

Python Shortcuts for internship

Sahas Talasila

INTRODUCTION

```
# variables.py - Understanding Variables and Data Types

# Variables are used to store values in Python.
# You don't need to declare types explicitly; Python infers them.

# 1. Integer variable
age = 25 # A whole number
print("Age:", age, "| Type:", type(age)) # Outputs: Age: 25 | Type: <class 'int'>

# 2. Float variable
height = 5.9 # A decimal (floating-point) number
print("Height:", height, "| Type:", type(height))
# Outputs: Height: 5.9 | Type: <class 'float'>

# 3. String variable
name = "Sahas" # Text (a sequence of characters)
print("Name:", name, "| Type:", type(name))
# Outputs: Name: Sahas | Type: <class 'str'>

# 4. Boolean variable
is_learning = True # Boolean value (True or False)
print("Learning:", is_learning, "| Type:", type(is_learning))
# Outputs: Learning: True | Type: <class 'bool'>

# 5. Lists - Ordered collections of items
numbers = [1, 2, 3, 4, 5] # A list of integers
print("Numbers:", numbers, "| Type:", type(numbers))
# Outputs: Numbers: [1, 2, 3, 4, 5] | Type: <class 'list'>

# 6. Dictionaries - Key-value pairs for storing structured data
person = {"name": "Sahas", "age": 25, "height": 5.9}
print("Person:", person, "| Type:", type(person))
#Outputs: Person: {'name': 'Sahas', 'age': 25, 'height': 5.9} | Type: <class 'dict'>

# Type conversion
age_as_string = str(age) # Convert integer to string
height_as_int = int(height) # Convert float to integer (loss of precision)
print("Converted age:", age_as_string, "| Type:", type(age_as_string))
# Outputs: Converted age: 25 | Type: <class 'str'>
print("Converted height:", height_as_int, "| Type:", type(height_as_int))
```

```
# Outputs: Converted height: 5 / Type: <class 'int'>
```

1.1 Control Flow

```
# control_flow.py - Understanding Control Flow in Python

# Control flow structures determine the logic of how a program executes.

### Conditional Statements (if, elif, else)
age = 20

if age < 18:
    print("You are a minor.")
elif 18 <= age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")

# Explanation:
# - The `if` block executes only if the condition `age < 18` is True.
# - The `elif` block executes if `18 <= age < 65` is True.
# - If none of the above conditions are met, the `else` block runs.

### Loops (for and while)
# `for` loop: Iterating over a sequence (list, tuple, range)
numbers = [10, 20, 30, 40, 50]

print("Iterating using a for loop:")
for num in numbers:
    print(num)  # Prints each number in the list

# `while` loop: Repeats execution while a condition remains True
count = 0
print("Iterating using a while loop:")
while count < 5:
    print("Count is:", count)
    count += 1  # Increment count (prevents infinite loop)

### Exception Handling
# Prevents the program from crashing due to runtime errors
try:
```

```

x = int(input("Enter a number: ")) # User inputs a value
result = 10 / x # May cause division by zero
print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input! Please enter a valid number.")
finally:
    print("Execution complete.")

# Explanation:
# - `try`: Code that may cause an error.
# - `except ZeroDivisionError`: Handles cases where division by zero occurs.
# - `except ValueError`: Catches invalid input (e.g., user enters non-numeric values).
# - `finally`: Executes regardless of errors.

```

1.2 Functions and Recursion

```

# functions.py - Understanding Functions and Recursion in Python

# Functions allow reusable blocks of code that can be executed multiple times.

### Defining and Calling Functions
def greet(name):
    """Function to greet a user by name."""
    print(f"Hello, {name}! Welcome!")

# Calling the function
greet("Sahas") # Outputs: Hello, Sahas! Welcome!

### Arguments and Return Values
def add_numbers(a, b):
    """Function that returns the sum of two numbers."""
    return a + b # The `return` statement sends back a result

# Storing function output in a variable
result = add_numbers(5, 7)
print("Sum:", result) # Outputs: Sum: 12

```

```

### Default Arguments
def power(base, exponent=2):
    """Function with a default exponent of 2 (square)."""
    return base ** exponent

print("Default exponent (square):", power(3))    # Outputs: exponent (square): 9
print("Custom exponent:", power(2, 3))          # Outputs: Custom exponent: 8

### Recursion - When a function calls itself
def factorial(n):
    """Computes factorial using recursion (n! = n * (n-1) * (n-2)...*1)."""
    if n == 0 or n == 1:    # Base case: Factorial of 0 or 1 is always 1
        return 1
    return n * factorial(n - 1)    # Recursive case: n multiplied by factorial(n-1)

print("Factorial of 5:", factorial(5))    # Outputs: Factorial of 5: 120

# Explanation:
# - Base case prevents infinite recursion.
# - Each call reduces `n` until it reaches 1.
# - The function builds the result step-by-step as it returns from recursion.

```

1.3 OOP Python edition

```

# oop.py - Deep Dive into Object-Oriented Programming (OOP)

# A class is a blueprint for creating objects. An object is an instance of a class.

### 1. Defining a Class and Creating Objects
class Person:
    """A class that represents a person."""

    def __init__(self, name, age):
        """Constructor method (__init__): Initializes object attributes."""
        self.name = name    # Instance attribute
        self.age = age      # Instance attribute

    def introduce(self):
        """Instance method: Uses attributes of the object."""
        return f"Hello, my name is {self.name} and I am {self.age} years old."

```

```

# Creating instances (objects)
person1 = Person("Sahas", 25)
person2 = Person("Alice", 30)

print(person1.introduce()) # Outputs: Hello, my name is Sahas and I am 25 years old.
print(person2.introduce()) # Outputs: Hello, my name is Alice and I am 30 years old.

### 2. Encapsulation (Restricting Direct Access to Attributes)
class BankAccount:
    """Encapsulated bank account class (uses private attributes)."""

    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance # Private attribute (cannot be accessed directly)

    def deposit(self, amount):
        """Deposits money into the account."""
        if amount > 0:
            self.__balance += amount
            return f"Deposited £{amount}. New balance: £{self.__balance}"
        return "Invalid deposit amount."

    def get_balance(self):
        """Accesses the private attribute."""
        return f"Balance for {self.owner}: £{self.__balance}"

# Creating an account
account = BankAccount("Sahas", 1000)

# Accessing balance indirectly
print(account.get_balance()) # Outputs: Balance for Sahas: £1000

# Trying to access private attribute directly (fails)
# print(account.__balance) # This would raise an AttributeError

### 3. Inheritance (Extending a Class)
# A subclass inherits attributes and methods from a parent class.

class Employee(Person): # Employee class inherits from Person
    """Employee class extending Person."""

```

```

def __init__(self, name, age, job_title, salary):
    """Extend constructor: Call parent class constructor using super()"""
    super().__init__(name, age)
    self.job_title = job_title
    self.salary = salary

def work(self):
    """Instance method specific to Employee."""
    return f"{self.name} works as a {self.job_title} and  

    earns £{self.salary} annually."

# Creating an Employee object
employee1 = Employee("Sahas", 25, "Software Engineer", 50000)
print(employee1.introduce())
# Outputs inherited method: Hello, my name is Sahas and I am 25 years old.
print(employee1.work())
# Outputs subclass method: Sahas works as a Software Engineer and earns £50000 annually.

### 4. Polymorphism (Using Same Methods in Different Ways)
# Polymorphism allows different classes to use the same method name.

class Dog:
    """Dog class with speak method."""
    def speak(self):
        return "Woof!"

class Cat:
    """Cat class with speak method."""
    def speak(self):
        return "Meow!"

# Using polymorphism
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak()) # Outputs: Woof! Meow!

---

### 5. Special Methods (`__init__`, `__str__`, `__repr__`)
# Python provides special methods to customize object behavior.

```



```

class Book:
    """Book class showcasing special methods."""

    def __init__(self, title, author, pages):
        """Initializer method."""
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        """Readable representation of object (used when printing)."""
        return f'{self.title} by {self.author}, {self.pages} pages.'

    def __repr__(self):
        """Technical representation (used in debugging)."""
        return f"Book(title={self.title}, author={self.author}, pages={self.pages})"

# Creating a book object
book1 = Book("Python Mastery", "Sahas", 350)
print(book1)  # Outputs: 'Python Mastery' by Sahas, 350 pages.
print(repr(book1))  # Outputs: Book(title=Python Mastery, author=Sahas, pages=350)

---

### 6. Private & Protected Variables
# Private (`__var`) and protected (`_var`) attributes control access levels.

class SecureData:
    """Demonstrates private and protected variables."""

    def __init__(self):
        self._protected_var = "This is protected"  # Accessible in subclasses
        self.__private_var = "This is private"  # Not directly accessible

    def access_private(self):
        """Accessing private attribute via method."""
        return self.__private_var

# Creating instance
data = SecureData()

print(data._protected_var)  # Outputs: This is protected

```

```
# print(data.__private_var) # This would raise an error (private variable)

print(data.access_private()) # Outputs: This is private (accessed via method)
```

1.4 Extra Python OOP

```
# advanced_oop.py - Comprehensive Guide to Object-Oriented Programming (OOP)

# -----
# 1. CLASS OPERATIONS
# -----

class Vehicle:
    """Base class representing a generic vehicle."""

    # Class attribute (shared across all instances)
    vehicle_count = 0

    def __init__(self, brand, model, year):
        """Constructor method (__init__) - Initializes instance attributes."""
        self.brand = brand # Instance attribute
        self.model = model
        self.year = year
        Vehicle.vehicle_count += 1 # Incrementing class attribute

    def display_info(self):
        """Instance method - Provides details about the vehicle."""
        return f"{self.year} {self.brand} {self.model}"

# Creating instances
car1 = Vehicle("Toyota", "Corolla", 2020)
car2 = Vehicle("Ford", "Focus", 2022)

print(car1.display_info()) # Outputs: 2020 Toyota Corolla
print(car2.display_info()) # Outputs: 2022 Ford Focus
print("Total Vehicles:", Vehicle.vehicle_count) # Outputs: Total Vehicles: 2

# -----
# 2. ENCAPSULATION & ACCESS MODIFIERS
# -----

class BankAccount:
```

```

"""Encapsulation Example - Restricting direct access to attributes."""

def __init__(self, owner, balance):
    self.owner = owner
    self._protected_balance = balance
    # Protected variable (can be accessed by subclasses)
    self.__private_balance = balance
    # Private variable (cannot be accessed directly)

def deposit(self, amount):
    """Deposits money."""
    if amount > 0:
        self.__private_balance += amount
        return f"Deposited £{amount}. New balance: £{self.__private_balance}"
    return "Invalid deposit amount."

def get_balance(self):
    """Access private attribute via method."""
    return f"Balance for {self.owner}: £{self.__private_balance}"

# Creating an account
account = BankAccount("Sahas", 1000)
print(account.get_balance()) # Outputs: Balance for Sahas: £1000
# print(account.__private_balance) # This would raise an AttributeError

# -----
# 3. INHERITANCE & METHOD OVERRIDING
# -----

class Employee:
    """Base class for employees."""
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def get_info(self):
        return f"{self.name} earns £{self.salary} per year."

class Developer(Employee): # Developer inherits from Employee
    def __init__(self, name, salary, programming_language):
        super().__init__(name, salary) # Call parent class constructor
        self.programming_language = programming_language

```

```

def get_info(self): # Method overriding
    return f"{self.name} writes {self.programming_language} code and
    earns £{self.salary}."

dev1 = Developer("Sahas", 60000, "Python")
print(dev1.get_info()) # Outputs: Sahas writes Python code and earns £60000.

# -----
# 4. ABSTRACT CLASSES & INTERFACES
# -----

from abc import ABC, abstractmethod # Import abstract class module

class Shape(ABC): # Abstract base class
    """Abstract class representing geometric shapes."""

    @abstractmethod
    def area(self):
        """Abstract method - Must be implemented by subclasses."""
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        """Implements required abstract method."""
        return 3.14 * self.radius ** 2

circle1 = Circle(5)
print("Circle Area:", circle1.area()) # Outputs: Circle Area: 78.5

# -----
# 5. MULTIPLE INHERITANCE
# -----

class A:
    def method_a(self):
        return "Method A"

class B:
    def method_b(self):
        return "Method B"

```

```

class C(A, B):  # Multiple inheritance
    def method_c(self):
        return "Method C"

obj = C()
print(obj.method_a())  # Outputs: Method A
print(obj.method_b())  # Outputs: Method B
print(obj.method_c())  # Outputs: Method C

# -----
# 6. OPERATOR OVERLOADING
# -----

class Vector:
    """Operator overloading example - Adding two vectors using + operator."""

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

    def __add__(self, other):  # Overloading the + operator
        return Vector(self.x + other.x, self.y + other.y)

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

v1 = Vector(2, 3)
v2 = Vector(4, 5)
result = v1 + v2  # Uses overloaded + operator
print(result)  # Outputs: Vector(6, 8)

# -----
# 7. METAPROGRAMMING (CLASS ATTRIBUTES & DECORATORS)
# -----

def log_method(func):
    """Custom decorator to log method calls."""
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}...")
        return func(*args, **kwargs)
    return wrapper

```

```

class Logger:
    """Example of using decorators in a class."""

    @log_method # Applying decorator
    def action(self):
        return "Action executed!"

logger = Logger()
print(logger.action()) # Outputs: Calling action... Action executed!

# -----
# 8. DESIGN PATTERNS - SINGLETON
# -----
class Singleton:
    """Singleton pattern ensures only one instance is created."""

    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

s1 = Singleton()
s2 = Singleton()
print(s1 is s2) # Outputs: True (both references point to the same instance)

```

1.5 File Formats and Conversion

```

# full_file_operations.py - Comprehensive File Handling Operations in Python

import os
import json
import csv
import shutil # For copying/moving files
import zipfile # For ZIP compression
import tarfile # For TAR compression

# -----
# 1. TEXT FILE OPERATIONS

```

```

# -----

file_path = "example.txt"

# Writing (overwrite mode 'w')
with open(file_path, "w") as file:
    file.write("Hello, Sahas!\nWelcome to Python file handling.\n")

# Appending ('a' mode)
with open(file_path, "a") as file:
    file.write("Appending new content.\n")

# Reading ('r' mode)
with open(file_path, "r") as file:
    content = file.read()
    print("Text File Content:\n", content)

# Reading line-by-line
with open(file_path, "r") as file:
    for line in file:
        print("Line:", line.strip())

# -----
# 2. JSON FILE OPERATIONS
# -----

json_data = {"name": "Sahas", "age": 25, "languages": ["Python", "C", "Lua"]}

# Writing JSON ('w' mode)
with open("data.json", "w") as json_file:
    json.dump(json_data, json_file, indent=4)

# Reading JSON ('r' mode)
with open("data.json", "r") as json_file:
    loaded_json = json.load(json_file)
    print("Loaded JSON:", loaded_json)

# -----
# 3. CSV FILE OPERATIONS
# -----

csv_file = "data.csv"

```

```

# Writing CSV ('w' mode)
with open(csv_file, "w", newline="") as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(["Name", "Age", "Language"])
    writer.writerow(["Sahas", "25", "Python"])
    writer.writerow(["Alice", "30", "C++"])

# Reading CSV ('r' mode)
with open(csv_file, "r") as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print("CSV Row:", row)

# -----
# 4. BINARY FILE OPERATIONS
# -----

binary_file = "binary_file.bin"
binary_data = b"\x89PNG\r\n\x1a\n\x00\x00\x00IHDR"

# Writing Binary ('wb' mode)
with open(binary_file, "wb") as bin_file:
    bin_file.write(binary_data)

# Reading Binary ('rb' mode)
with open(binary_file, "rb") as bin_file:
    loaded_bin = bin_file.read()
    print("Loaded Binary Data:", loaded_bin)

# -----
# 5. FILE PATH OPERATIONS
# -----

print("Current Directory:", os.getcwd())
print("Does 'example.txt' exist?", os.path.exists(file_path))
print("Absolute Path:", os.path.abspath(file_path))

# -----
# 6. FILE MANAGEMENT (COPYING/MOVING/DELETING)
# -----

```



```

shutil.copy(file_path, "copy_example.txt") # Copy file
shutil.move("copy_example.txt", "moved_example.txt") # Move file
os.remove("moved_example.txt") # Delete file

# -----
# 7. FILE COMPRESSION (ZIP & TAR)
# -----

# Create a ZIP file
with zipfile.ZipFile("compressed.zip", "w") as zipf:
    zipf.write(file_path) # Add file to ZIP

# Extract ZIP file
with zipfile.ZipFile("compressed.zip", "r") as zipf:
    zipf.extractall("extracted_files") # Extract to folder

# Create a TAR file
with tarfile.open("compressed.tar.gz", "w:gz") as tarf:
    tarf.add(file_path) # Add file to TAR

# Extract TAR file
with tarfile.open("compressed.tar.gz", "r:gz") as tarf:
    tarf.extractall("extracted_tar_files")

# -----
# 8. ERROR HANDLING IN FILE OPERATIONS
# -----

try:
    with open("non_existent.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: File does not exist!")
except IOError:
    print("Error: Issue with reading the file.")

```

2.1 NumPy

```
# full_numpy_operations.py - Comprehensive NumPy Operations in Python

import numpy as np

# -----
# 1. ARRAY CREATION & INITIALIZATION
# -----

# Creating NumPy arrays from lists
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array

# Special arrays
zeros = np.zeros((3, 3)) # 3x3 zero matrix
ones = np.ones((2, 2)) # 2x2 ones matrix
identity_matrix = np.eye(4) # 4x4 identity matrix
random_array = np.random.rand(3, 3) # Random values in range [0,1]
range_array = np.arange(1, 10, 2) # [1, 3, 5, 7, 9]
linspace_array = np.linspace(0, 1, 5) # 5 evenly spaced values between 0 and 1

print("Array from list:\n", arr1)
print("Identity Matrix:\n", identity_matrix)
print("Linspace Array:\n", linspace_array)

# -----
# 2. ARRAY SHAPE, SIZE & DATA TYPES
# -----

print("Shape of arr2:", arr2.shape) # (2, 3)
print("Size of arr2:", arr2.size) # Total number of elements
print("Data type of arr1:", arr1.dtype) # Data type of elements

# Changing data type
arr_float = arr1.astype(float) # Convert to float type

# -----
# 3. INDEXING, SLICING & ADVANCED SELECTION
# -----
```

```

print("First element:", arr1[0]) # Access single element
print("First row of arr2:", arr2[0]) # Access row
print("Element at row 1, col 2:", arr2[1, 2]) # Access specific element

# Slicing
print("First two elements:", arr1[:2]) # First two values
print("First row, first two columns:\n", arr2[:1, :2]) # Partial matrix slice

# Advanced Indexing
bool_mask = arr1 > 2 # Boolean mask selection
filtered_values = arr1[bool_mask] # Extract elements matching condition
print("Filtered Values:", filtered_values)

# -----
# 4. MATRIX OPERATIONS
# -----

matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

# Basic operations
print("Matrix Addition:\n", matrix1 + matrix2)
print("Element-wise Multiplication:\n", matrix1 * matrix2)

# Dot product
dot_product = np.dot(matrix1, matrix2)
print("Dot Product:\n", dot_product)

# -----
# 5. LINEAR ALGEBRA FUNCTIONS
# -----

print("Transpose:\n", matrix1.T)
print("Inverse:\n", np.linalg.inv(matrix1))
print("Determinant:", np.linalg.det(matrix1))
print("Eigenvalues:", np.linalg.eigvals(matrix1)) # Eigenvalues of matrix
print("QR Decomposition:\n", np.linalg.qr(matrix1)) # QR factorization

# -----
# 6. STATISTICAL FUNCTIONS
# -----

```

```

stats_array = np.array([[1, 2, 3], [4, 5, 6]])

print("Mean:", np.mean(stats_array))
print("Median:", np.median(stats_array))
print("Variance:", np.var(stats_array))
print("Standard Deviation:", np.std(stats_array))
print("Min:", np.min(stats_array))
print("Max:", np.max(stats_array))

# -----
# 7. SORTING, SEARCHING & FILTERING
# -----

sorted_arr = np.sort(arr2, axis=1) # Sort rows
print("Sorted Array:\n", sorted_arr)

unique_values = np.unique(arr2) # Unique elements
print("Unique Values:", unique_values)

# -----
# 8. CONCATENATION, STACKING & BROADCASTING
# -----

arr_a = np.array([[1, 2], [3, 4]])
arr_b = np.array([[5, 6], [7, 8]])

concat_horiz = np.hstack((arr_a, arr_b)) # Horizontal stack
concat_vert = np.vstack((arr_a, arr_b)) # Vertical stack

print("Horizontal Concatenation:\n", concat_horiz)
print("Vertical Concatenation:\n", concat_vert)

vector = np.array([1, 2])
broadcasted_matrix = matrix1 + vector # Adds vector to every row
print("Broadcasted Addition:\n", broadcasted_matrix)

# -----
# 9. ADVANCED NUMPY OPERATIONS
# -----

# Reshaping arrays

```

```

reshaped = arr1.reshape((2, 2))
print("Reshaped Array:\n", reshaped)

# Flattening a matrix
flattened = matrix1.flatten()
print("Flattened Matrix:", flattened)

# -----
# 10. FOURIER TRANSFORM & POLYNOMIAL FITTING
# -----

signal = np.array([0, 1, 0, -1])
fft_result = np.fft.fft(signal) # Fast Fourier Transform
print("FFT Result:", fft_result)

# Polynomial fitting
x = np.array([0, 1, 2, 3])
y = np.array([1, 3, 7, 13])
coefficients = np.polyfit(x, y, 2) # Fit quadratic polynomial
print("Polynomial Coefficients:", coefficients)

# -----
# 11. HISTOGRAM ANALYSIS
# -----

data_samples = np.random.randn(1000) # Generate random data
histogram, bins = np.histogram(data_samples, bins=10)
print("Histogram Bins:", bins)
print("Histogram Counts:", histogram)

# -----
# 12. MEMORY OPTIMIZATION & PERFORMANCE
# -----

arr_large = np.random.rand(1000000) # Large dataset
print("Memory size (bytes):", arr_large.nbytes) # Checking memory usage

```

2.2 Matplotlib and Seaborn usage

```

# visualization.py - Comprehensive Data Visualization with Matplotlib & Seaborn

```

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Generate sample data
np.random.seed(42)
data = np.random.randn(1000) # Normal distribution

# Create a DataFrame for Seaborn
df = pd.DataFrame({
    "Category": np.random.choice(["A", "B", "C"], size=100),
    "Values": np.random.randint(1, 100, size=100)
})

# -----
# 1. BASIC MATPLOTLIB PLOTS
# -----

# Line Plot
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.figure(figsize=(8, 4))
plt.plot(x, y, label="Sine Wave", color="blue", linestyle="--")
plt.title("Line Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.grid()
plt.show()

# Scatter Plot
plt.figure(figsize=(6, 4))
plt.scatter(np.random.rand(50), np.random.rand(50), color="red", marker="o")
plt.title("Scatter Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()

# -----
# 2. HISTOGRAM & PIE CHART
# -----

```

```

# Histogram
plt.figure(figsize=(6, 4))
plt.hist(data, bins=20, color="purple", edgecolor="black", alpha=0.7)
plt.title("Histogram Example")
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.show()

# Pie Chart
sizes = [40, 30, 20, 10]
labels = ["A", "B", "C", "D"]
plt.figure(figsize=(5, 5))
plt.pie(sizes, labels=labels, autopct="%.1f%%", startangle=140)
plt.title("Pie Chart Example")
plt.show()

# -----
# 3. MULTIPLE SUBPLOTS
# -----

fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Line plot
axes[0, 0].plot(x, y, color="blue")
axes[0, 0].set_title("Line Plot")

# Scatter plot
axes[0, 1].scatter(np.random.rand(50), np.random.rand(50), color="red")
axes[0, 1].set_title("Scatter Plot")

# Histogram
axes[1, 0].hist(data, bins=20, color="green", alpha=0.7)
axes[1, 0].set_title("Histogram")

# Pie chart
axes[1, 1].pie(sizes, labels=labels, autopct="%.1f%%")
axes[1, 1].set_title("Pie Chart")

plt.tight_layout()
plt.show()

```

```

# -----
# 4. SEABORN VISUALIZATION TECHNIQUES
# -----

# Boxplot
plt.figure(figsize=(6, 4))
sns.boxplot(x=df["Category"], y=df["Values"], palette="Set2")
plt.title("Seaborn Boxplot Example")
plt.show()

# Violin Plot
plt.figure(figsize=(6, 4))
sns.violinplot(x=df["Category"], y=df["Values"], palette="coolwarm")
plt.title("Seaborn Violin Plot Example")
plt.show()

# Strip Plot (Scatter on Categories)
plt.figure(figsize=(6, 4))
sns.stripplot(x=df["Category"], y=df["Values"], jitter=True, palette="Set3")
plt.title("Seaborn Strip Plot Example")
plt.show()

# -----
# 5. REGRESSION & DISTRIBUTION PLOTS
# -----

# Regression Plot
sns.lmplot(x="Values", y="Values", hue="Category", data=df, height=5)
plt.title("Seaborn Regression Plot")
plt.show()

# Distribution Plot (Histogram + KDE)
plt.figure(figsize=(6, 4))
sns.histplot(data, kde=True, bins=30, color="darkred")
plt.title("Seaborn Distribution Plot")
plt.show()

# KDE Plot
plt.figure(figsize=(6, 4))
sns.kdeplot(data, shade=True, color="darkblue")
plt.title("Seaborn KDE Density Plot")
plt.show()

```



```
# -----
# 6. PAIR PLOTS & HEATMAPS
# -----

# Pair Plot
sns.pairplot(df, hue="Category", palette="husl")
plt.title("Seaborn Pairplot Example")
plt.show()

# Heatmap (Correlation Matrix)
corr_matrix = df.corr()
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm")
plt.title("Seaborn Heatmap Example")
plt.show()
```

Linear regression models the relationship between an independent variable x and a dependent variable y :

$$y = \beta_0 + \beta_1 x + \epsilon$$

where:

- β_0 (intercept) controls the baseline prediction,
- β_1 (slope) determines the influence of x on y ,
- ϵ represents the error (residual).

To estimate β_0 and β_1 , we minimize the **sum of squared errors**, defined as:

$$J(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_i))^2$$

Solving for partial derivatives:

$$\beta_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

where \bar{x} and \bar{y} are the **means** of x and y .

2.3 Error Metrics: RMSE, SSE, R²

2.3.1 Root Mean Squared Error (RMSE)

RMSE measures the standard deviation of residuals:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

where \hat{y}_i is the predicted value.

2.3.2 Sum of Squared Errors (SSE)

SSE quantifies total residual variation:

$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

2.3.3 Coefficient of Determination (R²)

R^2 measures how well regression explains data variance:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

where the denominator represents **total variance**.

—

2.4 Gradient Descent Optimization

Instead of solving equations directly, we iteratively optimize parameters using **Gradient Descent**:

$$\beta_1^{new} = \beta_1^{old} - \alpha \frac{\partial J}{\partial \beta_1}$$

$$\beta_0^{new} = \beta_0^{old} - \alpha \frac{\partial J}{\partial \beta_0}$$

where:

- α is the learning rate,
- $\frac{\partial J}{\partial \beta}$ are gradients computed from data.

—

2.5 Python Implementation

Below is the Python implementation using NumPy, formatted using ‘minted’ with a gray background.

```
import numpy as np

# Sample data
X = np.array([1, 2, 3, 4, 5])
Y = np.array([2, 4, 5, 4, 5])

# Compute means
X_mean = np.mean(X)
Y_mean = np.mean(Y)

# Compute slope (beta_1)
numerator = np.sum((X - X_mean) * (Y - Y_mean))
denominator = np.sum((X - X_mean) ** 2)
beta_1 = numerator / denominator

# Compute intercept (beta_0)
beta_0 = Y_mean - beta_1 * X_mean

# Compute Predictions
Y_pred = beta_0 + beta_1 * X

# Compute RMSE
rmse = np.sqrt(np.mean((Y - Y_pred) ** 2))

# Compute SSE
sse = np.sum((Y - Y_pred) ** 2)

# Compute R2 Score
sst = np.sum((Y - Y_mean) ** 2)
r_squared = 1 - (sse / sst)

# Print results
print(f"Linear Regression Equation: y = {beta_0:.2f} + {beta_1:.2f}x")
print(f"RMSE: {rmse:.4f}")
print(f"SSE: {sse:.4f}")
print(f"R2 Score: {r_squared:.4f}")

# Gradient Descent Optimization
alpha = 0.01 # Learning rate
```

```

beta_0_gd, beta_1_gd = 0, 0  # Initial parameters

for epoch in range(1000):
    Y_pred_gd = beta_0_gd + beta_1_gd * X
    error = Y_pred_gd - Y

    # Compute gradients
    grad_beta_0 = np.mean(error)
    grad_beta_1 = np.mean(error * X)

    # Update parameters
    beta_0_gd -= alpha * grad_beta_0
    beta_1_gd -= alpha * grad_beta_1

print(f"Optimized Parameters using Gradient Descent: beta_0 =
{beta_0_gd:.2f}, beta_1 = {beta_1_gd:.2f}")

```

Gradient Descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of the negative gradient.

For a given function $J(\theta)$, the update rule for **Gradient Descent** is:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{\partial J}{\partial \theta}$$

where:

- θ represents the parameters to be optimized,
- α is the **learning rate**,
- $\frac{\partial J}{\partial \theta}$ is the **gradient**.

TYPES OF GRADIENT DESCENT

Gradient Descent can be implemented in different ways depending on how we update the parameters.

3.1 Batch Gradient Descent

Batch Gradient Descent updates the parameters **using the entire dataset** at each iteration:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta)$$

Pros: - More stable updates as gradients are computed on the full dataset. - Converges smoothly to

the optimal solution.

****Cons****: - Computationally expensive for large datasets. - Slower compared to other methods.

3.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates the parameters ****using a single sample**** at a time:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta^{(i)})$$

****Pros****: - Faster updates since each iteration processes only one example. - Can escape local minima due to randomness.

****Cons****: - High variance in updates, making convergence noisier. - Requires tuning of learning rate carefully.

3.3 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent finds a middle ground, updating parameters ****using a subset (mini-batch) of data****:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta^{(batch)})$$

****Pros****: - Computational efficiency: balances stability and speed. - Allows parallel computation on GPUs.

****Cons****: - Needs careful tuning of batch size to optimize performance.

COMPARISON OF GRADIENT DESCENT METHODS

Method	Update Frequency	Computational Cost	Convergence Stability
Batch GD	Full Dataset	High	Stable
SGD	Single Sample	Low	Noisy
Mini-Batch GD	Small Subset	Medium	Balanced

PYTHON IMPLEMENTATION

Below is the Python implementation using ‘minted’ with a gray background.

```
import numpy as np
import matplotlib.pyplot as plt

# Sample data
```

```

X = np.array([1, 2, 3, 4, 5])
Y = np.array([2, 4, 5, 4, 5])

# Initialize parameters
beta_0, beta_1 = 0, 0
alpha = 0.01 # Learning rate
epochs = 1000
batch_size = 2 # Mini-batch size

# Store history for visualization
history_beta_0, history_beta_1 = [], []

# Mini-Batch Gradient Descent
for epoch in range(epochs):
    indices = np.random.choice(len(X), batch_size, replace=False)
    X_batch, Y_batch = X[indices], Y[indices]

    Y_pred = beta_0 + beta_1 * X_batch
    error = Y_pred - Y_batch

    # Compute gradients
    grad_beta_0 = np.mean(error)
    grad_beta_1 = np.mean(error * X_batch)

    # Update parameters
    beta_0 -= alpha * grad_beta_0
    beta_1 -= alpha * grad_beta_1

    history_beta_0.append(beta_0)
    history_beta_1.append(beta_1)

# Print final parameters
print(f"Optimized Parameters: beta_0 = {beta_0:.2f}, beta_1 = {beta_1:.2f}")

# Plot Gradient Descent Progress
plt.figure(figsize=(8, 5))
plt.plot(history_beta_1, label="beta_1", color="red")
plt.plot(history_beta_0, label="beta_0", color="blue")
plt.xlabel("Epochs")
plt.ylabel("Parameter Values")
plt.title("Gradient Descent Optimization Progress")
plt.legend()

```

```
plt.show()
```

INTRODUCTION

This document presents a Python implementation of logistic regression using advanced optimization techniques (Adam, RMSProp, Momentum) and regularization (L1/L2 penalties).

OPTIMIZATION ALGORITHMS

7.1 Momentum

Momentum helps accelerate gradient descent in directions with consistent gradients. The update rule is:

$$v_t = \beta v_{t-1} + \eta \nabla J(\theta_t) \quad (1)$$

$$\theta_t = \theta_{t-1} - v_t \quad (2)$$

where v_t is velocity, β is the momentum coefficient, and η is the learning rate.

```
v = np.zeros_like(weights)
for epoch in range(epochs):
    gradient = compute_gradient(X, y)
    v = beta1 * v + lr * gradient
    weights -= v
```

7.2 RMSProp

RMSProp adapts learning rates by maintaining an exponentially decaying average of squared gradients.

$$s_t = \beta s_{t-1} + (1 - \beta)(\nabla J(\theta_t))^2 \quad (3)$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t} + \epsilon} \nabla J(\theta_t) \quad (4)$$

```
s = np.zeros_like(weights)
epsilon = 1e-8
for epoch in range(epochs):
    gradient = compute_gradient(X, y)
    s = beta2 * s + (1 - beta2) * (gradient ** 2)
    weights -= (lr / (np.sqrt(s) + epsilon)) * gradient
```

7.3 Adam Optimization

Adam combines momentum and RMSProp for more efficient training.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla J(\theta_t) \quad (5)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla J(\theta_t))^2 \quad (6)$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t \quad (7)$$

```
m, v = np.zeros_like(weights), np.zeros_like(weights)
beta1, beta2 = 0.9, 0.999
for epoch in range(epochs):
    gradient = compute_gradient(X, y)
    m = beta1 * m + (1 - beta1) * gradient
    v = beta2 * v + (1 - beta2) * (gradient ** 2)
    weights -= (lr / (np.sqrt(v) + epsilon)) * m
```

REGULARIZATION TECHNIQUES

8.1 L1 Regularization (Lasso)

L1 regularization encourages sparsity in weights:

$$L1 = \lambda \sum |w_i| \quad (8)$$

```
loss += reg_strength * np.sum(np.abs(weights))
gradient += reg_strength * np.sign(weights)
```

8.2 L2 Regularization (Ridge)

L2 regularization penalizes large weights:

$$L2 = \lambda \sum w_i^2 \quad (9)$$

```
loss += reg_strength * np.sum(weights ** 2)
gradient += 2 * reg_strength * weights
```

LOGISTIC REGRESSION WITH OPTIMIZATION

Finally, we integrate these concepts into a logistic regression model.


```

import numpy as np

class LogisticRegression:
    def __init__(self, lr=0.01, epochs=1000, optimizer="adam",
reg_type=None, reg_strength=0.01):
        self.lr = lr
        self.epochs = epochs
        self.optimizer = optimizer
        self.reg_type = reg_type
        self.reg_strength = reg_strength
        self.beta1, self.beta2 = 0.9, 0.999
        self.epsilon = 1e-8

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def compute_loss(self, y_true, y_pred):
        loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
        if self.reg_type == "l1":
            loss += self.reg_strength * np.sum(np.abs(self.weights))
        elif self.reg_type == "l2":
            loss += self.reg_strength * np.sum(self.weights ** 2)
        return loss

    def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros(n)
        self.bias = 0
        v, s = np.zeros(n), np.zeros(n)

        for epoch in range(self.epochs):
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)

            dw = (1/m) * np.dot(X.T, (y_pred - y))
            db = (1/m) * np.sum(y_pred - y)

            if self.optimizer == "momentum":
                v = self.beta1 * v + self.lr * dw
                self.weights -= v
            elif self.optimizer == "rmsprop":
                s = self.beta2 * s + (dw ** 2)
                self.weights -= (self.lr / (np.sqrt(s) + self.epsilon)) * dw
            elif self.optimizer == "adam":

```

```

        v = self.beta1 * v + (1 - self.beta1) * dw
        s = self.beta2 * s + (1 - self.beta2) * (dw ** 2)
        v_corr = v / (1 - self.beta1 ** (epoch + 1))
        s_corr = s / (1 - self.beta2 ** (epoch + 1))
        self.weights -= (self.lr / (np.sqrt(s_corr) + self.epsilon)) * v_corr

    self.bias -= self.lr * db

def predict(self, X):
    return self.sigmoid(np.dot(X, self.weights) + self.bias)

```