

Module 1: Introduction, Setup, and Fundamentals

1.1 Python Fundamentals & History

What is Python?

Python is a high-level, interpreted, general-purpose programming language created by Guido van Rossum and first released in 1991. It is designed for **readability** and features an often-cited simple syntax, making it an excellent choice for beginners.

Feature	Explanation
High-Level	Programmers do not need to manage memory or complex system details (like CPU architecture). Focus is on solving problems, not system management.
Interpreted	The code is executed line-by-line by an interpreter, not compiled into machine code all at once before running. This makes debugging easier.
Dynamically Typed	You don't need to declare a variable's data type (like int or string) before using it. The type is checked and assigned during runtime.
General-Purpose	Can be used for almost any kind of application, including web development, data analysis, scripting, and more.

Applications

- **Web Development (Backend):** Using frameworks like Django and Flask.
- **Data Science & Machine Learning:** Libraries like NumPy, Pandas, Scikit-learn.
- **Automation/Scripting:** Automating repetitive tasks (file management, system administration).
- **Software Testing:** Used for writing test scripts.

The Python Interpreter (REPL)

Python's interpreter allows you to execute code interactively, line-by-line. This is often called the **Read-Eval-Print-Loop (REPL)**.

- **Read:** Reads the user input.
- **Eval:** Evaluates the input expression.
- **Print:** Prints the result.
- **Loop:** Returns to the beginning to read the next input.

Example (In a terminal):

```
Python
>>> 2 + 3
5
>>> print("Hello")
Hello
```

1.2 Environment Setup

Installation and IDE

1. **Installation:** Download the official installer from python.org. Ensure the option "Add Python to PATH" is checked during installation.
2. **IDE/Code Editor:** While you can use a simple text editor, an **Integrated Development Environment (IDE)** or advanced code editor is highly recommended for writing larger programs.
 - o **VS Code:** (Popular code editor)
 - o **PyCharm:** (Dedicated Python IDE, good for large projects)

Writing and Executing a .py File

To run a script (a program saved as a file):

1. **Write the code:** Save your code in a file named, for example, `first_script.py`.

Python

```
# first_script.py
print("Executing my first Python script!")
```

2. **Execute from terminal:** Navigate to the file's directory and run the command:

Bash

```
python first_script.py
```

Understanding PVM and Bytecode

When you run a Python file, the following happens:

1. **Compilation:** The Python source code (.py file) is first compiled into an intermediate form called **Bytecode**.
2. **Bytecode File:** This bytecode is often saved as a .pyc file (Python Compiled).
3. **Execution:** The **Python Virtual Machine (PVM)** is the runtime engine that executes the bytecode. The PVM is responsible for managing memory and running the program.

1.3 Syntax Basics and Conventions

Keywords and Identifiers

- **Keywords:** Reserved words that have special meaning to the interpreter. You cannot use them as variable names. (e.g., if, for, while, def, class, return).
- **Identifiers:** Names given to variables, functions, classes, etc.
 - **Rules:** Must start with a letter (A-z) or underscore (_). Cannot be a keyword. Case-sensitive (age is different from Age).

Indentation: The Crucial Rule

In many languages, code blocks (like those following an if or for statement) are defined by curly braces ({}). In Python, code blocks are defined by indentation.

- All statements within a block **must** have the same level of indentation (usually 4 spaces).
- Inconsistent indentation will result in an IndentationError.

Example:

```
Python
# CORRECT indentation
if True:
    print("This runs")
    print("This also runs because it's at the same level")

# INCORRECT indentation (Syntax Error)
if True:
    print("This runs")
    print("This breaks the code block!") # Two spaces instead of four
```

Comments

Comments are non-executable notes used to explain the code.

- **Single-line comments:** Use the # symbol.

Python

```
# This is a comment, the interpreter ignores it.
temperature = 25
```

- **Multi-line comments (Docstrings):** Python uses **triple quotes** (""""...""" or ""...""") primarily for Docstrings (documentation strings, see Module 5), but they can also serve as multi-line comments.

Statements and Expressions

- **Statement:** An instruction that the Python interpreter can execute. It performs an action.
 - **Examples:** Variable assignment, printing output, a loop (for i in range(5)).

Python

```
x = 10      # Assignment statement
print("Hello")  # Print statement
```

- **Expression:** A combination of values, variables, operators, and function calls that the interpreter evaluates to produce a single value.
 - *Examples:* Mathematical calculations, comparisons, function calls.

Python

```
10 + 5    # Evaluates to 15
'a' * 3   # Evaluates to 'aaa'
```

PEP 8: Code Style Guidelines

PEP 8 is the official style guide for writing readable Python code. Adhering to it makes your code consistent and easier for others (and your future self) to understand.

- **Key Rules:**
 - Use 4 spaces per indentation level.
 - Use meaningful variable names (e.g., `user_name` instead of `u_n`).
 - Limit lines to a maximum of 79 characters.
 - Use blank lines to separate functions and classes.

Module 2: Variables, Data Types, and Operators

2.1 Variables and Assignment

A **variable** is a name that refers to a value stored in the computer's memory. The process of creating a variable and linking it to a value is called **assignment**.

Defining Variables (No explicit declaration needed)

In Python, you don't need to specify the variable's type before using it (unlike C++ or Java). You simply assign a value using the single equals sign (=).

Example Program:

Python

```
# Simple variable assignment
age = 30
name = "Alice"
is_student = False
pi_value = 3.14159

# The variables are created and assigned values instantly.
print(f"Name: {name}, Age: {age}")
```

Variable Naming Rules and Conventions (e.g., PEP 8)

- **Rules (Must Follow):**

1. Can only contain letters (a-z, A-Z), digits (0-9), and the underscore (_).
2. Must start with a letter or an underscore. Cannot start with a digit.
3. Variable names are **case-sensitive** (e.g., Age is different from age).
4. Cannot be a reserved Python keyword (e.g., if, for, while, print).

- **Conventions (PEP 8 - Best Practice):**

1. Use **snake_case** (all lowercase, words separated by underscores).
 - *Good:* total_sales, user_name
 - *Bad:* TotalSales, totalsales
2. Choose meaningful, descriptive names.
 - *Good:* price_per_unit
 - *Bad:* ppu (unless commonly understood abbreviation)
3. A single leading underscore (_variable) suggests an internal or private use (a convention, not strictly enforced).

Dynamic Typing (Type is checked at runtime)

Python is **dynamically typed**, meaning the type of a variable is determined by the value it currently holds, and a variable can change its type during program execution.

Example Program:

Python

```
x = 10    # x is an integer (int)
```

```
print(f"Type of x: {type(x)}")  
x = "Hello" # Now x is a string (str)  
print(f"Type of x: {type(x)}")
```

The `id()` and `type()` functions

- **`type(variable)`:** Returns the type of the object the variable refers to.
- **`id(variable)`:** Returns the identity (memory address) of the object. For two variables to be identical (using `is`), they must have the same `id()`.

Example Program:

Python

```
number = 42  
name = "Python"
```

```
print(f"The value is: {number}, its type is: {type(number)}, and its memory ID is: {id(number)}")  
print(f"The value is: {name}, its type is: {type(name)}, and its memory ID is: {id(name)}")
```

```
a = 10  
b = 10  
print(f"ID of a: {id(a)}")  
print(f"ID of b: {id(b)}")  
# Since 10 is an immutable object, a and b often point to the same memory location
```

2.2 Built-in Data Types (Primitives)

Numeric Types: `int`, `float`, `complex`

- **`int (Integer)`:** Whole numbers (positive, negative, or zero) without a decimal point.
- **`float (Floating Point)`:** Numbers with a decimal point.
- **`complex (Complex Number)`:** Numbers with a real and an imaginary part, written as `$a + bj$`.

Example Program:

Python

```
integer_num = 100  
float_num = 100.0 # Even though it's 100, the decimal point makes it a float  
large_num = 9876543210  
pi = 3.14159265  
complex_num = 3 + 4j
```

```
print(f"Type of {integer_num}: {type(integer_num)}")  
print(f"Type of {float_num}: {type(float_num)}")  
print(f"Type of {complex_num}: {type(complex_num)}")
```

Boolean Type: `bool (True, False)`

The Boolean type represents truth values. It has only two possible values: `True` and `False` (note the capitalization). These are crucial for control flow.

Example Program:

```
Python
is_active = True
is_logged_in = False

print(f"Is active: {is_active}, Type: {type(is_active)}")

# Booleans can be treated as numbers (True is 1, False is 0)
result = is_active + is_logged_in # 1 + 0 = 1
print(f"Result of arithmetic: {result}")
```

Type Conversion (Type Casting): int(), float(), str(), etc.

Python provides built-in functions to convert (cast) values from one type to another.

Function	Purpose
int(x)	Converts \$x\$ to an integer.
float(x)	Converts \$x\$ to a floating-point number.
str(x)	Converts \$x\$ to a string.
bool(x)	Converts \$x\$ to a boolean.

Example Program:

```
Python
num_str = "123"
decimal_num = 7.99
is_data = 1

# String to Integer
int_val = int(num_str) # 123
print(f"String '{num_str}' as int: {int_val}")

# Float to Integer (truncates the decimal part)
int_from_float = int(decimal_num) # 7
print(f"Float {decimal_num} as int: {int_from_float}")

# Integer/Float to String
str_val = str(int_val) + " dollars"
print(f"String + int: {str_val}")

# Integer to Boolean (0 is False, any non-zero number is True)
bool_val = bool(is_data) # True
print(f"Integer {is_data} as bool: {bool_val}")
```

2.3 Operators

Operators are special symbols or keywords that perform an operation on one or more values (operands).

Arithmetic Operators

Perform mathematical operations.

Operator	Description	Example	Result
+	Addition	\$5 + 2\$	7
-	Subtraction	\$5 - 2\$	3
*	Multiplication	\$5 * 2\$	10
/	Division (always returns a float)	\$5 / 2\$	2.5
//	Floor Division (discards fractional part)	\$5 // 2\$	2
%	Modulus (remainder of the division)	\$5 \% 2\$	1
**	Exponentiation	\$5 ** 2\$	25

Example Program:

Python

```
a = 10  
b = 3
```

```
print(f"Addition (a + b): {a + b}" )      # 13  
print(f"True Division (a / b): {a / b}" )    # 3.333... (float)  
print(f"Floor Division (a // b): {a // b}" ) # 3 (int)  
print(f"Modulus (a % b): {a % b}" )          # 1 (Remainder)  
print(f"Exponent (a ** b): {a ** b}" )        # 1000 (10*10*10)
```

Comparison Operators

Compare two values and return a **Boolean** (True or False).

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

Example Program:

Python

```
x = 10
y = 12
print(f"x == y: {x == y}") # False
print(f"x != y: {x != y}") # True
print(f"x <= 10: {x <= 10}") # True
```

Assignment Operators

Combine an arithmetic operator with the assignment operator (=) for a shorthand notation.

Operator	Equivalent to
+=	$x = x + y$
-=	$x = x - y$
*=	$x = x * y$
/=	$x = x / y$

Operator	Equivalent to
%=	x = x % y

Example Program:

```
Python
counter = 5
counter += 3 # Same as counter = counter + 3
print(f"After += 3: {counter}") # 8

price = 100
price *= 0.8 # Same as price = price * 0.8 (20% discount)
print(f"After *= 0.8: {price}") # 80.0
```

Logical Operators: and, or, not

Used to combine conditional statements. They also return a **Boolean** value.

Operator	Description
and	Returns True if both operands are true.
or	Returns True if at least one operand is true.
not	Negates the Boolean value (Flips True to False and vice versa).

Example Program:

```
Python
sunny = True
warm = False

# and
can_swim = sunny and warm # True and False -> False
print(f"Can swim (sunny and warm): {can_swim}")

# or
can_go_outside = sunny or warm # True or False -> True
print(f"Can go outside (sunny or warm): {can_go_outside}")

# not
not_warm = not warm # not False -> True
print(f"Is it not warm: {not_warm}")
```

Identity Operators: is, is not (Comparing memory location/object identity)

- **is:** Returns True if two variables point to the **same object** in memory (i.e., their `id()` is the same).
- **is not:** Returns True if two variables do **not** point to the same object.

Note: This is different from `==` which checks if the *values* are equal.

Example Program:

Python

```
list_a = [1, 2, 3]
list_b = [1, 2, 3]
list_c = list_a # c and a now reference the exact same list object

print(f"list_a == list_b: {list_a == list_b}") # True (Values are equal)
print(f"list_a is list_b: {list_a is list_b}") # False (Different objects/memory IDs)

print(f"list_a == list_c: {list_a == list_c}") # True
print(f"list_a is list_c: {list_a is list_c}") # True (Same object/memory ID)
```

Membership Operators: in, not in (Checking presence in a sequence)

- **in:** Returns True if a value is present in a sequence (like a string, list, or tuple).
- **not in:** Returns True if a value is **not** present in a sequence.

Example Program:

Python

```
my_text = "Hello Python"
vowels = ['a', 'e', 'i', 'o', 'u']

print(f"'P' in my_text: {'P' in my_text}")      # True
print(f"'z' not in my_text: {'z' not in my_text}") # True
print(f"'e' in vowels: {'e' in vowels}")        # True
```

Operator Precedence and Associativity

Precedence determines the order in which operators are evaluated (e.g., multiplication before addition, like in `$2 + 3 * 4$`). **Associativity** defines the order of evaluation for operators with the same precedence (usually left-to-right, except for exponentiation `**`, which is right-to-left).

Order of Precedence (Highest to Lowest, simplified):

1. **Parentheses ()**
2. **Exponentiation ****
3. **Unary operators (+, -, ~)**
4. **Multiplication, Division, Modulus, Floor Division (*, /, //, %)**
5. **Addition and Subtraction (+, -)**
6. **Comparison operators (==, !=, <, >, etc.)**
7. **Identity/Membership operators (is, in)**

8. Logical operators (not, and, or)

Example Program:

Python

```
# Standard Math: 10 + (2 * 5) = 20
result_1 = 10 + 2 * 5
print(f"Result 1 (10 + 2 * 5): {result_1}") # 20 (Multiplication before Addition)

# Use Parentheses to override: (10 + 2) * 5 = 60
result_2 = (10 + 2) * 5
print(f"Result 2 ((10 + 2) * 5): {result_2}") # 60
```

2.4 Input and Output

Using the print() function

The print() function displays output to the console.

Formatting Output, sep, end

- **sep (separator):** Specifies how to separate the arguments passed to print(). Default is a single space.
- **end:** Specifies what to print at the end of the output. Default is a newline character (\n).

Example Program:

Python

```
print("Hello", "World", "!", sep="---")
# Output: Hello---World---!

print("The first line.", end=" ")
print("The second line is concatenated.")
# Output: The first line. The second line is concatenated.

print("List of items:")
for item in ['A', 'B', 'C']:
    print(item, end=", ")
# Output: List of items: A, B, C,
```

Using the input() function to get user input

The input() function pauses the program and waits for the user to type something and press Enter. **It always returns the input as a string.**

Example Program:

Python

```
user_name = input("Please enter your name: ")
age_str = input("Please enter your age: ")

# Input returns a string, so we must convert it for calculations
```

```
user_age = int(age_str)
print(f"Hello, {user_name}! You will be {user_age + 1} next year.")
```

String Formatting: % operator, .format() method, f-strings

These methods allow you to embed variable values into a string elegantly.

1. % operator (Old style, C-like):

Python

```
item = "Laptop"
price = 999.50
print("The price of the %s is $%.2f." % (item, price))
```

2. .format() method (Newer style):

Python

```
item = "Keyboard"
price = 75.00
print("The price of the {} is ${:.2f}.".format(item, price))

# You can use index or names:
print("Name: {1}, ID: {0}".format(101, "Dave"))
```

3. f-strings (Formatted String Literals - Best practice in modern Python):

Prepend the string with an f and embed expressions directly inside curly braces {}.

Example Program (f-strings):

Python

```
product = "Monitor"
discount = 0.15
original_price = 450.00
final_price = original_price * (1 - discount)

print(f"Product: {product}")
# Formatting for currency ($ and 2 decimal places)
print(f"Original Price: ${original_price:.2f}")
# Can perform calculations inside {}
print(f"Discounted Price: ${final_price:.2f}")
# Conditional formatting
print(f"Status: {'Expensive' if final_price > 300 else 'Affordable'}")
```

Module 3: Control Flow Statements

Control flow statements are used to control the flow of execution of the program. They fall into two main categories: **Conditional** (Decision Making) and **Looping** (Iteration).

3.1 Conditional Statements (Decision Making)

Conditional statements execute a block of code only if a specified condition is True.

The if statement

The simplest decision structure. The code block beneath if is executed only if the condition is true.

Real-Time Example: Checking for a positive bank balance.

Python

```
balance = 5000.00
withdrawal_amount = 6000.00

if balance >= withdrawal_amount:
    balance -= withdrawal_amount
    print(f"Withdrawal successful. New balance: ${balance:.2f}")

print("Transaction completed.")
# Since 6000 > 5000, the 'if' block is skipped.
```

The if-else statement

Provides two possible paths of execution. If the if condition is True, the if block executes; otherwise, the else block executes.

Real-Time Example: User authentication check (login).

Python

```
stored_password = "password123"
user_input = input("Enter password: ")

if user_input == stored_password:
    print("✓ Login successful. Welcome back!")
    is_authenticated = True
else:
    print("✗ Login failed. Incorrect password.")
    is_authenticated = False
```

The if-elif-else chain

Used when you have multiple conditions to check sequentially. The conditions are evaluated from top to bottom. As soon as a condition is True, its corresponding block is executed, and the entire chain is exited. The else block (optional) executes if none of the preceding conditions are true.

Real-Time Example: Grading system calculation.

Python

score = 85

```
if score >= 90:  
    grade = 'A'  
elif score >= 80: # This runs only if score < 90  
    grade = 'B'  
elif score >= 70: # This runs only if score < 80  
    grade = 'C'  
else:  
    grade = 'F'  
  
print(f"Student score: {score}, Grade: {grade}") # Output: Grade: B
```

Nested if statements

Placing one if or if-else structure inside another if or else block. This allows for complex, multi-level decision-making.

Real-Time Example: E-commerce shipping qualification.

Python

is_prime_member = True

order_total = 75.50

```
if is_prime_member:  
    if order_total >= 50:  
        shipping_cost = 0.00  
        print("🎉 Prime member with order over $50: Free shipping!")  
    else:  
        shipping_cost = 5.00  
        print("Prime member, but order under $50. Shipping cost: $5.00")  
else:  
    # Non-Prime Member logic  
    shipping_cost = 10.00  
    print("Not a Prime member. Shipping cost: $10.00")
```

3.2 Looping Statements (Iteration)

Looping statements allow a block of code to be executed repeatedly.

The while loop

Executes a block of code **as long as** its condition remains True. You must ensure the condition eventually becomes False to avoid an infinite loop.

Real-Time Example: Countdown timer.

Python

timer = 5

```

print("Starting in...")
while timer > 0:
    print(timer)
    timer -= 1 # Crucial step to change the condition
    # In a real application, you'd use a time delay function here
print("GO!")

```

The for loop (Iterating over sequences/iterables)

Used for iterating over a sequence (like a list, tuple, dictionary, set, or string) or other iterable objects. It executes the code block once for each item in the sequence.

Real-Time Example: Processing items in a shopping cart.

Python

```

shopping_cart = ["milk", "bread", "eggs", "juice"]

print("Items in your cart:")
for item in shopping_cart:
    # 'item' takes the value of each element sequentially
    print(f"- {item.capitalize()}")
    # Real-time use: Calculate tax, update inventory, etc.

```

The range() function

Generates a sequence of numbers, often used to control for loops that need to run a specific number of times.

- `range(stop)`: Generates numbers from \$0\$ up to (but not including) stop.
- `range(start, stop)`: Generates numbers from start up to (but not including) stop.
- `range(start, stop, step)`: Generates numbers with the specified step size.

Real-Time Example: Displaying monthly sales data.

Python

```

# Iterating a fixed number of times (12 months)
monthly_sales = [1200, 1500, 1100, 1800, 2000, 2500, 2200, 1900, 1700, 1600, 2100, 2400]

```

```

print("Quarterly Sales Report:")
# i goes from 0 to 11 (12 months)
for i in range(len(monthly_sales)):
    # Using floor division to determine the quarter (0, 0, 0, 1, 1, 1, ...)
    quarter = (i // 3) + 1

    print(f"Month {i+1} (Q{quarter}): ${monthly_sales[i]}")

```

```

# Example using step: printing only odd numbers
print("\nOdd numbers from 1 to 10:")
for num in range(1, 11, 2):
    print(num, end=" ")

```

Nested loops

A loop inside another loop. The inner loop executes completely for every single iteration of the outer loop.

Real-Time Example: Generating a multiplication table or processing a 2D matrix (like seating charts).

Python

```
# Generating a small multiplication table
size = 3

print("\n--- Multiplication Table (3x3) ---")
for i in range(1, size + 1): # Outer loop (rows)
    for j in range(1, size + 1): # Inner loop (columns)
        # print(f"{i*j:4}", end="") # :4 is for padding
        print(i * j, end="\t") # \t is a tab space
    print() # Prints a newline after the inner loop finishes (end of row)
```

3.3 Loop Control Statements

These statements alter the normal execution flow of a loop.

break (Exiting the loop)

Immediately terminates the loop (both for and while) and transfers execution to the statement immediately following the loop.

Real-Time Example: Searching for a specific product in a large inventory.

Python

```
inventory = ["TV", "Phone", "Laptop", "Mouse", "Keyboard"]
target = "Laptop"

print("\nSearching inventory...")
for product in inventory:
    if product == target:
        print(f"⌚ Found the {target}! Stopping search.")
        break # Exit the loop as soon as the item is found
    print(f"Checking {product}...")

# Code execution continues here, outside the loop.
print("Search complete.")
```

continue (Skipping the current iteration)

Stops the current iteration of the loop and moves the execution to the beginning of the next iteration.

Real-Time Example: Skipping weekend days when processing weekly tasks.

Python

```
working_days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

print("\nProcessing tasks for the week:")
for day in working_days:
    if day == "Sat" or day == "Sun":
        print(f"Skipping {day}. Weekend!")
    continue # Skips the rest of the code in the loop body for Sat/Sun
```

```
# This code only runs for Mon-Fri
print(f"Working on tasks for {day}...")
```

pass (Null operation statement)

The pass statement is a null operation; nothing happens when it executes. It is often used as a placeholder where a statement is syntactically required but you don't want any code to run.

Real-Time Example: Defining an incomplete function or condition block for later implementation.

Python

```
user_role = "admin"

if user_role == "admin":
    # Developer needs to add admin-specific code later
    pass
elif user_role == "user":
    print("Standard user functions loaded.")
else:
    # Handle unknown roles
    pass
```

else clause with loops (Executes if the loop finishes without a break)

The else block associated with a for loop executes *only if* the loop completes its full cycle (i.e., it was **not** terminated early by a break statement). This is most common in search operations.

Real-Time Example: Checking if an item is *not* found after checking the entire sequence.

Python

```
database = ["A101", "B202", "C303"]
search_id = "D404"

print("\nSearching database for ID D404...")
for item in database:
    if item == search_id:
        print(f"ID {search_id} found in database!")
        break
else:
    # This executes because the 'break' statement was NOT hit
    print(f"ID {search_id} was NOT found after checking all records.")
```

Module 4: Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions allow us to break a large program into smaller, manageable, and modular chunks.

4.1 Defining and Calling Functions

Syntax: def keyword

Functions are defined using the `def` keyword, followed by the function name, a set of parentheses `()`, and a colon `:`. The function body is indented.

Real-Time Example: Creating a reusable function for temperature conversion.

Python

```
def celsius_to_fahrenheit(celsius):
    """
    Converts a temperature from Celsius to Fahrenheit.
    Formula: F = C * (9/5) + 32
    """
    fahrenheit = celsius * (9/5) + 32
    return fahrenheit
```

Function Call and Execution Flow

To execute the code inside a function, you must **call** it by using its name followed by parentheses and any required arguments. When a function is called, the program's execution jumps to the function's body, executes the statements, and then returns to where it was called from.

Example Program:

```
Python
# Function Call
temp_c = 25
temp_f = celsius_to_fahrenheit(temp_c) # Execution jumps to the function

print(f"{temp_c}°C is equal to {temp_f:.2f}°F")
# Output: 25°C is equal to 77.00°F
```

Docstrings ("""...""")

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It's used to explain what the function does and how to use it. They are accessed using `help(function_name)` or `function_name.__doc__`.

Example: (See the celsius_to_fahrenheit function above for the docstring example)

```
Python
help(celsius_to_fahrenheit)
# Output will display the docstring content
```

The return statement

The return statement is used to exit a function and pass an object (value) back to the caller.

1. A function can return any type of object (data type, list, even another function).
2. If the return statement is omitted, the function implicitly returns the special value **None**.
3. Once return is executed, the function immediately terminates.

Real-Time Example: Returning multiple values (e.g., calculation and status).

Python

```
def calculate_discount(price, discount_rate):
    """Calculates the final price after applying a discount."""
    if discount_rate >= 1.0:
        return price, "Error: Discount must be less than 100%"

    final_price = price * (1 - discount_rate)
    # Returns a tuple containing two values
    return final_price, "Success"

# Unpacking the returned tuple
final_price, status = calculate_discount(price=200, discount_rate=0.25)
print(f"Status: {status}, Final Price: ${final_price:.2f}")

# Example of implicit None return
def log_message(msg):
    print(f"LOG: {msg}")
    # No return statement

result = log_message("App started")
print(f"Return value of log_message: {result}") # Output: Return value of log_message: None
```

4.2 Arguments and Parameters

Parameters are the names defined in the function definition. **Arguments** are the actual values passed to the function when it is called.

Positional Arguments

The arguments are matched to the parameters based on their position/order.

Example Program:

```
Python
def check_credentials(username, password):
    # username is matched to the 1st argument, password to the 2nd
    print(f"Checking credentials for: {username}")
```

```
# ... logic ...

check_credentials("admin_user", "secure_pass") # Positional matching
```

Keyword Arguments

Arguments are matched to parameters using the parameter name, which allows them to be passed out of order.

Example Program:

Python

```
def process_order(item_name, quantity, customer_id):
    print(f"Order for {item_name} ({quantity}) placed by customer ID {customer_id}")

# Arguments are passed using their keyword, order does not matter
process_order(quantity=2, customer_id="C456", item_name="T-shirt")
```

Default Arguments

Parameters can be given a default value in the function definition. If the caller does not provide an argument for that parameter, the default value is used. Default arguments must be defined **after** any non-default arguments.

Real-Time Example: Logging function with an optional severity level.

Python

```
def log_event(message, level="INFO"): # 'level' has a default value
    """Logs an application event with an optional severity level."""
    print(f"[{level.upper()}]: {message}")

log_event("User logged in successfully")      # Uses default level="INFO"
log_event("Database connection failed", "ERROR") # Overrides default
```

Variable-Length Arguments: *args (Non-Keyword) and **kwargs (Keyword)

These allow a function to accept an arbitrary, unknown number of arguments.

- ***args (Non-Keyword Arguments):** Collects a variable number of positional arguments into a **tuple**.
- ****kwargs (Keyword Arguments):** Collects a variable number of keyword arguments into a **dictionary**.

Real-Time Example: Function to calculate the sum of any number of values (*args) and configure settings (**kwargs).

Python

```
# *args example
def calculate_total_sum(*numbers):
    """Calculates the sum of all passed numbers."""
    # 'numbers' is a tuple of all positional arguments
    total = sum(numbers)
    return total

print(f"Total sales: {calculate_total_sum(10, 20, 30, 45.5)}")
# Output: Total sales: 105.5
```

```

# **kwargs example
def configure_device(device_name, **settings):
    """Applies configuration settings to a device."""
    print(f"Configuring device: {device_name}")
    # 'settings' is a dictionary
    for key, value in settings.items():
        print(f" - {key}: {value}")

configure_device("Router-R1", ip_address="192.168.1.1", port=80, enable_qos=True)

```

4.3 Scope and Lifetime

Scope refers to the region of a program where a variable is accessible. **Lifetime** is the period during which the variable exists in memory.

Local vs. Global variables

- **Local:** Variables defined inside a function. They exist only while the function is executing and cannot be accessed from outside the function.
- **Global:** Variables defined outside all functions. They are accessible throughout the program, including inside functions (read-only access by default).

Example Program:

```

Python
global_counter = 0 # Global variable

def update_counter():
    local_variable = 10 # Local variable, dies when function ends

    # We can read the global variable
    print(f"Inside function, global_counter is: {global_counter}")

    # ERROR: Cannot directly modify a global variable without 'global' keyword
    # global_counter = 5 # This would create a new local variable named global_counter

update_counter()
# print(local_variable) # ERROR: local_variable is not defined outside the function
print(f"Outside function, global_counter is: {global_counter}")

```

LEGB Rule (Local, Enclosing, Global, Built-in)

Python uses this rule to determine the order in which scopes are searched for a variable name:

1. **Local:** Inside the current function.
2. **Enclosing:** Inside enclosing functions (for nested functions).
3. **Global:** At the top level of the module (script).
4. **Built-in:** Names pre-assigned by Python (e.g., print, len, range).

The global and nonlocal keywords

- **global:** Used inside a function to explicitly indicate that a variable being assigned to is the global variable.

- **nonlocal:** Used in nested functions to refer to a variable in the nearest enclosing scope that is *not* the global scope.

Example Program:

Python

```
count = 10 # Global variable

def modify_scopes():
    # 1. Accessing and modifying the Global scope
    global count
    count += 10 # Modifies the global 'count'

    x = 5 # Enclosing scope variable
    def nested_func():
        # 2. Accessing and modifying the Enclosing scope
        nonlocal x
        x += 1

        y = 100 # Local scope variable

        nested_func()
        print(f"Enclosing x after modification: {x}") # x is 6

    modify_scopes()
    print(f"Global count after modification: {count}") # count is 20
```

4.4 Functional Programming Concepts (Core)

Functional programming emphasizes functions as primary, avoiding state and mutable data.

Lambda Functions (Anonymous functions)

A small, single-expression function that does not have a formal name. They are defined using the `lambda` keyword.

- **Syntax:** `lambda arguments: expression`

Real-Time Example: Using a lambda function as a key for sorting data.

Python

```
data = [("apple", 150), ("banana", 100), ("cherry", 200)]
# Sort the list of tuples based on the second element (the weight/number)
# The lambda function takes one element (x) and returns x[1]
sorted_data = sorted(data, key=lambda x: x[1])

print(f"Sorted by weight: {sorted_data}")
# Output: [('banana', 100), ('apple', 150), ('cherry', 200)]
```

Built-in higher-order functions: map(), filter()

These functions take other functions (often lambdas) as arguments.

- **map(func, iterable):** Applies a given function to all items in an input list (or other iterable) and returns a map object (which can be converted to a list).
- **filter(func, iterable):** Creates an iterator that filters out elements from an iterable for which a function returns False.

Real-Time Example: Processing a list of prices.

Python

```
prices = [10.50, 25.00, 5.99, 50.00]
```

```
# 1. map(): Apply 10% tax to all prices
# tax_func takes x and returns x * 1.10
taxed_prices = list(map(lambda p: p * 1.10, prices))
print(f"Taxed Prices: {taxed_prices}")
```

```
# 2. filter(): Select only prices greater than $20
# filter_func takes x and returns True only if x > 20
high_prices = list(filter(lambda p: p > 20, prices))
print(f"High Prices (> $20): {high_prices}")
```

Introduction to reduce() (from functools module)

The reduce() function applies a rolling computation to sequential pairs of values in a list. It returns a single final value.

Real-Time Example: Calculating the product of all numbers in a list.

Python

```
from functools import reduce
```

```
numbers = [1, 2, 3, 4, 5]
```

```
# The lambda accumulates the result.
# accumulator starts with the first item (1), current with the second (2).
# Next iteration: accumulator is 1*2, current is 3, and so on.
product = reduce(lambda accumulator, current: accumulator * current, numbers)

print(f"Product of numbers: {product}") # Output: 120 (1*2*3*4*5)
```

Recursion (Basic concept and examples)

Recursion is a technique where a function calls itself, either directly or indirectly. Every recursive function must have two parts:

1. **Base Case:** The condition that stops the recursion (prevents infinite calls).
2. **Recursive Step:** The part where the function calls itself, usually with a smaller input.

Real-Time Example: Calculating the factorial of a number ($n! = n \times (n-1)!$).

Python

```
def factorial(n):
```

```
    """Calculates n! using recursion."""
```

```
    # Base Case: Stop condition
```

```

if n == 0 or n == 1:
    return 1

# Recursive Step: Function calls itself with smaller input
else:
    # Example: factorial(4) returns 4 * factorial(3)
    return n * factorial(n - 1)

print(f"Factorial of 5: {factorial(5)}") # Output: 120

```

Core Data Structures (Collections)

5.1 Strings (Immutable Sequence)

Strings are sequences of Unicode characters. They are **immutable**, meaning once created, their contents cannot be changed.

Creating Strings

You can create strings using single, double, or triple quotes. Triple quotes are often used for multi-line strings or docstrings.

Creation Method	Example	Real-time Use Case
Single Quotes	name = 'Alice'	Simple, single-line text data.
Double Quotes	message = "Hello, World!"	Preferred for consistency, allows easy use of single quotes inside (e.g., "It's time").
Triple Quotes	html_content = """<html>...</html>"""	Storing multi-line text, like HTML templates, SQL queries, or poems.

Python

```

# Real-time Example: Storing a user-provided comment
user_comment = "This is a great product!" # Double quotes
license_agreement = ""

Please read the terms below.
By using this software, you agree to all conditions.
(c) 2024 TechCorp
''' # Triple quotes for multi-line text
print(f"Comment: {user_comment}")
print(license_agreement)

```

Indexing (Positive and Negative)

You access individual characters in a string using their index (position).

- **Positive Indexing:** Starts from **0** for the first character and goes up to **\$n-1\$** (where **\$n\$** is the string length).
- **Negative Indexing:** Starts from **-1** for the last character and goes down to **-\$n\$**.

Python

```
# Real-time Example: Extracting the first and last initial of a user's ID
user_id = "PYTHON_DEV_2024"
first_char = user_id[0] # P (Positive index 0)
last_char = user_id[-1] # 4 (Negative index -1)

print(f"User ID: {user_id}")
print(f"First character: {first_char}")
print(f"Last character: {last_char}")
```

Slicing **[\$start:\$stop:\$step]\$**

Slicing extracts a substring. The slice includes the character at start but **excludes** the character at stop.

- **start:** Index where the slice begins (inclusive). Default is 0.
- **stop:** Index where the slice ends (exclusive). Default is end of string.
- **step:** The step size to take. Default is 1.

Python

```
# Real-time Example: Parsing a timestamp string "YYYY-MM-DD HH:MM:SS"
timestamp = "2025-12-12 18:00:25"

# Extract the Date part
date_part = timestamp[0:10] # or timestamp[:10] -> '2025-12-12'

# Extract the Time part
time_part = timestamp[11:] # -> '18:00:25'

# Extract the Year (using step of 2 to skip hyphen/sePARATOR)
# This example is slightly contrived but shows the step
year_with_sep = timestamp[0:4] # '2025'
every_other_char = timestamp[::2] # '20-1-1 8025' (Step 2)

print(f"Full Timestamp: {timestamp}")
print(f"Date: {date_part}")
print(f"Time: {time_part}")
```

String Methods

Method	Description	Real-time Example
<code>len(s)</code>	Returns the length of the string (a built-in function).	Validating a password length: if <code>len(password) < 8:</code>

Method	Description	Real-time Example
.lower()	Converts all characters to lowercase.	Normalizing user input for a case-insensitive search: search_key.lower()
.upper()	Converts all characters to uppercase.	Formatting an abbreviation: country_code.upper()
.strip()	Removes leading and trailing whitespace.	Cleaning user form input: username.strip()
.split(sep)	Splits the string into a list of substrings using a delimiter (sep).	Parsing a CSV row: row_list = line.split(',')
.join(iterable)	Joins elements of an iterable (like a list) into a single string using the string as a separator.	Constructing a file path: '\\'.join(['C:', 'Users', 'Doc'])
.find(sub)	Returns the lowest index of substring sub found, or -1 if not found.	Checking if an email contains '@': email.find('@')
.replace(old, new)	Returns a new string with all occurrences of old replaced by new.	Sanitizing text by removing profanity: comment.replace('bad', '****')

Python

```
# Real-time Example: Processing a Log Entry
log_entry = " ERROR: Connection failed, retry 1. "

# 1. Clean up (strip) and convert to lowercase for analysis
cleaned_entry = log_entry.strip().lower()
print(f"1. Cleaned: {cleaned_entry}"") # 'error: connection failed, retry 1.'

# 2. Check for the error keyword (find)
error_index = cleaned_entry.find("error")
print(f"2. Error found at index: {error_index}") # 0

# 3. Split the entry to separate the type and message
parts = cleaned_entry.split(': ')
log_type = parts[0]
log_message = parts[1]
print(f"3. Type: {log_type}, Message: {log_message}")

# 4. Join components for a new standardized format
standard_log = " | ".join([log_type.upper(), "SYSTEM_A", log_message.capitalize()])
print(f"4. Standard Log: {standard_log}") # 'ERROR | SYSTEM_A | Connection failed, retry 1.'
```

String Formatting

Method	Syntax	Description
% operator	"%s %d" % ("text", 10)	Older C-style formatting. %s for string, %d for integer, %f for float.
.format() method	"{} {}".format("text", 10)	Uses curly braces as placeholders. Can use positional or keyword arguments.
f-strings	f"text {var}"	Recommended method (Python 3.6+). Prefix with f and embed variables/expressions directly in curly braces.

Python

```
# Real-time Example: Generating a confirmation message
order_id = "TX-98765"
product_name = "Laptop"
price = 1250.75
tax_rate = 0.08

# Using .format()
msg_format = "Order ID: {}, Product: {}, Total: ${:.2f} (Includes {:.0%} Tax)".format(
    order_id, product_name, price * (1 + tax_rate), tax_rate
)

# Using f-strings (Recommended)
msg_fstring = f"Order ID: **{order_id}**. Product: {product_name}. Total: **${price * (1 + tax_rate):.2f}** (Includes {tax_rate:.0%} Tax)"

print(f"Format method: {msg_format}")
print(f"F-string method: {msg_fstring}")
```

String Immutability

Immutability means you **cannot** change a string in place. Any string method (like `.lower()`, `.replace()`) returns a **new string**; it does not modify the original.

Python

```
original_tag = "python"
modified_tag = original_tag.upper() # A new string is created

print(f"Original: {original_tag}") # 'python'
print(f"Modified: {modified_tag}") # 'PYTHON'

# Attempting to change an item will result in a TypeError
# original_tag[0] = 'P' # Uncommenting this line causes: TypeError: 'str' object does not support item assignment
```

5.2 Lists (Mutable Sequence)

Lists are ordered sequences of items, and they are **mutable**, meaning their contents can be changed after creation. Items in a list do not have to be of the same type.

Creating and Accessing Lists

Lists are created using square brackets []. Accessing uses indexing, just like strings.

Python

```
# Real-time Example: Tracking user scores in a game
user_scores = [150, 200, 180, 250, 150]
user_names = ["Priya", "Amit", "Chris", "Diya"]

# Accessing
print(f"Highest score (first element): {user_scores[0]}")
print(f"The second user: {user_names[1]}") # Amit
```

List Indexing and Slicing

Identical to strings, lists support positive and negative indexing, and slicing [start:stop:step].

Python

```
# Extracting a leaderboard's top 3
leaderboard = user_scores[0:3] # [150, 200, 180]
last_two = user_scores[-2:] # [250, 150]
print(f"Leaderboard top 3 scores: {leaderboard}")
```

List Methods

These methods *modify* the list in place (mutability) or return information about the list.

Method	Description	Real-time Use Case
.append(item)	Adds a single item to the end of the list.	Adding a new task to a to-do list.
.insert(index, item)	Inserts an item at a specified index.	Inserting a high-priority item at the beginning of a queue.
.extend(iterable)	Appends all elements from an iterable (like another list) to the end.	Merging results from multiple database queries.
.remove(item)	Removes the first occurrence of a specific item by value. Raises ValueError if not found.	Removing a specific element from a list of user permissions.
.pop(index)	Removes and returns the item at a given index (defaults to the last item).	Dequeueing an item from a list used as a stack or queue.

Method	Description	Real-time Use Case
.sort()	Sorts the list elements in place (ascending by default).	Arranging search results by price or date.
.reverse()	Reverses the order of elements in place.	Displaying most recent log entries first.

Python

```
# Real-time Example: Managing a Shopping Cart
```

```
shopping_cart = ["Apples", "Milk", "Bread"]
```

```
print(f"Initial Cart: {shopping_cart}") # ['Apples', 'Milk', 'Bread']
```

```
# 1. Add a new item (append)
```

```
shopping_cart.append("Eggs")
```

```
print(f"After Append: {shopping_cart}") # ['Apples', 'Milk', 'Bread', 'Eggs']
```

```
# 2. Add an item at a specific position (insert)
```

```
shopping_cart.insert(1, "Bananas") # Insert at index 1
```

```
print(f"After Insert: {shopping_cart}") # ['Apples', 'Bananas', 'Milk', 'Bread', 'Eggs']
```

```
# 3. Remove an item by value (remove)
```

```
shopping_cart.remove("Bread")
```

```
print(f"After Remove: {shopping_cart}") # ['Apples', 'Bananas', 'Milk', 'Eggs']
```

```
# 4. Remove the last item and process it (pop)
```

```
last_item = shopping_cart.pop() # Removes 'Eggs'
```

```
print(f"Removed item: {last_item}, Current Cart: {shopping_cart}")
```

```
# 5. Sort the cart alphabetically (sort)
```

```
shopping_cart.sort()
```

```
print(f"After Sort: {shopping_cart}") # ['Apples', 'Bananas', 'Milk']
```

List Mutability and Copying (Shallow vs. Deep Copy)

Because lists are mutable, special care is needed when copying.

- **Assignment (\$=\$):** Just creates a new reference to the **same** list object. Changing one affects the other.
- **Shallow Copy (e.g., slicing [:] or .copy()):** Creates a **new** list object. If the list contains simple immutable objects (numbers, strings), it works like a deep copy. If it contains **mutable objects** (like other lists), only the *references* to those nested objects are copied. Changing a *nested* mutable object affects both lists.
- **Deep Copy (using copy.deepcopy()):** Creates a new list object and recursively copies all nested objects.

Python

```
import copy
```

```
# Example with simple (immutable) elements
```

```
list_a = [1, 2, 3]
```

```
list_b = list_a[:] # Shallow copy using slicing
```

```
list_b[0] = 99
```

```
print(f"List A: {list_a}, List B: {list_b}") # A: [1, 2, 3], B: [99, 2, 3] (Independent)
```

```

# Example with mutable (nested list) elements
original_list = [10, [20, 30], 40]

# Shallow Copy
shallow_copy = original_list.copy()
shallow_copy[1][0] = 999 # Modify the nested list [20, 30]

print(f"Original (Shallow): {original_list}") # [10, [999, 30], 40] <- MODIFIED!
print(f"Shallow Copy: {shallow_copy}") # [10, [999, 30], 40]

# Deep Copy
deep_copy = copy.deepcopy(original_list)
deep_copy[1][0] = 111 # Modify the nested list

print(f"Original (Deep): {original_list}") # [10, [999, 30], 40] <- *STILL* MODIFIED from SHALLOW copy above
print(f"Deep Copy: {deep_copy}") # [10, [111, 30], 40]
# Note: To see deep copy success, you would run the deep copy code *before* the shallow copy change.

```

List Comprehensions (Basic Syntax)

A concise way to create lists. Syntax: [expression for item in iterable if condition].

Python

```

# Real-time Example: Filtering and transforming data
data = [10, 25, 42, 60, 75, 90]

# 1. Filter: Get only the scores > 50
high_scores = [score for score in data if score > 50] # [60, 75, 90]

# 2. Transform: Convert temperatures from Celsius to Fahrenheit (C * 9/5 + 32)
celsius_temps = [20, 25, 30, 35]
fahrenheit_temps = [temp * 9/5 + 32 for temp in celsius_temps] # [68.0, 77.0, 86.0, 95.0]

print(f"High Scores: {high_scores}")
print(f"Fahrenheit: {fahrenheit_temps}")

```

Using Lists as Stacks and Queues (Brief)

- **Stack (LIFO: Last-In, First-Out):** Use `.append()` to push (add) and `.pop()` to pop (remove from the end).
- **Queue (FIFO: First-In, First-Out):** Use `.append()` to enqueue (add to the end) and `.pop(0)` to dequeue (remove from the beginning). *Note: `pop(0)` is slow for large lists. For a fast queue, use `collections.deque`.*

Python

```

# Stack Example: Managing browser history (Last visited page is the first to go back to)
history = ["Google", "Wikipedia", "Python Docs"]
history.append("StackOverflow") # Push
last_page = history.pop() # Pop -> 'StackOverflow'

# Queue Example: Processing print jobs
print_queue = ["Job A", "Job B", "Job C"]
print_queue.append("Job D") # Enqueue
next_job = print_queue.pop(0) # Dequeue -> 'Job A'

```

5.3 Tuples (Immutable Sequence)

Tuples are ordered sequences of items, similar to lists, but they are **immutable**. They are generally used for heterogeneous data (different types) that belongs together, like coordinate pairs or database records.

Creating and Accessing Tuples

Tuples are created using parentheses (), though they are often optional. Accessing uses indexing and slicing, just like strings and lists.

Python

```
# Real-time Example: Storing a fixed GPS coordinate
gps_coord = (34.0522, -118.2437)
db_record = ("user_101", "active", "2025-01-15")

print(f"Latitude: {gps_coord[0]}")
print(f"Status: {db_record[1]}")
```

Packing and Unpacking Tuples

- **Packing:** Simply creating a tuple from a sequence of values.
- **Unpacking:** Assigning the elements of a tuple to individual variables. This is extremely common for function returns.

Python

```
# Packing (The tuple is created implicitly)
settings = 1024, True, "High"

# Unpacking (Commonly used for multiple assignment or iterating through key-value pairs)
max_res, is_enabled, quality = settings
print(f"Resolution: {max_res}, Enabled: {is_enabled}, Quality: {quality}")

# Real-time Example: Swapping variables (a classic tuple unpacking trick)
a = 10
b = 20
a, b = b, a # Swaps the values!
print(f"After swap, a: {a}, b: {b}") # a: 20, b: 10
```

Tuple Immutability

Tuples cannot be modified after creation. They are faster than lists and can be used as keys in dictionaries (unlike lists).

Python

```
status_code = (200, "OK")

# Attempting to change an item will result in a TypeError
# status_code[0] = 404 # TypeError: 'tuple' object does not support item assignment
```

Single-item tuple

A tuple with a single item must include a trailing comma (,) to distinguish it from a simple expression enclosed in parentheses.

Python

```
single_item_tuple = (10,) # This is a tuple
```

```

not_a_tuple = (10)    # This is just an integer (10)
print(f"Type of (10,: {type(single_item_tuple)}") # <class 'tuple'>
print(f"Type of (10): {type(not_a_tuple)}")    # <class 'int'>

```

5.4 Dictionaries (Mutable Mapping)

Dictionaries are unordered collections of **Key-Value** pairs. They are **mutable**. They are used to map one item (the key) to another (the value), providing efficient lookup by key.

Creating and Accessing Dictionaries

Dictionaries are created using curly braces {} with colon-separated key-value pairs: {key1: value1, key2: value2}.

- **Accessing:** Use square brackets [key]. If the key is not found, it raises a `KeyError`.

Python

```
# Real-time Example: Storing configuration settings
```

```
config = {
    "host": "localhost",
    "port": 8080,
    "max_connections": 50,
    "debug_mode": True
}
```

```
# Accessing values
```

```
server_port = config["port"]
print(f"Server Port: {server_port}") # 8080
```

```
# Adding/Updating a key-value pair
```

```
config["debug_mode"] = False
config["timeout"] = 30
print(f"New Config: {config}")
```

Keys and Values

- **Keys:** Must be **immutable** and **unique**. Common key types are strings, numbers, and tuples. Lists and dictionaries cannot be keys.
- **Values:** Can be any data type (mutable or immutable).

Dictionary Methods

Method	Description	Real-time Use Case
<code>.keys()</code>	Returns a view object of all keys.	Iterating over all available configuration options.

Method	Description	Real-time Use Case
<code>.values()</code>	Returns a view object of all values.	Calculating the sum of all item prices.
<code>.items()</code>	Returns a view object of (key, value) tuples.	Iterating over key-value pairs for display or processing.
<code>.get(key, default)</code>	Returns the value for key if it exists, otherwise returns default (defaults to None). Recommended for safe access.	Safely accessing a setting that might not be in the config file.
<code>.pop(key)</code>	Removes the key and returns its value. Raises KeyError if key is not found.	Extracting and processing a task from a queue (using task ID as key).
<code>.update(other_dict)</code>	Merges another dictionary or key-value pairs into the current dictionary. Overwrites keys if they already exist.	Applying global default settings over user-specific settings.

Python

```
# Real-time Example: Processing user profile data
user_profile = {
    "username": "coder_xyz",
    "email": "c@example.com",
    "joined": "2024-10-01",
    "premium": False
}

# 1. Safe access (get)
status = user_profile.get("status", "No status provided")
print(f"Status: {status}") # No status provided

# 2. Get keys and iterate
print("User Fields:", list(user_profile.keys()))

# 3. Get key-value pairs (items)
print("Profile:")
for key, value in user_profile.items():
    print(f"- {key.capitalize()}: {value}")

# 4. Update the profile (update)
new_data = {"premium": True, "last_login": "2025-12-12"}
user_profile.update(new_data)
print(f"After Update: {user_profile}")
```

Dictionary Comprehensions (Basic Syntax)

A concise way to create dictionaries. Syntax: `{key_expr: value_expr for item in iterable if condition}`.

Python

```
# Real-time Example: Creating a dictionary from two lists
product_ids = ["P101", "P102", "P103"]
product_names = ["Monitor", "Keyboard", "Mouse"]

# Create ID -> Name mapping
product_map = {
    id: name
    for id, name in zip(product_ids, product_names)
}
print(f"Product Map: {product_map}") # {'P101': 'Monitor', 'P102': 'Keyboard', 'P103': 'Mouse'}
```

```
# Example 2: Inverting a dictionary (if values are unique and immutable)
inverted_map = {
    v: k for k, v in product_map.items()
}
print(f"Inverted Map: {inverted_map}") # {'Monitor': 'P101', 'Keyboard': 'P102', 'Mouse': 'P103'}
```

5.5 Sets (Mutable, Unordered Collection of Unique Elements)

Sets are unordered collections of unique, immutable elements. They are highly optimized for membership testing and performing mathematical set operations.

Creating Sets (Differences from dictionary creation)

Sets are created using curly braces {} or the set() constructor.

Crucial Difference: An **empty dictionary** is created with {}, while an **empty set** is created with set().

Python

```
# Real-time Example: Tracking unique visitors to a website
session_ids = {101, 102, 103, 101, 104} # Duplicate 101 is automatically removed
print(f"Unique Sessions: {session_ids}") # {101, 102, 103, 104} (Order is not guaranteed)

# Creating an empty set
active_users = set()
```

Set Operations

Sets support mathematical set operations.

Operator	Method	Description	Real-time Use Case
`	`	.union()	All unique elements in both sets.
&	.intersection()	Elements common to both sets.	Finding users who bought <i>both</i> Product A and Product B.

Operator	Method	Description	Real-time Use Case
-	.difference()	Elements in the first set but not the second.	Finding users who logged in on Day 1 <i>but not</i> Day 2.
^	.symmetric_difference()	Elements in either set, but not in both.	Finding users who made a purchase <i>or</i> returned an item, but <i>not</i> both.

Python

```
# Real-time Example: Analyzing customer product purchases
product_a_buyers = {"user_A", "user_B", "user_C"}
product_b_buyers = {"user_B", "user_D", "user_E"}

# 1. Union: All customers who bought at least one product
all_buyers = product_a_buyers | product_b_buyers
print(f"All Buyers (Union): {all_buyers}") # {'user_A', 'user_B', 'user_C', 'user_D', 'user_E'}

# 2. Intersection: Customers who bought both products
both_buyers = product_a_buyers & product_b_buyers
print(f"Both Buyers (Intersection): {both_buyers}") # {'user_B'}

# 3. Difference: Buyers of A but not B
a_only_buyers = product_a_buyers - product_b_buyers
print(f"A Only (Difference): {a_only_buyers}") # {'user_A', 'user_C'}
```

Set Methods

Method	Description	Real-time Use Case
.add(item)	Adds a single item to the set.	Adding a new session ID to a set of active sessions.
.remove(item)	Removes an item. Raises KeyError if the item is not present.	Removing a known processed ID.
.discard(item)	Removes an item if present. Does not raise an error if the item is not present.	Safely removing a temporary file name that may or may not exist in a list of files to clean up.

Python

```
# Real-time Example: Managing temporary file permissions
temp_files_to_clean = set()

# Add new files
temp_files_to_clean.add("temp_1.log")
temp_files_to_clean.add("temp_2.bak")

# Process and remove known file (remove)
```

```
temp_files_to_clean.remove("temp_1.log")
# Attempt to remove a file that might not be there (discard - safe)
temp_files_to_clean.discard("temp_99.tmp") # No error raised

print(f"Remaining files to clean: {temp_files_to_clean}") # {'temp_2.bak'}
```

frozenset (Immutable version of set)

A frozenset is an immutable set. Once created, you cannot add or remove elements. This makes it hashable, so it can be used as a **key in a dictionary** or as an **element in another set**.

Python

```
# Real-time Example: Using a fixed set of permissions as a dictionary key
# This can't be done with a regular mutable set.
permissions = frozenset(["read", "write"])

access_levels = {
    permissions: "Administrator",
    frozenset(["read"]): "Guest"
}

print(f"Access level for {list(permissions)}: {access_levels[permissions]}")
```

6. Modules and Packages

6.1 Modules

What is a Module?

A module is essentially a **single Python file** (with the .py extension) that contains definitions, statements, and functions. Modules allow you to logically organize your Python code, making it reusable and easier to maintain.

Real-time Analogy: Think of a module as a specialized **tool kit** (e.g., a math tool kit or a file-handling tool kit). When you need to perform a specific task, you import the relevant tool kit instead of rewriting the tools yourself.

Creating a Module

Creating a module is as simple as saving code in a .py file.

Example: data_processor.py Let's create a module for processing sales data.

Python

```
# --- File: data_processor.py ---
# Global variable for configuration
TAX_RATE = 0.05
```

```

def calculate_total(price, quantity):
    """Calculates the subtotal (price * quantity)."""
    return price * quantity

def apply_tax(subtotal):
    """Applies the global TAX_RATE to the subtotal."""
    return subtotal * (1 + TAX_RATE)

def format_currency(amount):
    """Formats a number as a currency string."""
    return f"${amount:.2f}"

# Initial print statement runs only once when imported
print("Data Processor Module Initialized.")

```

Importing Modules: import and from...import

There are two primary ways to bring the functionalities of a module into your current script:

1. **import module_name:** Imports the entire module. You must use the module name as a prefix to access its contents (module_name.function()).
2. **from module_name import item1, item2:** Imports only specific items (functions, variables, classes) directly into your current namespace. You can use them without the module prefix.

Example: main_script.py

Python

--- File: main_script.py ---

1. Importing the entire module

```

import data_processor
# Accessing using the prefix
subtotal = data_processor.calculate_total(price=100.00, quantity=3)
taxed_total = data_processor.apply_tax(subtotal)
final_output = data_processor.format_currency(taxed_total)

```

```
print(f"\n--- Using 'import data_processor' ---")
```

```
print(f"Tax Rate: {data_processor.TAX_RATE}")
```

```
print(f"Final Total: {final_output}")
```

2. Importing specific items (avoids prefix)

```
from data_processor import calculate_total, TAX_RATE # Renaming TAX_RATE
```

```
print(f"\n--- Using 'from...import' ---")
```

```
# The variable is now named SALES_TAX in this script
```

```
print(f"Sales Tax: {SALES_TAX}")
```

```
# The function can be called directly
```

```
quick_subtotal = calculate_total(50, 2)
```

```
print(f"Quick Subtotal: {quick_subtotal}")
```

Module Search Path (sys.path)

When you use an import statement, the Python interpreter searches for the requested module in a specific order:

- Current Directory:** The directory where the executing script resides.
- PYTHONPATH Directories:** Directories specified in the PYTHONPATH environment variable.
- Standard Library Directories:** Directories containing built-in Python modules (like os, sys, math).
- Site-packages Directory:** The location for third-party modules (like requests, numpy) installed via pip.

This path is stored in the sys.path list.

Real-time Example: Debugging import errors.

Python

```
# --- Example: Viewing the Search Path ---
import sys

print("\n--- Module Search Path (sys.path) ---")
for path in sys.path:
    print(path)

# If your module isn't found, you typically need to:
# 1. Ensure it's in the same directory.
# 2. If it's elsewhere, you can temporarily add its path:
# sys.path.append("/path/to/my/custom/modules")
# import my_module # This will now work
```

The dir() function

The built-in dir() function returns a list of names (variables, functions, classes) defined in a module or currently available in the scope. It is useful for **discovery** and **introspection**.

Python

```
# Using dir() on our imported module
import data_processor

print("\n--- Contents of data_processor module (dir()) ---")
# Prints: ['TAX_RATE', '__builtins__', ..., 'apply_tax', 'calculate_total', 'format_currency']
print(dir(data_processor))

# Using dir() without arguments shows the current scope
# print(dir()) # Shows items imported/defined in main_script.py
```

Using __name__ == '__main__'

Every Python module has a built-in special variable called __name__.

- When the file is executed directly by the interpreter, Python sets __name__ to the string '__main__'.
- When the file is imported as a module into another script, Python sets __name__ to the module's name (e.g., 'data_processor').

This condition is used to define code that should **only run when the script is executed directly**, not when it is imported. This is crucial for adding unit tests or demonstration code to a module without running them every time it's imported.

Example: Enhancing data_processor.py

```

Python
# --- File: data_processor.py (Revisited) ---
# ... (functions remain the same) ...

# Test/Demonstration Code (runs only when this file is executed directly)
if __name__ == '__main__':
    print("\n--- Running module as a standalone script (Demo) ---")

# Simple test case
item_price = 45.99
item_qty = 2

subtotal = calculate_total(item_price, item_qty)
final = apply_tax(subtotal)

print(f"Item Price: {format_currency(item_price)}")
print(f"Subtotal: {format_currency(subtotal)}")
print(f"Total with Tax ({TAX_RATE*100}%%): {format_currency(final)}")

```

6.2 Packages

What is a Package?

A package is a way to organize related modules into a directory hierarchy.

- A **module** is a single file (.py).
- A **package** is a directory containing related modules and a special file called `__init__.py`.

Real-time Analogy: A package is like a **library** (e.g., a "Financial Analysis" library) which contains specialized tool kits (modules) like `calculations.py`, `reporting.py`, and `data_fetching.py`.

Creating a Package (Role of `__init__.py`)

A directory becomes a Python package when it contains a file named `__init__.py`. This file can be empty, but its presence signals to Python that the directory should be treated as a package.

In modern Python (3.3+), a directory can be considered a namespace package even without `__init__.py`, but including it is still standard practice for simple packages.

Example: Creating a reporting Package

```

/my_project
└── main.py
└── reporting/      <-- Package Directory
    ├── __init__.py   <-- Identifies 'reporting' as a package
    ├── report_generator.py <-- Module 1
    └── email_sender.py  <-- Module 2

```

report_generator.py

```

Python
def generate_summary_report(data):
    return f"Summary: Processed {len(data)} records."

```

email_sender.py

Python

```
def send_email(recipient, subject, body):
    return f"Email sent to {recipient} with subject: '{subject[:20]}...'"
```

Importing from Packages

To use modules and functions within a package, you use the dot (.) notation to specify the package and module hierarchy.

Example: main.py

Python

```
# --- File: main.py ---
```

```
# 1. Importing a specific function from a module inside the package
from reporting.report_generator import generate_summary_report
```

```
# 2. Importing the entire module from the package
import reporting.email_sender
```

```
data = [1, 2, 3, 4, 5, 6]
```

```
# Using the function imported directly
summary = generate_summary_report(data)
print(f"Report Output: {summary}")
```

```
# Using the function imported via the module path
email_result = reporting.email_sender.send_email(
    recipient="admin@corp.com",
    subject="Weekly Sales Report",
    body="Attached is the detailed report..."
)
print(f"Email Status: {email_result}")
```

7. Object-Oriented Programming (OOP) - The Core Pillars

7.1 Classes and Objects

Defining a Class (The Blueprint)

A **Class** is a blueprint or a template for creating objects. It defines a set of attributes (data/variables) and methods (functions) that the created objects will possess.

Real-time Example: A Vehicle class.

Creating an Object (The Instance)

An **Object** (or instance) is a concrete entity built from the class blueprint. Each object has its own unique set of attribute values.

Python

```
# 1. Defining the Class
class Vehicle:
    # Class Variable (shared by all instances)
    WHEELS = 4

    # 2. Creating an Object (Instance) - This calls the __init__ method
    def __init__(self, make, model, year):
        # Instance Variables (unique to each instance)
        self.make = make
        self.model = model
        self.year = year
        self.is_running = False # Initial state

    # Instance Method
    def start_engine(self):
        self.is_running = True
        return f"{self.make} {self.model}'s engine started."

    def get_details(self):
        return f"{self.year} {self.make} {self.model} (Wheels: {Vehicle.WHEELS})"

# Creating Objects (Instances)
car1 = Vehicle("Toyota", "Camry", 2022)
car2 = Vehicle("Honda", "Civic", 2024)

# Accessing methods and attributes
print(car1.get_details())
print(car2.start_engine())
```

Instance Variables vs. Class Variables

Variable Type	Definition	Access/Scope	Real-time Use Case
Instance Variable	Defined inside __init__ using self.variable = value.	Unique to each object/instance.	car1.model, car2.year
Class Variable	Defined outside __init__ at the class level.	Shared among all objects of the class. Accessed via ClassName.variable.	Vehicle.WHEELS (All cars have 4 wheels).

Python

```
# Accessing
print(f"Car 1 Model: {car1.model}") # Instance Variable
print(f"Car 2 Wheels: {car2.WHEELS}") # Accessing Class Variable via instance (good)

# Modifying the Class Variable (affects all instances)
Vehicle.WHEELS = 6
print(f"After modification, Car 1 Wheels: {car1.WHEELS}") # 6
```

The self Variable

The `self` variable is a reference to the **instance** of the class currently being operated on. It is the first argument in all instance methods (including `__init__`). It is how an instance method accesses the instance's attributes.

When you call `car1.start_engine()`, Python automatically passes `car1` as the `self` argument to the method.

Constructors (`__init__`) and Destructors (`__del__`)

- **`__init__(self, ...)` (Constructor):** This method is automatically called when a new object is created. Its main purpose is to **initialize** the instance's attributes.
- **`__del__(self)` (Destructor):** This method is called when an object's reference count drops to zero and it is about to be garbage-collected. It is rarely used in modern Python, but can be useful for **cleanup** operations (e.g., closing file handles, database connections).

Python

```
class DatabaseConnection:  
    def __init__(self, host):  
        self.host = host  
        print(f"Connection established to {self.host}.") # Resource allocation  
  
    def __del__(self):  
        # Resource deallocation/cleanup  
        print(f"Connection to {self.host} closed and object destroyed.")  
  
# Constructor called  
db_conn = DatabaseConnection("production_server")  
print("Working with the database...")  
# Destructor is called automatically when the object is no longer referenced  
del db_conn  
# Or when the program exits
```

7.2 Encapsulation

Encapsulation is the principle of bundling data (attributes) and the methods (functions) that operate on that data into a single unit (the class). It also involves **data hiding**, restricting direct access to an object's internal state.

Bundling Data and Methods Together

We achieved this in the `Vehicle` class by having attributes (`make`, `model`) and methods (`start_engine`, `get_details`) together.

Access Specifiers (Public, Protected, Private Conventions)

Python does not have strict access specifiers like C++ or Java. Instead, it uses **naming conventions** to suggest access intent:

1. **Public (No prefix):** Accessible from anywhere. (e.g., `self.make`)
2. **Protected (Single underscore _):** Intended for internal use, especially within the class and its subclasses. Should be treated as internal by convention. (e.g., `self._engine_id`)
3. **Private (Double underscore __):** Triggers **name mangling** (Python changes the name to `_ClassName__variable`) to make it harder to access from outside the class, although technically still possible. This is used for attributes you *really* don't want external code touching. (e.g., `self.__password`)

```

Python
class BankAccount:
    def __init__(self, balance):
        # Public: Accessible directly
        self.account_type = "Savings"
        # Protected: Conventionally internal
        self._minimum_balance = 100
        # Private: Name-mangled (stronger hiding)
        self.__balance = balance # Store sensitive data privately

    def get_balance(self): # Public accessor method
        return f"Current balance: ${self.__balance:.2f}"

    # Object creation
    user_acc = BankAccount(1000.00)

    # Access
    print(f"Type: {user_acc.account_type}") # Accessing Public variable (OK)

    # Attempt to access protected/private variables directly
    print(f"Minimum (Protected): {user_acc._minimum_balance}") # Access possible, but bad practice
    # print(f"Balance (Private): {user_acc.__balance}") # ERROR: AttributeError

    # Accessing private attribute via name mangling (Discouraged!)
    print(f"Balance (Mangled Access): {user_acc._BankAccount__balance}") # Accesses the mangled name

```

Using Getters and Setters (Property Decorators)

To control access to internal attributes (like `__balance`), we use public methods called **Getters** (to read the data) and **Setters** (to modify the data, often with validation logic).

Python's **@property decorator** allows you to define methods that can be accessed like attributes, making the code cleaner while still enforcing control.

```

Python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance # Private

    # Getter: Allows reading the balance as an attribute (user_acc.balance)
    @property
    def balance(self):
        return self.__balance

    # Setter: Allows setting the balance as an attribute (user_acc.balance = 500)
    @balance.setter
    def balance(self, new_amount):
        if new_amount < 0:
            raise ValueError("Balance cannot be negative.")
        self.__balance = new_amount
        print("Balance updated successfully.")

user_acc = BankAccount(1000)

# 1. Using the getter method (accessed like an attribute)
print(f"Initial Balance: {user_acc.balance}")

```

```
# 2. Using the setter method (accessed like an attribute with validation)
user_acc.balance = 500
# user_acc.balance = -100 # This would raise the ValueError
```

7.3 Inheritance

Inheritance is a mechanism that allows a new class (the **subclass** or **derived class**) to inherit properties (attributes and methods) from an existing class (the **superclass** or **base class**). This promotes code reuse.

Single, Multi-Level, and Multiple Inheritance (MRO)

1. **Single Inheritance:** A subclass inherits from only one superclass.
2. **Multi-Level Inheritance:** A class inherits from a subclass, which in turn inherited from another class (e.g., A → B → C).
3. **Multiple Inheritance:** A subclass inherits from two or more superclasses. This can lead to the **Diamond Problem** if methods are duplicated.

Python uses the **Method Resolution Order (MRO)** (specifically the C3 Linearization algorithm) to determine the order in which base classes are searched for methods. You can view the MRO using `ClassName.mro()`.

Python

```
# Single Inheritance Example
class Employee: # Base Class
    def __init__(self, name):
        self.name = name

    def get_info(self):
        return f"Employee: {self.name}"

class Developer(Employee): # Subclass
    def __init__(self, name, language):
        super().__init__(name) # Call parent constructor
        self.language = language

    def code(self):
        return f"{self.name} is coding in {self.language}."

dev1 = Developer("Alice", "Python")
print(dev1.get_info()) # Inherited from Employee
print(dev1.code()) # Specific to Developer
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to tailor inherited behavior.

Python

```
class Manager(Employee):
    def __init__(self, name, team_size):
        super().__init__(name)
        self.team_size = team_size

    # Overriding the get_info method from the Employee class
    def get_info(self):
```

```
# You can call the superclass method using super()
parent_info = super().get_info()
return f"{parent_info} (Manager of {self.team_size} people.)"

mgr1 = Manager("Bob", 5)
print(mgr1.get_info()) # Calls the overridden Manager method
```

Using the super() function

The `super()` function is used to give access to methods and attributes of the parent or sibling class. It is most commonly used in `__init__` to call the parent's constructor and initialize inherited attributes.

7.4 Polymorphism

Polymorphism means "many forms." In OOP, it refers to the ability of different objects to respond to the same method call in their own way.

Method Overloading (Simulated)

Method Overloading means defining multiple methods in the same class with the same name but different numbers or types of arguments.

Python does **not** support traditional method overloading. However, it can be simulated using **default arguments** or variable-length arguments (`*args, **kwargs`).

Python

```
class Calculator:
    # Simulating overloading with default arguments
    def add(self, a, b, c=None):
        if c is None:
            # Handles two arguments
            return a + b
        else:
            # Handles three arguments
            return a + b + c

calc = Calculator()
print(f"2 arguments: {calc.add(5, 10)}")  # 15
print(f"3 arguments: {calc.add(5, 10, 15)}") # 30
```

Method Overriding (Run-time Polymorphism)

As demonstrated in the Inheritance section, method overriding allows a derived class to replace a method's implementation. When the method is called on an object, the decision of which version to execute is made at **run-time** based on the object's actual type.

Python

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def speak(self):
        return "Woof!"
```

```

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Polymorphic list: objects of different classes
animals = [Dog(), Cat()]

# The same method call (speak()) yields different results
for animal in animals:
    print(f"The animal says: {animal.speak()}")

```

Operator Overloading (e.g., using `__add__` to redefine `+`)

Python allows you to redefine how standard operators (like `+`, `-`, `*`) behave when applied to instances of your custom class. This is done by implementing special methods (Dunder methods, like `__add__`).

Python

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Operator Overloading for the '+' operator
    def __add__(self, other):
        # Defines the operation V1 + V2
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Vector(new_x, new_y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(5, 10)
v2 = Vector(3, 2)

# Using the '+' operator calls the __add__ method
v3 = v1 + v2
print(f"Vector V3: {v3}") # Output: Vector(8, 12)

```

7.5 Data Abstraction

Data Abstraction means showing only essential information to the user and hiding the complex implementation details.

- The user of the `BankAccount` class only interacts with `user_acc.balance` and does not need to know the attribute is internally stored as `self.__balance` or how the validation logic in the setter works.

Using the `abc` module (Abstract Base Classes)

Python achieves formal abstraction using the built-in `abc` (Abstract Base Classes) module. This allows you to define a blueprint class that **must** have certain methods implemented by any concrete subclass.

- An **Abstract Class** cannot be instantiated directly.
- An **Abstract Method** is a method declared in the abstract class but has no implementation (no body).

```

Python
from abc import ABC, abstractmethod

# Abstract Class: Cannot be instantiated
class PaymentProcessor(ABC):

    # Concrete Method (has implementation)
    def process_transaction(self, amount):
        print(f"--- Processing ${amount} transaction ---")
        return self._authorize_payment(amount) # Calls the abstract method

    # Abstract Method: Must be implemented by subclasses
    @abstractmethod
    def _authorize_payment(self, amount):
        pass # No implementation here

# Concrete Subclass: Must implement the abstract method
class CreditCardProcessor(PaymentProcessor):

    def _authorize_payment(self, amount):
        # Real-world: Connects to a payment gateway API
        print("Authorizing via credit card gateway...")
        if amount < 500:
            return "SUCCESS: CC authorized"
        return "FAILURE: CC limit exceeded"

# Example Usage
# processor = PaymentProcessor() # TypeError: Can't instantiate abstract class

cc_proc = CreditCardProcessor()
status = cc_proc.process_transaction(450.00)
print(f"Transaction Status: {status}")

# If CreditCardProcessor failed to implement _authorize_payment,
# Python would raise a TypeError when trying to instantiate it.

```

8. Errors, Exceptions, and File Handling

8.1 Errors and Exceptions

Syntax Errors (Compile-time)

A **Syntax Error** occurs when the code violates the grammatical rules of the Python language. The interpreter cannot parse the code and will stop execution before the program even starts.

Real-time Example: Missing a colon, parenthesis, or having improper indentation.

Python

```

# --- Syntax Error Example ---
# if 5 > 2      <-- SyntaxError: expected ':'
#   print("Five is greater than two")
# print("Hello"  <-- SyntaxError: unclosed parenthesis

```

Logical Errors and Runtime Errors (Exceptions)

- Logical Errors:** The code runs without crashing, but the output is incorrect because the logic is flawed. These are the hardest to debug.
 - Example:* Calculating area as $A = L + W$ instead of $A = L \times W$. The program runs but gives the wrong area.
- Runtime Errors (Exceptions):** The program starts executing but encounters an unexpected situation (like dividing by zero, or trying to access a file that doesn't exist) that forces it to terminate abruptly. These are called **Exceptions**.

Common Built-in Exceptions

Exception	Description	Real-time Example
<code>TypeError</code>	An operation or function is applied to an object of an inappropriate type.	'5' + 10 (Adding string and integer)
<code>ValueError</code>	A function receives an argument of the correct type but an inappropriate value.	<code>int('hello')</code> (Cannot convert string to integer)
<code>ZeroDivisionError</code>	Occurs when dividing a number by zero.	<code>10 / 0</code>
<code>IndexError</code>	A sequence index (string, list, tuple) is out of range.	<code>my_list = [10]; my_list[1]</code>
<code>KeyError</code>	Trying to access a dictionary key that doesn't exist.	<code>my_dict = {'a': 1}; my_dict['b']</code>
<code>FileNotFoundException</code>	Trying to open a file that does not exist.	<code>open('nonexistent.txt', 'r')</code>
<code>AttributeError</code>	Trying to access an attribute or method that an object does not have.	'hello'.append('x') (Strings don't have an append method)

8.2 Exception Handling

Exception handling allows a program to gracefully manage runtime errors instead of crashing.

The `try`, `except`, `else`, and `finally` blocks

- try:** The code that might raise an exception is placed here.
- except:** If an exception occurs in the try block, the code in the except block is executed.
- else:** (Optional) Code that executes ONLY if the try block completes **without** raising an exception.
- finally:** (Optional) Code that executes **always**, regardless of whether an exception occurred or was handled. Often used for cleanup operations.

Python

```
# Real-time Example: Robust User Input Handling
def get_positive_integer(prompt):
    while True:
        try:
            # 1. Code that might fail (convert input to int)
            user_input = input(prompt)
            number = int(user_input)

            # 2. Logical check that might raise an error
            if number <= 0:
                # 3. Raising an Exception (see next topic)
                raise ValueError("Input must be a positive number.")

        except ValueError as e:
            # 4. Handle specific exception
            print(f"Invalid input: {e}. Please try again.")

    except Exception as e:
        # Catching any other unexpected error
        print(f"An unexpected error occurred: {e}")

    else:
        # Executes if NO exception was raised in try block
        print("Input successfully validated.")
        return number

    finally:
        # Executes always, cleanup/logging can go here
        print("--- Input attempt finished ---")

# quantity = get_positive_integer("Enter quantity: ")
# print(f"You entered: {quantity}")
```

Handling Specific Exceptions

It's best practice to handle specific exceptions rather than a general `Exception`. This prevents catching unexpected errors (like `SystemExit`) and gives you context-specific recovery logic.

Python

```
data = {'apple': 1.5, 'banana': 0.5}
price = 0

try:
    item_name = 'orange'
    # This might raise KeyError
    price = data[item_name]

    # This might raise ZeroDivisionError
    ratio = 10 / price

except KeyError:
    print(f"Error: Product '{item_name}' not found in catalog.")
    price = 0.0 # Set default value

except ZeroDivisionError:
    print("Error: Price cannot be zero for division.")
```

```
except Exception as e:  
    # Fallback for any other unexpected error  
    print(f"An unknown error occurred: {e}")  
  
# This will print: Error: Product 'orange' not found in catalog.
```

Raising Exceptions (raise keyword)

You can manually trigger an exception using the `raise` keyword. This is useful for enforcing business rules or signaling errors from within functions.

Python

```
def validate_age(age):  
    if not isinstance(age, int):  
        raise TypeError("Age must be an integer.")  
    if age < 0 or age > 120:  
        raise ValueError("Age must be between 0 and 120.")  
    return age  
  
try:  
    # validate_age("twenty") # Raises TypeError  
    valid_age = validate_age(150) # Raises ValueError  
except ValueError as e:  
    print(f"Validation Error: {e}")
```

User-Defined Exceptions (Basic)

For complex applications, you can create your own custom exceptions by defining a new class that inherits from the base `Exception` class (or one of its specialized subclasses).

Python

```
class InsufficientFundsError(Exception):  
    """Custom exception raised when a withdrawal exceeds the balance."""  
    def __init__(self, requested, available):  
        self.requested = requested  
        self.available = available  
        super().__init__(f"Requested {requested}, but only {available} available.")  
  
    def withdraw(balance, amount):  
        if amount > balance:  
            raise InsufficientFundsError(amount, balance)  
        return balance - amount  
  
    # Real-time usage  
    current_balance = 500  
    try:  
        new_balance = withdraw(current_balance, 700)  
        print(f"New Balance: {new_balance}")  
    except InsufficientFundsError as e:  
        print(f"Transaction failed: {e}")
```

8.3 File Handling

File handling involves reading data from external files and writing data to them.

File Operations: Open, Read, Write, Close

1. **Open:** Use the built-in `open()` function. It returns a file object.
 - o `file_object = open('filename', 'mode')`
2. **Read/Write:** Use methods on the file object (e.g., `read()`, `write()`).
3. **Close:** Use the `.close()` method to release the file handle, freeing system resources. Failing to close files can lead to data corruption or resource exhaustion.

File Modes

Mode	Description	Real-time Use Case
'r'	Read (Default). Error if file doesn't exist.	Reading a configuration file.
'w'	Write . Creates the file if it doesn't exist, overwrites content if it does.	Generating a report from scratch.
'a'	Append . Creates the file if it doesn't exist, adds new data to the end.	Writing new log entries.
'r+'	Read and Write. Pointer at the beginning.	Modifying the start of a file.
'w+'	Write and Read. Overwrites or creates.	Creating a temp file, writing, and then immediately reading back.
't'	Text mode (Default). Handles encoding/decoding.	Reading a standard text file.
'b'	Binary mode. Used for images, executables, etc.	Reading/writing image data or encrypted files.

Reading Methods

Method	Description	Real-time Use Case
<code>read(size)</code>	Reads the entire file content as a single string, or up to size bytes/characters.	Loading a small configuration file into a single variable.
<code>readline()</code>	Reads a single line from the file.	Processing a file line-by-line in a loop.

Method	Description	Real-time Use Case
<code>readlines()</code>	Reads all lines into a list of strings , where each element is one line.	Getting all user names from a delimited text file.

Python

```
# Real-time Example: Reading a Log File
try:
    file = open("application.log", 'r')
    first_line = file.readline()
    all_content = file.read() # Reads remaining content from current position
    print(f"First Line: {first_line.strip()}")
    # print(f"Remaining Content: {all_content}")

except FileNotFoundError:
    print("Error: application.log not found.")

finally:
    if 'file' in locals() and not file.closed:
        file.close()
```

Writing Methods

Method	Description	Real-time Use Case
<code>write(string)</code>	Writes the specified string to the file. Returns the number of characters written.	Writing a single message or piece of data.
<code>writelines(list_of_strings)</code>	Writes a list of strings to the file. Does not automatically add newlines (\n).	Writing a list of processed results or a CSV file.

Python

```
# Real-time Example: Writing processed data to a new CSV file
data_to_write = [
    "ID,Name,Status\n",
    "1,Alice,Active\n",
    "2,Bob,Inactive\n"
]

try:
    file = open("processed_data.csv", 'w')
    file.writelines(data_to_write)
    print("Data written to processed_data.csv")

finally:
    file.close()
```

The `with` statement and Context Managers (Recommended)

The **with statement** is the recommended way to handle files. It automatically handles the setup and teardown of the resource, guaranteeing that the `.close()` method is called even if an exception occurs. This uses Python's **Context Management Protocol**.

Python

```
# Real-time Example: Robust and safe log file append
log_entry = f"[{datetime.datetime.now()}] Server health check completed.\n"

with open("server_activity.log", 'a') as log_file:
    # File is automatically opened and assigned to log_file variable
    log_file.write(log_entry)
    # The file is automatically closed when the 'with' block is exited,
    # even if an exception occurs during the write operation.

print("Log entry recorded and file safely closed.")
```

8.4 Working with Standard Library Modules (Core)

Python's standard library is vast and provides core functionality without installing external packages.

os: Operating System Interaction

The `os` module provides a way of interacting with the operating system, useful for file and directory manipulation.

Function	Description	Real-time Use Case
<code>os.getcwd()</code>	Returns the current working directory.	Logging the location of script execution.
<code>os.mkdir('name')</code>	Creates a new directory.	Setting up project folders after installation.
<code>os.path.join(...)</code>	Smartly joins path components using the correct OS separator (/ or \).	Building platform-independent file paths.
<code>os.path.exists('path')</code>	Checks if a file or directory exists.	Validating configuration file presence before starting a server.
<code>os.listdir('dir')</code>	Returns a list of files and directories in the specified path.	Listing all processed reports in an output folder.

```
Python
import os
```

```
print(f"Current Directory: {os.getcwd()}")
output_dir = "reports_2025"
```

```

# Create directory if it doesn't exist (robust)
if not os.path.exists(output_dir):
    os.mkdir(output_dir)

# Create a platform-independent path
file_path = os.path.join(output_dir, "Q1_Summary.txt")
print(f"Generated Path: {file_path}")

```

sys: System Parameters and Functions

The `sys` module provides access to system-specific parameters and functions.

Function/Variable	Description	Real-time Use Case
<code>sys.argv</code>	A list of command-line arguments passed to the script.	Reading input parameters (like a filename or mode) from the terminal.
<code>sys.path</code>	List of directories Python searches for modules (covered in Module 6).	Debugging import issues.
<code>sys.exit()</code>	Exits the Python interpreter.	Terminating the program on a fatal error.

```

Python
import sys

```

```

# Real-time Example: Reading command line arguments
# To run this: python your_script.py input.txt debug
# sys.argv will be: ['your_script.py', 'input.txt', 'debug']

if len(sys.argv) > 1:
    filename = sys.argv[1]
    print(f"Processing file specified in argument: {filename}")
else:
    print("No input file specified. Using default.")


```

math: Mathematical Functions

The `math` module provides standard mathematical functions and constants.

```

Python
import math

radius = 5.0
# Calculate area of a circle: pi * r^2
area = math.pi * math.pow(radius, 2)
print(f"Area: {area:.2f}")

# Calculate square root
root = math.sqrt(25) # 5.0

```

```

print(f"Square Root: {root}")

# Trigonometric functions
angle_deg = 30
angle_rad = math.radians(angle_deg)
sin_value = math.sin(angle_rad)
print(f"Sin({angle_deg}°): {sin_value:.2f}")

```

datetime: Working with Dates and Times

The `datetime` module supplies classes for manipulating date and time.

Class/Method	Description	Real-time Use Case
<code>datetime.datetime.now()</code>	Returns the current local date and time.	Timestamping log entries or transactions.
<code>datetime.date(Y, M, D)</code>	Represents a date object.	Storing a user's birthday or payment date.
<code>.strftime(format_string)</code>	Formats date/time objects into a specified string format.	Displaying dates to users in MM/DD/YYYY format.
<code>datetime.timedelta</code>	Represents a duration/difference between two dates/times.	Calculating age or remaining subscription days.

```

Python
import datetime

```

```

# 1. Current timestamp
now = datetime.datetime.now()
print(f"Current Time: {now}")

# 2. Formatting a date for a user (MM/DD/YYYY HH:MM)
formatted_date = now.strftime("%m/%d/%Y %H:%M")
print(f"Formatted: {formatted_date}")

# 3. Calculate future date (timedelta)
today = datetime.date.today()
thirty_days = datetime.timedelta(days=30)
renewal_date = today + thirty_days

print(f"Today: {today}")
print(f"Subscription Renewal Date: {renewal_date}")

```