

# Python

EVERYTHING ABOUT OPERATORS AND DATA TYPE

# Today's Agenda

- Operator business use case
- Precedence Of Python Operator
- Type casting
- Input Function

# Business Use Case Of Operators

- **Scenario:**

A company is dealing with a large amount of data that needs to be transmitted over the network. To optimize the transmission speed and reduce costs, the company decides to compress the data. One approach to data compression is to pack multiple smaller pieces of data into a single larger data type, effectively reducing the total number of bits needed.

- **Example:**

Suppose the company is dealing with data that contains multiple attributes, each represented by a small range of values. For instance, they have:

- ❑ A product category ID (0-15): can be represented by 4 bits.
- ❑ A region code (0-3): can be represented by 2 bits.
- ❑ A status flag (0-1): can be represented by 1 bit.

Instead of storing these attributes separately, they can be packed into a single byte (8 bits) using bitwise operations.

- `product_category_id << 3`: Shifts the product category ID 3 bits to the left, making room for the region code and status flag.
- `region_code << 1`: Shifts the region code 1 bit to the left, making room for the status flag.
- `|` (bitwise OR): Combines the shifted values into a single byte.

```
# Assign values to the attributes
product_category_id = 9 # 1001 in binary (4 bits)
region_code = 2         # 10 in binary (2 bits)
status_flag = 1         # 1 in binary (1 bit)

# Pack the attributes into a single byte using bitwise operators
compressed_data = (product_category_id << 3) | (region_code << 1) | status_flag

# Now `compressed_data` is a single byte that contains all the information.
print(f"Compressed Data: {compressed_data} (Binary: {compressed_data:08b})")
```

Output:

less

Compressed Data: 77 (Binary: 01001101)

# Precedence of Python Operators

- The combination of values, variables, operators, and function calls is an expression. The Python interpreter can evaluate a valid expression.

```
>>> 5 - 7  
-2
```

Here `5 - 7` is an expression. There can be more than one operator in an expression.

- To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out.

For example, multiplication has higher precedence than subtraction.

```
# Multiplication has higher precedence  
# than subtraction  
>>> 10 - 4 * 2  
2
```

But we can change this order using parentheses `()` as it has higher precedence than multiplication.

```
# Parentheses () has higher precedence  
>>> (10 - 4) * 2  
12
```

# Type Casting

## WHAT IS TYPE CASTING?

- The process of converting one data type into another.
- Essential when working with different data types in a single operation.

## WHY IS IT IMPORTANT?

- Ensures data compatibility.
- Prevents type errors in operations involving different data types.

# Types of Type Casting

## IMPLICIT TYPE CASTING (AUTOMATIC)

- Performed automatically by Python.
- Python converts smaller data types to larger data types to avoid data loss.
- Example: int to float.

## EXPLICIT TYPE CASTING (MANUAL)

- Performed by the programmer using built-in functions.
- Allows control over the conversion process.
- Example: `str()` to convert a number to a string.

## IMPLICIT TYPE CASTING EXAMPLE

python

```
num_int = 100 # Integer  
num_float = 1.5 # Float  
result = num_int + num_float  
print(result) # Output: 101.5
```

- Python automatically converts num\_int (int) to float before addition.
- The result is a float value.



## EXPLICIT TYPE CASTING EXAMPLE

Common Functions:

- `int()`: Converts to integer
- `float()`: Converts to float.
- `str()`: Converts to string.
- `list()`, `tuple()`, `dict()`: Convert to respective data structures.

python

```
num_str = "123"  
num_int = int(num_str)  
print(type(num_int)) # Output: <class 'int'>
```

The string "123" is explicitly converted to an integer using `int()`.

# Handling Type Casting Errors

## COMMON ISSUES

- Invalid conversions (e.g., converting a non-numeric string to an integer).
- Data loss during conversion (e.g., converting a float to an int).
- Example of an Error

```
invalid_str = "ABC"  
num_int = int(invalid_str) # Raises ValueError
```

## BEST PRACTICES

- Validate data before casting.
- Use try-except blocks to handle potential errors.



## USE CASE: DATA INTEGRATION FOR A RETAIL BUSINESS

### **Objective:**

To consolidate and analyze this data efficiently, the data engineering team needs to standardize the data types across all sources.

- Scenario: A retail company wants to integrate data from multiple sources—such as online sales, in-store sales, customer feedback, and inventory systems—into a centralized data warehouse. These data sources have different formats and data types. For example, sales data might be stored as strings in one system and as integers in another, while dates might be stored as strings in various formats across systems.



## STEPS INVOLVED

Extract data from multiple sources

SQL databases, CSV files, JSON API's

- Example : A column `sales_amount` is stored as a string in a CSV file but as a float in the SQL database.
- Converting Strings to Numbers : Sales data stored as strings (e.g., "1500", "2000") needs to be converted to integers or floats for aggregation and analysis.
- Converting Strings to Dates : Dates stored as strings in different formats need to be standardized.
- Handling Null Values : Missing or null values might be represented as "None" or "NaN" strings and need to be converted to appropriate data types.

# The input() Function in Python

## WHAT IS INPUT()?

- A built-in Python function that captures user input from the console.
- Returns the input as a string, regardless of the user's input type.

## WHY IS IT IMPORTANT?

- Enables interactive programs where the user provides data at runtime.
- Commonly used in applications that require user interaction.

# Basic Usage of input()

## SYNTAX

- Syntax :

```
user_input = input("Prompt message")
```

- Example:

```
name = input("Enter your name: ")  
print(f"Hello, {name}!")
```

## EXPLANATION

- The user is prompted with "Enter your name: ".
- The input is stored in the name variable and then used in the greeting.

## CONVERTING INPUT DATA TYPES

python

```
age = input("Enter your age: ")  
age = int(age) # Convert the input to an integer  
print(f"Next year, you will be {age + 1} years old.")
```

- Default Behaviour: The input() function always returns a string.
- Type Conversion: Use type casting to convert the input to the desired type.
- Explanation: The string input "25" is converted to an integer using int(), allowing arithmetic operations.

# HANDLING INVALID INPUT

```
try:
    age = int(input("Enter your age: "))
    print(f"Next year, you will be {age + 1} years old.")
except ValueError:
    print("Please enter a valid number.")
```

```
while True:
    age = input("Enter your age: ")
    if age.isdigit():
        age = int(age)
        break
    else:
        print("Invalid input. Please enter a numeric value.")
print(f"Next year, you will be {age + 1} years old.")
```

- Potential Issues: If the input cannot be converted to the desired type, it raises a `ValueError`. `Img(1)`
- Using try-except to Handle Errors: Prevents the program from crashing due to invalid input.
- Advanced Usage: Input Validation  
Continuously prompt the user until a valid input is received. `Img(2)`
- **Explanation:** The loop continues until the user enters a valid numeric input. `Img(2)`



# Use Cases of input() in Applications

- **Survey Forms:** Collecting user information like name, age, and preferences.
- **Interactive Games:** Asking players for their moves or choices.
- **Command-Line Tools:** Accepting parameters or options from the user at runtime.

```
choice = input("Do you want to continue? (yes/no): ").lower()
if choice == "yes":
    print("Continuing...")
else:
    print("Exiting...")
```

# Best Practices for Using input()

- **Prompt Clearly:** Provide clear instructions so the user knows what to enter.
- **Validate Input:** Always validate and handle potential errors.
- **Consider User Experience:** Offer default values or guidance if the input is complex. Example -

```
username = input("Enter your username (default: guest): ") or "guest"  
print(f"Welcome, {username}!")
```

# The eval() Function in Python

## WHAT IS EVAL()?

- A built-in Python function that parses the expression passed to it and executes Python code within it.
- Used to evaluate string expressions as Python code.

## WHY IS IT IMPORTANT?

- Enables dynamic evaluation of expressions at runtime.
- Useful for executing code stored in text or responding to user inputs.

## SYNTAX OF EVAL()

```
name = eval(input("enter name :"))
```

- Basic Usage of eval():
  - ❑ Example 1: Evaluating a Mathematical Expression

```
expression = "2 + 3 * 4"  
result = eval(expression)  
print(result) # Output: 14
```

- ❑ Example 2: Using eval() with Variables

```
x = 10  
expression = "x * 2"  
result = eval(expression)  
print(result) # Output: 20
```

# Security Risks of eval()

- **Code Injection:** If eval() is used with untrusted input, it can execute arbitrary code, leading to security vulnerabilities.

## Example of Risk:

python

```
user_input = "__import__('os').system('rm -rf /')"  
eval(user_input) # Dangerous if executed!
```

## Why This Is Dangerous:

- **Execution of Malicious Code:**
  - The command `rm -rf /` is a highly destructive shell command that attempts to delete all files and directories from the root directory (`/`) on a Unix-like system, such as Linux or macOS.
  - If this code were executed on a machine with sufficient permissions, it could effectively wipe out the entire filesystem, leading to catastrophic data loss.
- **Code Injection Attack:**
  - If `eval()` is used to execute user-provided input without any validation or sanitization, an attacker could provide a malicious input that gets executed, potentially compromising the system.
  - This is a classic example of a code injection attack, where an attacker injects and executes arbitrary code.

# Which is best while working in business eval or input?

- When working in a business environment, especially where security, reliability, and data integrity are critical, `input()` is generally preferred over `eval()`.
- Here's why:

## 1. Security:

- `input()`: Captures user input as a string and requires explicit conversion or processing to use it, making it safer and less prone to security vulnerabilities.
- `eval()`: Evaluates the string as Python code, which can be dangerous if the input is not strictly controlled. Malicious users could inject harmful code.

## 2. Control:

- `input()`: Provides more control over what the program does with the input. You can validate, sanitize, and convert the input as needed before processing it.
- `eval()`: Executes the input as code directly, which could lead to unintended consequences if the input is unexpected or contains errors.

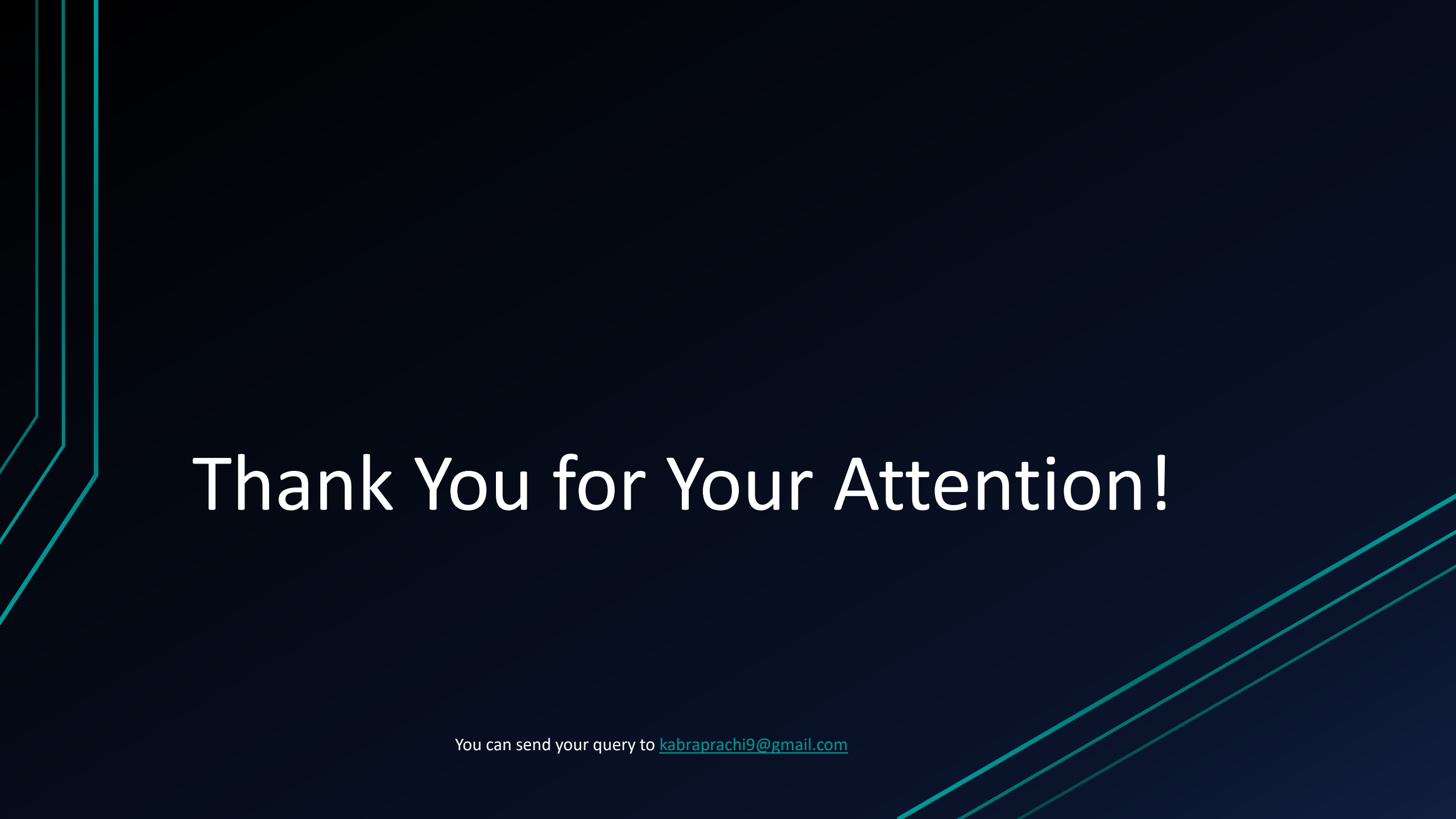
## 3. Reliability:

- `input()`: Forces you to explicitly handle different data types and input formats, leading to more robust code. You can easily add checks and error handling.
- `eval()`: While powerful, it can lead to unexpected results if the expression is complex or if the input isn't properly validated, making the code less reliable.

## Use Cases:

`input()`: Ideal for most business applications, such as collecting user input, processing data, or interacting with users in a controlled and secure way.

`eval()`: Best suited for situations where dynamic evaluation of expressions is absolutely necessary, but with strict validation and control mechanisms in place.



# Thank You for Your Attention!

You can send your query to [kabraprachi9@gmail.com](mailto:kabraprachi9@gmail.com)