Python Programming Notes: Advanced Concepts.

Day 1: September 11, 2025

Functions and Object-Oriented Programming (OOP)

Functions: An In-Depth Look

A **function** is a block of code that is organized, reusable, and performs a single, related action. This helps to break down large programs into smaller, manageable parts.

- **Types of Functions:**
    - **Built-in Functions:** Functions that are part of Python's core, like print(), len(), sum(), and input().
    - **User-defined Functions:** Functions you create yourself to perform specific tasks in your code.
- **Types of Arguments:**
    - **Positional Arguments:** Arguments passed to a function based on their position or order.

```
def describe_pet(animal, name):
    print(f"I have a {animal} named {name}.")
describe_pet("dog", "Buddy")
```

    - **Keyword Arguments:** Arguments specified by name, allowing you to pass them in any order.

```
describe_pet(name="Buddy", animal="dog")
```

    - **Default Arguments:** A parameter is assigned a default value in the function definition, which is used if no argument is provided for it.

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")
greet("Alice")          # Uses default: "Hello, Alice!"
greet("Bob", "Hi")      # Overrides default: "Hi, Bob!"
```

- **Arbitrary Arguments (*args and **kwargs):**
    - *args (non-keyworded arguments): Allows a function to accept any number of positional arguments. They are packed into a **tuple**.
    - **kwargs (keyworded arguments): Allows a function to accept any number of keyword arguments. They are packed into a **dictionary**.

```
def show_info(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)
show_info(1, "apple", age=25, city="New York")
# Output:
# Positional arguments: (1, 'apple')
# Keyword arguments: {'age': 25, 'city': 'New York'}
```

- **Variable Scope: The global Keyword:** The global keyword is used to modify a global variable from within a function.

```
x = 10  # A global variable
def modify_x():
    global x
    x = 20
modify_x()
print(x) # Output: 20
```

Object-Oriented Programming (OOP): The Four Pillars in Detail

OOP is a powerful programming paradigm that structures code around objects, which have both data and behavior.

- **1. Encapsulation** Encapsulation is the principle of bundling an object's data and methods into a single unit (a class). It protects the internal state of an object from being modified in an uncontrolled way. In Python, we achieve this through naming conventions that signal "private" attributes.
    - **Public vs. "Private" Attributes:**

- **Public:** Attributes without a leading underscore. They can be accessed and modified from outside the class.
- **"Protected" (_):** Attributes with a single leading underscore. This is a convention to indicate they are for internal use and should not be accessed from outside.
- **"Private" (__):** Attributes with a double leading underscore. Python internally "mangles" the name to make it harder to access from outside the class, but it's not truly private.

```
class UserAccount:
def __init__(self, username, password):
    self.username = username
    self.__password = password  # "Private" attribute

  def get_password_length(self):
    return len(self.__password)

my_account = UserAccount("student", "secure_password123")
print(my_account.username)
print(my_account.get_password_length())
```

- **2. Inheritance: A Deeper Dive** Inheritance is a core OOP concept where a new class (child or derived class) can inherit the properties (attributes) and behaviors (methods) of an existing class (parent or base class). This promotes code reuse and creates a logical hierarchy.

**super()` function:** The `super()` function is used to call a method from the parent class. It's often used in the child class's `__init__` method to initialize the parent class's attributes.

Types of Inheritance:

Single Inheritance: A child class inherits from only one parent class.

Example:

```
class Animal:
  def __init__(self, species):
```

```
        self.species = species
    def speak(self):
        print("The animal makes a sound.")


class Dog(Animal):  # Dog inherits from a single parent, Animal
    def __init__(self, species, name):
        super().__init__(species) # Call the parent's constructor
        self.name = name
    def speak(self):
        print(f"{self.name} says Woof!")


my_dog = Dog("Canine", "Buddy")
my_dog.speak()
print(f"Buddy is a {my_dog.species}.")
```

Multiple Inheritance: A child class inherits from more than one parent class.

Example:

```
class Programmer:
    def write_code(self):
        print("Writing code...")


class Musician:
    def play_music(self):
        print("Playing music...")


class Genius(Programmer, Musician): # Genius inherits from both
    def create(self):
        self.write_code()
        self.play_music()
```

```
davinci = Genius()
davinci.create()
```

Multilevel Inheritance: A child class inherits from a parent class, which itself inherits from another parent class.

Example:

```
class Grandparent:
    def __init__(self, name):
        self.name = name
    def legacy(self):
        print(f"{self.name} has a rich history.")

class Parent(Grandparent):
    def __init__(self, name, job):
        super().__init__(name)
        self.job = job
    def work(self):
        print(f"{self.name} works as a {self.job}.")

class Child(Parent):
    def __init__(self, name, job, school):
        super().__init__(name, job)
        self.school = school
    def study(self):
        print(f"{self.name} studies at {self.school}.")

alice = Child("Alice", "Engineer", "University of Python")
alice.study()
alice.work()
```

```
alice.legacy()
```

Hierarchical Inheritance: Multiple child classes inherit from a single parent class.

Example:

```
class Vehicle:
    def move(self):
        print("The vehicle moves.")


class Car(Vehicle):
    def drive(self):
        print("Driving the car.")


class Bicycle(Vehicle):
    def pedal(self):
        print("Pedaling the bicycle.")


my_car = Car()
my_bike = Bicycle()
my_car.move()
my_bike.move()
```

- **3. Polymorphism** Polymorphism means "many forms." In OOP, it allows different objects to respond to the same method call in their own unique way. The classic example involves a base class and multiple derived classes that override a method.
    - **Method Overriding:** A child class provides a specific implementation for a method that is already defined in its parent class.

```
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")
```

```
class Dog(Animal):
    def make_sound(self):
        # The Dog class overrides the make_sound method
        print("Woof woof!")

class Cat(Animal):
    def make_sound(self):
        # The Cat class also overrides the make_sound method
        print("Meow!")

# Here's the magic of polymorphism: the same function call
# behaves differently depending on the object type.
def animal_action(animal):
    animal.make_sound()

dog_obj = Dog()
cat_obj = Cat()

animal_action(dog_obj) # Calls Dog's method
animal_action(cat_obj) # Calls Cat's method
```

- **4. Abstraction** Abstraction is about hiding the complex internal details of an object and exposing only the essential functionalities to the user. In Python, this is often achieved using **abstract base classes** from the abc module. An abstract class cannot be instantiated on its own; it must be inherited by a child class that provides an implementation for its abstract methods.
    - **Abstract Base Class:** A blueprint for other classes. It defines methods that must be implemented by any concrete child class.

```
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")
```

```
class Dog(Animal):
    def make_sound(self):
        # The Dog class overrides the make_sound method
        print("Woof woof!")

class Cat(Animal):
    def make_sound(self):
        # The Cat class also overrides the make_sound method
        print("Meow!")

# Here's the magic of polymorphism: the same function call
# behaves differently depending on the object type.
def animal_action(animal):
    animal.make_sound()

dog_obj = Dog()
cat_obj = Cat()

animal_action(dog_obj) # Calls Dog's method
animal_action(cat_obj) # Calls Cat's method
```

Day 2: September 12, 2025

Control Statements and Logical Operators

Control statements determine the order in which your code is executed. Logical operators help you build complex conditions for these statements.

Logical Operators: Building Complex Conditions

Logical operators combine multiple conditional expressions into a single True or False result.

and: Returns True if **both** operands are True.

Truth Table:

Operand 1	Operand 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

or: Returns True if **at least one** operand is True.

Truth Table:

Operand 1	Operand 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

- **not:** Reverses the logical state. not True is False, and not False is True.
- **Sample Code with Logical Operators:**

```
# Example 1: `and`
is_admin = True
is_logged_in = True
```

```
if is_admin and is_logged_in:
    print("Access granted to admin panel.")


# Example 2: `or`
is_weekend = False
has_day_off = True
if is_weekend or has_day_off:
    print("Time to relax!")


# Example 3: `not`
is_valid = False
if not is_valid:
    print("Data is invalid. Please correct it.")
```

Control Statements: if, elif, else

This structure allows your program to make decisions and follow a specific "branch" of code.

- **if:** The most basic control statement. The code block is executed only if the condition is True.

```
score = 95
if score > 90:
    print("You got an A.")
```

- **if...else:** Provides a default path to take if the initial condition is False.

```
age = 17
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

- **if...elif...else:** Used to check for another condition if the previous if or elif conditions were False. You can have multiple elif blocks.

```
grade = 85
if grade >= 90:
    print("Excellent! You got an A.")
elif grade >= 80:
    print("Great job! You got a B.")
elif grade >= 70:
    print("Good effort. You got a C.")
else:
    print("You got a D or F. Keep studying!")
```

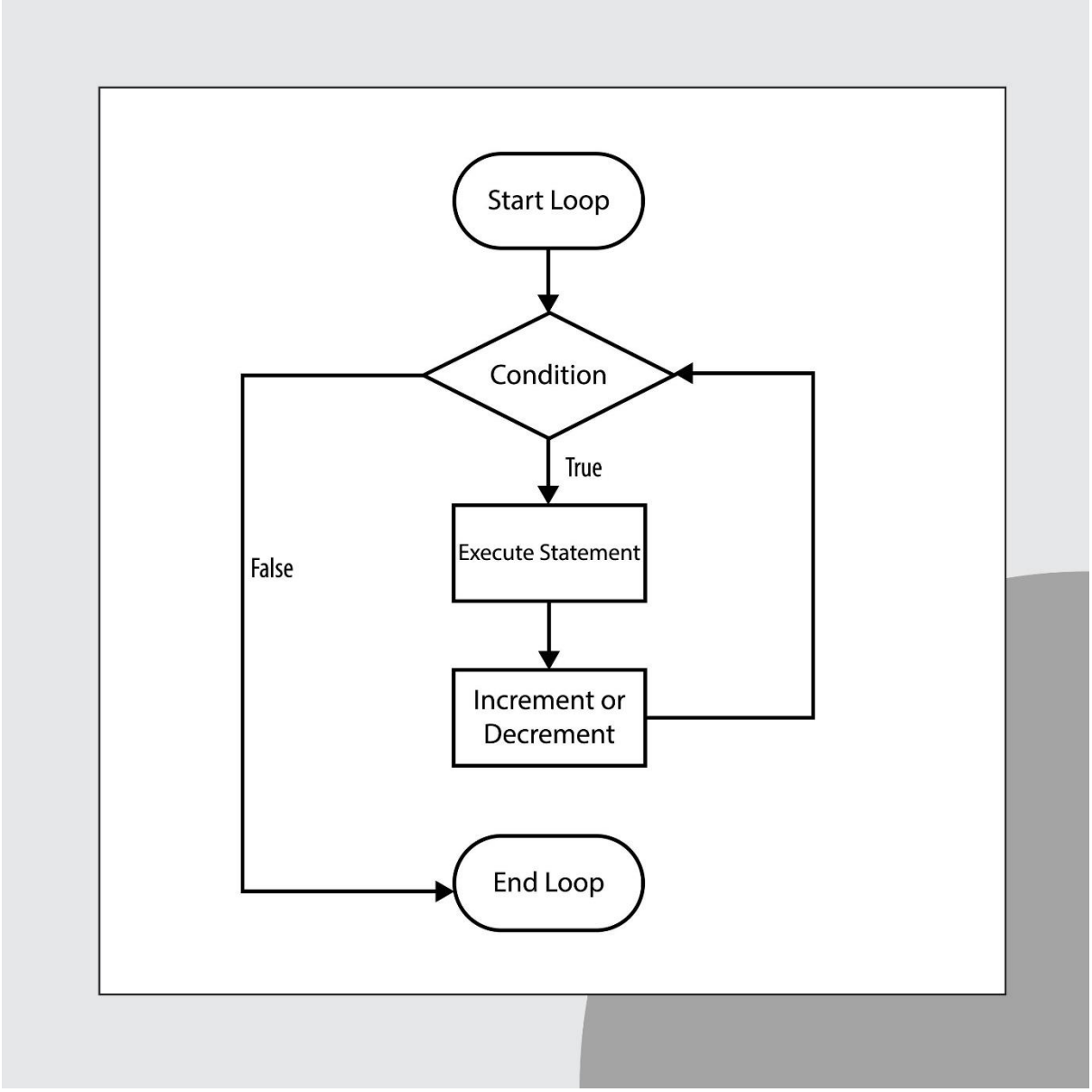
Day 3: September 15, 2025

Looping: The Power of Repetition

Looping statements are used to execute a block of code repeatedly.

for Loop: Iteration over a Sequence

A **for loop** is used to iterate over the items of any sequence (a list, tuple, dictionary, string, or range). It's great when you know the number of times you need to loop.



- **Iterating over Different Data Types:**

```
# Looping through a list of items
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
```

```
print(fruit)


# Looping through characters in a string
for char in "Python":
print(char)


# Looping through a dictionary's keys and values
student = {"name": "Charlie", "age": 20, "major": "Computer Science"}
for key, value in student.items():
print(f"Key: {key}, Value: {value}")
```

- **Useful Functions with for Loops:**

```
range(start, stop, step): Generates a sequence of numbers.
for i in range(1, 10, 2):
   print(i, end=" ") # Output: 1 3 5 7 9


enumerate(iterable): Returns both the index and the value of each item.
for index, fruit in enumerate(fruits):
   print(f"Fruit at index {index} is {fruit}.")


zip(iterable1, iterable2, ...): Combines items from multiple iterables.
names = ["Alice", "Bob"]
ages = [25, 30]
for name, age in zip(names, ages):
   print(f"{name} is {age} years old.")
```

while Loop: Repetition Based on a Condition

A **while loop** repeats as long as its condition is True. This is useful when the number of iterations is unknown beforehand.

- **Example with a Counter:**

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1
```

- **Example with a Sentinel Value:**

```
user_input = ""
while user_input.lower() != "quit":
    user_input = input("Enter a word (or 'quit' to exit): ")
    print(f"You entered: {user_input}")
```

- **Loop Control Statements:**
    - break: Exits the loop immediately.

```
for number in range(10):
    if number == 5:
        break
    print(number) # Output: 0 1 2 3 4
```

    - continue: Skips the rest of the code in the current iteration and moves to the next one.

```
for number in range(5):
    if number == 2:
        continue
    print(number) # Output: 0 1 3 4
```

- pass: A null statement. It is used as a placeholder where a statement is syntactically required but you want no action to be performed.

```
for number in range(5):
    if number % 2 == 0:
        pass # This does nothing, just a placeholder
    else:
        print(number) # Output: 1 3
```

Day 4: September 16, 2025

Exception Handling: Robust and Graceful Error Management

Exception handling is a critical part of writing stable, user-friendly programs. An **exception** is a special type of error that occurs during the execution of your program, but it can be "handled" or "caught" to prevent the program from crashing.

- **The Problem:** Without exception handling, a simple error like a user typing a letter instead of a number will cause your program to stop with a ValueError.
- # This will crash if the user enters a non-integer
- user_age = int(input("Enter your age: "))

- **The Solution: The try...except Block:** The try block contains the code that might cause an error. The except block contains the code that will run if an error occurs. This allows you to gracefully handle the error and give the user a helpful message.
    - **Basic try...except:**

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except:
    print("An error occurred.")
```

    - **Handling Specific Exceptions:**

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input. Please enter a valid number.")
```

    - **The try...except...else Block:** The else block runs only if the try block completes successfully without any exceptions.

```
try:
        num = int(input("Enter a number: "))
        print("The number is:", num)
except ValueError:
```

```
    print("Invalid input.")
else:
    print("The try block executed successfully!")
```

- **The try...except...finally Block:** The finally block **always** runs, regardless of whether an exception occurred. It's perfect for cleanup tasks, such as closing files or connections.

```
file = None
try:
    file = open("my_file.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("The file was not found.")
finally:
    if file:
        file.close()
        print("File has been closed.")
```

- **Comprehensive Example:**

```
while True:
    try:
        num1_str = input("Enter a number: ")
        num2_str = input("Enter a second number: ")

        num1 = int(num1_str) # Could raise a ValueError
        num2 = int(num2_str) # Could raise a ValueError
```

```
        result = num1 / num2 # Could raise a ZeroDivisionError

    except ValueError:
        print("Invalid input. Please enter a valid number.")

    except ZeroDivisionError:
        print("You cannot divide by zero. Please try again.")

    except Exception as e:
        # This is a generic handler for any other unexpected error
        print(f"An unexpected error occurred: {e}")

    else:
        # The 'else' block runs ONLY if the 'try' block was successful.
        print(f"The result is {result}.")
        break  # Exit the loop on success

    finally:
        # The 'finally' block always runs, no matter what.
        # It's perfect for cleanup, like closing files or connections.
        print("Operation attempt complete.")
```