data Type	Example
Numeric	int, float, complex	10, 3.14, 2 + 3j
Text	str	"Python", 'Hello'
Boolean	bool	True, False
None	NoneType	None
Sequence	list, tuple, range	[1,2,3], (1,2,3), range(5)
Mapping	dict	{"name": "Riya"}
Set	set, frozenset	{1,2,3}, frozenset({1,2,3})

For this level, we will focus on the **core types:** int, float, str, bool, and NoneType.

2.3 Numeric Data Types

1. Integer (int)

- Represents **whole numbers** (no decimals).
- Can be **positive**, **negative**, or **zero**.

Example:

```
roll_no = 25

temperature = -5

age = 0
```

Used for:

- Counting, IDs, indexing, loops, etc.

Real-Life Example:

```
# Total marks in 3 subjects
math, science, english = 85, 90, 88
total = math + science + english
print("Total Marks:", total)
```

2. Floating-Point Numbers (float)

- Represents **real numbers** with decimals or in scientific notation.
- Used for **precise calculations** and **measurements**.

Example:

```
price = 99.50
pi = 3.14159
distance = 5.6e2   # 5.6 × 10^2 = 560.0
```

Used for:

- Money, physics, temperature, BMI, etc.

Real-Life Example:

```
# BMI Calculation
weight = 60.5
height = 1.65
bmi = weight / (height ** 2)
print("Your BMI is:", round(bmi, 2))
```

3. Complex Numbers (complex)

- Represents numbers with **real** and **imaginary** parts.
- Written as a + bj where j is the imaginary unit.

Example:

```
z = 2 + 3j
print(z.real)   # 2.0
print(z.imag)   # 3.0
```

Used for:

- Scientific computing, AI signal processing, robotics (electrical phase analysis).

2.4 String Data Type (str)

Definition

A **string** is a sequence of characters enclosed in single ' ', double " ", or triple quotes ''' '''.

Example:

```
message = "Hello, Python!"
quote = 'Learning is fun!'
para = """Python is simple,
easy, and powerful."""
```

Common String Operations

Operation	Example	Result
Concatenation	"Hello" + "World"	"HelloWorld"
Repetition	"Hi! " * 3	"Hi! Hi! Hi! "
Indexing	"Code"[0]	'C'
Slicing	"Python"[0:4]	'Pyth'
Length	len("Hello")	5

String Formatting

Combine text and variables using f-strings or .format():

```
name = "Riya"
age = 16
```

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

Real-Life Example:

```
user = "Aarav"

print("Welcome,", user, "to CodeSketchers!")
```

2.5 String Formatting: Placeholders & Format Modifiers

When you print values in Python, you often need to combine **text + variables**, or control how numbers appear (rounding, spacing, alignment).
String formatting helps you create **clean, readable, and professional-looking output**.

Python supports three formatting styles:

A. f-Strings (Modern & Recommended)

This is the cleanest and most readable method.

Example

```
name = "Riya"

age = 17
```

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

Formatting numbers

```
pi = 3.14159

print(f"Rounded value: {pi:.2f}")      # 3.14
```

Alignment

```
print(f"{'Item':10}{'Price':>8}")

print(f"{'Apple':10}{55.5:>8.2f}")
```

B. str.format() Method

Allows flexible formatting using {} placeholders.

Basic Example

```
print("Hello, {}!".format("Aarav"))
```

Number formatting

```
price = 1250.456

print("Price: {:.2f}".format(price))    # 1250.46
```

Alignment

```
print("{:<10} {:>5}".format("Name", "Age"))

print("{:<10} {:>5}".format("Riya", 16))
```

What this means:

1. **{'Item':10}**
   Prints the word *Item* in a column **10 characters wide**, aligned to the left.
   (Python adds extra spaces on the right.)

2. **{'Price':>8}**
   Prints *Price* in a column **8 characters wide**, aligned to the **right**.

3. **{'Apple':10}**
   Prints *Apple* left-aligned in a 10-character column.

4. **{55.5:>8.2f}**
   Prints the number **55.5**

   - right-aligned in a width of **8**
   - formatted to **2 decimal places** (55.50)

Purpose:

This is used to create **clean, table-like output**, where text and numbers line up neatly like in a bill, receipt, or report.

C. Percent % Formatting (Older Style)

Still used in some programs.

Example

```
score = 95.678

print("Score: %.2f" % score)      # 95.68
```

Useful Format Modifiers

Modifier	Meaning	Example	Output
.2f	2 decimal places	{pi:.2f}	3.14
>10	Right-align in width 10	{score:>10}	(spaces) 95
<10	Left-align	{name:<10}	Riya
^10	Center align	{item:^10}	Apple
,	Thousands separator	{salary:,.2f}	1,23,456.78
+	Show + or – sign	{value:+.1f}	+5.0
03d	Zero-padding	{num:03d}	007

These modifiers work in both **f-strings** and **format()**.

Examples

Formatting a bill

```
item = "Notebook"
price = 45.5
quantity = 3
print(f"Item: {item}")
print(f"Total: {price * quantity:.2f}")
```

Percentage

```
accuracy = 0.91234
print(f"Accuracy: {accuracy:.2%}")    # 91.23%
```

Table format

```
print(f"{'Name':10}{'Marks':>10}")
print(f"{'Riya':10}{85:>10}")
print(f"{'Aarav':10}{90:>10}")
```

String formatting helps you neatly combine text and variables, control decimal places, align output, and create clean, professional results.

2.6 Boolean Data Type (bool)

Definition

Booleans represent **True** or **False** values, used for **decision-making**.

Example:

```
is_active = True

is_subscribed = False
```

Comparison Example:

```
a, b = 10, 20

print(a > b)    # False

print(a != b)   # True
```

Used For:

- Conditions, loops, logical operations.

Real-Life Example:

```
is_raining = True

if is_raining:

  print("Take an umbrella!")

else:

  print("Enjoy the sunshine!")
```

2.7 NoneType

Definition

None represents **no value**, **empty**, or **null**.

Example:

```
student_email = None
```

```
print(student_email)
```

Used For:

- Placeholder variables
- Function defaults
- Resetting variables

Real-Life Example:

```
temperature = None
if temperature is None:
  print("Sensor not connected!")
```

2.8 Mutable vs Immutable Data Types

Type	Mutable (Can Change)	Immutable (Cannot Change)
Definition	Values can be modified after creation	Value cannot be changed after creation
Examples	list, dict, set	int, float, str, bool, tuple
Analogy	Whiteboard → can erase & rewrite	Printed paper → cannot change once printed

Code Example:

```
# Mutable
colors = ["red", "blue"]
colors.append("green")
print(colors)


# Immutable
name = "Aarav"
name = name + " Kumar"   # Creates new string
print(name)
```

2.9 Checking and Converting Data Types”

2.9.1 Type Casting Errors

Type conversion works only when the value is valid for the target type.
Invalid conversions will cause an error.

Examples of Invalid Type Casting

```
int("hello")      # Error → cannot convert text to number

float("abc")      # Error

int("12.5")       # Error → convert float string directly? Not allowed

bool("False")     # True → non-empty string always becomes True
```

Common Mistakes Students Make

- Converting text that is not a valid number
- Forgetting that "123abc" cannot be converted
- Assuming "False" becomes False (it becomes **True**)

Correct Way

```
int("25")   # ✅ works

float("12.5") # ✅ works
```

Safe Conversion Tip

Always check if a string is numeric before converting.

```
num = "45"

if num.isdigit():

  print(int(num))
```

2.10 Checking and Converting Data Types

Checking Type

Use the type() function to know the data type of a variable:

```
x = 12

y = 5.8
```

```
z = "Hello"
print(type(x))  # <class 'int'>
print(type(y))  # <class 'float'>
print(type(z))  # <class 'str'>
```

Type Conversion (Typecasting)

Convert one data type to another using built-in functions.

Function	Purpose	Example	Result
int()	Converts to integer	int(3.5)	3
float()	Converts to float	float(5)	5.0
str()	Converts to string	str(25)	"25"
bool()	Converts to boolean	bool(0)	False

Example:

```
num_str = "50"
num = int(num_str)
print(num + 10)   # Output: 60
```

3. Type Conversion in Python

Python converts data types in two ways: **Implicit (automatic)** and **Explicit (manual)**.

3.1 Implicit Type Conversion (Automatic Promotion)

Python automatically converts one data type to another **when it is safe and lossless**.

Why does Python convert automatically?

To ensure the final result makes mathematical sense.

Example:

```
x = 10      # int
y = 5.5     # float
```

```
result = x + y

print(result)    # 15.5
```

What happened internally?

Python converted 10 → 10.0 because a float can hold more precision than an integer.

Another example: Boolean auto-conversion

```
value = True + 4

print(value)    # 5
```

Internally:

- True → 1
- False → 0

Important Note

Implicit conversion **never converts string ↔ numbers** automatically.
This prevents unexpected errors.

3.2 Explicit Type Conversion (Manual Typecasting)

You convert one type to another using built-in functions.

Common Type Conversion Functions

Function	Converts To	Example	Output
int()	Integer	int("50")	50
float()	Float	float("3.14")	3.14
str()	String	str(25)	"25"
bool()	Boolean	bool(1)	True
type()	Returns type	type(3.5)	<class 'float'>

Examples

```
num_str = "100"

num = int(num_str)
```

```
print(num + 50)    # 150
```

Boolean Conversion Rules

- 0, "", None, [], {} → False
- Everything else → True

Example:

```
bool(0)    # False
bool("Hi")  # True
```

Type Conversion Errors

Some conversions are impossible.

Example:

```
int("hello")     # Error
float("4.5.6")    # Error
```

Reason: The text cannot be interpreted as a number.

User Input in Python

User input is essential for interactive programs.

4.1 The input() Function

input() ALWAYS returns a **string**, no matter what the user types.

Example:

```
name = input("Enter your name: ")
print("Hello,", name)
```

Internal Behavior

- Python pauses and waits for user input
- Converts whatever is typed into a string
- Stores it in a variable

4.2 Converting User Input into Numbers

Since input returns a string, convert it using:

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

If you don't convert, arithmetic operations will fail.

Example:

```
x = input("Enter a number: ")
y = input("Enter another number: ")
print(x + y)
```

If you type 5 and 7, output is:

57 (string concatenation)

But after conversion:

```
x = int(x)
y = int(y)
print(x + y)      # 12
```

4.3 Real-Life Practical Tasks

1. Greeting User

```
name = input("Enter your name: ")
print(f"Welcome, {name}! Happy Learning.")
```

2. Add Two Numbers

```
a = float(input("Enter number 1: "))
b = float(input("Enter number 2: "))
print("Sum =", a + b)
```

3. Calculate Age After 10 Years

```
age = int(input("Your age: "))
```

```
print("After 10 years, you will be", age + 10)
```

4. Simple Salary Calculator

```
hourly_rate = float(input("Hourly rate: "))

hours = float(input("Hours worked: "))

print("Total Salary:", hourly_rate * hours)
```

5. Operators in Python

Operators allow Python to perform calculations, comparisons, and decision-making.

5.1 Arithmetic Operators

Operator	Operator Name	Operator Function	Example and Explanation
+	Addition	Add two/more values	x = 4; y = 5; => x + y => 9
-	Subtraction	Subtract two/more values	x = 10; y = 4; => x - y => 6
*	Multiplication	Multiply two/more values	x = 4; y = 5; => x * y => 20
/	Division	Divide two values, and produce the n	x = 10; y = 4; => x / y => 2.25
%	Modulus	Gives the remainder after one value is divided with other	x = 14; y = 4; => x % y => 2
**	Exponential	Gives the exponential values	x = 3; y = 4; => x ** y => 81
//	Floor Division	Gives the integer value on dividing one value with other	x = 14; y = 4; => x // y => 3

5.2 Assignment Operators

Used to update variable values.

Operator	Example	Equivalent Expression (m=15)	Result
=	y = a+b	y = 10 + 20	30
+=	m +=10	m = m+10	25
-=	m -=10	m = m-10	5
*=	m *=10	m = m*10	150
/=	m /=10	m = m/10	1.5
%=	m %=10	m = m%10	5
**=	m**=2	m = m**2 or $m = m^2$	225
//=	m//=10	m = m//10	1

Operator	Meaning	Example
=	Assign value	x = 10
+=	Add and assign	x += 3 → x = x + 3
-=	Subtract and assign	x -= 2
*=	Multiply and assign	x *= 4
/=	Divide and assign	x /= 3
%=	Modulo and assign	x %= 5
//=	Floor divide and assign	x //= 2
**=	Power and assign	x **= 3

Deep Example

```
salary = 20000
salary += 5000    # 25000
salary *= 2       # 50000
```

5.3 Comparison (Relational) Operators

Used to compare values (result is True/False).

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

5.4 Logical Operators

Combine conditions.

Operator	Meaning	Example
and	True if both conditions true	age > 18 and is_student
or	True if one condition true	age > 18 or is_student
not	Reverses condition	not True → False

Operand 1	Operand 2	Result
True	True	True
True	False	False
False	True	False
False	False	False
True	Expression	Expression
False	Expression	False

Deep Example

age = 17

```
has_id = True
print(age >= 18 and has_id)   # False
print(age >= 18 or has_id)    # True
```

Built-in Functions

Function	Purpose
int()	Convert to integer
float()	Convert to float
str()	Convert to string
bool()	Convert to boolean
type()	Check data type
input()	Take user input

2.11 Real-Life Mini Activities

1. Calculate Simple Interest

```
p = 1000
r = 5
t = 2
si = (p * r * t) / 100
print("Simple Interest:", si)
```

2. Check Student Eligibility

```
age = 17
if age >= 18:
   print("Eligible to vote")
else:
   print("Not eligible yet")
```

3. Temperature Converter

```
celsius = 37

fahrenheit = (celsius * 9/5) + 32

print("Temperature in Fahrenheit:", fahrenheit)
```

2.12 Summary

- Data types define **what kind of data** a variable holds.
- Python automatically assigns types → **Dynamic Typing**.
- Core data types include:
  int, float, str, bool, and NoneType.
- Mutable types can be changed, immutable types cannot.
- Use type() to check and int(), float(), str() to convert.
- Understanding data types helps build logic for real-world programs.