For Loop with Else

The else block in a for loop executes after the loop has iterated through all items in the iterable.

# Example of a for loop with an else clause

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

for number in numbers:

print(number)

else:

print("Finished iterating over the list.")

1

2

3

4

5

Finished iterating over the list.

In this example, the else block runs after the loop finishes iterating through all the numbers.

**While Loop with Else**

The else block in a while loop executes when the condition of the loop becomes false.

# Example of a while loop with an else clause

count = 0

while count < 5:

print(count)

count += 1

else:

print("Count reached 5.")

0

1

2

3

4

Count reached 5.

In this case, the else block runs after the while loop condition (count < 5) is no longer true.

**3. Using Break with Loops and Else**

If the loop is terminated by a break statement, the else block will **not** execute.

**Example with For Loop:**

# Example of a for loop with a break statement and an else clause

for number in range(5):

if number == 3:

print("Breaking out of the loop.")

break

print(number)

else:

print("This will not be printed because the loop was exited with break.")

0

1

2

Breaking out of the loop.

In this example, the loop breaks when number equals 3, so the else block does not execute.

**4. Example with While Loop and Break:**

# Example of a while loop with a break statement and an else clause

count = 0

while count < 5:

if count == 3:

print("Breaking out of the loop.")

break

print(count)

count += 1

else:

print("This will not be printed because the loop was exited with break.")

0

1

2

Breaking out of the loop.  
--------------------------------------------------------------------------------------------------------------------------------------

1. **Defining a Function**

To define a function in Python, you use the def keyword, followed by the function name and parentheses (). Inside the parentheses, you can specify any parameters the function will accept.

def function\_name(parameters):

# Code block

return result # Optional

def greet():

print("Hello, World!")

greet() # Output: Hello, World!

**2. Function Arguments**

Function arguments are the values you pass to a function when you call it. There are several types of arguments in Python:

**1. Positional Arguments**

Positional arguments are the most common type of arguments. They are passed to the function in the order they are defined.

def add(a, b):

return a + b

result = add(5, 3) # 5 is passed as a, and 3 is passed as b

print(result) # Output: 8

**2. Keyword Arguments**

Keyword arguments allow you to pass arguments by explicitly specifying the parameter names. This can improve readability and allows you to pass arguments in any order.

def describe\_pet(pet\_name, animal\_type):

print(f"I have a {animal\_type} named {pet\_name}.")

describe\_pet(animal\_type="hamster", pet\_name="Harry") # Output: I have a hamster named Harry.

**3. Default Arguments**

You can set default values for parameters in the function definition. If no value is provided for those parameters when the function is called, the default value is used.

def greet(name, greeting="Hello"):

return f"{greeting}, {name}!"

print(greet("Alice")) # Output: Hello, Alice!

print(greet("Bob", "Hi")) # Output: Hi, Bob!

**4. Variable-Length Arguments**

Sometimes you may want to pass a variable number of arguments to a function. Python allows you to do this using the \*args and \*\*kwargs syntax.

* \*args is used to send a non-keyworded variable-length argument list.
* \*\*kwargs is used to send a keyworded variable-length argument list.

def add\_numbers(\*args):

return sum(args)

print(add\_numbers(1, 2, 3)) # Output: 6

print(add\_numbers(1, 2, 3, 4, 5)) # Output: 15

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

for key, value in kwargs.items():

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

print\_info(name="Alice", age=25, city="New York")

3. Returning Values from Functions

Functions can return values using the return statement. When a return statement is executed, the function terminates, and the value is sent back to the caller.

def multiply(a, b):

return a \* b

result = multiply(4, 5) # result will be 20

print(result) # Output: 20

**4. Docstrings**

You can add a documentation string (docstring) to your function to describe its purpose. This is helpful for users who want to understand what the function does.

def divide(a, b):

"""Divide a by b and return the result."""

return a / b

print(divide.\_\_doc\_\_) # Output: Divide a by b and return the result.

**Lambda**

In Python, anonymous functions are defined using the lambda keyword. These functions are often referred to as "lambda functions" and are typically used for short, simple operations where defining a full function using def would be unnecessary.

**Characteristics of Lambda Functions:**

1. **Anonymous**: Lambda functions do not have a name (hence the term "anonymous").
2. **Single Expression**: They can only contain a single expression and cannot include statements or annotations.
3. **Return Value**: The result of the expression is implicitly returned.

**Syntax**

The syntax for a lambda function is as follows:

lambda arguments: expression

**Examples**

**1. Basic Lambda Function**

Here’s a simple example of a lambda function that adds two numbers:

add = lambda x, y: x + y

result = add(5, 3)

print(result) # Output: 8

**2. Using Lambda with Built-in Functions**

Lambda functions are often used with functions like map(), filter(), and reduce().

**Using map():**

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

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

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

**Using filter():**

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

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

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

Using reduce(): To use reduce(), you need to import it from the functools module.

from functools import reduce

numbers = [1, 2, 3, 4]

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

print(product) # Output: 24

Recursive Functions

def recursive\_function(parameters):

if base\_case\_condition:

return base\_case\_value # Base case

else:

return recursive\_function(modified\_parameters) # Recursive case

**SAMPLE**

def factorial(n):

if n == 0: # Base case

return 1

else: # Recursive case

return n \* factorial(n - 1)

result = factorial(5)

print(result) # Output: 120

**Advantages:**

* **Simplicity**: Recursive solutions can be simpler and more elegant than their iterative counterparts.
* **Problem-Solving**: They are particularly useful for problems that naturally fit a recursive structure, such as tree traversals or combinatorial problems.

**Disadvantages:**

* **Performance**: Recursive functions can be less efficient due to overhead from function calls and increased memory usage from maintaining the call stack.
* **Stack Overflow**: Deep recursion can lead to stack overflow errors, especially in Python, which has a default recursion limit (usually around 1000).

User Defined Function

def greet():

print("Hello, World!")

greet() # Output: Hello, World!

**Function Returning Multiple Values**

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

**return a + b, a - b, a \* b, a / b**

**sum\_result, diff\_result, prod\_result, div\_result = calculate(10, 2)**

**print(sum\_result, diff\_result, prod\_result, div\_result)**

**# Output: 12 8 20 5.0**

**Module 2**

**### File Handling in Python**

**File handling in Python refers to the process of creating, reading, writing, and manipulating files on your computer. Python provides built-in functions and methods for file operations, allowing you to work with various file types (like text files, binary files, etc.) easily.**

**### Basic File Operations**

**Here are the fundamental operations you can perform on files in Python:**

**1. \*\*Opening a File\*\*: You can open a file using the `open()` function. This function requires the filename and the mode in which you want to open the file.**

**2. \*\*Reading from a File\*\*: You can read the content of a file using methods such as `read()`, `readline()`, and `readlines()`.**

**3. \*\*Writing to a File\*\*: You can write content to a file using methods such as `write()` and `writelines()`.**

**4. \*\*Closing a File\*\*: Always close the file after operations using the `close()` method to free up system resources.**

**### File Modes**

**When opening a file, you can specify different modes:**

**- `'r'`: Read (default mode). Opens a file for reading.**

**- `'w'`: Write. Opens a file for writing (creates a new file or truncates an existing file).**

**- `'a'`: Append. Opens a file for appending content.**

**- `'b'`: Binary mode. Opens a file in binary format (e.g., for images).**

**- `'x'`: Exclusive creation. Fails if the file already exists.**

**- `'t'`: Text mode (default mode).**

**### Example: File Handling in Python**

**#### 1. Writing to a File**

**```python**

**# Writing to a file**

**with open('example.txt', 'w') as file:**

**file.write("Hello, World!\n")**

**file.write("This is a file handling example.\n")**

**```**

**#### 2. Reading from a File**

**```python**

**# Reading from a file**

**with open('example.txt', 'r') as file:**

**content = file.read()**

**print(content)**

**```**

**\*\*Output:\*\***

**```**

**Hello, World!**

**This is a file handling example.**

**```**

**#### 3. Reading Line by Line**

**```python**

**# Reading line by line**

**with open('example.txt', 'r') as file:**

**for line in file:**

**print(line.strip()) # Use strip() to remove newline characters**

**```**

**#### 4. Appending to a File**

**```python**

**# Appending to a file**

**with open('example.txt', 'a') as file:**

**file.write("Adding more content to the file.\n")**

**```**

**#### 5. Reading All Lines into a List**

**```python**

**# Reading all lines into a list**

**with open('example.txt', 'r') as file:**

**lines = file.readlines()**

**print(lines)**

**```**

**### Using `with` Statement**

**Using the `with` statement is a good practice when working with files. It automatically handles closing the file for you, even if an error occurs during file operations.**

**### Handling Exceptions**

**You can use `try` and `except` blocks to handle exceptions that may occur during file operations, such as trying to open a file that does not exist.**

**```python**

**try:**

**with open('non\_existent\_file.txt', 'r') as file:**

**content = file.read()**

**except FileNotFoundError:**

**print("The file does not exist.")**

**```**

**---**

**Exception Handling**

# Exception Handling in Python

try:

# Code that may raise an exception

num = int(input("Enter a number: "))

result = 10 / num

print(f"Result: {result}")

except ValueError:

# Handle ValueError (e.g., invalid input)

print("Invalid input! Please enter a valid integer.")

except ZeroDivisionError:

# Handle ZeroDivisionError (division by zero)

print("Error! You can't divide by zero.")

except Exception as e:

# Handle any other exception

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

else:

# This block executes if no exceptions were raised

print("The operation completed successfully.")

finally:

# This block always executes, regardless of exceptions

print("Execution finished.")

**Key Points:**

* **try**: Block where you write code that may raise exceptions.
* **except**: Block to handle specific exceptions.
* **else**: Optional block that runs if no exceptions occur.
* **finally**: Block that always runs after try/except, regardless of the outcome.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Sqlite – Database Programing

import sqlite3

# Connect to a database (or create one if it doesn't exist)

conn = sqlite3.connect('example.db')

# Create a cursor object to interact with the database

cursor = conn.cursor()

# Create a table

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

age INTEGER NOT NULL

)

''')

# Insert data into the table

cursor.execute('''

INSERT INTO users (name, age) VALUES (?, ?)

''', ('Alice', 30))

cursor.execute('''

INSERT INTO users (name, age) VALUES (?, ?)

''', ('Bob', 25))

# Commit the changes

conn.commit()

# Query the database

cursor.execute('SELECT \* FROM users')

rows = cursor.fetchall()

# Display the results

for row in rows:

print(row)

# Update a record

cursor.execute('''

UPDATE users SET age = ? WHERE name = ?

''', (31, 'Alice'))

# Delete a record

cursor.execute('''

DELETE FROM users WHERE name = ?

''', ('Bob',))

# Commit the changes

conn.commit()

# Close the connection

conn.close()

**Common SQL Commands**

* **CREATE TABLE**: Create a new table.
* **INSERT INTO**: Insert new records into a table.
* **SELECT**: Retrieve records from a table.
* **UPDATE**: Modify existing records.
* **DELETE**: Remove records from a table.
* **DROP TABLE**: Delete a table from the database.

**Introduction to Classes in Python**

Classes are blueprints for creating objects (instances) that bundle data (attributes) and functionality (methods) together. This is a fundamental concept in object-oriented programming.

class Dog:

# Class attribute

species = "Canis familiaris"

def \_\_init\_\_(self, name, age):

# Instance attributes

self.name = name

self.age = age

def bark(self):

return f"{self.name} says woof!"

def get\_age(self):

return f"{self.name} is {self.age} years old."

# Creating instances (objects) of the Dog class

dog1 = Dog("Buddy", 3)

dog2 = Dog("Max", 5)

# Accessing instance methods and attributes

print(dog1.bark()) # Output: Buddy says woof!

print(dog1.get\_age()) # Output: Buddy is 3 years old.

print(dog2.bark()) # Output: Max says woof!

print(dog2.get\_age()) # Output: Max is 5 years old.

# Accessing class attribute

print(Dog.species) # Output: Canis familiaris

print(dog1.species) # Output: Canis familiaris (access via instance)

**Key Concepts**

1. **Class Definition**:
   * Defined using the class keyword followed by the class name (PascalCase).
2. **Constructor (\_\_init\_\_ method)**:
   * Initializes instance attributes and is called automatically when a new object of the class is created.
3. **Instance Attributes**:
   * Unique to each object (instance) and are defined using self.
4. **Methods**:
   * Functions defined within a class that operate on instance attributes.
   * Can be instance methods, class methods (using @classmethod), or static methods (using @staticmethod).
5. **Class Attributes**:
   * Shared among all instances of the class and are defined directly inside the class.

**What is self :**

In Python, self is a conventional name used for the first parameter of instance methods in a class. It refers to the instance of the class itself and is used to access instance attributes and methods within the class. Here’s a breakdown of what self is and how it works:

**Key Points about self**

1. **Reference to the Instance**:
   * self allows you to refer to instance variables and methods from within class methods. It is not a keyword in Python but a naming convention. You could technically name it anything, but using self is the standard practice.
2. **Accessing Instance Attributes**:
   * You can use self to access attributes that are associated with the specific instance of the class.
3. **Distinguishing Between Instance Variables and Local Variables**:
   * Using self helps distinguish between instance variables (attributes of the object) and local variables within methods.
4. **Not Automatically Passed**:
   * When you call a method on an instance of a class, Python automatically passes the instance as the first argument to the method. You don’t need to include it in the method call.

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name # Instance variable

self.age = age # Instance variable

def bark(self):

return f"{self.name} says woof!"

def get\_age(self):

return f"{self.name} is {self.age} years old."

# Create an instance of the Dog class

my\_dog = Dog("Buddy", 4)

# Accessing methods

print(my\_dog.bark()) # Output: Buddy says woof!

print(my\_dog.get\_age()) # Output: Buddy is 4 years old.