Type casting in Python refers to the process of converting one data type to another. This is particularly useful when you need to ensure that the operations you are performing are compatible with the data types involved. Python supports two types of casting:

1. **Implicit Casting** (automatically done by Python)
2. **Explicit Casting** (manually done by the programmer)

**1. Implicit Type Casting**

In implicit casting, Python automatically converts one data type to another when it makes sense. This is often done when combining different types in expressions to avoid loss of information.

**Example of Implicit Type Casting:**

a =15

b =10.5

c= a+b

print("The sum of int and float is:", c)

Output:The sum of int and float is: 25.5

### 2. Explicit Type Casting

Explicit type casting is when the programmer manually converts one data type to another using Python's built-in functions. Some common type casting functions are:

* int(): Converts to integer
* float(): Converts to float
* str(): Converts to string
* list(): Converts to list
* tuple(): Converts to tuple
* set(): Converts to set
* dict(): Converts to dictionary

#### Example of Explicit Type Casting:

# String to Integer

s = "123"

n = int(s)

print(n)            # Output: 123

print(type(n))      # Output: <class 'int'>

# Float to Integer (Note: decimals are truncated, not rounded)

f = 9.8

i = int(f)

print(i)            # Output: 9

print(type(i))      # Output: <class 'int'>

# Integer to String

num = 50

s = str(num)

print(s)            # Output: '50'

print(type(s))      # Output: <class 'str'>

# List to Set (removes duplicates)

lst = [1, 2, 3, 2, 4]

st = set(lst)

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

print(type(st))     # Output: <class 'set'>

List of Data types that can be converted and few cant

In Python, certain types can be converted to others, while some conversions are not allowed due to data incompatibility. Below is a list of conversions for int, string, and float, specifying what data types they **can** and **cannot** be converted to.

### 1. ****Integer (****int****)****

#### Can be Converted To:

* **float**: Converts an integer to a floating-point number.

python

Copy code

num = 5

print(float(num)) # Output: 5.0

* **str**: Converts an integer to its string representation.

python

Copy code

num = 5

print(str(num)) # Output: "5"

* **bool**: Converts an integer to a boolean. 0 becomes False, all other values become True.

python

Copy code

num = 5

print(bool(num)) # Output: True

print(bool(0)) # Output: False

#### Cannot be Converted To:

* **list**, **set**, **tuple**, **dict**: Direct conversion of an integer to these data types is not allowed because an integer is a single value, and these types represent collections of values.

python

Copy code

# These will raise a TypeError

list(5) # TypeError: 'int' object is not iterable

set(5) # TypeError: 'int' object is not iterable

tuple(5) # TypeError: 'int' object is not iterable

### 2. ****String (****str****)****

#### Can be Converted To:

* **int**: Converts a numeric string to an integer (only if the string represents an integer).

python

Copy code

s = "123"

print(int(s)) # Output: 123

* + **Note**: This conversion will raise an error if the string contains non-integer values.
* **float**: Converts a numeric string to a float (only if the string represents a valid float).

python

Copy code

s = "123.45"

print(float(s)) # Output: 123.45

* **list**, **set**, **tuple**: Converts a string to a list, set, or tuple, where each character becomes an individual element.

python

Copy code

s = "abc"

print(list(s)) # Output: ['a', 'b', 'c']

print(set(s)) # Output: {'a', 'b', 'c'}

print(tuple(s)) # Output: ('a', 'b', 'c')

* **bool**: Converts a string to a boolean. An empty string becomes False, and any non-empty string becomes True.

python

Copy code

s = ""

print(bool(s)) # Output: False

print(bool("abc")) # Output: True

#### Cannot be Converted To:

* **dict**: Direct conversion of a string to a dictionary is not allowed, as dictionaries require key-value pairs.

python

Copy code

# This will raise a ValueError

dict("key") # ValueError: dictionary update sequence element #0 has length 1; 2 is required

* **int or float** (for non-numeric strings): If the string contains alphabetic or special characters that cannot represent a number, conversion will fail.

python

Copy code

s = "abc"

int(s) # ValueError: invalid literal for int()

### 3. ****Float (****float****)****

#### Can be Converted To:

* **int**: Converts a floating-point number to an integer by truncating the decimal part (no rounding).

python

Copy code

f = 5.67

print(int(f)) # Output: 5

* **str**: Converts a floating-point number to its string representation.

python

Copy code

f = 5.67

print(str(f)) # Output: "5.67"

* **bool**: Converts a floating-point number to a boolean. 0.0 becomes False, all other values become True.

python

Copy code

f = 0.0

print(bool(f)) # Output: False

print(bool(5.67)) # Output: True

#### Cannot be Converted To:

* **list**, **set**, **tuple**, **dict**: Direct conversion of a float to these types is not allowed.

python

Copy code

# These will raise a TypeError

list(5.67) # TypeError: 'float' object is not iterable

set(5.67) # TypeError: 'float' object is not iterable

tuple(5.67) # TypeError: 'float' object is not iterable

Summary Table:

| **From** | **Can Be Converted To** | **Cannot Be Converted To** |
| --- | --- | --- |
| int | float, str, bool | list, set, tuple, dict |
| str | int, float, list, set, tuple, bool | dict (direct), int/float for non-numeric strings |
| float | int, str, bool | list, set, tuple, dict |

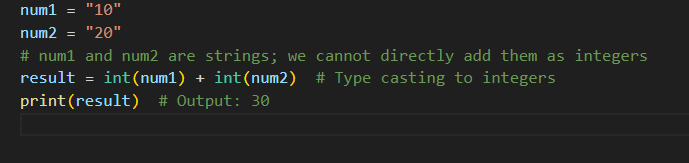
**Why we need to use type casting?**

**T**ype casting is important in Python for several reasons, particularly when handling operations or working with various data types. Here are key reasons why we need to use type casting:

### 1. ****Ensuring Correct Data Types for Operations****

Certain operations in Python require specific data types. For example, mathematical operations require numeric types (integers or floats), while string concatenation works only with strings.

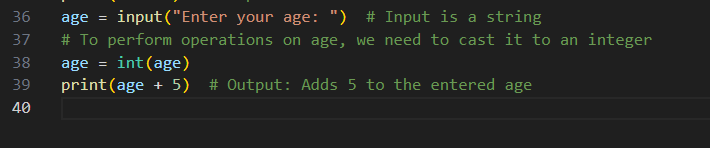
Ex



### 2. ****Interacting with User Input****

In Python, input from users is always treated as a string, regardless of what the user enters. Type casting allows you to convert this input into a data type that can be used for further operations, such as mathematical calculations.

Ex:

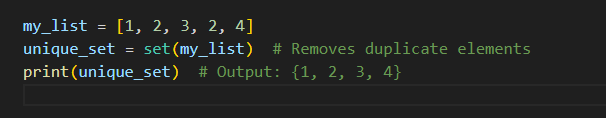


### 3. ****Converting Between Data Structures****

When handling data from various sources (such as lists, tuples, or dictionaries), you often need to convert one type of structure to another. Type casting allows for this conversion, ensuring you can use the appropriate operations for the data structure.

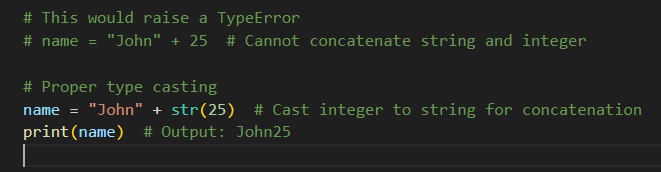
#### Example:

* **From List to Set**: If you want to remove duplicates from a list, you can convert the list to a set.



### 4. ****Preventing Errors during Operations****

When working with mixed data types, using type casting can prevent errors that occur due to incompatible types. For example, adding a string and an integer without type casting results in an error.

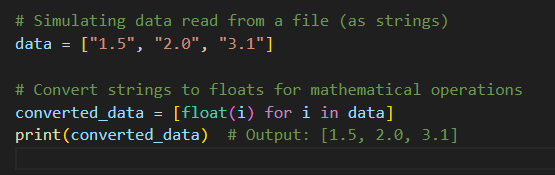


### 5. ****Working with External Data (Files, APIs, Databases)****

When importing data from external sources (like files, APIs, or databases), the data often comes in formats that may not match the expected types. Type casting helps you convert the data into the correct format for processing.

#### Example:

If you are reading numeric data from a file, it might come in as strings. You need to cast it to integers or floats to perform any calculations.



### 6. ****Handling JSON and Data Serialization****

When working with JSON or other serialization formats, Python data types need to be converted for storage or transfer. For instance, JSON supports only certain data types, so type casting may be necessary when converting between Python and JSON objects.

data = {

    "name": "Alice",

    "age": 25,  # Integer in Python, but in JSON, it must be a number

}

# Convert Python object to JSON

json\_data = json.dumps(data)

print(json\_data)  # Output: '{"name": "Alice", "age": 25}'

### 7. ****Ensuring Consistency Across Data Types****

Type casting ensures consistent data types when working with collections, databases, or APIs. For example, casting all elements in a list to a common data type makes it easier to process.

#### Example:

mixed\_list = [1, "2", 3.5, "4.0"]

# Ensure all elements are floats

consistent\_list = [float(i) for i in mixed\_list]

print(consistent\_list)  # Output: [1.0, 2.0, 3.5, 4.0]