|  |  |
| --- | --- |
| Python Examples  With Explanations |  |
| This booklet is designed to provide readers with a practical and hands-on approach to learning Python programming through a variety of examples and use cases. Whether you are a beginner looking to get started with Python or an experienced developer seeking to enhance your skills, this booklet offers valuable insights and code samples to help you achieve your goals.  By Randall Fadler September 2024 |  |

Contents

[Introduction 4](#_Toc177468508)

[Python Looping 4](#_Toc177468509)

[for Loop 4](#_Toc177468510)

[while Loop 5](#_Toc177468511)

[For and While Loop Summary 6](#_Toc177468512)

[Rotating a Matrix 6](#_Toc177468513)

[Multiplying Matrixes. 8](#_Toc177468514)

[Using NumPy for Matrix Multiplication 9](#_Toc177468515)

[File Operations 10](#_Toc177468516)

[Reading and writing to a text file 10](#_Toc177468517)

[Reading and writing CSV files 10](#_Toc177468518)

[Manipulating Data in Files 10](#_Toc177468519)

[Calculating the total salary from the CSV file 11](#_Toc177468520)

[Working with File Paths 11](#_Toc177468521)

[Reading from and Writing to an Excel File 11](#_Toc177468522)

[Collecting User Input and Appending to CSV File 12](#_Toc177468523)

[File Operations Summary 16](#_Toc177468524)

[Exception Handling 16](#_Toc177468525)

[Basic Exception Handling 16](#_Toc177468526)

[Handling Multiple Exceptions 16](#_Toc177468527)

[Using else and finally Blocks 17](#_Toc177468528)

[Custom Exception Handling 17](#_Toc177468529)

[Nested Exception Handling 18](#_Toc177468530)

[Exception Handling Summary 19](#_Toc177468531)

[Dictionaries and Sets 19](#_Toc177468532)

[Dictionaries 19](#_Toc177468533)

[Basic Dictionary Operations 19](#_Toc177468534)

[Nested Dictionaries 21](#_Toc177468535)

[Dictionary Methods 22](#_Toc177468536)

[Sets 22](#_Toc177468537)

[Set Operations (Union, Intersection, Difference) 23](#_Toc177468538)

[Set Methods 24](#_Toc177468539)

[Dictionary and Sets Summary 25](#_Toc177468540)

[2-D Lists – Matrixes 25](#_Toc177468541)

[Creating and Accessing Elements in a 2D List 25](#_Toc177468542)

[Iterating Through a 2D List 26](#_Toc177468543)

[Modifying Elements in a 2D List 26](#_Toc177468544)

[Adding and Removing Rows and Columns 27](#_Toc177468545)

[Matrix Operations (Addition and Multiplication) 28](#_Toc177468546)

[Matrixes Summary 29](#_Toc177468547)

[Classes and Objects 29](#_Toc177468548)

[Defining a Simple Class 29](#_Toc177468549)

[Adding More Methods 30](#_Toc177468550)

[Inheritance 31](#_Toc177468551)

[Example Database Class 32](#_Toc177468552)

[Modules 34](#_Toc177468553)

[Library Management System 34](#_Toc177468554)

[Create the book.py Module 34](#_Toc177468555)

[Create the member.py Module 35](#_Toc177468556)

[Create the library.py Module 36](#_Toc177468557)

[Create the main.py Script 36](#_Toc177468558)

[Running the Code 37](#_Toc177468559)

[Decorators 38](#_Toc177468560)

[Example 38](#_Toc177468561)

[Generators 39](#_Toc177468562)

[Example 39](#_Toc177468563)

[Context Managers 40](#_Toc177468564)

[Example 40](#_Toc177468565)

[Concurrency 41](#_Toc177468566)

[Threading 41](#_Toc177468567)

[Multiprocessing 42](#_Toc177468568)

[Tic Tac Toe 43](#_Toc177468569)

[Complete Source Code 43](#_Toc177468570)

[Code Breakdown and Explanations 47](#_Toc177468571)

[Conway’s Game of Life 54](#_Toc177468572)

[Complete Source Code 54](#_Toc177468573)

[Code Breakdown and Explanation 55](#_Toc177468574)

# Introduction

Welcome to “Example Python Scripts”, a guide designed to provide you with practical and illustrative Python code examples. Whether you are a novice programmer taking your first steps into the world of coding or an experienced developer looking to expand your repertoire, this booklet is crafted to cater to your needs. *You can also use this booklet as a reference guide.*

Python, known for its simplicity and versatility, has become one of the most popular programming languages in the world. Its applications range from web development and data analysis to artificial intelligence and automation. In this booklet, you will find a diverse collection of Python scripts that cover a wide array of topics, including basic syntax, data structures, file handling, web scraping, and more. Each script is accompanied by detailed explanations to help you understand the underlying concepts and techniques. By working through these examples, you will not only enhance your coding skills but also gain valuable insights into best practices and efficient programming. So, grab your favorite code editor, and let’s embark on this exciting journey to explore the endless possibilities of Python programming together! So, let’s dig in.

# Python Looping

Looping is a fundamental concept in programming that allows you to execute a block of code repeatedly. Python provides two primary types of loops: for loops and while loops. These loops help automate repetitive tasks, making your code more efficient and easier to manage. Understanding how to use loops effectively is essential for any Python programmer.

## for Loop

The for loop is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) and execute a block of code for each element in the sequence.

**Iterating Over a List**

fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:

print(fruit)

**Explanation**: This for loop iterates over the list fruits and prints each fruit.

**Iterating Over a Range of Numbers**

for i in range(5):

print(i)

**Explanation**: This for loop iterates over a range of numbers from 0 to 4 and prints each number.

**Iterating Over a Dictionary**

student\_grades = {'Alice': 85, 'Bob': 92, 'Charlie': 78}

for student, grade in student\_grades.items():

print(f"{student}: {grade}")

**Explanation**: This for loop iterates over the key-value pairs in the dictionary student\_grades and prints each student’s name and grade.

## while Loop

The while loop is used to execute a block of code as long as a specified condition is true.

**Basic**while**Loop**

count = 0

while count < 5:

print(count)

count += 1

**Explanation**: This while loop prints the value of count and increments it by 1 until count is no longer less than 5.

**Using**break**to Exit a Loop**

count = 0

while True:

print(count)

count += 1

if count == 5:

break

**Explanation**: This while loop runs indefinitely until the break statement is encountered when count equals 5, which exits the loop.

**Using**continue**to Skip an Iteration**

count = 0

while count < 5:

count += 1

if count == 3:

continue

print(count)

**Explanation**: This while loop increments count and skips printing the value when count equals 3 using the continue statement.

## For and While Loop Summary

Looping is an essential concept in Python that allows you to execute a block of code multiple times. The two primary types of loops are for loops and while loops:

* for**Loop**: Used to iterate over a sequence (list, tuple, dictionary, set, or string) and execute a block of code for each element.
  + Example: Iterating over a list, range of numbers, or dictionary.
* while**Loop**: Used to execute a block of code as long as a specified condition is true.
  + Example: Basic while loop, using break to exit a loop, and using continue to skip an iteration.

## Rotating a Matrix

import tkinter as tk

from tkinter import messagebox

rainbow\_colors = [

'red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'

]

class RainbowMatrixApp:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("Rainbow Matrix")

self.matrix = {}

self.create\_matrix()

self.create\_control\_buttons()

def create\_matrix(self):

for i in range(10):

for j in range(10):

color\_index = (i + j) % len(rainbow\_colors)

color = rainbow\_colors[color\_index]

button = tk.Button(self.root, bg=color, width=5, height=2)

button.grid(row=i, column=j)

self.matrix[(i, j)] = button

def rotate\_matrix(self):

rotated\_matrix = {}

for i in range(10):

for j in range(10):

rotated\_matrix[(j, 9 - i)] = self.matrix[(i, j)]

for (i, j), button in rotated\_matrix.items():

button.grid(row=i, column=j)

self.matrix = rotated\_matrix

def create\_control\_buttons(self):

rotate\_button = tk.Button(self.root, text="Rotate 90°", command=self.rotate\_matrix)

rotate\_button.grid(row=10, column=0, columnspan=5)

exit\_button = tk.Button(self.root, text="Exit", command=self.root.quit)

exit\_button.grid(row=10, column=5, columnspan=5)

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

app = RainbowMatrixApp(root)

root.mainloop()

**Explanation**

1. **Import Required Libraries**:
   * tkinter is imported for creating the GUI.
   * messagebox is imported for displaying messages (if needed).
2. **Define the Colors of the Rainbow**:
   * A list rainbow\_colors is created containing the colors of the rainbow.
3. **Create the 10x10 Matrix with Buttons**:
   * A class RainbowMatrixApp is defined to encapsulate the application logic.
   * The \_\_init\_\_ method initializes the application, creates the matrix, and adds control buttons.
   * The create\_matrix method creates a 10x10 grid of buttons, each with a background color from the rainbow.
   * The rotate\_matrix method rotates the matrix 90 degrees by rearranging the buttons in the grid.
   * The create\_control\_buttons method adds buttons to rotate the matrix and exit the application.
4. **Run the Application**:
   * The RainbowMatrixApp class is instantiated, and the Tkinter main loop is started.

## Multiplying Matrixes.

Matrix multiplication is a fundamental operation in linear algebra with numerous applications in computer science, engineering, and mathematics. Let’s go through the concept of matrix multiplication with examples and detailed explanations.

**Matrix Multiplication**

Matrix multiplication involves multiplying two matrices to produce a third matrix. The number of columns in the first matrix must equal the number of rows in the second matrix. The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

**Example 1: Multiplying Two 2x2 Matrices**

# Define matrices A and B

A = [

[1, 2],

[3, 4]

]

B = [

[5, 6],

[7, 8]

]

# Initialize the result matrix with zeros

result = [

[0, 0],

[0, 0]

]

# Perform matrix multiplication

for i in range(len(A)):

for j in range(len(B[0])):

for k in range(len(B)):

result[i][j] += A[i][k] \* B[k][j]

# Print the result

for r in result:

print(r)

**Explanation:**

1. **Matrix Definition:** We define two matrices (A) and (B).
2. **Result Matrix Initialization:** We initialize the result matrix with zeros.
3. **Matrix Multiplication:** We use three nested loops to perform the multiplication. The outer loop iterates over the rows of (A), the middle loop iterates over the columns of (B), and the inner loop iterates over the elements of the row of (A) and the column of (B).
4. **Result Calculation:** We calculate the sum of the products of corresponding elements and store it in the result matrix.
5. **Print Result:** Finally, we print the resulting matrix.

## Using NumPy for Matrix Multiplication

import numpy as np

# Define matrices A and B

A = np.array([

[1, 2],

[3, 4]

])

B = np.array([

[5, 6],

[7, 8]

])

# Perform matrix multiplication using np.dot

result = np.dot(A, B)

# Print the result

print(result)

**Explanation:**

1. **Import NumPy:** We import the NumPy library.
2. **Matrix Definition:** We define matrices (A) and (B) as NumPy arrays.
3. **Matrix Multiplication:** We use the np.dot function to perform matrix multiplication.
4. **Print Result:** We print the resulting matrix.

# File Operations

## Reading and writing to a text file

# Writing to a text file

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

file.write("Name, Age, Salary\n")

file.write("Alice, 30, 70000\n")

file.write("Bob, 25, 50000\n")

file.write("Charlie, 35, 80000\n")

# Reading from a text file

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

content = file.read()

print(content)

## Reading and writing CSV files

import csv

# Writing to a CSV file

with open('example.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerow(["Name", "Age", "Salary"])

writer.writerow(["Alice", 30, 70000])

writer.writerow(["Bob", 25, 50000])

writer.writerow(["Charlie", 35, 80000])

# Reading from a CSV file

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

reader = csv.reader(file)

for row in reader:

print(row)

## Manipulating Data in Files

# Reading and manipulating data records from a CSV file

data = []

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

reader = csv.reader(file)

header = next(reader) # Skip the header row

for row in reader:

data.append(row)

# Adding a new record

data.append(["David", 28, 60000])

# Writing the updated data back to the CSV file

with open('example.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerow(header) # Write the header row

writer.writerows(data)

## Calculating the total salary from the CSV file

total\_salary = 0

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

reader = csv.reader(file)

next(reader) # Skip the header row

for row in reader:

total\_salary += int(row[2]) # Assuming the salary is in the third column

print(f"Total Salary: {total\_salary}")

## Working with File Paths

import os

# Getting the current working directory

current\_directory = os.getcwd()

print(f"Current Directory: {current\_directory}")

# Creating a new directory

new\_directory = os.path.join(current\_directory, 'new\_folder')

os.makedirs(new\_directory, exist\_ok=True)

print(f"New Directory Created: {new\_directory}")

# Moving a file to the new directory

source = 'example.csv'

destination = os.path.join(new\_directory, 'example.csv')

os.rename(source, destination)

print(f"File moved to: {destination}")

# Listing files in the new directory

files = os.listdir(new\_directory)

print(f"Files in {new\_directory}: {files}")

## Reading from and Writing to an Excel File

You must first install the openpyxl and pandas modules via pip.

>> pip install openpyxl pandas

import pandas as pd

# Creating a DataFrame

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [30, 25, 35, 28],

'Salary': [70000, 50000, 80000, 60000]

}

df = pd.DataFrame(data)

# Writing the DataFrame to an Excel file

df.to\_excel('example.xlsx', index=False)

print("Data written to example.xlsx")

## Collecting User Input and Appending to CSV File

import csv

import os

# Define the CSV file name

csv\_file = 'user\_data.csv'

# Check if the file exists to determine if we need to write the header

file\_exists = os.path.isfile(csv\_file)

# Function to get the number of records in the CSV file

def get\_record\_count(file):

with open(file, 'r') as f:

reader = csv.reader(f)

next(reader) # Skip the header

return sum(1 for row in reader)

# Open the CSV file in append mode

with open(csv\_file, 'a', newline='') as file:

writer = csv.writer(file)

# Write the header if the file does not exist

if not file\_exists:

writer.writerow(['Name', 'Address', 'City', 'State', 'Zip Code'])

writer.writerow(['Total Records', '', '', '', '', 0])

while True:

# Collect user input

name = input("Enter Name (or type 'STOP' to end): ")

if name.upper() == 'STOP':

break

address = input("Enter Address: ")

city = input("Enter City: ")

state = input("Enter State: ")

zip\_code = input("Enter Zip Code: ")

# Append the data to the CSV file

writer.writerow([name, address, city, state, zip\_code])

print("Record added.")

# Read the existing data

with open(csv\_file, 'r') as file:

lines = file.readlines()

# Update the "Total Records" row

record\_count = get\_record\_count(csv\_file)

lines[1] = f'Total Records,,,,,{record\_count}\n'

# Write the updated data back to the file

with open(csv\_file, 'w', newline='') as file:

file.writelines(lines)

print(f"Data collection terminated. Total records: {record\_count}")

Code Explanation

**1. Importing Libraries**

import csv

import os

* **csv**: This library is used for reading from and writing to CSV files.
* **os**: This library provides functions for interacting with the operating system, such as checking if a file exists.

**2. Defining the CSV File Name**

csv\_file = 'user\_data.csv'

* **csv\_file**: This variable stores the name of the CSV file.

**3. Checking if the File Exists**

file\_exists = os.path.isfile(csv\_file)

* **file\_exists**: This variable checks if the CSV file already exists.

**4. Function to Get the Number of Records**

**Python**

def get\_record\_count(file):

with open(file, 'r') as f:

reader = csv.reader(f)

next(reader) # Skip the header

return sum(1 for row in reader)

* **get\_record\_count**: This function reads the CSV file, skips the header row, and counts the number of records (excluding the header).

**5. Opening the CSV File in Append Mode**

with open(csv\_file, 'a', newline='') as file:

writer = csv.writer(file)

# Write the header if the file does not exist

if not file\_exists:

writer.writerow(['Name', 'Address', 'City', 'State', 'Zip Code'])

writer.writerow(['Total Records', '', '', '', '', 0])

* **with open(csv\_file, ‘a’, newline=‘’) as file**: Opens the CSV file in append mode.
* **writer = csv.writer(file)**: Creates a CSV writer object.
* **if not file\_exists**: If the file does not exist, writes the header row and the “Total Records” row with an initial count of 0.

**6. Collecting User Input and Appending Data**

while True:

# Collect user input

name = input("Enter Name (or type 'STOP' to end): ")

if name.upper() == 'STOP':

break

address = input("Enter Address: ")

city = input("Enter City: ")

state = input("Enter State: ")

zip\_code = input("Enter Zip Code: ")

# Append the data to the CSV file

writer.writerow([name, address, city, state, zip\_code])

print("Record added.")

* **while True**: Starts an infinite loop to collect user input.
* **name = input("Enter Name (or type ‘STOP’ to end): ")**: Prompts the user to enter their name. If the user types “STOP”, the loop breaks.
* **address, city, state, zip\_code**: Prompts the user to enter their address, city, state, and zip code.
* **writer.writerow([name, address, city, state, zip\_code])**: Appends the user input to the CSV file.
* **print(“Record added.”)**: Prints a confirmation message.

**7. Reading the Existing Data**

# Read the existing data

with open(csv\_file, 'r') as file:

lines = file.readlines()

* **with open(csv\_file, ‘r’) as file**: Opens the CSV file in read mode.
* **lines = file.readlines()**: Reads all lines from the file into a list.

**8. Updating the “Total Records” Row**

# Update the "Total Records" row

record\_count = get\_record\_count(csv\_file)

lines[1] = f'Total Records,,,,,{record\_count}\n'

* **record\_count = get\_record\_count(csv\_file)**: Gets the current record count.
* **lines[1] = f’Total Records,{record\_count}\n’**: Updates the second line (index 1) with the “Total Records” row and the current record count.

**9. Writing the Updated Data Back to the File**

# Write the updated data back to the file

with open(csv\_file, 'w', newline='') as file:

file.writelines(lines)

* **with open(csv\_file, ‘w’, newline=‘’) as file**: Opens the CSV file in write mode.
* **file.writelines(lines)**: Writes the updated lines back to the file.

**10. Printing the Final Message**

print(f"Data collection terminated. Total records: {record\_count}")

* **print(f"Data collection terminated. Total records: {record\_count}")**: Prints a final message with the total number of records.

**Summary**

This script:

1. Checks if the CSV file exists and writes the header and “Total Records” row if it doesn’t.
2. Collects user input for name, address, city, state, and zip code, and appends the data to the CSV file.
3. Updates the “Total Records” row with the current record count, ensuring it is always on the second line.
4. Writes the updated data back to the file.

## File Operations Summary

These example demonstrates:

* Writing to and reading from text files.
* Handling CSV files for structured data.
* Manipulating data records by adding new entries.
* Calculating the total of a numeric column.
* Working with file paths to create directories, move files, and list directory contents.

# Exception Handling

Exception handling in Python is a powerful feature that allows you to manage and respond to errors gracefully. Here are some examples demonstrating how to use exception handling in various scenarios:

## Basic Exception Handling

try:

# Attempt to divide by zero

result = 10 / 0

except ZeroDivisionError:

# Handle the exception

print("Error: Division by zero is not allowed.")

**Explanation**: This example attempts to divide a number by zero, which raises a ZeroDivisionError. The except block catches the exception and prints an error message.

## Handling Multiple Exceptions

try:

# Attempt to open a non-existent file

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

content = file.read()

except FileNotFoundError:

# Handle file not found error

print("Error: The file was not found.")

except IOError:

# Handle other I/O errors

print("Error: An I/O error occurred.")

**Explanation**: This example attempts to open a file that does not exist, which raises a FileNotFoundError. The except blocks handle both FileNotFoundError and other I/O errors.

## Using else and finally Blocks

try:

# Attempt to read from a file

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

content = file.read()

except FileNotFoundError:

# Handle file not found error

print("Error: The file was not found.")

else:

# Execute if no exceptions occur

print("File content:")

print(content)

finally:

# Execute regardless of whether an exception occurs

print("Execution completed.")

**Explanation**: This example includes else and finally blocks. The else block executes if no exceptions occur, and the finally block executes regardless of whether an exception occurs.

## Custom Exception Handling

class CustomError(Exception):

"""Custom exception class"""

pass

def check\_value(value):

if value < 0:

raise CustomError("Negative value is not allowed.")

try:

# Attempt to check a negative value

check\_value(-10)

except CustomError as e:

# Handle the custom exception

print(f"Error: {e}")

**Explanation**: This example includes else and finally blocks. The else block executes if no exceptions occur, and the finally block executes regardless of whether an exception occurs.

## Nested Exception Handling

try:

try:

# Attempt to convert a string to an integer

number = int("abc")

except ValueError:

# Handle value error

print("Error: Invalid input. Cannot convert to integer.")

raise

except Exception as e:

# Handle any other exceptions

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

Explanation: This example demonstrates nested exception handling. The inner try block catches a ValueError and re-raises it. The outer try block catches any other exceptions and prints an error message.

## Exception Handling Summary

* **Graceful Error Handling**: Exception handling allows programs to continue running smoothly even when errors occur.
* **Specificity**: Multiple except blocks enable handling specific exceptions differently.
* **Cleanup Actions**: The finally block ensures that cleanup actions are performed regardless of exceptions.
* **Custom Exceptions**: Custom exceptions provide a way to handle specific error conditions unique to your application.
* **Nested Handling**: Nested exception handling allows for more granular control over error management.

By incorporating these practices, you can write robust and resilient Python code that can handle a wide range of error conditions gracefully.

# Dictionaries and Sets

Python offers a variety of powerful data structures, among which dictionaries and sets stand out for their unique capabilities and efficiency. **Dictionaries** in Python are mutable, unordered collections of key-value pairs, where each key is unique. They provide a fast and efficient way to store and retrieve data based on a unique key, making them ideal for tasks that require quick lookups, such as counting occurrences, storing configurations, or mapping relationships. On the other hand, **sets** are unordered collections of unique elements. They are particularly useful for membership testing, eliminating duplicates, and performing mathematical operations like unions, intersections, and differences. Sets are highly efficient for these operations due to their underlying hash table implementation. Together, dictionaries and sets offer versatile tools for managing and manipulating data, making them indispensable in a Python programmer’s toolkit.

## Dictionaries

## Basic Dictionary Operations

# Creating a dictionary

student\_grades = {

'Alice': 85,

'Bob': 92,

'Charlie': 78,

'David': 90

}

# Accessing values

print(student\_grades['Alice']) # Output: 85

# Adding a new key-value pair

student\_grades['Eve'] = 88

# Updating a value

student\_grades['Alice'] = 95

# Removing a key-value pair

del student\_grades['Charlie']

# Iterating through a dictionary

for student, grade in student\_grades.items():

print(f"{student}: {grade}")

# Checking if a key exists

if 'Bob' in student\_grades:

print("Bob's grade is recorded.")

**Explanation**:

* **Creating a dictionary**: student\_grades is a dictionary where the keys are student names and the values are their grades.
* **Accessing values**: student\_grades['Alice'] retrieves the grade for Alice.
* **Adding a new key-value pair**: student\_grades['Eve'] = 88 adds a new student, Eve, with a grade of 88.
* **Updating a value**: student\_grades['Alice'] = 95 updates Alice’s grade to 95.
* **Removing a key-value pair**: del student\_grades['Charlie'] removes Charlie and his grade from the dictionary.
* **Iterating through a dictionary**: The for loop iterates through the dictionary, printing each student’s name and grade.
* **Checking if a key exists**: The if statement checks if Bob’s grade is recorded in the dictionary.

## Nested Dictionaries

# Creating a nested dictionary

class\_grades = {

'Math': {

'Alice': 85,

'Bob': 92

},

'Science': {

'Charlie': 78,

'David': 90

}

}

# Accessing nested values

print(class\_grades['Math']['Alice']) # Output: 85

# Adding a new nested dictionary

class\_grades['History'] = {'Eve': 88}

# Updating a nested value

class\_grades['Science']['Charlie'] = 80

# Iterating through a nested dictionary

for subject, grades in class\_grades.items():

print(f"{subject} Grades:")

for student, grade in grades.items():

print(f" {student}: {grade}")

**Explanation**:

* **Creating a nested dictionary**: class\_grades is a dictionary where each key is a subject, and the value is another dictionary of student grades.
* **Accessing nested values**: class\_grades['Math']['Alice'] retrieves Alice’s grade in Math.
* **Adding a new nested dictionary**: class\_grades['History'] = {'Eve': 88} adds a new subject, History, with Eve’s grade.
* **Updating a nested value**: class\_grades['Science']['Charlie'] = 80 updates Charlie’s grade in Science to 80.
* **Iterating through a nested dictionary**: The outer for loop iterates through subjects, and the inner for loop iterates through student grades within each subject.

## Dictionary Methods

# Creating a dictionary

inventory = {

'apples': 10,

'bananas': 5,

'oranges': 8

}

# Using dictionary methods

keys = inventory.keys()

values = inventory.values()

items = inventory.items()

print(keys) # Output: dict\_keys(['apples', 'bananas', 'oranges'])

print(values) # Output: dict\_values([10, 5, 8])

print(items) # Output: dict\_items([('apples', 10), ('bananas', 5), ('oranges', 8)])

# Using get() method

print(inventory.get('apples')) # Output: 10

print(inventory.get('grapes', 'Not Found')) # Output: Not Found

# Using pop() method

inventory.pop('bananas')

print(inventory) # Output: {'apples': 10, 'oranges': 8}

**Explanation**:

* **keys()**: Returns a view object of the dictionary’s keys.
* **values()**: Returns a view object of the dictionary’s values.
* **items()**: Returns a view object of the dictionary’s key-value pairs.
* **get()**: Retrieves the value for a given key, with an optional default value if the key is not found.
* **pop()**: Removes a key-value pair from the dictionary and returns the value.

## Sets

# Creating a set

fruits = {'apple', 'banana', 'cherry'}

# Adding an element

fruits.add('orange')

# Removing an element

fruits.remove('banana')

# Checking membership

print('apple' in fruits) # Output: True

# Iterating through a set

for fruit in fruits:

print(fruit)

**Explanation**:

* **Creating a set**: fruits is a set containing unique fruit names.
* **Adding an element**: fruits.add('orange') adds ‘orange’ to the set.
* **Removing an element**: fruits.remove('banana') removes ‘banana’ from the set.
* **Checking membership**: 'apple' in fruits checks if ‘apple’ is in the set.
* **Iterating through a set**: The for loop iterates through the set, printing each fruit.

## Set Operations (Union, Intersection, Difference)

# Creating sets

set\_a = {1, 2, 3, 4}

set\_b = {3, 4, 5, 6}

# Union

union\_set = set\_a.union(set\_b)

print(union\_set) # Output: {1, 2, 3, 4, 5, 6}

# Intersection

intersection\_set = set\_a.intersection(set\_b)

print(intersection\_set) # Output: {3, 4}

# Difference

difference\_set = set\_a.difference(set\_b)

print(difference\_set) # Output: {1, 2}

# Symmetric Difference

symmetric\_difference\_set = set\_a.symmetric\_difference(set\_b)

print(symmetric\_difference\_set) # Output: {1, 2, 5, 6}

**Explanation**:

* **Union**: Combines elements from both sets, removing duplicates.
* **Intersection**: Returns elements that are common to both sets.
* **Difference**: Returns elements that are in set\_a but not in set\_b.
* **Symmetric Difference**: Returns elements that are in either set, but not in both.

## Set Methods

# Creating a set

numbers = {1, 2, 3, 4, 5}

# Using set methods

numbers.add(6)

numbers.discard(3)

numbers.update([7, 8, 9])

print(numbers) # Output: {1, 2, 4, 5, 6, 7, 8, 9}

# Using copy() method

numbers\_copy = numbers.copy()

print(numbers\_copy) # Output: {1, 2, 4, 5, 6, 7, 8, 9}

# Using clear() method

numbers.clear()

print(numbers) # Output: set()

**Explanation**:

* **add()**: Adds an element to the set.
* **discard()**: Removes an element from the set if it exists.
* **update()**: Adds multiple elements to the set.
* **copy()**: Creates a shallow copy of the set.
* **clear()**: Removes all elements from the set.

## Dictionary and Sets Summary

These examples demonstrate various ways to use dictionaries and sets in Python:

* Basic dictionary operations such as creating, accessing, adding, updating, and removing key-value pairs.
* Nested dictionaries for more complex data structures.
* Dictionary methods for retrieving keys, values, and items, as well as using get() and pop().
* Basic set operations such as adding, removing, and checking membership.
* Set operations like union, intersection, difference, and symmetric difference.
* Set methods for adding, discarding, updating, copying, and clearing elements.

# 2-D Lists – Matrixes

In Python, a matrix can be represented as a list of lists, where each inner list represents a row of the matrix. This is often referred to as a 2-dimensional (2D) list. Here are some examples and explanations to help you understand how to work with 2D lists in Python:

## Creating and Accessing Elements in a 2D List

# Creating a 2D list (matrix)

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

# Accessing elements in the 2D list

print(matrix[0][0]) # Output: 1 (first row, first column)

print(matrix[1][2]) # Output: 6 (second row, third column)

print(matrix[2][1]) # Output: 8 (third row, second column)

**Explanation**:

* **Creating a 2D list**: matrix is a list of lists, where each inner list represents a row of the matrix.
* **Accessing elements**: You can access elements in the 2D list using two indices: matrix[row][column].

## Iterating Through a 2D List

# Creating a 2D list (matrix)

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

# Iterating through the 2D list

for row in matrix:

for element in row:

print(element, end=' ')

print()

**Explanation**:

* **Iterating through the 2D list**: The outer for loop iterates through each row, and the inner for loop iterates through each element in the row. The end=' ' argument in the print function ensures that the elements are printed on the same line, and the print() function at the end of the inner loop moves to the next line.

## Modifying Elements in a 2D List

# Creating a 2D list (matrix)

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

# Modifying elements in the 2D list

matrix[0][0] = 10

matrix[1][2] = 20

matrix[2][1] = 30

# Printing the modified matrix

for row in matrix:

print(row)

**Explanation**:

* **Modifying elements**: You can modify elements in the 2D list by assigning new values to specific indices: matrix[row][column] = new\_value.

## Adding and Removing Rows and Columns

# Creating a 2D list (matrix)

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

# Adding a new row

new\_row = [10, 11, 12]

matrix.append(new\_row)

# Removing a row

matrix.pop(1) # Removes the second row

# Adding a new column

for row in matrix:

row.append(0)

# Removing a column

for row in matrix:

row.pop(1) # Removes the second column

# Printing the modified matrix

for row in matrix:

print(row)

**Explanation**:

* **Adding a new row**: Use the append method to add a new row to the matrix.
* **Removing a row**: Use the pop method to remove a row from the matrix.
* **Adding a new column**: Iterate through each row and use the append method to add a new element to each row.
* **Removing a column**: Iterate through each row and use the pop method to remove an element from each row.

## Matrix Operations (Addition and Multiplication)

# Creating two 2D lists (matrices)

matrix1 = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

matrix2 = [

[9, 8, 7],

[6, 5, 4],

[3, 2, 1]

]

# Matrix addition

result\_addition = [

[matrix1[i][j] + matrix2[i][j] for j in range(len(matrix1[0]))]

for i in range(len(matrix1))

]

# Matrix multiplication

result\_multiplication = [

[sum(matrix1[i][k] \* matrix2[k][j] for k in range(len(matrix1))) for j in range(len(matrix2[0]))]

for i in range(len(matrix1))

]

# Printing the results

print("Matrix Addition:")

for row in result\_addition:

print(row)

print("\nMatrix Multiplication:")

for row in result\_multiplication:

print(row)

**Explanation**:

* **Matrix addition**: The result\_addition matrix is created by adding corresponding elements from matrix1 and matrix2.
* **Matrix multiplication**: The result\_multiplication matrix is created by performing the dot product of rows from matrix1 and columns from matrix2.

## Matrixes Summary

These examples demonstrate various ways to work with 2D lists (matrices) in Python:

* Creating and accessing elements in a 2D list.
* Iterating through a 2D list.
* Modifying elements in a 2D list.
* Adding and removing rows and columns.
* Performing matrix operations such as addition and multiplication.

# Classes and Objects

**Classes** are blueprints for creating objects. They encapsulate data for the object and methods to manipulate that data. An **object** is an instance of a class. When you create an object, you are instantiating a class.

## Defining a Simple Class

Let’s start with a basic example of defining a class and creating an object.

# Define a class named 'Person'

class Person:

# Constructor method to initialize the object

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

self.name = name

self.age = age

# Method to display person's details

def display(self):

print(f"Name: {self.name}, Age: {self.age}")

# Create an object of the class 'Person'

person1 = Person("Alice", 30)

# Call the display method

person1.display()

**Explanation:**

1. **Class Definition:** We define a class named Person using the class keyword.
2. **Constructor Method:** The \_\_init\_\_ method is a special method called a constructor. It initializes the object’s attributes (name and age).
3. **Method Definition:** We define a method display to print the person’s details.
4. **Object Creation:** We create an object person1 of the class Person and pass the name “Alice” and age 30 as arguments.
5. **Method Call:** We call the display method on the person1 object to print the details.

## Adding More Methods

Let’s add more methods to the class to demonstrate additional functionality.

# Define a class named 'Person'

class Person:

# Constructor method to initialize the object

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

self.name = name

self.age = age

# Method to display person's details

def display(self):

print(f"Name: {self.name}, Age: {self.age}")

# Method to update the person's age

def update\_age(self, new\_age):

self.age = new\_age

# Create an object of the class 'Person'

person1 = Person("Bob", 25)

# Call the display method

person1.display()

# Update the age and display again

person1.update\_age(26)

person1.display()

**Explanation:**

1. **Class Definition:** We define a class named Person with a constructor method and a display method.
2. **Additional Method:** We add a method update\_age to update the person’s age.
3. **Object Creation:** We create an object person1 of the class Person and pass the name “Bob” and age 25 as arguments.
4. **Method Calls:** We call the display method to print the initial details, then update the age using update\_age, and call display again to print the updated details.

## Inheritance

Inheritance allows a class to inherit attributes and methods from another class. Let’s see an example.

# Define a base class named 'Person'

class Person:

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

self.name = name

self.age = age

def display(self):

print(f"Name: {self.name}, Age: {self.age}")

# Define a derived class named 'Student' that inherits from 'Person'

class Student(Person):

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

super().\_\_init\_\_(name, age)

self.student\_id = student\_id

def display\_student(self):

print(f"Name: {self.name}, Age: {self.age}, Student ID: {self.student\_id}")

# Create an object of the derived class 'Student'

student1 = Student("Charlie", 20, "S12345")

# Call the display\_student method

student1.display\_student()

**Explanation:**

1. **Base Class Definition:** We define a base class named Person with a constructor and a display method.
2. **Derived Class Definition:** We define a derived class named Student that inherits from Person using the super() function to call the constructor of the base class.
3. **Additional Attribute:** We add an additional attribute student\_id to the Student class.
4. **Additional Method:** We add a method display\_student to print the student’s details, including the student ID.
5. **Object Creation:** We create an object student1 of the class Student and pass the name “Charlie”, age 20, and student ID “S12345” as arguments.
6. **Method Call:** We call the display\_student method on the student1 object to print the details.

## Example Database Class

Let’s design a Database class to hold employee information such as name, job name, job rank, salary, etc. We’ll use classes and objects in Python to achieve this.

**Step-by-Step Design**

1. **Define the**Employee**Class:** This class will hold individual employee details.
2. **Define the**Database**Class:** This class will manage a collection of Employee objects and provide methods to add, remove, and display employee information.

**Defining the**Employee**Class**

class Employee:

def \_\_init\_\_(self, name, job\_name, job\_rank, salary):

self.name = name

self.job\_name = job\_name

self.job\_rank = job\_rank

self.salary = salary

def display(self):

print(f"Name: {self.name}, Job Name: {self.job\_name}, Job Rank: {self.job\_rank}, Salary: ${self.salary}")

# Example of creating an Employee object

employee1 = Employee("Alice", "Software Engineer", "Senior", 90000)

employee1.display()

**Explanation:**

1. **Class Definition:** We define a class named Employee with a constructor method to initialize the attributes (name, job\_name, job\_rank, salary).
2. **Display Method:** We define a method display to print the employee’s details.
3. **Object Creation:** We create an Employee object employee1 and call the display method to print the details.

**Defining the**Database**Class**

class Database:

def \_\_init\_\_(self):

self.employees = []

def add\_employee(self, employee):

self.employees.append(employee)

def remove\_employee(self, name):

self.employees = [emp for emp in self.employees if emp.name != name]

def display\_all(self):

for emp in self.employees:

emp.display()

# Example of creating a Database object and adding employees

db = Database()

db.add\_employee(Employee("Alice", "Software Engineer", "Senior", 90000))

db.add\_employee(Employee("Bob", "Data Scientist", "Junior", 70000))

# Display all employees

db.display\_all()

# Remove an employee and display again

db.remove\_employee("Alice")

db.display\_all()

**Explanation:**

1. **Class Definition:** We define a class named Database with a constructor method to initialize an empty list of employees.
2. **Add Employee Method:** We define a method add\_employee to add an Employee object to the list.
3. **Remove Employee Method:** We define a method remove\_employee to remove an employee by name.
4. **Display All Method:** We define a method display\_all to print the details of all employees in the database.
5. **Object Creation:** We create a Database object db and add two Employee objects to it.
6. **Display All Employees:** We call the display\_all method to print the details of all employees.
7. **Remove Employee:** We remove an employee by name and display the remaining employees.

These examples should give you a good understanding of how to design a Database class to manage employee information using classes and objects in Python.

# Modules

Modules in Python are files containing Python code that can define functions, classes, and variables. They help in organizing and reusing code. Let’s go through an example of creating and using a module.

## Library Management System

Let’s create a more complex example involving multiple modules to simulate a small library management system. We’ll have the following modules:

1. book.py: Defines the Book class.
2. member.py: Defines the Member class.
3. library.py: Manages the library operations.
4. main.py: The main script to interact with the library system.

## Create the book.py Module

# book.py

class Book:

def \_\_init\_\_(self, title, author, isbn):

self.title = title

self.author = author

self.isbn = isbn

self.is\_borrowed = False

def display\_info(self):

status = "Borrowed" if self.is\_borrowed else "Available"

print(f"Title: {self.title}, Author: {self.author}, ISBN: {self.isbn}, Status: {status}")

**Explanation:**

* **Class Definition:** We define a Book class with attributes title, author, isbn, and is\_borrowed.
* **Method Definition:** We define a method display\_info to print the book’s details.

## Create the member.py Module

# member.py

class Member:

def \_\_init\_\_(self, name, member\_id):

self.name = name

self.member\_id = member\_id

self.borrowed\_books = []

def borrow\_book(self, book):

if not book.is\_borrowed:

book.is\_borrowed = True

self.borrowed\_books.append(book)

print(f"{self.name} borrowed {book.title}")

else:

print(f"{book.title} is already borrowed")

def return\_book(self, book):

if book in self.borrowed\_books:

book.is\_borrowed = False

self.borrowed\_books.remove(book)

print(f"{self.name} returned {book.title}")

else:

print(f"{self.name} did not borrow {book.title}")

def display\_borrowed\_books(self):

print(f"{self.name} has borrowed the following books:")

for book in self.borrowed\_books:

book.display\_info()

**Explanation:**

* **Class Definition:** We define a Member class with attributes name, member\_id, and borrowed\_books.
* **Method Definitions:** We define methods borrow\_book, return\_book, and display\_borrowed\_books to manage the member’s borrowed books.

## Create the library.py Module

# library.py

from book import Book

from member import Member

class Library:

def \_\_init\_\_(self):

self.books = []

self.members = []

def add\_book(self, book):

self.books.append(book)

print(f"Added book: {book.title}")

def add\_member(self, member):

self.members.append(member)

print(f"Added member: {member.name}")

def display\_books(self):

print("Library books:")

for book in self.books:

book.display\_info()

def display\_members(self):

print("Library members:")

for member in self.members:

print(f"Name: {member.name}, Member ID: {member.member\_id}")

**Explanation:**

* **Class Definition:** We define a Library class with attributes books and members.
* **Method Definitions:** We define methods add\_book, add\_member, display\_books, and display\_members to manage the library’s books and members.

## Create the main.py Script

# main.py

from book import Book

from member import Member

from library import Library

# Create a library instance

library = Library()

# Add books to the library

book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "1234567890")

book2 = Book("1984", "George Orwell", "0987654321")

library.add\_book(book1)

library.add\_book(book2)

# Add members to the library

member1 = Member("Alice", "M001")

member2 = Member("Bob", "M002")

library.add\_member(member1)

library.add\_member(member2)

# Display library books and members

library.display\_books()

library.display\_members()

# Borrow and return books

member1.borrow\_book(book1)

member1.display\_borrowed\_books()

member1.return\_book(book1)

member1.display\_borrowed\_books()

**Explanation:**

* **Import Modules:** We import the Book, Member, and Library classes from their respective modules.
* **Library Instance:** We create an instance of the Library class.
* **Add Books and Members:** We add books and members to the library.
* **Display Books and Members:** We display the library’s books and members.
* **Borrow and Return Books:** We demonstrate borrowing and returning books.

## Running the Code

To see the output, run the main.py script. The output will show the library’s books and members, and the borrowing and returning of books.

# Decorators

Decorators in Python are a powerful and flexible tool that allows you to modify the behavior of functions or methods without altering their actual code. They are implemented as higher-order functions, meaning they take a function as an argument and return a new function that typically extends or enhances the original function’s behavior. Decorators are often used for cross-cutting concerns such as logging, authentication, caching, and performance monitoring. By using the @decorator\_name syntax, you can apply a decorator to a function in a clean and readable manner. Advanced usage of decorators can involve chaining multiple decorators, passing arguments to decorators, and even creating class-based decorators for more complex scenarios. This allows for a high degree of code reuse and separation of concerns, making your codebase more modular and maintainable.

## Example

import functools

def log\_decorator(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")

result = func(\*args, \*\*kwargs)

print(f"{func.\_\_name\_\_} returned {result}")

return result

return wrapper

@log\_decorator

def add(a, b):

return a + b

print(add(3, 4))

In this example, log\_decorator logs the function call details and the result. It is defined by def log\_decorator(func) and then called by @log\_decorator. Thus when the function add runs, it will also document the running of the function.

# Generators

Generators in Python are a sophisticated tool for creating iterators in a memory-efficient manner. Unlike regular functions that return a single value and terminate, generators use the yield keyword to produce a sequence of values lazily, meaning they generate values on-the-fly and only when requested. This makes them particularly useful for handling large datasets or streams of data where storing the entire dataset in memory would be impractical. Generators maintain their state between successive calls, allowing them to resume execution right after the last yield statement. This statefulness enables complex iteration patterns and can be leveraged to implement custom iteration logic. Advanced usage of generators includes generator expressions, which provide a concise way to create generators, and the use of the send() method to inject values into a generator’s execution, enabling coroutines and more interactive data processing workflows. By utilizing generators, developers can write more efficient and readable code, especially in scenarios involving large-scale data processing or real-time data streams.

## Example

def fibonacci(n):

a, b = 0, 1

for \_ in range(n):

yield a

a, b = b, a + b

for num in fibonacci(10):

print(num)

This generator produces the first n Fibonacci numbers without storing them all in memory. You can see that the function is a generator because of the use of the yield statement. This, this code sets some variables, establishes a range based upon the passed parameter. Then Inside the loop, it yields, or returns, a, the current number, and sets a and b to the Fibonacci formula and continues the loop until the end of the range.

The for loop then calls the Fibonacci function for 10 iterations and produces the Fibonacci numbers.

# Context Managers

Context managers in Python are constructs that allow for the setup and teardown of resources in a clean and efficient manner, ensuring that resources are properly managed. They are most commonly used with the with statement, which guarantees that the resource is released or cleaned up after its block of code is executed, even if an exception occurs. This is particularly useful for managing file operations, network connections, and threading locks. A context manager is typically implemented using the \_\_enter\_\_ and \_\_exit\_\_ methods in a class. The \_\_enter\_\_ method is executed when the with block is entered, and the \_\_exit\_\_ method is executed when the block is exited.

## Example

class DatabaseConnection:

def \_\_init\_\_(self, db\_name):

self.db\_name = db\_name

def \_\_enter\_\_(self):

self.conn = self.connect\_to\_database(self.db\_name)

return self.conn

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.conn.close()

def connect\_to\_database(self, db\_name):

# Simulate a database connection

print(f"Connecting to database {db\_name}")

return self

def close(self):

print("Closing database connection")

with DatabaseConnection('my\_database') as db\_conn:

# Perform database operations

print("Performing database operations")

In this example, the DatabaseConnection class manages a simulated database connection. The \_\_enter\_\_ method establishes the connection, and the \_\_exit\_\_ method ensures that the connection is closed, even if an error occurs during the database operations. This pattern helps in writing more robust and maintainable code by abstracting resource management details.

# Concurrency

Concurrency in Python refers to the ability to run multiple tasks or operations simultaneously, which can significantly improve the performance of programs, especially those that involve I/O-bound or high-latency operations. Python provides several ways to achieve concurrency, including threading, multiprocessing, and asynchronous programming with asyncio.

## Threading

Threading allows you to run multiple threads (smaller units of a process) concurrently. However, due to the Global Interpreter Lock (GIL) in CPython, threading is more suitable for I/O-bound tasks rather than CPU-bound tasks.

**Example: Threading for I/O-bound Tasks**

import threading

import time

def print\_numbers():

for i in range(5):

print(i)

time.sleep(1)

def print\_letters():

for letter in 'abcde':

print(letter)

time.sleep(1)

thread1 = threading.Thread(target=print\_numbers)

thread2 = threading.Thread(target=print\_letters)

thread1.start()

thread2.start()

thread1.join()

thread2.join()

In this example, two threads run concurrently, one printing numbers and the other printing letters.

## Multiprocessing

Multiprocessing involves running multiple processes, each with its own Python interpreter and memory space. This bypasses the GIL and is suitable for CPU-bound tasks.

**Example: Multiprocessing for CPU-bound Tasks**

import multiprocessing

import time

import multiprocessing

import time

def compute\_square(n):

time.sleep(0)

return n \* n

if \_\_name\_\_ == "\_\_main\_\_":

with multiprocessing.Pool(processes=8) as pool:

results = pool.map(compute\_square, range(10000))

print(results)

Here, multiple processes compute the square of numbers concurrently, leveraging multiple CPU cores. It runs quite fast.

# Tic Tac Toe

Tic-tac-toe, also known as “noughts and crosses,” has a rich history that dates back to ancient civilizations. Here are some key points about its origins:

* [**Ancient Egypt**: Evidence of a similar game, called “Seega,” has been found carved into temple roofs, suggesting that the ancient Egyptians played an early version of tic-tac-toe1](https://tictactoefree.com/tips/tic-tac-toe-history-and-origins).
* [**Roman Empire**: The Romans had their own version of the game called “Terni Lapilli,” which was played with different symbols and etched onto public monuments1](https://tictactoefree.com/tips/tic-tac-toe-history-and-origins).
* [**Ancient India**: In ancient India, a game called “Pada” was played with shells or tiny stones, aiming to achieve a specific pattern1](https://tictactoefree.com/tips/tic-tac-toe-history-and-origins).

The modern version of tic-tac-toe that we know today first appeared in mid-19th century Britain. [In 1858, a popular tabloid released the game under the title "Noughts and Crosses"2](https://bing.com/search?q=tic+tac+toe+invention+history).

## Complete Source Code

The following source code uses Tkinter as the framework for displaying the graphical board and handling the mouse clicks. It also incorporates an algorithm for the move engine called minimax.

Type in this script and try to beat it!

# +------------------------------------------------------+

# |Tic Tac Toe |

# | Writen by Randall Fadler and CoPilot |

# | September 16, 2024 |

# | |

# | This Tic Tac Toe game is using the Minimax algorithm |

# | for the move engine. The Minimax algorithm is a |

# | decision-making algorithm used in game theory and |

# | artificial intelligence to determine the optimal move|

# | for a player, assuming that the opponent is also |

# | playing optimally. Can you beat it?? |

# +------------------------------------------------------+

import tkinter as tk

import random

import math

from tkinter import messagebox

class TicTacToe:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("Tic Tac Toe")

self.board = [' ' for \_ in range(9)]

self.current\_player = "X"

self.human = "X"

self.computer = "O"

self.human\_score = 0

self.computer\_score = 0

self.tie\_score = 0

self.empty\_square = ' '

self.create\_widgets()

def show\_popup(self,message):

messagebox.showinfo("Game Over", message)

def create\_widgets(self):

self.buttons = []

for i in range(9):

button = tk.Button(self.root, text=' ', font='Arial 20', width=5, height=2, command=lambda i=i: self.on\_button\_click(i))

button.grid(row=i//3, column=i%3)

self.buttons.append(button)

self.score\_label = tk.Label(self.root, text=f"Human: {self.human\_score} Computer: {self.computer\_score} Tie: {self.tie\_score}", font='Arial 15')

self.score\_label.grid(row=3, column=0, columnspan=3)

def on\_button\_click(self, index):

if self.board[index]== self.empty\_square and self.current\_player == self.human:

self.board[index] = self.human

self.buttons[index].config(text=self.human)

if self.check\_winner(self.board,self.human):

self.human\_score += 1

self.update\_score()

self.show\_popup("You Win!")

self.reset\_board()

elif ' ' not in self.board:

print('its in tie')

self.show\_popup("It's a tie!")

self.tie\_score += 1

self.update\_score()

print('its after popup')

self.reset\_board()

else:

self.current\_player = self.computer

self.computer\_move()

# Minimax algorithm

def minimax(self, board, depth, is\_maximizing):

if self.check\_winner(board, self.computer):

return 1

elif self.check\_winner(board, self.human):

return -1

elif ' ' not in board:

return 0

if is\_maximizing:

best\_score = -math.inf

for i in range(9):

if board[i]== self.empty\_square:

board[i] = self.computer

score = self.minimax(board, depth + 1, False)

board[i] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = math.inf

for i in range(9):

if board[i]== self.empty\_square:

board[i] = self.human

score = self.minimax(board, depth + 1, True)

board[i] = ' '

best\_score = min(score, best\_score)

return best\_score

# Function to make the best move

def best\_move(self, board):

best\_score = -math.inf

move = 0

for i in range(9):

if board[i]== self.empty\_square:

board[i] = self.computer

score = self.minimax(board, 0, False)

print(score)

board[i] = ' '

if score is not None and score > best\_score:

best\_score = score

print('best\_score', best\_score)

move = i

print(move)

return move

def computer\_move(self):

# available\_moves = [i for i, x in enumerate(self.board) if x== self.empty\_square]

# move = random.choice(available\_moves)

move = self.best\_move(self.board)

self.board[move] = self.computer

self.buttons[move].config(text=self.computer)

if self.check\_winner(self.board, self.computer):

self.computer\_score += 1

self.update\_score()

self.show\_popup('You loose!')

self.reset\_board()

elif ' ' not in self.board:

self.reset\_board()

else:

self.current\_player = self.human

def check\_winner(self, board, player):

win\_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)]

return any(all(self.board[i] == player for i in condition) for condition in win\_conditions)

def update\_score(self):

self.score\_label.config(text=f"Human: {self.human\_score} Computer: {self.computer\_score} Tie: {self.tie\_score}")

def reset\_board(self):

self.board = [' ' for \_ in range(9)]

for button in self.buttons:

button.config(text=' ')

self.current\_player = self.human

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

game = TicTacToe(root)

root.mainloop()

## Code Breakdown and Explanations

The minimax function in your Tic Tac Toe game follows the **Minimax algorithm**, which is a decision-making algorithm used in game theory and artificial intelligence. Here’s a brief overview of how it works:

**Minimax Algorithm**

1. **Game Tree Construction**: The algorithm constructs a game tree where each node represents a possible game state. The root node is the current state of the game, and the child nodes represent the possible moves.
2. **Recursive Evaluation**: The algorithm recursively evaluates the game tree by simulating all possible moves for both players (the maximizing player and the minimizing player).
3. **Maximizing Player**: The algorithm assumes that the maximizing player (in this case, the computer) will try to maximize their score. It selects the move that leads to the highest possible score.
4. **Minimizing Player**: Conversely, the algorithm assumes that the minimizing player (the human) will try to minimize the maximizing player’s score. It selects the move that leads to the lowest possible score for the maximizing player.
5. **Terminal States**: The algorithm evaluates terminal states (win, lose, or draw) and assigns scores:
   * **Win for Computer**: +1
   * **Win for Human**: -1
   * **Draw**: 0
6. **Backtracking**: The algorithm backtracks through the game tree, propagating the scores up to the root node. At each level, it chooses the move that maximizes or minimizes the score, depending on the player’s turn.

**Example**

In your minimax function:

* The function checks for terminal states (win, lose, or draw) and returns the corresponding score.
* If it’s the maximizing player’s turn (computer), it tries to maximize the score by exploring all possible moves and selecting the one with the highest score.
* If it’s the minimizing player’s turn (human), it tries to minimize the score by exploring all possible moves and selecting the one with the lowest score.

The Minimax algorithm ensures that the computer makes the optimal move, assuming that the human player is also playing optimally.

So with that behind us, let’s break down the functions in this Tic Tac Toe game script and explain what each one does:

\_\_init\_\_(self, root)

This is the constructor method for the TicTacToe class. It initializes the game board, sets the current player, and creates the game widgets.

* self.root = root: Assigns the root window to the self.root attribute.
* self.root.title("Tic Tac Toe"): Sets the title of the window.
* self.board = [' ' for \_ in range(9)]: Initializes the game board with 9 empty spaces.
* self.current\_player = "X": Sets the current player to “X”.
* self.human = "X": Sets the human player to “X”.
* self.computer = "O": Sets the computer player to “O”.
* self.human\_score = 0, self.computer\_score = 0, self.tie\_score = 0: Initializes the scores.
* self.empty\_square = ' ': Defines the symbol for an empty square.
* self.create\_widgets(): Calls the method to create the game widgets.

show\_popup(self, message)

Displays a popup message when the game is over.

* messagebox.showinfo("Game Over", message): Shows an information message box with the game over message.

create\_widgets(self)

Creates the game buttons and score label.

* self.buttons = []: Initializes an empty list to store the buttons.
* for i in range(9): Loops through the range of 9 to create 9 buttons.
* button = tk.Button(...): Creates a button with specific properties.
* button.grid(row=i//3, column=i%3): Places the button in a 3x3 grid.
* self.buttons.append(button): Adds the button to the list of buttons.
* self.score\_label = tk.Label(...): Creates a label to display the scores.
* self.score\_label.grid(row=3, column=0, columnspan=3): Places the score label below the buttons.

on\_button\_click(self, index)

Handles the button click event.

* if self.board[index] == self.empty\_square and self.current\_player == self.human: Checks if the clicked square is empty and if it’s the human’s turn.
* self.board[index] = self.human: Marks the square with the human’s symbol.
* self.buttons[index].config(text=self.human): Updates the button text to the human’s symbol.
* if self.check\_winner(self.board, self.human): Checks if the human has won.
* self.human\_score += 1: Increments the human’s score.
* self.update\_score(): Updates the score display.
* self.show\_popup("You Win!"): Shows a popup message that the human has won.
* self.reset\_board(): Resets the game board.
* elif ' ' not in self.board: Checks if the board is full (tie).
* self.show\_popup("It's a tie!"): Shows a popup message that the game is a tie.
* self.tie\_score += 1: Increments the tie score.
* self.update\_score(): Updates the score display.
* self.reset\_board(): Resets the game board.
* else: If it’s the computer’s turn.
* self.current\_player = self.computer: Sets the current player to the computer.
* self.computer\_move(): Calls the method for the computer’s move.

minimax(self, board, depth, is\_maximizing)

Implements the Minimax algorithm to determine the optimal move.

* if self.check\_winner(board, self.computer): Checks if the computer has won.
* return 1: Returns a score of 1 if the computer has won.
* elif self.check\_winner(board, self.human): Checks if the human has won.
* return -1: Returns a score of -1 if the human has won.
* elif ' ' not in board: Checks if the board is full (tie).
* return 0: Returns a score of 0 if the game is a tie.
* if is\_maximizing: If it’s the maximizing player’s turn (computer).
* best\_score = -math.inf: Initializes the best score to negative infinity.
* for i in range(9): Loops through the board.
* if board[i] == self.empty\_square: Checks if the square is empty.
* board[i] = self.computer: Marks the square with the computer’s symbol.
* score = self.minimax(board, depth + 1, False): Recursively calls the minimax function for the minimizing player.
* board[i] = ' ': Resets the square.
* best\_score = max(score, best\_score): Updates the best score.
* return best\_score: Returns the best score.
* else: If it’s the minimizing player’s turn (human).
* best\_score = math.inf: Initializes the best score to positive infinity.
* for i in range(9): Loops through the board.
* if board[i] == self.empty\_square: Checks if the square is empty.
* board[i] = self.human: Marks the square with the human’s symbol.
* score = self.minimax(board, depth + 1, True): Recursively calls the minimax function for the maximizing player.
* board[i] = ' ': Resets the square.
* best\_score = min(score, best\_score): Updates the best score.
* return best\_score: Returns the best score.

best\_move(self, board)

Determines the best move for the computer using the Minimax algorithm.

* best\_score = -math.inf: Initializes the best score to negative infinity.
* move = 0: Initializes the best move to 0.
* for i in range(9): Loops through the board.
* if board[i] == self.empty\_square: Checks if the square is empty.
* board[i] = self.computer: Marks the square with the computer’s symbol.
* score = self.minimax(board, 0, False): Calls the minimax function to get the score for the move.
* board[i] = ' ': Resets the square.
* if score is not None and score > best\_score: Checks if the score is better than the best score.
* best\_score = score: Updates the best score.
* move = i: Updates the best move.
* return move: Returns the best move.

computer\_move(self)

Handles the computer’s move.

* move = self.best\_move(self.board): Gets the best move for the computer.
* self.board[move] = self.computer: Marks the square with the computer’s symbol.
* self.buttons[move].config(text=self.computer): Updates the button text to the computer’s symbol.
* if self.check\_winner(self.board, self.computer): Checks if the computer has won.
* self.computer\_score += 1: Increments the computer’s score.
* self.update\_score(): Updates the score display.
* self.show\_popup('You lose!'): Shows a popup message that the human has lost.
* self.reset\_board(): Resets the game board.
* elif ' ' not in self.board: Checks if the board is full (tie).
* self.reset\_board(): Resets the game board.
* else: If it’s the human’s turn.
* self.current\_player = self.human: Sets the current player to the human.

check\_winner(self, board, player)

Checks if a player has won the game.

* win\_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)]: Defines the winning conditions.
* return any(all(self.board[i] == player for i in condition) for condition in win\_conditions): Checks if any of the winning conditions are met.

update\_score(self)

Updates the score display.

* self.score\_label.config(text=f"Human: {self.human\_score} Computer: {self.computer\_score} Tie: {self.tie\_score}"): Updates the score label text.

reset\_board(self)

Resets the game board.

* self.board = [' ' for \_ in range(9)]: Resets the board to empty squares.
* for button in self.buttons: Loops through the buttons.
* button.config(text=' '): Resets the button text to empty.
* self.current\_player = self.human: Sets the current player to the human.

# Conway’s Game of Life

Conway’s Game of Life is a cellular automaton devised by mathematician John Conway. It’s a zero-player game, meaning its evolution is determined by its initial state, requiring no further input. Let’s create a Python script for Conway’s Game of Life and then break down the code with explanations.

## Complete Source Code

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.animation as animation

# Define the size of the grid

GRID\_SIZE = 50

# Initialize the grid with random values

grid = np.random.choice([0, 1], GRID\_SIZE \* GRID\_SIZE, p=[0.8, 0.2]).reshape(GRID\_SIZE, GRID\_SIZE)

def update(frameNum, img, grid):

# Copy the grid to apply the rules

newGrid = grid.copy()

for i in range(GRID\_SIZE):

for j in range(GRID\_SIZE):

# Count the number of live neighbors

total = int((grid[i, (j-1)%GRID\_SIZE] + grid[i, (j+1)%GRID\_SIZE] +

grid[(i-1)%GRID\_SIZE, j] + grid[(i+1)%GRID\_SIZE, j] +

grid[(i-1)%GRID\_SIZE, (j-1)%GRID\_SIZE] + grid[(i-1)%GRID\_SIZE, (j+1)%GRID\_SIZE] +

grid[(i+1)%GRID\_SIZE, (j-1)%GRID\_SIZE] + grid[(i+1)%GRID\_SIZE, (j+1)%GRID\_SIZE]))

# Apply Conway's rules

if grid[i, j] == 1:

if (total < 2) or (total > 3):

newGrid[i, j] = 0

else:

if total == 3:

newGrid[i, j] = 1

# Update the data of the image

img.set\_data(newGrid)

grid[:] = newGrid[:]

return img,

# Set up the animation

fig, ax = plt.subplots()

img = ax.imshow(grid, interpolation='nearest')

ani = animation.FuncAnimation(fig, update, fargs=(img, grid), frames=10, interval=200, save\_count=50)

plt.show()

## Code Breakdown and Explanation

1. **Import Libraries:**

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.animation as animation

* We import the necessary libraries: numpy for numerical operations, matplotlib.pyplot for plotting, and matplotlib.animation for creating animations.

1. **Define the Grid Size:**

**Python**

GRID\_SIZE = 50

* We define the size of the grid. In this case, it’s a 50x50 grid.

1. **Initialize the Grid:**

grid = np.random.choice([0, 1], GRID\_SIZE \* GRID\_SIZE, p=[0.8, 0.2]).reshape(GRID\_SIZE, GRID\_SIZE)

* We initialize the grid with random values, where 80% of the cells are dead (0) and 20% are alive (1).
* np.random.choice([0, 1], ...): This function randomly selects elements from the list [0, 1].
* GRID\_SIZE \* GRID\_SIZE: This specifies the total number of elements to be selected. If GRID\_SIZE is 50, then GRID\_SIZE \* GRID\_SIZE is 2500, meaning 2500 elements will be selected.
* p=[0.8, 0.2]: This specifies the probabilities associated with each element in the list [0, 1]. Here, 0 has a probability of 0.8 (80%) and 1 has a probability of 0.2 (20%). This means that 80% of the selected elements will be 0 and 20% will be 1.
* .reshape(GRID\_SIZE, GRID\_SIZE):
  + This method reshapes the 1D array of 2500 elements into a 2D array (matrix) with dimensions GRID\_SIZE x GRID\_SIZE. If GRID\_SIZE is 50, the resulting matrix will be 50 rows by 50 columns.

1. **Update Function:**

def update(frameNum, img, grid):

newGrid = grid.copy()

for i in range(GRID\_SIZE):

for j in range(GRID\_SIZE):

total = int((grid[i, (j-1)%GRID\_SIZE] + grid[i, (j+1)%GRID\_SIZE] +

grid[(i-1)%GRID\_SIZE, j] + grid[(i+1)%GRID\_SIZE, j] +

grid[(i-1)%GRID\_SIZE, (j-1)%GRID\_SIZE] + grid[(i-1)%GRID\_SIZE, (j+1)%GRID\_SIZE] +

grid[(i+1)%GRID\_SIZE, (j-1)%GRID\_SIZE] + grid[(i+1)%GRID\_SIZE, (j+1)%GRID\_SIZE]))

if grid[i, j] == 1:

if (total < 2) or (total > 3):

newGrid[i, j] = 0

else:

if total == 3:

newGrid[i, j] = 1

img.set\_data(newGrid)

grid[:] = newGrid[:]

return img,

* The update function is called for each frame of the animation. It applies Conway’s rules to update the grid:
  + **Live Cell:** A live cell with fewer than 2 or more than 3 live neighbors dies.
  + **Dead Cell:** A dead cell with exactly 3 live neighbors becomes alive.
* The grid is updated, and the image data is set to the new grid.

1. **Set Up the Animation:**

fig, ax = plt.subplots()

img = ax.imshow(grid, interpolation='nearest')

ani = animation.FuncAnimation(fig, update, fargs=(img, grid), frames=10, interval=200, save\_count=50)

plt.show()

* We set up the animation using FuncAnimation from matplotlib.animation. The update function is called for each frame, and the interval between frames is set to 200 milliseconds.
* Finally, we display the animation using plt.show().