# Module 3: Advanced Python Programming

## Question 1: Printing on Screen

### 1. Introduction to the print() function in Python

The print() function is one of the most fundamental and frequently used built-in functions in Python. Its primary purpose is to output data to the standard output device, which is typically the console or terminal. In Python 3.x, print() is a function, whereas in Python 2.x, it was a statement. This distinction allows for more powerful and flexible usage in modern Python development.

The basic syntax of the function is:
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

- ***objects**: This indicates that the function can take multiple arguments (variable length). You can pass one or more objects to be printed. If multiple objects are passed, they are separated by the sep character.
- **sep**: The separator between the objects. It defaults to a single space ' '.
- **end**: The string appended after the last value. It defaults to a newline character \n.
- **file**: An object with a write(string) method; if not present, it defaults to sys.stdout.
- **flush**: A boolean, specifying whether to forcibly flush the stream.

### 2. Formatting Outputs

While the basic print() function is useful, professional applications require precise control over how data is presented. Python provides three primary methods for string formatting.

A. Old Style Formatting (% Operator)
This method uses the % operator (modulo) to substitute values into a string, similar to printf in C language.

- %s: String
- %d: Integer
- %f: Floating point number

B. The str.format() Method
Introduced in Python 2.6 and 3.0, this method offers more power and flexibility. It uses curly braces {} as placeholders.

- **Positional Arguments:** print("Hello {0}, your age is {1}".format("Alice", 25))
- **Keyword Arguments:** print("Hello {name}, your age is {age}".format(name="Alice", age=25))
- **Alignment:** You can align text left (<), right (>), or center (^) within a specific width.

C. f-strings (Formatted String Literals)
Introduced in Python 3.6, f-strings are the most modern, readable, and efficient way to format strings. By prefixing a string with f or F, you can embed expressions directly inside string literals using curly braces {}.
**Comparison of Formatting Methods:**

| **Feature** | **% Operator** | **.format() method** | **f-strings** |
|---|---|---|---|
| **Syntax** | Verbose | Moderate | Concise |
| **Readability** | Low | Medium | High |
| **Performance** | Slowest | Medium | Fastest |
| **Type Support** | Strict types | Flexible | Flexible |

**Visual Diagram: The Anatomy of an f-string**

```
  f  "The value is { value : .2f }"
  ^      ^         ^       ^
Prefix   Literal    Variable  Format Specifier
```

Understanding print() goes beyond simply displaying text. Mastering sep, end, and modern formatting techniques like f-strings is crucial for creating user-friendly command-line interfaces, generating readable logs, and debugging code effectively.

## Question 2: Reading Data from Keyboard

**1. Using the input() Function**

Interactive programs require a way to accept data from the user. in Python 3, the input() function is used for this purpose. It pauses program execution and waits for the user to type something on the keyboard and press Enter.

- **Syntax:** variable = input([prompt])
- **Prompt:** An optional string argument that is displayed to the user before the input is taken. This is crucial for User Experience (UX) to let the user know what is expected.

**Important Characteristic:** The input() function *always* returns the data as a **string** (str), regardless of whether the user types text, numbers, or special characters. This is a common source of bugs for beginners who expect mathematical operations to work immediately on input numbers.

**2. Type Conversion (Type Casting)**

Since input() returns a string, we often need to convert this data into other types (Integers, Floats, Booleans) to perform operations. This process is called Type Casting.

- **int()**: Converts a string to an integer. If the string contains non-numeric characters (like 'abc' or '12.5'), it raises a ValueError.
- **float()**: Converts a string to a floating-point number.
- **eval()**: A powerful but dangerous function that evaluates a string as a Python expression. For example, eval("2 + 2") returns integer 4.

**Best Practices for Input Handling:**

1. **Always use a prompt:** input("Enter age: ") is better than input().
2. **Handle Exceptions:** When casting input, always wrap the code in a try-except block to prevent the program from crashing if the user enters invalid data.
3. **Validation:** Check if the input meets specific criteria (e.g., age cannot be negative) after conversion.

Security Warning on eval():
While eval() allows for flexible input (automatically detecting int, float, or list), it poses a significant security risk. If a user inputs malicious code (like importing the os module and deleting files), eval() will execute it. Therefore, explicit casting using int() or float()

is preferred in professional environments.

**Data Flow Diagram: Input Processing**

```
[ User Keyboard ]
     |
     v
[ input() Function ] --> Captures raw bytes
     |
     v
[ String Object ]   --> "25" (Text format)
     |
     v
[ Type Casting ]    --> int("25")
     |
     v
[ Integer Object ]  --> 25 (Numeric format)
     |
     v
[ Variable Memory ]
```

Reading data effectively involves not just capturing keystrokes but also validating types, handling errors, and guiding the user via prompts. This forms the basis of dynamic and interactive Python applications.

## Question 3: Opening and Closing Files

### 1. File Handling in Python

File handling is a mechanism that allows Python to interact with the file system to store data permanently. Unlike variables, which lose data when a program terminates, files allow data persistence. The process generally involves three steps:

1. **Open** the file.
2. **Process** the file (Read/Write).
3. **Close** the file.

### 2. The open() Function

The built-in open() function is the gateway to file manipulation.

- **Syntax:** file_object = open(file_name, access_mode, buffering)
- **Returns:** A file object (also called a handle) that provides methods to modify the file.

### 3. File Access Modes

The mode argument specifies what you intend to do with the file. Choosing the wrong mode can result in data loss (e.g., overwriting a file you intended to read).

| Mode | Name | Description | Pointer Position |
|---|---|---|---|
| **'r'** | Read Only | Opens a file for reading. (Default mode). Raises error if file doesn't exist. | Beginning |
| **'w'** | Write Only | Opens a file for writing. Overwrites existing file or creates a new one. | Beginning |

| | | | |
|---|---|---|---|
| **'a'** | Append Only | Opens a file for appending. Data is added to the end. Creates file if not exists. | End |
| **'r+'** | Read & Write | Opens for both reading and writing. | Beginning |
| **'w+'** | Write & Read | Opens for both, but overwrites the file first. | Beginning |
| **'x'** | Exclusive Creation | Creates a new file. Fails if the file already exists. | Beginning |
| **'b'** | Binary Mode | Added to other modes (e.g., 'rb', 'wb') for images/videos. | - |

**4. The Importance of close()**

The close() method flushes any unwritten information in the buffer to the file and closes the file object, freeing up system resources.

- **Syntax:** file_object.close()

**Risks of not closing files:**

- **Data Corruption:** Data sitting in the buffer might not be written to the disk.
- **Resource Leaks:** Operating systems have limits on how many files can be open at once. Not closing files can hit this limit.
- **File Locks:** Other programs may be unable to access the file if your script is holding it open.

The Context Manager (with statement)
In modern Python, it is considered best practice to use the with statement. It

automatically handles closing the file, even if an exception occurs during processing.

```
# Professional approach
with open('data.txt', 'r') as file:
    content = file.read()
# File is automatically closed here
```

Mastering file modes and the open/close lifecycle is critical for data persistence. The distinction between 'w' (overwrite) and 'a' (append) is particularly vital to prevent accidental data loss.

## Question 4: Reading and Writing Files

**1. Writing to Files**

Once a file is opened in a writable mode ('w', 'a', 'r+'), Python provides two primary methods to insert data.

- **write(string)**: Writes a single string to the file. It does *not* automatically add a newline character (\n). You must include it manually if you want line breaks.
  - *Example:* f.write("Hello World\n")
- **writelines(list_of_strings)**: Writes a list of strings to the file. Despite the name, it does not add newlines between the strings in the list.
  - *Example:* f.writelines(["Line 1\n", "Line 2\n"])

**2. Reading from Files**

When a file is opened in a readable mode ('r', 'r+'), we can extract data using three main methods, depending on memory constraints and processing needs.

- **read([size])**:
  - Reads the entire content of the file into a single string variable.
  - If size is specified, it reads that many bytes.
  - *Use case:* Small text files where you need to process the whole text at once.
  - *Risk:* Can crash the program if the file is larger than available RAM (MemoryError).
- **readline()**:
  - Reads a single line from the file, including the newline character at the end.
  - The file cursor moves to the start of the next line after reading.
  - *Use case:* Processing large files line-by-line to save memory.
- **readlines()**:
  - Reads all lines and returns them as a **list of strings**.
  - *Use case:* When you need to iterate over lines or access specific line numbers (e.g., lines[4]).

**3. The File Cursor (Seek and Tell)**

File handling involves a "cursor" that marks the current position in the file.

- **tell()**: Returns the current position of the cursor (in bytes).
- **seek(offset, from_what)**: Moves the cursor to a specific location.
  - 0: Beginning of file.
  - 1: Current position.

- 2: End of file.

**Visualizing File Reading Methods**

| **Method** | **Output Type** | **Memory Usage** | **Best For** |
|---|---|---|---|
| read() | String | High (All contents) | Small config files, regex search |
| readline() | String | Low (One line) | Large logs, data streams |
| readlines() | List | High (All contents) | Sorting lines, filtering lines |

Choosing the right read/write method depends heavily on the size of the data. For massive datasets, iterating over the file object directly (which acts like readline) is the most Pythonic and memory-efficient approach.

## Question 5: Exception Handling

**1. Introduction to Exceptions**

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Unlike syntax errors (which prevent the code from running), exceptions are **runtime errors**. Examples include ZeroDivisionError, FileNotFoundError, and TypeError.

**2. The try, except, else, and finally Blocks**

Python uses a structured block approach to handle these errors gracefully so the program doesn't crash.

- **try block**: Contains the code that might raise an exception. This is the "risky" code.
- **except block**: Contains the code that executes *if* an exception occurs in the try block. You can specify which exception to catch (e.g., except ValueError:).
- **else block** (Optional): Executes only if **no** exceptions were raised in the try block. It is useful for code that should run only on success.
- **finally block** (Optional): Executes **always**, regardless of whether an exception occurred or not. This is critical for cleanup activities like closing files or database connections.

**3. Handling Multiple Exceptions**

A single piece of code might fail for various reasons. For example, opening a file and dividing a number from it could fail because the file is missing (FileNotFoundError) or the number is zero (ZeroDivisionError).

```
try:
    # Risky code
except FileNotFoundError:
    print("File missing")
except ZeroDivisionError:
    print("Cannot divide by zero")
except Exception as e:
    print(f"Unknown error: {e}") # Catch-all for other errors
```

**4. Custom Exceptions**

Python allows developers to define their own exceptions by creating a new class that inherits from the built-in Exception class. This is useful for business logic errors (e.g., InsufficientFundsError in a banking app).

**5. Flow of Execution Diagram**

```
     [ Start try block ]
              |
     Error Occurred?
      /             \
   YES               NO
    |                 |
[ Find matching ]   [ Run else block ]
[ except block  ]     |
    |                 |
[ Run except code ]   |
    |                 |
     \               /
      v             v
   [ Run finally block ]
              |
     [ End of logic ]
```

Robust software must anticipate failure. Exception handling converts "crashes" into "error messages," allowing the user to correct their action or the system to degrade gracefully. The finally block is particularly important for resource management safety.

## Question 6: Class and Object (OOP Concepts)

**1. Object-Oriented Programming (OOP) Paradigm**

Python is a multi-paradigm language, but its core is Object-Oriented. OOP is a programming paradigm based on the concept of "objects," which can contain data (attributes) and code (methods). It aims to model real-world entities.

**2. Class: The Blueprint**

A **Class** is a user-defined prototype (blueprint) for an object. It defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Syntax:**
  ```
  class Car:
      pass
  ```

**3. Object: The Instance**

An **Object** is a unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

- **Instantiation:** my_car = Car()

**4. The __init__ Method and self**

- **__init__**: This is a special method (constructor) that is automatically called when a new instance of a class is created. It is used to initialize the object's attributes.
- **self**: This represents the instance of the class itself. By using self, we can access the attributes and methods of the class in Python. It binds the attributes with the given arguments.

**5. Local vs. Global Variables in OOP Context**

- **Local Variables**: Defined inside a method and can only be accessed within that specific method. They are destroyed when the method execution finishes.
- **Instance Variables (via self)**: Variables that are owned by a specific instance of a class. self.name is accessible anywhere inside the class.
- **Class Variables**: Variables shared by all instances of a class. Defined directly inside the class body but outside any methods.

**Diagram: Class vs Object**

```
   [ CLASS: Dog ]              [ OBJECTS (Instances) ]
   Blueprint                   Real Entities
 +-------------------+         +-------------------+
 | Attributes:       |         | Name: Buddy       |
 |  - Breed          |  -----> | Breed: Pug        |
 |  - Color          |         +-------------------+
 |                   |
 | Methods:          |         +-------------------+
 |  - Bark()         |  -----> | Name: Rex         |
 |  - Eat()          |         | Breed: German Sh.|
 +-------------------+         +-------------------+
```

OOP structures code into modular, reusable pieces. Classes provide the structure, while objects provide the specific data. Understanding the scope of variables (local vs instance vs class) is key to managing state correctly within an application.

## Question 7: Inheritance

### 1. Definition of Inheritance

Inheritance is a fundamental concept in Object-Oriented Programming that allows a class (called the **Child** or **Derived** class) to inherit attributes and methods from another class (called the **Parent** or **Base** class).

**Benefits:**

- **Code Reusability:** Write code once in the parent class and use it in multiple child classes.
- **Extensibility:** Add new features to an existing class without modifying it.
- **Transitivity:** If B inherits from A, and C inherits from B, then C also inherits from A.

### 2. Types of Inheritance in Python

Python supports five distinct types of inheritance:

1. **Single Inheritance:** A child class inherits from only one parent class.
   - *Structure:* Parent A -> Child B
2. **Multiple Inheritance:** A child class inherits from more than one parent class. Python resolves method conflicts using the Method Resolution Order (MRO).
   - *Structure:* Parent A, Parent B -> Child C
3. **Multilevel Inheritance:** A child class inherits from a parent, which in turn inherits from another parent.
   - *Structure:* Grandparent A -> Parent B -> Child C
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
   - *Structure:* Parent A -> Child B, Child C, Child D
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

### 3. The super() Function

The super() function returns a temporary object of the superclass that allows you to call its methods. This is commonly used in the __init__ method to ensure the parent class is initialized correctly before adding the child's own initialization logic.

**Visual Representation of Inheritance Types**

```
Single    Multiple     Multilevel   Hierarchical
 [A]     [A]  [B]          [A]        [A]
  |       \  /         |        /  \
  |        v           v        /   \
 [B]       [C]          [B]       [B]    [C]
                   |
                   v
                  [C]
```

Method Resolution Order (MRO)
In Multiple Inheritance, if two parents have a method with the same name, Python follows a specific order (Depth-First, Left-to-Right) to decide which method to call. This order can be viewed using ClassName.mro().

Inheritance models "is-a" relationships (e.g., a Car is a Vehicle). While powerful, excessive use of complex inheritance (like Hybrid) can make code hard to debug. super() is essential for maintainable cooperative inheritance hierarchies.

## Question 8: Method Overloading and Overriding

**1. Polymorphism in Python**

Polymorphism means "many forms." In programming, it refers to the ability of a function or method to behave differently based on the input object or the context. The two main ways to achieve this in OOP are Method Overloading and Method Overriding.

**2. Method Overloading (Compile-time Polymorphism)**

Method Overloading is the ability to define multiple methods with the *same name* but *different parameters* (different type or number of arguments) within the same class.

- **Python's Unique Stance:** Python **does not support** method overloading in the traditional sense found in Java or C++. If you define two methods with the same name, the second one simply overwrites the first one.
- **How to achieve it in Python:** We simulate overloading using default arguments or variable-length arguments (*args).

```
# Simulation of Overloading
def add(a, b=0, c=0):
    return a + b + c
# Can be called as add(1), add(1, 2), or add(1, 2, 3)
```

**3. Method Overriding (Run-time Polymorphism)**

Method Overriding occurs when a child class provides a specific implementation of a method that is already defined in its parent class. The method in the child class must have the same name and the same parameters as the one in the parent.

- **Purpose:** To change the behavior of an inherited method.
- **Mechanism:** When the method is called on the child object, Python looks in the child class first. If found, it executes that version (overriding the parent).

**Comparison Table**

| Feature | Method Overloading | Method Overriding |
|---|---|---|
| **Location** | Inside the same class | In separate classes (Parent & Child) |
| **Parameters** | Must be different (number/type) | Must be the same |
| **Inheritance** | Not required | Mandatory |
| **Binding** | Static/Compile-time (Conceptually) | Dynamic/Run-time |
| **Python Support** | Not natively supported | Fully supported |

**Diagram: Overriding Flow**

```
[ Parent Class: Animal ]
    Method: speak() -> Returns "Some sound"
        ^
        | (Inherits)
        |
[ Child Class: Dog ]
    Method: speak() -> Returns "Bark"

Code:
dog = Dog()
dog.speak()  --> Python checks Dog class first. Found it!
            Executes "Bark". (Parent method ignored)
```

While Overloading offers convenience in call signatures, Overriding is the heart of dynamic polymorphism, allowing generic code (e.g., animal.speak()) to work with any specific subclass implementation automatically.

## Question 9: SQLite3 and PyMySQL (Database Connectors)

**1. Database Connectivity in Python**

Python acts as a powerful frontend for database applications. To interact with databases, Python uses **Database Connectors** or **Drivers**. These are modules that follow the Python Database API Specification (DB-API), ensuring a consistent way to access different databases.

**2. SQLite3**

- **What is it?** SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.
- **Integration:** The sqlite3 module is built into the Python Standard Library. No installation is needed.
- **Use Case:** Excellent for local storage, mobile apps, prototyping, and small-to-medium web applications. It is file-based (the whole DB is a single file).

**3. PyMySQL**

- **What is it?** A pure-Python MySQL client library.
- **Integration:** It is not built-in; it must be installed via pip (pip install pymysql).
- **Use Case:** Used to connect Python applications to a MySQL server. This is suitable for large-scale, multi-user applications where the database resides on a server.

**4. Steps to Connect and Execute Queries**

Regardless of the connector (SQLite or PyMySQL), the workflow follows the DB-API standard:

1. **Import Module:** import sqlite3 or import pymysql.
2. **Create Connection:** Use connect() to establish a link to the DB.
   - conn = sqlite3.connect('database.db')
3. **Create Cursor:** The cursor object allows you to execute SQL commands and traverse records.
   - cursor = conn.cursor()
4. **Execute Query:** Use the execute() method to run SQL (CREATE, INSERT, SELECT).

    - cursor.execute("SELECT * FROM students")
5. **Commit:** For modifications (INSERT, UPDATE, DELETE), you must call conn.commit() to save changes.
6. **Fetch Data:** For SELECT queries, use fetchone(), fetchall(), etc.
7. **Close:** Close the cursor and connection.

**Architecture Diagram**

```
+-----------------+       +------------------+       +-------------------+
| Python Script   | <----> | Cursor Object    | <----> | Database Engine   |
| (Application)   |       | (Traversal)      |       | (SQLite/MySQL)    |
+-----------------+       +------------------+       +-------------------+
       |                       ^                       ^
       |                       |                       |
       +-----> Connection Object -+--------------------------+
```

Database connectors bridge the gap between Python's logic and data persistence. SQLite is perfect for standalone apps due to its zero-configuration nature, while PyMySQL is essential for server-based enterprise environments.

## Question 10: Search and Match Functions

### 1. The re Module (Regular Expressions)

Python’s re module provides support for regular expressions (Regex), which are sequences of characters that define a search pattern. They are extremely powerful for string manipulation, validation (like email or password checks), and parsing.

Two of the most commonly confused functions in this module are re.match() and re.search().

### 2. re.match() Function

- **Behavior:** This function attempts to match the pattern **only at the beginning** of the string.
- **Logic:** If the pattern is found at the start (index 0), it returns a match object. If the pattern exists but starts at index 1 or later, it returns None.
- **Syntax:** re.match(pattern, string, flags=0)

### 3. re.search() Function

- **Behavior:** This function scans through the **entire string** and returns the **first location** where the regular expression pattern produces a match.
- **Logic:** It checks every position in the string. If a match is found anywhere, it returns a match object. If no match is found after scanning the whole string, it returns None.
- **Syntax:** re.search(pattern, string, flags=0)

### Comparison Example

Let's look at a string: "Python is amazing" and the pattern "is".

- **re.match("is", "Python is amazing")**:
  - Result: None.
  - Reason: The string starts with "P", not "is". match stops immediately.
- **re.search("is", "Python is amazing")**:
  - Result: <re.Match object; span=(7, 9), match='is'>.
  - Reason: It scans past "Python " and finds "is".

### Visualizing the Difference

```
String:  "H  e  l  l  o     W  o  r  l  d"
Indices:  0  1  2  3  4  5  6  7  8  9 10

Pattern: "World"

re.match("World")
  |
  +---> Checks Index 0 ("H"). Match? NO. -> STOP. Return None.

re.search("World")
  |
  +---> Checks Index 0 ("H"). Match? NO.
  +---> Checks Index 1 ("e"). Match? NO.
  ...
  +---> Checks Index 6 ("W"). Match? YES! -> STOP. Return Object.
```

The Match Object

Both functions return a Match Object on success. This object contains methods like:

- group(): Return the string matched.
- start(): Return the starting position.
- end(): Return the ending position.
- span(): Return a tuple (start, end).

The distinction is crucial: use match() when the structure requires the pattern to be a prefix (like checking if a URL starts with 'http'). Use search() for finding a pattern hidden anywhere inside a block of text.