**Core Python Basics**

Topics :

1.Variables,

2.Data Types,

3.Input/Output

4.Operators

5.Control Structures (if-else, loops)

**Control Structures**

* if and else : testing a condition and acting on it.
* for : execute a loop a fixed number of times.
* while : execute a loop while a condition is true.
* repeat : execute an infinite loop (must break out of it to stop)
* break : break the execution of a loop.
* next : skip an interation of a loop.­

**If-Else Statements**

The if-else statement allows you to execute certain blocks of code based on whether a condition is true or false.

-----Syntax----

**if condition:**

**# code to execute if condition is true**

**else:**

**# code to execute if condition is false**

**\\**

**if condition:**

**# Code to execute if condition is True**

**else:**

**# Code to execute if none of the above conditions are True**

**Real time example:**

**age = 18**

**if age >= 18:**

**print("You are an adult.")**

**else:**

**print("You are a minor.")**

**//**

**temperature = 30**

**if temperature > 25:**

**print("It's hot outside.")**

**elif temperature > 15:**

**print("It's warm outside.")**

**else:**

**print("It's cold outside.")**

**---//---**

**Loops**

Loops are used to execute a block of code multiple times.

**a. For Loop**

**Purpose:** Iterate over a sequence (like a list, tuple, string) or a range of numbers.

**Syntax:**

**for variable in sequence:**

**# Code to execute for each item in the sequence**

**fruits = ["apple", "banana", "cherry"]**

**for fruit in fruits:**

**print(fruit)**

**////**

**for i in range(5):**

**print(i)**

**While Loop**

**Purpose:** Execute a block of code as long as a condition is True.

**Syntax:**

python

Copy code

while condition:

# Code to execute as long as condition is True

**Example:**

python

Copy code

count = 0

while count < 5:

print(count)

count += 1

**c. Nested Loops**

Loops can be nested within each other, allowing for more complex iterations.

**Example:**

python

Copy code

for i in range(3):

for j in range(2):

print(f"i: {i}, j: {j}")

**3. Break and Continue**

**Break:** Exits the nearest enclosing loop.

**Continue:** Skips the remaining code inside the nearest enclosing loop and proceeds to the next iteration.

**Example with Break:**

python

Copy code

for i in range(10):

if i == 5:

break

print(i)

**Example with Continue:**

python

Copy code

for i in range(10):

if i % 2 == 0:

continue

print(i)

**6.Functions**

Functions are a fundamental concept in programming that allow you to encapsulate code into reusable blocks. They help in organizing code, avoiding repetition, and improving readability and maintainability. Here’s an overview of how functions work:

### 1. Defining a Function

**Purpose:** Define a block of code that can be executed when called.

**Syntax:**

**def function\_name(parameters):**

**# Code to execute**

**return value # Optional**

**////**

**def greet(name):**

**print(f"Hello, {name}!")**

**greet(“nagu”)**

### Calling a Function

**Purpose:** Execute the code inside a function by invoking it with appropriate arguments.

**Syntax:**

**function\_name(arguments)**

### Parameters and Arguments

* **Parameters** are the variables listed as part of a function definition.
* **Arguments** are the values passed to the function when it is called.

**def add(a, b):**

**return a + b**

**result = add(3, 5) # Here, 3 and 5 are arguments**

**print(result)**

**# Output: 8**

### Keyword Arguments

**Purpose:** Allow you to specify arguments by name, making the function call more readable.

**Syntax:**

python

Copy code

def introduce(name, age):

print(f"My name is {name} and I am {age} years old.")

**Example:**

python

Copy code

introduce(age=25, name="Alice") # Output: My name is Alice and I am 25 years old.

**Variable-Length Arguments**

**Purpose:** Allow a function to accept an arbitrary number of arguments.

* **\*args** for non-keyword arguments
* **\*\*kwargs** for keyword arguments

def print\_numbers(\*args):

for num in args:

print(num)

def print\_info(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_numbers(1, 2, 3, 4)

# Output:

# 1

# 2

# 3

# 4

print\_info(name="Alice", age=30)

# Output:

# name: Alice

# age: 30

7.Recursion Basics

Problems to Solve:

1. Problem 1: Find the Sum of Digits of a Number

Question: Write a Python function that takes an integer as input and returns the sum of its digits.

G Solution:

python

def sum of digits (n):

return sumtint (digit) for digit-in-str(n))

print(sum of digits (123))

Output: 6

2. Problem 2: Find the Greatest of Three Numbers

Question: Write a Python program that takes three numbers and returns the greatest.

Solution:

python

def greatest of three (a, b, c):

return max(a, b, & Technologies

**List comprehensions**

List comprehensions in Python provide a concise way to create lists. They can be used to generate lists based on existing lists, and they offer a more readable and expressive way to construct lists compared to traditional loops.

### Basic Syntax

The basic syntax of a list comprehension is:

python

Copy code

[expression for item in iterable if condition]

 **expression**: The value or transformation to apply to each item.

 **item**: The variable representing each element in the iterable.

 **iterable**: The collection you are looping through (e.g., a list, range, etc.).

 **condition** (optional): A filter to include only items that satisfy the condition.

 **Simple List Comprehension**:

python

Copy code

# Creating a list of squares of numbers from 0 to 9

squares = [x\*\*2 for x in range(10)]

print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]



**Lambda functions**

Lambda functions in Python are small, anonymous functions defined with the `lambda` keyword. They are often used for short, throwaway functions that are not intended to be reused. Lambda functions are particularly useful in conjunction with functions like `map()`, `filter()`, and `reduce()` to perform functional programming tasks.

Syntax of Lambda Functions

The syntax of a lambda function is:

```python

**lambda arguments: expression**

```

- \*\*`arguments`\*\*: The input parameters.

- \*\*`expression`\*\*: The single expression that is evaluated and returned.

### Examples of Lambda Functions

```python

# A lambda function that adds 10 to a given number

add\_ten = lambda x: x + 10

print(add\_ten(5)) # Output: 15

```

### Using Lambda Functions with `map()`

The `map()` function applies a given function to all items in an iterable (like a list) and returns an iterator that yields the results.

\*\*Syntax\*\*:

```python

map(function, iterable)

```

\*\*Example\*\*:

```python

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

squared\_numbers = map(lambda x: x\*\*2, numbers)

print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

```

### Using Lambda Functions with `filter()`

The `filter()` function applies a given function to each item in an iterable and returns an iterator containing only the items for which the function returns `True`.

\*\*Syntax\*\*:

```python

filter(function, iterable)

```

\*\*Example\*\*:

```python

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

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4]

```

### Using Lambda Functions with `reduce()`

The `reduce()` function (from the `functools` module) applies a given function cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value.

\*\*Syntax\*\*:

```python

from functools import reduce

reduce(function, iterable)

```

\*\*Example\*\*:

```python

from functools import reduce

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

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 120

```

### Summary

- \*\*`lambda` functions\*\*: Anonymous functions defined using `lambda`.

- \*\*`map()`\*\*: Applies a function to all items in an iterable.

- \*\*`filter()`\*\*: Filters items in an iterable based on a function's criteria.

- \*\*`reduce()`\*\*: Reduces an iterable to a single value by applying a function cumulatively.

Lambda functions, combined with `map()`, `filter()`, and `reduce()`, allow for concise and expressive functional programming in Python.

**Exception handling**

Exception handling in Python is a mechanism to manage and respond to runtime errors in a controlled way. Instead of letting your program crash when an error occurs, you can use exception handling to catch errors, take corrective actions, and maintain the normal flow of your program.

### Basic Syntax

The basic structure of exception handling in Python uses the `try`, `except`, `else`, and `finally` blocks:

```python

try:

# Code that might raise an exception

risky\_code()

except SomeException as e:

# Code that runs if an exception occurs

handle\_exception(e)

else:

# Code that runs if no exception occurs

success\_code()

finally:

# Code that always runs, regardless of whether an exception occurred or not

cleanup\_code()

```

### Explanation of Blocks

1. \*\*`try` Block\*\*:

- Contains code that might raise an exception.

- If an exception occurs, the code in the `except` block is executed.

2. \*\*`except` Block\*\*:

- Catches and handles exceptions.

- You can specify the type of exception to catch. If you don't specify, it catches all exceptions.

- You can access the exception object through an optional variable (e.g., `as e`).

3. \*\*`else` Block\*\*:

- Optional.

- Executes if no exception is raised in the `try` block.

- Useful for code that should run only if the `try` block succeeds.

4. \*\*`finally` Block\*\*:

- Optional.

- Executes regardless of whether an exception occurred or not.

- Often used for cleanup actions, like closing files or releasing resources.

### Examples

1. \*\*Basic Exception Handling\*\*:

```python

try:

x = 1 / 0

except ZeroDivisionError as e:

print(f"Error: {e}")

```

\*\*Output\*\*:

```

Error: division by zero

```

2. \*\*Handling Multiple Exceptions\*\*:

```python

try:

x = int("not a number")

except ValueError as e:

print(f"ValueError: {e}")

except ZeroDivisionError as e:

print(f"ZeroDivisionError: {e}")

```

\*\*Output\*\*:

```

ValueError: invalid literal for int() with base 10: 'not a number'

```

3. \*\*Using `else` and `finally`\*\*:

```python

try:

result = 10 / 2

except ZeroDivisionError:

print("Cannot divide by zero.")

else:

print(f"Result: {result}")

finally:

print("This will always execute.")

```

\*\*Output\*\*:

```

Result: 5.0

This will always execute.

```

4. \*\*Handling All Exceptions\*\*:

```python

try:

# Code that might raise any exception

risky\_code()

except Exception as e:

print(f"An error occurred: {e}")

```

\*\*Note\*\*: Using a broad `except Exception` is generally discouraged unless you have a specific reason. It's better to catch specific exceptions to avoid masking unexpected bugs.

### Best Practices

- \*\*Catch Specific Exceptions\*\*: Handle known errors explicitly to avoid masking other bugs.

- \*\*Avoid Bare `except:`\*\*: Catching all exceptions (`except:` without specifying an exception type) can hide bugs.

- \*\*Use `finally` for Cleanup\*\*: Use the `finally` block for actions that must occur regardless of exceptions (e.g., closing files or releasing resources).

- \*\*Log Exceptions\*\*: For complex applications, consider logging exceptions instead of just printing them.

Proper exception handling improves the robustness and user experience of your code by allowing it to handle errors gracefully and continue execution or terminate in a controlled manner.