Data Handling

1. Expressions

An expression is a valid combination of variables, constants, operators, and function calls that the Python interpreter evaluates to produce a single value.

Example:

```
result = 5 + 3 * 2  # The right side, "5 + 3 * 2", is an expression.
name = "Alice"      # Even a single value is an expression.
```

2. Types of Expressions: An expression is a combination of values, variables, operators, and function calls that can be evaluated to produce a single value.

Expressions are categorized based on the operators and operands involved.

(i) Arithmetic Expressions: Use arithmetic operators (+, -, *, /, %, //, **).

```
(a + b) * (c - d)
```

(ii) Relational (Comparison) Expressions: Use relational operators (==, !=, >, <, >=, <=). They evaluate to True or False.

```
age >= 18
```

(iii) Logical Expressions: Use logical operators (and, or, not). They combine relational expressions.

```
marks > 33 and attendance > 75
```

(iv) Assignment Expression: Uses the assignment operator (=) or its variants (+=, -=, etc.).

```
total = 100
count += 1
```

(v) String Expressions: Use string operators (+ for concatenation, * for repetition).

```
name= “India”
"Hello " + name
```

3. Operator Precedence

This defines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated first.

Analogy: In mathematics, we do multiplication before addition (BODMAS/PEMDAS). Python has a similar rulebook.

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus
`+` `-`	Addition and subtraction
`<<` `>>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==` `!=` `>` `>=` `<` `<=` `is` `is not` `in` `not in`	Comparisons, identity, and membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

Example:

```
result = 5 + 3 * 2 ** 2

# Step 1: 2 ** 2 = 4  (Exponentiation highest)
```

```
# Step 2: 3 * 4 = 12   (Multiplication next)

# Step 3: 5 + 12 = 17  (Addition last)

print(result) # Output: 17
```

Rule of Thumb: When in doubt, use parentheses ( ) to force the order of evaluation.

4. Operator Associativity

When multiple operators of the same precedence appear in an expression, associativity determines the direction of evaluation.

Left-to-Right Associativity: Most operators (like +, -, *, /, %, //).

10 - 5 - 2 is evaluated as (10 - 5) - 2 = 3

Right-to-Left Associativity: Assignment (=) and exponentiation (**).

a = b = 5 is evaluated as b = 5, then a = b.

2 ** 3 ** 2 is evaluated as 2 ** (3 ** 2) = 2 ** 9 = 512

5. Typecasting

The process of converting a variable from one data type to another.

Why is it needed? To perform operations between compatible types and to ensure data is in the correct format.

(a) Implicit Typecasting: Python automatically converts a lower data type (integer) to a higher data type (float) to prevent data loss.

```
a = 5    # integer

b = 2.5  # float

c = a + b # c becomes 7.5 (float)
```

(b) Explicit Typecasting: The programmer manually converts the data type using built-in functions.

* int(): Converts to integer.
* float(): Converts to float.
* str(): Converts to string.

```
num_str = "123"
num_int = int(num_str)  # Converts string "123" to integer 123
number = 100
text = "The number is " + str(number) # Converts integer to string for concatenation
```

6. Standard Library Modules

Python's standard library is a collection of pre-written code (modules and packages) that provides common functionality. You need to import a module to use its features.

Syntax:

```
import module_name
from module_name import function_name
```

7. Math Module

Provides mathematical functions and constants.

```
import math
```

Common Functions/Constants in math module:

* math.sqrt(x): Returns square root of x.
* math.pow(x, y): Returns x raised to the power y.
* math.ceil(x): Returns smallest integer >= x.
* math.floor(x): Returns largest integer <= x.

* math.fabs(x): Returns absolute value of x.
* math.pi: Constant value of π (approx. 3.14159...)

8. Statistics Module

Provides functions for calculating mathematical statistics of numeric data.

import statistics

Common Functions:

* statistics.mean(data): Returns the average (arithmetic mean).
* statistics.median(data): Returns the middle value.
* statistics.mode(data): Returns the most frequent value.
* statistics.stdev(data): Returns the standard deviation.

9. Random Module: It is used to generate random numbers.

import random

Common Functions:

* random.random(): Returns a random float between 0.0 and 1.0.
* random.randint(a, b): Returns a random integer N such that a <= N <= b.
* random.randrange(a, b): Returns a random integer N such that a >= N < b.
* random.choice(sequence): Returns a random element from a non-empty sequence (like a list).
* random.shuffle(sequence): Shuffles the sequence in place.

10. Debugging

The process of identifying and removing bugs (errors) from a program is called debugging.

Methods of debugging:

* **Code Reading:** Manually tracing through the code.

* **Print Statement Debugging:** Inserting print() statements to check variable values at different points.

* **Using a Debugger:** Using tools (like in IDLE or PyCharm) to set breakpoints and step through code line-by-line.

11. Errors in a Program

A. Compile-time Errors:

Errors detected by the Python interpreter during the parsing (translation) phase, before the code is executed.

i. Syntax Error:

Violation of Python's grammatical rules (e.g., missing colon, unmatched parentheses, incorrect indentation).

Example:

```
if x == 5                    # SyntaxError: Missing colon ':'
print("Hello World"          # SyntaxError: Unclosed parenthesis
```

ii. Semantic Error:

The statement is syntactically correct but doesn't make any sense. Python often catches these as syntax errors or NameError.

Example:

```
a + = 5          # SyntaxError: Invalid syntax (space between + and =)
```

B. Runtime Errors:

Errors that occur during the execution of the program. The code is syntactically correct, but an operation fails. It is also called Exceptions.

Examples:

* ZeroDivisionError: Division by zero.
* ValueError: Invalid conversion, e.g., int("abc").
* NameError: Using a variable that has not been defined.
* IndexError: List index out of range.
* FileNotFoundError: Trying to open a file that doesn't exist.

C. Logical Errors:

It is the most difficult to find. The program runs without crashing and produces an output, but the output is incorrect. It is due to some flaw in the program's logic or algorithm. It is not detected by Python.

Example:

```
# Program to calculate average of two numbers
a = 10
b = 20
average = a + b / 2                    # Logical Error! Should be (a + b) / 2
print(average)                         # Output: 20.0 (Incorrect)
                                       # Correct output should be 15.0
```

12. Exceptions:

Runtime errors are represented as Exception Objects. When a runtime error occurs, Python "raises" an exception. If not handled, it causes the program to stop.

Exceptions can be handled using **try...except** blocks to make the program more robust.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
```

```
        print(f"Result is {result}")
except ZeroDivisionError:
        print("You cannot divide by zero!")
except ValueError:
    print("That's not a valid number!")
```