

Python Notes: Days 1-10 (Angela Yu's 100 Days of Code)

Day 1: Working with Variables to Manage Data

Core Concepts:

- Printing to the console.
- String manipulation and concatenation.
- The `input()` function to get user data.
- Variables to store data.
- Debugging common errors.

1. The `print()` Function

The `print()` function is used to display output to the console. The text (a "string") to be printed must be enclosed in quotes (single `'` or double `"`).

```
# Prints a string to the console
print("Hello, World!")

# You can also print multiple lines
print("Hello, world!")
print("This is my first Python program.")
```

2. String Manipulation

You can combine strings (a process called **concatenation**) using the `+` operator. You can also include special characters like the newline character `\n`.

```
# String concatenation
print("Hello" + " " + "Angela") # Output: Hello Angela

# Using the newline character
print("Hello!\nThis will be on a new line.")
# Output:
# Hello!
# This will be on a new line.
```

3. The input() Function

This function prompts the user for an input and returns whatever they type as a **string**.

```
# The string inside input() is the prompt shown to the user
input("What is your name? ")

# You can combine print() and input()
print("Hello, " + input("What is your name? ") + "!")
```

4. Python Variables

Variables are containers for storing data values. You assign a value to a variable using the `=` operator.

- **Naming Rules:**
 - Must start with a letter or underscore (`_`).
 - Cannot start with a number.
 - Can only contain alpha-numeric characters and underscores (A-z, 0-9, and `_`).
 - Names are case-sensitive (`age` is different from `Age`).
 - Use `snake_case` for variable names (all lowercase with underscores between words).

```
# Assigning a value to a variable
name = "Jack"
print(name)

# The value of a variable can change
name = "Angela"
print(name)

# Using input() to set a variable's value
user_name = input("What is your name? ")
length = len(user_name)
print(length)
```



Day 1 Project: Band Name Generator

This project combines all the concepts from Day 1.

1. **Greet the user:** Use `print()` .
2. **Ask for the city they grew up in:** Use `input()` and store it in a variable (e.g., `city`).
3. **Ask for the name of their pet:** Use `input()` and store it in another variable (e.g., `pet_name`).
4. **Combine the names:** Use string concatenation to create the band name.
5. **Print the result:** Use `print()` to show the user their new band name.

```
#1. Create a greeting for your program.
print("Welcome to the Band Name Generator.")

#2. Ask the user for the city that they grew up in.
city = input("What's the name of the city you grew up in?\n")

#3. Ask the user for the name of a pet.
pet_name = input("What's your pet's name?\n")

#4. Combine the name of their city and pet and show them their band name.
print("Your band name could be " + city + " " + pet_name)
```

Day 2: Understanding Data Types and How to Manipulate Strings

Core Concepts:

- Primitive Data Types: **String**, **Integer**, **Float**, **Boolean**.

- Type checking with `type()` .
- Type conversion (casting).
- Mathematical operators.
- f-Strings for easy string formatting.

1. Data Types

- **String:** A sequence of characters. `"Hello"` , `"123"` .
- **Integer:** A whole number without decimals. `123` , `45` , `-10` .
- **Float:** A number with a decimal point. `3.14` , `10.5` .
- **Boolean:** Represents truth values, either `True` or `False` .

```
# String subscripting (pulling out a character)
```

```
print("Hello"[0]) # Output: H
```

```
print("Hello"[4]) # Output: o
```

```
# Integer
```

```
print(123 + 456) # Output: 579
```

```
# Use underscores for large numbers; Python ignores them
```

```
large_number = 123_456_789
```

```
print(large_number) # Output: 123456789
```

```
# Float
```

```
pi = 3.14159
```

```
# Boolean
```

```
is_true = True
```

```
is_false = False
```

2. Type Errors, Checking, and Conversion

You cannot concatenate a string with a number without converting it first.

```

# This will cause a TypeError
# num_char = len(input("What is your name?"))
# print("Your name has " + num_char + " characters.")

# Use type() to check the data type
num_char = len(input("What is your name?"))
print(type(num_char)) # Output: <class 'int'>

# Convert the integer to a string to fix the error (Type Casting)
new_num_char = str(num_char)
print("Your name has " + new_num_char + " characters.")

# Other conversions
a = float(123)      # a is 123.0
b = int(70.8)       # b is 70 (data loss!)

```

3. Mathematical Operators

- + : Addition
- - : Subtraction
- * : Multiplication
- / : Division (always results in a float)
- ** : Exponent (power of)
- Order of operations (PEMDAS) is followed: () ** * / + -

```

3 + 5    # 8
7 - 4    # 3
3 * 2    # 6
6 / 3    # 2.0
2 ** 3   # 8

```

4. f-Strings

An **f-String** makes it easy to embed variables and expressions inside a string. Just put an `f` before the opening quote.

```

score = 0
height = 1.8
is_winning = True

# Without f-String (messy)
# print("your score is " + str(score) + ", your height is " + str(height))

# With f-String (clean and easy)
print(f"your score is {score}, your height is {height}, you are winning is {is_winning}")

```

Day 2 Project: Tip Calculator

This project applies type conversion and f-strings to calculate a tip.

1. **Welcome message:** Use `print()` .
2. **Get total bill:** Use `input()` and convert it to a **float**.
3. **Get tip percentage:** Use `input()` and convert it to an **integer**.
4. **Get number of people:** Use `input()` and convert it to an **integer**.
5. **Calculate:**
 - `tip_as_percent = tip_percentage / 100`
 - `total_tip_amount = bill * tip_as_percent`
 - `total_bill = bill + total_tip_amount`
 - `bill_per_person = total_bill / people`
6. **Format and print the result:** Use an f-string to display the amount each person should pay, formatted to 2 decimal places.

```

print("Welcome to the tip calculator.")
bill = float(input("What was the total bill? $"))
tip = int(input("What percentage tip would you like to give? 10, 12, or 15? "))
people = int(input("How many people to split the bill? "))

tip_as_percent = tip / 100
total_tip_amount = bill * tip_as_percent
total_bill = bill + total_tip_amount
bill_per_person = total_bill / people

# Format the result to 2 decimal places
final_amount = "{:.2f}".format(bill_per_person)

print(f"Each person should pay: ${final_amount}")

```

Day 3: Control Flow with `if/else` and Logical Operators

Core Concepts:

- Conditional statements: `if` , `else` , `elif` .
- Comparison operators.
- Nested `if/else` statements.
- Logical operators: `and` , `or` , `not` .

1. `if/else` Statements

These statements allow your code to make decisions. The code inside the `if` block runs only if the condition is `True` . Otherwise, the code inside the `else` block runs.

```
water_level = 50
if water_level > 80:
    print("Drain water")
else:
    print("Continue")
```

2. Comparison Operators

- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to
- `==` : Equal to (check for equality)
- `!=` : Not equal to

```
# Check if a number is even or odd using the modulo operator (%)
number = int(input("Which number do you want to check? "))

if number % 2 == 0:
    print("This is an even number.")
else:
    print("This is an odd number.")
```

3. Nested if/else and elif

You can place `if/else` statements inside other `if/else` statements. The `elif` (else if) keyword lets you check multiple conditions in sequence.

```
# Nested if/else
height = int(input("What is your height in cm? "))
if height >= 120:
    print("You can ride the rollercoaster!")
    age = int(input("What is your age? "))
    if age < 12:
        print("Please pay $5.")
    else:
        print("Please pay $7.")
else:
    print("Sorry, you have to grow taller before you can ride.")

# Using elif for cleaner code
height = int(input("What is your height in cm? "))
if height >= 120:
    print("You can ride the rollercoaster!")
    age = int(input("What is your age? "))
    if age < 12:
        print("Child tickets are $5.")
    elif age <= 18:
        print("Youth tickets are $7.")
    else:
        print("Adult tickets are $12.")
else:
    print("Sorry, you have to grow taller before you can ride.")
```

4. Logical Operators

- A and B : True only if both A **and** B are True.
- C or D : True if either C **or** D (or both) are True.
- not E : Inverts the boolean value of E. not True is False .


```
# Example with logical operators
age = int(input("What is your age? "))
has_photo_id = True # A boolean variable

if age >= 18 and has_photo_id:
    print("You can buy a lottery ticket.")
else:
    print("You are not eligible.")
```

Day 3 Project: Treasure Island

This is a "choose your own adventure" game that uses multiple nested `if/elif/else` statements to guide the user through a story. The user's choices, captured with `input()`, determine the outcome.

```
print("Welcome to Treasure Island.")
print("Your mission is to find the treasure.")

choice1 = input('You\'re at a cross road. Where do you want to go? Type "left" or "right"\n').lower()

if choice1 == "left":
    choice2 = input('You come to a lake. There is an island in the middle of the lake. Type "wait" or "swim"\n').lower()
    if choice2 == "wait":
        choice3 = input("You arrive at the island unharmed. There is a house with 3 doors. One red, one blue and one yellow. Which one do you want to open?\n").lower()
        if choice3 == "red":
            print("It's a room full of fire. Game Over.")
        elif choice3 == "yellow":
            print("You found the treasure! You Win!")
        elif choice3 == "blue":
            print("You enter a room of beasts. Game Over.")
        else:
            print("You chose a door that doesn't exist. Game Over.")
    else:
        print("You get attacked by an angry trout. Game Over.")
else:
    print("You fell into a hole. Game Over.")
```

Day 4: Randomisation and Python Lists

Core Concepts:

- The `random` module.
- Python **Lists**: a data structure for storing ordered collections of items.
- Index errors.
- Nested lists.

1. The `random` Module

Python's `random` module allows you to generate pseudo-random numbers. You must **import** it first.

```
import random

# Generates a random integer between a and b (inclusive)
random_integer = random.randint(1, 10)
print(random_integer)

# Generates a random floating point number between 0.0 and 1.0 (not including 1.0)
random_float = random.random()
print(random_float)

# To get a float in a larger range, e.g., 0 to 5
random_float_in_range = random.random() * 5
print(random_float_in_range)
```

2. Lists

A **list** is a data structure that can hold multiple items in a specific order. It's mutable, meaning you can change its contents.

```
# Creating a list
fruits = ["Apple", "Banana", "Cherry"]
states_of_america = ["Delaware", "Pennsylvania", "New Jersey"]

# Accessing items by index (starts at 0)
print(states_of_america[0]) # Output: Delaware
print(states_of_america[-1]) # Output: New Jersey (negative index starts from the end)

# Modifying items in a list
states_of_america[1] = "Pencilvania"
print(states_of_america)

# Adding items to a list
states_of_america.append("Georgia") # Adds to the end
states_of_america.extend(["Ohio", "Iowa"]) # Adds another list to the end
print(states_of_america)
```

3. Index Errors

An `IndexError: list index out of range` occurs if you try to access an element at an index that doesn't exist. For a list with `n` items, the valid indices are `0` to `n-1`.

4. Nested Lists

You can put lists inside other lists.

```
fruits = ["Strawberries", "Nectarines", "Apples"]
vegetables = ["Spinach", "Kale", "Tomatoes"]

dirty_dozen = [fruits, vegetables]

print(dirty_dozen)
# Output: [['Strawberries', 'Nectarines', 'Apples'], ['Spinach', 'Kale', 'Tomatoes']]

print(dirty_dozen[1][1]) # Output: Kale (second list, second item)
```



Day 4 Project: Rock, Paper, Scissors

This project uses random number generation and lists to create a game against the computer.

1. **Store the choices:** Create a list of the game images (rock, paper, scissors).

2. **Get user input:** Ask the user for their choice (0 for Rock, 1 for Paper, 2 for Scissors) and convert it to an integer.
3. **Generate computer's choice:** Use `random.randint(0, 2)` to get a random choice for the computer.
4. **Display choices:** Print the ASCII art for both the user's and the computer's choice using the list.
5. **Determine the winner:** Use a series of `if/elif/else` statements to compare the user's choice and the computer's choice and declare a winner based on the game's rules.

```

import random

rock = '''
    _____
   ---'   ____ )
          (_____)
          (_____)
          (_____)
   ---.__(_____)
   ...

# ... (paper and scissors art omitted for brevity)

game_images = [rock, paper, scissors]

user_choice = int(input("What do you choose? Type 0 for Rock, 1 for Paper or 2 for Scissors.\n"))

if user_choice >= 3 or user_choice < 0:
    print("You typed an invalid number, you lose!")
else:
    print(game_images[user_choice])
    computer_choice = random.randint(0, 2)
    print("Computer chose:")
    print(game_images[computer_choice])

    if user_choice == 0 and computer_choice == 2:
        print("You win!")
    elif computer_choice == 0 and user_choice == 2:
        print("You lose")
    elif computer_choice > user_choice:
        print("You lose")
    elif user_choice > computer_choice:
        print("You win!")
    elif computer_choice == user_choice:
        print("It's a draw")

```

Day 5: Python Loops

Core Concepts:

- for loops for iterating over lists.

- The `range()` function.
- Calculating sums and averages with loops.

1. for Loops

A `for` loop is used for iterating over a sequence (like a list, tuple, dictionary, set, or string).

```
fruits = ["Apple", "Peach", "Pear"]
for fruit in fruits:
    print(fruit)
    print(fruit + " Pie")
print(fruits) # This is outside the loop, so it only runs once
```

2. The `range()` Function

The `range()` function generates a sequence of numbers, which is often used to control how many times a loop runs.

- `range(start, stop, step)`
 - **start** (optional): The starting number. Defaults to 0.
 - **stop** (required): The number to stop *before*. The range goes up to `stop - 1`.
 - **step** (optional): The increment amount. Defaults to 1.

```
# Loop 10 times (from 0 to 9)
```

```
for number in range(10):
    print(number)
```

```
# Loop from 1 to 10
```

```
for number in range(1, 11):
    print(number)
```

```
# Loop with a step
```

```
for number in range(1, 11, 3): # 1, 4, 7, 10
    print(number)
```

```
# Calculate the sum of numbers from 1 to 100
```

```
total = 0
for number in range(1, 101):
    total += number
print(total) # Output: 5050
```



Day 5 Project: Password Generator

This project uses `for` loops and the `random` module to generate a random password based on user-specified criteria.

1. **Define characters:** Create lists for letters, numbers, and symbols.
2. **Get user input:** Ask how many letters, symbols, and numbers they want in their password.
Convert inputs to integers.
3. **Generate the password (ordered):**
 - Use a `for` loop with `range()` to pick random letters and append them to a password list.
 - Do the same for symbols and numbers.
4. **Shuffle the password:**
 - Use `random.shuffle()` to randomize the order of the characters in the password list.
5. **Convert to string:**
 - Use another `for` loop to build the final password string from the shuffled list.
6. **Print the result.**

```

import random
letters = ['a', 'b', 'c', ..., 'Z']
numbers = ['0', '1', '2', ..., '9']
symbols = ['!', '#', '$', '%', '&', '(', ')', '*', '+']

print("Welcome to the PyPassword Generator!")
nr_letters = int(input("How many letters would you like in your password?\n"))
nr_symbols = int(input(f"How many symbols would you like?\n"))
nr_numbers = int(input(f"How many numbers would you like?\n"))

password_list = []

for char in range(1, nr_letters + 1):
    password_list.append(random.choice(letters))

for char in range(1, nr_symbols + 1):
    password_list.append(random.choice(symbols))

for char in range(1, nr_numbers + 1):
    password_list.append(random.choice(numbers))

# Shuffle the list
random.shuffle(password_list)

# Convert list back to a string
password = ""
for char in password_list:
    password += char

print(f"Your password is: {password}")

```

Day 6: Python Functions & Karel

Core Concepts:

- Defining and calling your own functions.
- The importance of **indentation** in Python.
- while loops.

(Note: Much of this day uses Reeborg's World, a specific coding environment to visualize concepts. The notes below focus on the Python syntax itself.)

1. Defining and Calling Functions

Functions allow you to bundle up code that performs a specific task. This makes your code more organized, reusable, and readable.

- Use the `def` keyword to define a function.
- The code inside the function must be **indented**.
- Call the function by writing its name followed by parentheses `()`.

```
# Defining the function
```

```
def my_function():  
    print("Hello")  
    print("Bye")
```

```
# Calling the function
```

```
my_function()
```

2. Indentation

Python does **not** use curly braces `{}` to define blocks of code like other languages. It uses **indentation** (whitespace at the beginning of a line). A standard convention is to use **4 spaces** for each level of indentation. Incorrect indentation will cause an `IndentationError`.

3. while Loops

A `while` loop will continue to execute a block of code as long as a certain condition is `True`.

Warning: If the condition never becomes `False`, you will create an **infinite loop**.

```
# This loop will run as long as the number is less than 5
```

```
number = 0
```

```
while number < 5:
```

```
    print(f"The number is {number}")
```

```
    number += 1 # Important: increment the number to eventually stop the loop
```

```
print("Loop finished.")
```



Day 6 Project: Escaping the Maze

This project is done in Reeborg's World. The goal is to write a program that can solve any maze. The key logic involves using a combination of `if/else` conditions and `while` loops to navigate Karel the robot. The logic often follows a "right-hand rule" for solving mazes.

Conceptual Python code:

```
# This is a conceptual representation of the maze logic
def turn_right():
    # Code to make the robot turn right (e.g., turn_left() three times)
    pass

# ... other robot commands like move(), at_goal(), wall_on_right(), front_is_clear()

while not at_goal():
    if wall_on_right():
        if front_is_clear():
            move()
        else:
            turn_left()
    else: # No wall on the right
        turn_right()
        move()
```

Day 7: Hangman Project - Part 1

Core Concepts:

- Breaking down a complex problem into a flowchart and smaller steps.
- Using `for` loops and `if` statements together.
- Checking if an item is in a list.

Project Steps

The goal of this day is to build the foundational logic for the Hangman game.

1. Setup:

- Create a word list.

- Use `random.choice()` to pick a random word from the list.
- Create a list called `display` with an underscore `_` for each letter in the chosen word.

2. Core Game Loop:

- Use a `while` loop to let the user keep guessing until the game is over (they've won or lost).
- Ask the user to guess a letter using `input()` and convert it to lowercase.

3. Check the Guess:

- Use a `for` loop to iterate through the `chosen_word`.
- Inside the loop, use an `if` statement to check if the current letter matches the user's `guess`.
- If it matches, replace the underscore in the `display` list at the same position with the guessed letter.

4. Display the result:

- Print the `display` list to show the user their progress.

```

import random

word_list = ["aardvark", "baboon", "camel"]
chosen_word = random.choice(word_list)
word_length = len(chosen_word)

#Create blanks
display = []
for _ in range(word_length):
    display += "_"

#TODO-1: - Use a while loop to let the user guess again.
#The loop should only stop once the user has guessed all the letters in the chosen_word and 'display' has no more blanks.
end_of_game = False

while not end_of_game:
    guess = input("Guess a letter: ").lower()

    #Check guessed letter
    for position in range(word_length):
        letter = chosen_word[position]
        if letter == guess:
            display[position] = letter

    print(display)

    #Check if there are no more "_" left in 'display'.
    if "_" not in display:
        end_of_game = True
        print("You win.")

```

Day 8: Functions with Inputs

Core Concepts:

- Functions that accept inputs (**parameters** and **arguments**).
- **Positional** vs. **Keyword** arguments.

1. Functions with Parameters

You can pass data into a function. The variable names inside the function definition's parentheses are called **parameters**.

```
# 'name' is the parameter
def greet(name):
    print(f"Hello, {name}")
    print(f"How do you do, {name}?")

# "Angela" is the argument being passed to the function
greet("Angela")
```

2. Positional vs. Keyword Arguments

- **Positional Arguments:** The arguments are matched to parameters based on their order. This is the default.
- **Keyword Arguments:** You explicitly state which parameter the argument is for. This makes the code more readable and the order doesn't matter.

```
# A function with multiple parameters
def greet_with(name, location):
    print(f"Hello {name}")
    print(f"What is it like in {location}?")

# Calling with positional arguments
greet_with("Jack Bauer", "Nowhere")

# Calling with keyword arguments (order can be switched)
greet_with(location="London", name="Angela")
```

Day 8 Project: Caesar Cipher

This project applies functions with inputs to encrypt and decrypt messages.

1. **Create a combined function:** Define a single function called `caesar()` that takes the `start_text`, `shift_amount`, and `cipher_direction` ('encode' or 'decode') as inputs.
2. **Handle the shift:** If the direction is 'decode', multiply the `shift_amount` by -1 to reverse the shift.
3. **Process text:** Loop through each character in the `start_text`.
 - If it's in the alphabet list, find its position.

- Calculate the new position by adding the (positive or negative) shift amount.
- Get the new character from the alphabet list and add it to the final text.
- If the character is not in the alphabet (e.g., a number or symbol), just add it to the final text without changing it.

4. **Print the result:** Print the encoded or decoded text.

5. **Add a game loop:** Use a `while` loop to ask the user if they want to go again.

```
alphabet = ['a', 'b', ..., 'z', 'a', 'b', ..., 'z'] # Doubled for wrapping
```

```
def caesar(start_text, shift_amount, cipher_direction):
```

```
    end_text = ""
```

```
    if cipher_direction == "decode":
```

```
        shift_amount *= -1 # Reverse the shift
```

```
    for char in start_text:
```

```
        if char in alphabet:
```

```
            position = alphabet.index(char)
```

```
            new_position = position + shift_amount
```

```
            end_text += alphabet[new_position]
```

```
        else:
```

```
            end_text += char
```

```
    print(f"The {cipher_direction}d text is {end_text}")
```

```
should_continue = True
```

```
while should_continue:
```

```
    direction = input("Type 'encode' to encrypt, type 'decode' to decrypt:\n")
```

```
    text = input("Type your message:\n").lower()
```

```
    shift = int(input("Type the shift number:\n"))
```

```
    shift = shift % 26 # Handle large shift numbers
```

```
    caesar(start_text=text, shift_amount=shift, cipher_direction=direction)
```

```
    result = input("Type 'yes' if you want to go again. Otherwise type 'no'.\n")
```

```
    if result == "no":
```

```
        should_continue = False
```

```
    print("Goodbye")
```

Day 9: Dictionaries & Nesting

Core Concepts:

- **Dictionaries:** A data structure for storing key-value pairs.
- Nesting lists and dictionaries.

1. Dictionaries

Dictionaries store data in {key: value} pairs. They are unordered, mutable, and indexed by a unique key.

```
# Creating a dictionary
programming_dictionary = {
    "Bug": "An error in a program that prevents it from running as expected.",
    "Function": "A piece of code that you can easily call over and over again.",
}

# Accessing a value by its key
print(programming_dictionary["Bug"])

# Adding a new item
programming_dictionary["Loop"] = "The action of doing something over and over again."
print(programming_dictionary)

# Creating an empty dictionary
empty_dictionary = {}

# Wiping an existing dictionary
# programming_dictionary = {}

# Looping through a dictionary
for key in programming_dictionary:
    print(key) # Prints the key
    print(programming_dictionary[key]) # Prints the value
```

2. Nesting

You can nest lists within dictionaries and dictionaries within dictionaries.

```

# Nesting a List in a Dictionary
travel_log = {
    "France": ["Paris", "Lille", "Dijon"],
    "Germany": ["Berlin", "Hamburg", "Stuttgart"],
}

# Nesting a Dictionary in a Dictionary
travel_log_detailed = {
    "France": {"cities_visited": ["Paris", "Lille"], "total_visits": 12},
    "Germany": {"cities_visited": ["Berlin", "Hamburg"], "total_visits": 5},
}

# Nesting a Dictionary inside a List
travel_log_list = [
    {
        "country": "France",
        "cities_visited": ["Paris", "Lille"],
        "total_visits": 12
    },
    {
        "country": "Germany",
        "cities_visited": ["Berlin", "Hamburg"],
        "total_visits": 5
    },
]

```

Day 9 Project: Secret Auction Program

This project uses a dictionary to store bids from different people and then determines the highest bidder.

1. **Initialize:** Create an empty dictionary to store bids, e.g., `bids = {}` .
2. **Loop for Bidders:** Create a `while` loop that continues as long as there are more bidders.
3. **Get Input:** Inside the loop, ask for the bidder's `name` and their `bid` amount (as an integer).
4. **Add to Dictionary:** Add the name and bid to the `bids` dictionary: `bids[name] = bid` .
5. **Check for More Bidders:** Ask if there are other users who want to bid. If 'no', exit the loop.
6. **Find the Winner:** After the loop, create a function to iterate through the `bids` dictionary.
 - Keep track of the `highest_bid` and the `winner` .
 - For each bidder, compare their bid to the current `highest_bid` . If it's higher, update `highest_bid` and `winner` .
7. **Announce Winner:** Print the name of the winner and the amount they bid.


```

# (Code assumes a function to clear the console is available)
bids = {}
bidding_finished = False

def find_highest_bidder(bidding_record):
    highest_bid = 0
    winner = ""
    for bidder in bidding_record:
        bid_amount = bidding_record[bidder]
        if bid_amount > highest_bid:
            highest_bid = bid_amount
            winner = bidder
    print(f"The winner is {winner} with a bid of ${highest_bid}")

while not bidding_finished:
    name = input("What is your name?: ")
    price = int(input("What is your bid?: $"))
    bids[name] = price
    should_continue = input("Are there any other bidders? Type 'yes' or 'no'.\n")
    if should_continue == "no":
        bidding_finished = True
        find_highest_bidder(bids)
    elif should_continue == "yes":
        # clear() # Clear the console screen
        pass

```

Day 10: Functions with Outputs

Core Concepts:

- The `return` keyword to get an output from a function.
- **Docstrings** for documenting functions.

1. Functions with Outputs (`return`)

Functions can process inputs and then **return** a result that can be stored in a variable or used elsewhere. The `return` keyword immediately exits the function and sends back the specified value.

```
def format_name(f_name, l_name):
    # Use the .title() method to capitalize the first letter of each word
    formatted_f_name = f_name.title()
    formatted_l_name = l_name.title()
    return f"{formatted_f_name} {formatted_l_name}"

# The returned value is stored in the variable
formatted_string = format_name("aNgElA", "yU")
print(formatted_string) # Output: Angela Yu
```

A function can have multiple `return` statements (e.g., inside an `if/else`), but it will exit as soon as it hits the first one.

2. Docstrings

Docstrings are strings placed as the very first line inside a function definition (using triple quotes `"""`) that explain what the function does. They are a crucial part of writing good, maintainable code.

```
def format_name(f_name, l_name):
    """
    Takes a first and last name and formats it to title case.
    Returns the formatted full name as a string.
    """
    if f_name == "" or l_name == "":
        return "You didn't provide valid inputs."

    formatted_f_name = f_name.title()
    formatted_l_name = l_name.title()
    return f"{formatted_f_name} {formatted_l_name}"

# You can access the docstring later
# print(format_name.__doc__)
```



Day 10 Project: Calculator

This project builds a calculator that uses a dictionary to map operation symbols to their respective functions, and uses functions with `return` values.

1. **Define Basic Functions:** Create functions for add, subtract, multiply, and divide. Each function should take two numbers as input and `return` the result.

2. **Create Operations Dictionary:** Create a dictionary where keys are the symbols (+ , - , * , /) and the values are the names of the functions you just created.

3. **Create the Calculator Function:**

- Prompt for the first number (num1).
- Loop through the dictionary keys to show the user the available operations.
- Create a while loop to allow for continuous calculations.
- Inside the loop:
 - Ask for the operation symbol.
 - Ask for the next number (num2).
 - Get the appropriate function from the dictionary using the symbol as the key.
 - Call that function with num1 and num2 to get the answer .
 - Print the calculation.
 - Ask the user if they want to continue calculating with the answer , start a new calculation, or exit. If they continue, set num1 equal to the answer and repeat the loop.

```

def add(n1, n2):
    return n1 + n2

def subtract(n1, n2):
    return n1 - n2

def multiply(n1, n2):
    return n1 * n2

def divide(n1, n2):
    return n1 / n2

operations = {
    "+": add,
    "-": subtract,
    "*": multiply,
    "/": divide
}

def calculator():
    num1 = float(input("What's the first number?: "))
    for symbol in operations:
        print(symbol)
    should_continue = True

    while should_continue:
        operation_symbol = input("Pick an operation: ")
        num2 = float(input("What's the next number?: "))
        calculation_function = operations[operation_symbol]
        answer = calculation_function(num1, num2)
        print(f"{num1} {operation_symbol} {num2} = {answer}")

        if input(f"Type 'y' to continue calculating with {answer}, or type 'n' to start a new calculation: ") == 'y':
            num1 = answer
        else:
            should_continue = False
            calculator() # Recursion to start over

calculator()

```

Python Notes: Days 11-20 (Angela Yu's 100 Days of Code)

Here are the detailed notes covering the intermediate Python section of the course, focusing on capstone projects, scope, debugging, Object-Oriented Programming (OOP), and GUI programming with Turtle.

Day 11: The Blackjack Capstone Project

This day is dedicated to applying all prior knowledge to build a complete text-based Blackjack game.

Key Concepts Applied

- **Functions:** Breaking down the game into logical parts like `deal_card()` , `calculate_score()` , `compare()` .
- **Lists:** To represent the deck of cards and the hands of the player and dealer.
- **random Module:** Using `random.choice()` to deal a card from the deck.
- **Loops & Conditionals:** `while` loops manage the game turns, and `if/elif/else` statements handle the complex game logic (busting, winning, drawing).

Project Logic & Structure

1. **Create the Deck:** The simplest approach is a list of integers representing card values. Ace is 11 by default.

```
cards = [11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]
```

2. **Deal Cards:** A function `deal_card()` should select a random card from the `cards` list.
3. **Calculate Score:** A function `calculate_score(hand)` takes a list of cards (a hand) and returns the sum.
 - **The Ace Rule (Crucial Logic):** If the total score is over 21 and an Ace (11) is in the hand, the Ace's value must change to 1.

```
def calculate_score(cards):
    # A score of 0 represents a Blackjack
    if sum(cards) == 21 and len(cards) == 2:
        return 0

    # Change Ace from 11 to 1 if score is over 21
    if 11 in cards and sum(cards) > 21:
        cards.remove(11)
        cards.append(1)
    return sum(cards)
```

4. Player's Turn:

- The player is in a `while` loop, asked to 'hit' (get another card) or 'stand'.
- The loop breaks if the player stands or busts (score > 21).

5. Dealer's Turn:

- After the player stands, the dealer automatically hits until their score is 17 or more.

6. Compare Scores: A final function compares the player's and dealer's scores to determine the winner, handling all cases (Blackjack, bust, higher score).

Day 12: Scope & Number Guessing Game

This day introduces a fundamental programming concept: **variable scope**.

Core Concepts: Scope

- **Local Scope:** Variables created inside a function are **local** to that function. They cannot be accessed from outside and are destroyed when the function completes.
- **Global Scope:** Variables created at the top level of a script are **global**. They can be accessed (read) from anywhere in the code.
- **Modifying Globals:** To change a global variable's value from inside a function, you must use the `global` keyword. **This is considered bad practice**. The proper way is to `return` the new value from the function and reassign the global variable.

```
# BAD PRACTICE 🙅
enemies = 1
def increase_enemies():
    global enemies
    enemies += 1

# GOOD PRACTICE 👍
player_health = 100
def take_damage(current_health):
    return current_health - 10

player_health = take_damage(player_health)
```

- **No Block Scope:** In Python, `if`, `while`, and `for` blocks do **not** create their own local scope.
- **Global Constants:** Variables in global scope that are not intended to be changed are written in ALL_CAPS_SNAKE_CASE by convention (e.g., `PI = 3.14159`).

Project: Number Guessing Game

- The computer picks a random number between 1 and 100.
- The user picks a difficulty ('easy' or 'hard'), which determines the number of guesses.
- A `while` loop runs as long as the user has guesses left.
- Inside the loop, the user's guess is compared to the number, and they are told if it's "Too high" or "Too low".
- If they guess correctly, they win. If they run out of guesses, they lose.

Day 13: Debugging: How to Find and Fix Errors

A crucial skill-based day focused on the art of debugging.

Key Debugging Techniques

1. **Describe the Problem:** Clearly state what you expected to happen versus what actually happened.
2. **Reproduce the Bug:** Find a reliable way to make the error occur. This is the most critical step.
3. **Play Computer:** Read through your code line-by-line, tracking the value of each variable in your head or on paper. This helps spot logical fallacies.
4. **Use `print()`:** The simplest debugger. Add print statements to check the value of variables at different stages of your program to see where things go wrong.

5. **Use a Debugger:** Learn to use the debugger in your IDE (like VS Code or PyCharm). It allows you to:
- **Set Breakpoints:** Pause your code at a specific line.
 - **Step Through:** Execute code one line at a time.
 - **Inspect Variables:** See the live values of all variables.

Day 14: Higher Lower Game Project

This project applies functions and data structures to create a "Higher Lower" follower count game.

Project Logic & Structure

- **Data:** The game uses a list of dictionaries, where each dictionary represents a person/brand and contains their `name`, `follower_count`, `description`, and `country`.
- **Game Flow:**
 - i. Pick two random accounts, 'A' and 'B', from the data.
 - ii. Display info for A: "Compare A: [Name], a [Description], from [Country]."
 - iii. Display info for B: "Against B: [Name], a [Description], from [Country]."
 - iv. Ask the user: "Who has more followers? Type 'A' or 'B':".
 - v. Check the user's answer against the `follower_count` of A and B.
 - vi. If correct, the score increases. Account 'B' becomes the new 'A', and a new 'B' is chosen. The game continues.
 - vii. If incorrect, the game ends and the final score is displayed.

Day 15: The Coffee Machine Project

A more complex project involving resource management, dictionaries, and processing transactions.

Project Logic & Structure

1. **Data:**
 - `MENU` : A dictionary containing coffee types. Each coffee is a dictionary with `ingredients` (another dictionary) and `cost`.
 - `resources` : A dictionary tracking available water, milk, and coffee.
2. **Main Loop:** The program continuously prompts the user for a drink (`espresso` / `latte` / `cappuccino`).

3. Special Commands:

- `off` : Terminates the program.
- `report` : Prints the current levels of all resources.

4. Key Functions:

- `check_resources(drink_ingredients)` : Checks if the machine has enough ingredients in `resources` to make the selected drink.
- `process_coins()` : Prompts the user to insert coins (quarters, dimes, etc.) and returns the total monetary value.
- `check_transaction(money_received, drink_cost)` : Verifies if the user paid enough. If so, it provides change and returns `True` . If not, it refunds the money and returns `False` .
- `make_coffee(drink_name, order_ingredients)` : If both resources and money are sufficient, this function deducts the used ingredients from the `resources` dictionary.

Day 16: Object-Oriented Programming (OOP)

A paradigm shift from procedural to object-oriented programming.

Core OOP Concepts

- **Class**: A blueprint for creating objects. Example: `Car` .
- **Object (Instance)**: An item created from a class. It has its own data and behaviors. Example:
`my_nissan = Car()` .
- **Attribute**: A variable associated with an object, representing its data or state. Example:
`my_nissan.fuel_level` .
- **Method**: A function that belongs to an object, defining its behavior. Example: `my_nissan.drive()` .
- **Constructor (`__init__`)**: A special method that is called when an object is created. It's used to set up the object's initial attributes.

Example

```
# The Class (Blueprint)
class Dog:
    # The Constructor
    def __init__(self, name, breed):
        # Attributes
        self.name = name
        self.breed = breed
        self.tricks = []

    # A Method
    def add_trick(self, trick):
        self.tricks.append(trick)

# Creating an Object (Instance)
my_dog = Dog("Fido", "Golden Retriever")

# Accessing attributes and calling methods
print(my_dog.name) # Output: Fido
my_dog.add_trick("sit")
print(my_dog.tricks) # Output: ['sit']
```

Day 17: The Quiz Project & Benefits of OOP

Applying OOP to build a modular and scalable quiz game.

Project Structure using Classes

1. **Question Class (question_model.py)**: Models a single question.
 - **Attributes**: text and answer .
 - **__init__** : Takes a question text and answer to create a new Question object.
2. **QuizBrain Class (quiz_brain.py)**: Manages the quiz logic.
 - **Attributes**: question_number , score , and question_list (a list of Question objects).
 - **Methods**:
 - still_has_questions() : Returns True if there are more questions in the list.
 - next_question() : Fetches the next question, prompts the user, and checks their answer.
 - check_answer(user_answer, correct_answer) : Compares answers, updates the score, and provides feedback.

3. `main.py` :

- Creates a list of `Question` objects (the "question bank").
- Creates a `QuizBrain` object, passing the question bank to it.
- Uses a `while` loop to run the quiz as long as `still_has_questions()` is true.

Day 18: Turtle & The Graphical User Interface (GUI)

An introduction to creating graphics using Python's built-in `turtle` module.

Core Concepts

- **Turtle Graphics:** A simple and fun way to draw shapes and patterns. You control a "turtle" (the pen) on a "screen" (the canvas).
- **Key Objects:** `Turtle()` and `Screen()` .
- **Tuples:** An immutable (unchangeable) data structure, written with parentheses `()` . Often used for things like RGB color values, e.g., `(255, 100, 50)` .
- **Importing:** You can import modules in different ways:
 - `import turtle`
 - `from turtle import Turtle, Screen` (most common for this module)
 - `import random_module as rm` (using an alias)

Project: Hirst Dot Painting

This day involves several drawing challenges, culminating in a project that recreates a Damien Hirst-style dot painting. The logic involves:

- Using the `colorgram` library to extract a list of RGB colors from an image.
- Creating a `Turtle` and setting its pen up.
- Using nested loops to create a grid of dots (e.g., 10x10).
- For each position in the grid, choose a random color from the extracted list and stamp a dot using `turtle.dot()` .
- Move the turtle to the next position without drawing.

Day 19: Instances, State, and Higher-Order Functions

Making interactive GUI programs with Turtle by handling user input.

Core Concepts

- **Instances & State:** You can create multiple `Turtle` objects. Each one is a separate **instance** with its own **state** (color, position, heading, etc.).
- **Higher-Order Functions & Event Listeners:**
 - An **event listener** is a function that waits for an event (like a key press).
 - The `screen.listen()` method activates the listener.
 - `screen.onkey(fun, key)` is a **higher-order function** because it takes another function (`fun`) as an argument. It binds a keyboard `key` to a specific function.
 - **Important:** You pass the function name *without parentheses*: `onkey(move_forward, "w")` .

Projects

1. Etch-A-Sketch:

- Create functions like `move_forward()` , `turn_left()` , etc.
- Bind these functions to keyboard keys (`w` , `a` , `s` , `d`).
- Create a `clear_screen()` function and bind it to `c` .

2. Turtle Race:

- Create multiple `Turtle` instances, each a different color.
- Line them up at a starting line on the left of the screen.
- Use a `while` loop to move each turtle forward by a random amount in each iteration.
- The first turtle to cross a finish line on the right wins.

Day 20: Build the Snake Game Part 1

Beginning the first multi-day project, focusing on setting up the game and the snake's movement.

Project Plan (Part 1)

1. Screen Setup:

- Create a `Screen` object.
- Use `screen.tracer(0)` to **turn off automatic animation**. This gives you manual control over when the screen updates, preventing flicker.

2. Create the Snake Class (`snake.py`):

- `__init__` :
 - Create an empty list `self.segments` .
 - Create the first 3 segments (as square `Turtle` objects) at positions (0,0), (-20,0), and (-40,0).

- Add these segments to the `self.segments` list.
- **move() method:** This is the core logic.
 - The snake moves by making each segment take the position of the one in front of it.
 - Use a `for` loop that iterates **backwards** over the segments.
 - `segment[2]` goes to `segment[1]` 's position.
 - `segment[1]` goes to `segment[0]` 's position.
 - Finally, the head (`segment[0]`) moves forward by 20 pixels.

3. Main Game Loop (`main.py`):

- Create a `Snake` object.
- Start a `while game_is_on:` loop.
- Inside the loop:
 - `screen.update()` : Manually refresh the screen to show the new positions of all segments at once.
 - `time.sleep(0.1)` : Add a small delay to control the game speed.
 - `snake.move()` : Call the snake's move method.

This day ends with a snake that moves continuously across the screen. The next steps will involve adding controls, food, and collision detection.

Day 21: The Snake Game (Part 2) - Inheritance & Slicing

This day focuses on completing the Snake Game by introducing two powerful concepts: **class inheritance** and **list/tuple slicing**.

Core Concepts

1. Class Inheritance

Inheritance allows a new class (the *child* or *subclass*) to inherit attributes and methods from an existing class (the *parent* or *superclass*). This promotes code reuse.

The syntax is `class ChildClass(ParentClass):` .

A key part of inheritance is initializing the parent class from within the child class using `super().__init__()` . This ensures that all the setup code from the parent class runs before you add the child-specific code.

Example: In the Snake Game, the `Food` and `Scoreboard` are specialized versions of a `Turtle` . Instead of rewriting all the `Turtle` setup, we inherit from it.

```
from turtle import Turtle

# Food class inherits from the Turtle class
class Food(Turtle):
    def __init__(self):
        # Initialize the parent Turtle class
        super().__init__()
        self.shape("circle")
        self.penup()
        self.shapesize(stretch_len=0.5, stretch_wid=0.5) # 10x10 circle
        self.color("blue")
        self.speed("fastest")
        # Custom method for this class
        self.refresh()

    def refresh(self):
        # Code to move food to a new random location
        pass
```

2. List and Tuple Slicing

Slicing is a concise way to access parts of a sequence (like a list or tuple). It's incredibly useful for things like collision detection in the Snake Game.

Syntax: `my_list[start:stop:step]`

- `start` : The index to start the slice from (inclusive). Defaults to 0.
- `stop` : The index to end the slice at (exclusive). Defaults to the end of the list.
- `step` : The amount to jump by. Defaults to 1.

Common Slicing Examples:

```
piano_keys = ["a", "b", "c", "d", "e", "f", "g"]

# Get elements from index 2 up to (but not including) index 5
print(piano_keys[2:5]) # Output: ['c', 'd', 'e']

# Get everything from index 2 to the end
print(piano_keys[2:]) # Output: ['c', 'd', 'e', 'f', 'g']

# Get everything from the beginning up to index 5
print(piano_keys[:5]) # Output: ['a', 'b', 'c', 'd', 'e']

# Get every other element
print(piano_keys[::2]) # Output: ['a', 'c', 'e', 'g']

# Get the list in reverse
print(piano_keys[::-1]) # Output: ['g', 'f', 'e', 'd', 'c', 'b', 'a']
```

Project Implementation: Snake Game

- **Detecting Collision with Tail:** To check if the snake's head collides with any part of its body, we can slice the `segments` list. We check if the head's position is near any segment *except* for the head itself.

```
# In the main game loop
for segment in snake.segments[1:]: # Slice to exclude the head (at index 0)
    if snake.head.distance(segment) < 10:
        game_is_on = False
        scoreboard.game_over()
```

Day 22: Building the Pong Game

This day is all about building another classic arcade game, Pong. It reinforces object-oriented programming by building the game from separate, interacting classes.

Project Architecture

The game is broken down into modular classes:

1. **main.py** : The main script that initializes the screen, creates objects from the other classes, and runs the game loop.
2. **paddle.py** : A `Paddle` class that inherits from `Turtle` . It handles creating the paddle shape and its movement (`go_up()` , `go_down()`).
3. **ball.py** : A `Ball` class, also inheriting from `Turtle` . It controls the ball's movement, bouncing logic, and reset position.
4. **scoreboard.py** : A `Scoreboard` class to display and update the scores for both players.

Project Implementation Notes

- **Creating the Paddles**: An instance of the `Paddle` class is created for each player. Their starting positions are set using coordinates.

```
# in main.py
r_paddle = Paddle((350, 0))
l_paddle = Paddle((-350, 0))
```

- **Ball Movement**: The ball moves continuously in the main game loop. Its direction is controlled by `x_move` and `y_move` attributes, which are inverted upon collision.

```
# in Ball class
def move(self):
    new_x = self.xcor() + self.x_move
    new_y = self.ycor() + self.y_move
    self.goto(new_x, new_y)
```

- **Detecting Wall Collision**: Check the ball's y-coordinate. If it exceeds the top or bottom boundary, reverse its `y_move` direction.


```
# in main.py loop
if ball.ycor() > 280 or ball.ycor() < -280:
    ball.bounce_y() # Method to reverse y_move
```

- **Detecting Paddle Collision:** Check the distance between the ball and each paddle. If the distance is small *and* the ball is close to the front edge of the paddle, reverse its `x_move` direction. This prevents the ball from getting "stuck" inside the paddle.

```
# in main.py loop
if (ball.distance(r_paddle) < 50 and ball.xcor() > 320) or \
    (ball.distance(l_paddle) < 50 and ball.xcor() < -320):
    ball.bounce_x() # Method to reverse x_move
```

- **Detecting a Miss:** If the ball goes past a paddle's x-coordinate, a player has scored. The scoreboard is updated, and the ball is reset to the center.

Day 23: The Turtle Crossing Capstone Project

This project combines everything learned so far: OOP, inheritance, event listeners, and collision detection to create a "Frogger"-style game.

Project Architecture

1. **Player class:** Represents the turtle the user controls. It only moves forward.
2. **CarManager class:** Manages the creation, movement, and speed of all the cars on the screen. It's responsible for generating cars randomly.
3. **Scoreboard class:** Keeps track of the player's level and displays "GAME OVER" when a collision occurs.

Project Implementation Notes

- **Managing Many Cars:** The `CarManager` class holds a list of all car objects (which are `Turtle` instances). In the game loop, a single method `car_manager.move_cars()` is called, which then iterates through its list and moves each car.
- **Random Car Generation:** To avoid a constant stream of cars, you can use `random.randint(1, 6)`. Only generate a new car if the random number is, for example, 1. This makes car generation sporadic.

- **Collision Detection:** In the main loop, iterate through all the cars managed by `CarManager` and check the player's distance to each one.

```
# in main.py loop
for car in car_manager.all_cars:
    if car.distance(player) < 20:
        game_is_on = False
        scoreboard.game_over()
```

- **Leveling Up:** Detect when the player has reached the finish line (a certain y-coordinate). When this happens, reset the player's position, increase the game level (which increases car speed), and update the scoreboard.

Day 24: Files, Directories, and Paths

This day moves away from Turtle graphics and into a fundamental programming concept: reading from and writing to files.

Core Concepts

1. Reading Files

The standard way to open and read a file in Python is using the `with open(...)` statement. This is preferred because it **automatically closes the file** for you, even if errors occur.

```
# Open and read the entire file into a string
with open("my_file.txt") as file:
    contents = file.read()
    print(contents)
```

2. File Paths: Absolute vs. Relative

- **Absolute Path:** The full path from the root directory of your computer. It's unambiguous but not portable.
 - Example: `C:/Users/YourUser/Documents/my_file.txt` (Windows) or `/Users/YourUser/Documents/my_file.txt` (Mac/Linux).
- **Relative Path:** The path from your current working directory. It's portable, as it doesn't depend on the user's specific folder structure.

- `./my_file.txt` or `my_file.txt` : The file is in the same folder as the script.
- `../data/my_file.txt` : The file is in a `data` folder one level up from the current directory.

3. Writing and Appending to Files

You can specify a `mode` when opening a file.

- `mode="w"` (**Write**): Opens the file for writing. **Warning:** This will delete all existing content in the file. If the file doesn't exist, it will be created.
- `mode="a"` (**Append**): Opens the file for appending. New content is added to the end of the file without deleting existing content. If the file doesn't exist, it will be created.

```
# Write mode (overwrites file)
with open("my_file.txt", mode="w") as file:
    file.write("New text.")
```

```
# Append mode (adds to end of file)
with open("my_file.txt", mode="a") as file:
    file.write("\nSome more text.")
```

Project Implementation: Mail Merge

The project for this day is a Mail Merge program.

1. **Read Names:** Open a file like `invited_names.txt` that contains a list of names, one per line. Use `file.readlines()` to get a list of names.
2. **Read Letter Template:** Open a file like `starting_letter.txt` which contains a placeholder, e.g., `[name]` .
3. **Create Custom Letters:** Loop through each name from the list. For each name:
 - Use the string `.replace()` method to replace the `[name]` placeholder in the template with the actual name.
 - Create a new file named `letter_for_John.txt` (or whatever the name is).
 - Write the new, personalized letter content into this file.

Day 25: Working with CSV Data and the Pandas Library

This day introduces one of the most powerful data analysis libraries in Python: **Pandas**. It makes working with tabular data (like spreadsheets or CSV files) incredibly simple.

Core Concepts

1. What is CSV?

CSV stands for **Comma-Separated Values**. It's a plain text format for storing spreadsheet-like data, where each row is a new line and columns are separated by commas.

2. The Pandas Library

Pandas provides two primary data structures:

- **Series** : A one-dimensional labeled array, essentially a single column of data.
- **DataFrame** : A two-dimensional labeled data structure with columns of potentially different types, like an entire spreadsheet.

Installation: `pip install pandas`

3. Reading CSV Files

Reading a CSV is a one-liner with Pandas.

```
import pandas

data = pandas.read_csv("weather_data.csv")
print(data)
```

4. Accessing Data

- **Accessing a Column (Series):**

```
# Both are equivalent
temperatures = data["temp"]
temperatures = data.temp
```

- **Accessing a Row:**

```
# Get the row where the day is "Monday"
monday_data = data[data.day == "Monday"]
print(monday_data)
```

- **Getting the Row with the Max Value:**

```
max_temp_row = data[data.temp == data.temp.max()]
print(max_temp_row)
```

- **Creating a DataFrame from Scratch:**

```
data_dict = {
    "students": ["Amy", "James", "Angela"],
    "scores": [76, 56, 65]
}
new_data = pandas.DataFrame(data_dict)
# Save to a CSV file
new_data.to_csv("new_data.csv")
```

Project Implementation: U.S. States Game

The project is a game where a map of the U.S. is shown, and the user has to guess the names of all 50 states.

1. **Load Data:** Use Pandas to load `50_states.csv` , which contains state names and their x, y coordinates on the map image.
2. **Get User Input:** Use Turtle's `screen.textinput()` to get the user's guess.
3. **Check Answer:** Check if the user's guess exists in the 'state' column of the DataFrame.
4. **Write State on Map:** If the guess is correct, get the corresponding row from the DataFrame to find its x and y coordinates. Create a new Turtle to write the state's name at that location on the map.
5. **Track Score:** Keep a list of correctly guessed states and update the score.
6. **End Game:** The game ends when all 50 states are guessed or the user types "Exit". If they exit, create a `states_to_learn.csv` file containing all the states they missed.

Day 26: List Comprehension and Dictionary Comprehension

Comprehensions are a unique and powerful feature of Python. They allow you to create new lists or dictionaries from existing ones in a single, readable line of code, replacing longer `for` loops.

Core Concepts

1. List Comprehension

Syntax: `new_list = [new_item for item in list if test]`

- `new_item` : An expression for the new item to be included.
- `for item in list` : The loop over the existing iterable.
- `if test` : (Optional) A condition to filter items.

Example 1: Basic List Comprehension

```
# Traditional Way
numbers = [1, 2, 3]
new_list = []
for n in numbers:
    add_1 = n + 1
    new_list.append(add_1)
# new_list is [2, 3, 4]

# With List Comprehension
numbers = [1, 2, 3]
new_list = [n + 1 for n in numbers]
# new_list is [2, 3, 4]
```

Example 2: Conditional List Comprehension

```
# Get a list of short names
names = ["Alex", "Beth", "Caroline", "Dave", "Eleanor", "Freddie"]

# With List Comprehension
short_names = [name for name in names if len(name) < 5]
# short_names is ["Alex", "Beth", "Dave"]

# Challenge: Get uppercase versions of long names
long_names_upper = [name.upper() for name in names if len(name) > 5]
# long_names_upper is ['CAROLINE', 'ELEANOR', 'FREDDIE']
```

2. Dictionary Comprehension

Syntax: `new_dict = {new_key:new_value for (key, value) in dict.items() if test}`

Example: Students with high scores

```
import random

student_scores = {student: random.randint(50, 100) for student in ["Alex", "Beth", "Caroline"]}
# student_scores -> {'Alex': 78, 'Beth': 92, 'Caroline': 65}

# Create a new dictionary with students who passed (score >= 60)
passed_students = {student: score for (student, score) in student_scores.items() if score >= 60}
# passed_students -> {'Alex': 78, 'Beth': 92}
```

Project Implementation: NATO Phonetic Alphabet

The project is to create a program that converts a word into its NATO phonetic alphabet equivalent (e.g., "A" -> "Alfa", "B" -> "Bravo").

1. **Load Data:** Use Pandas to read `nato_phonetic_alphabet.csv`.
2. **Create Dictionary:** Use a **dictionary comprehension** to create a dictionary from the DataFrame, where keys are the letters and values are the phonetic codes.

```
# {new_key:new_value for (index, row) in df.iterrows()}
phonetic_dict = {row.letter: row.code for (index, row) in nato_df.iterrows()}
```

3. **Get User Input:** Ask the user for a word.
4. **Create Phonetic List:** Use a **list comprehension** to iterate through the user's word and look up each letter in the dictionary created in step 2.

```
word = input("Enter a word: ").upper()
output_list = [phonetic_dict[letter] for letter in word]
print(output_list)
# Input: "Angela" -> Output: ['Alfa', 'November', 'Golf', 'Echo', 'Lima', 'Alfa']
```

Day 27: Tkinter, *args , **kwargs and Creating GUI Programs

This day introduces **Tkinter**, Python's built-in library for creating graphical user interfaces (GUIs). It also covers advanced Python function arguments: `*args` and `**kwargs` .

Core Concepts

1. Creating a Window and Widgets

Everything in a Tkinter app is a **widget**. The main window is the root widget, and other widgets (buttons, labels, etc.) are placed inside it.

```
import tkinter

# 1. Create the main window
window = tkinter.Tk()
window.title("My First GUI Program")
window.minsize(width=500, height=300)

# 2. Create a Label widget
my_label = tkinter.Label(text="I am a Label", font=("Arial", 24, "bold"))
# 3. Place the widget on the screen
my_label.pack() # .pack() is one way to manage layout

# Must be at the very end of the program
window.mainloop() # Keeps the window open
```

2. *args : Unlimited Positional Arguments

The `*args` syntax allows a function to accept any number of positional arguments. Inside the function, `args` is a **tuple** of all the arguments passed.


```
def add(*args):
    # args is a tuple, e.g., (1, 2, 3)
    total = 0
    for n in args:
        total += n
    return total

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

3. **kwargs : Unlimited Keyword Arguments

The `**kwargs` syntax allows a function to accept any number of keyword arguments. Inside the function, `kwargs` is a **dictionary**.

```
def calculate(n, **kwargs):
    # kwargs is a dictionary, e.g., {'add': 3, 'multiply': 5}
    n += kwargs.get("add", 0) # Use .get() to avoid errors if key doesn't exist
    n *= kwargs.get("multiply", 1)
    return n

print(calculate(2, add=3, multiply=5)) # Output: 25
```

4. Tkinter Layout Managers

You can't mix different layout managers in the same container.

- `pack()` : Simple but less precise. Stacks widgets on top of each other or side-by-side.
- `place()` : Very precise. You specify exact x and y coordinates. Can be tedious for complex layouts.
- `grid()` : The most flexible. Organizes widgets in a grid of rows and columns. Great for forms and structured layouts.

```
# Example using grid
button = tkinter.Button(text="Click Me")
button.grid(column=1, row=1)

entry = tkinter.Entry(width=10)
entry.grid(column=3, row=2)
```

Project Implementation: Mile to Kilometer Converter

This project builds a simple utility GUI.

1. **Window Setup:** Create a `Tk` window.
2. **Widgets:**
 - An `Entry` widget for the user to type in the miles.
 - Four `Label` widgets: "is equal to", the result "0", "Miles", and "Km".
 - A `Button` widget labeled "Calculate".
3. **Layout:** Use the `.grid()` layout manager to arrange the widgets in a clean 3x3 grid.
4. **Functionality:**
 - Create a function `button_clicked()` that is linked to the button's `command` option.
 - Inside this function, get the text from the `Entry` widget using `.get()`.
 - Convert the value from miles to kilometers.
 - Update the result label's text using the `.config()` method:
`result_label.config(text=f"{km_result}")`.

Day 28: Pomodoro GUI Application with Tkinter

This project builds a functional Pomodoro productivity timer, applying the Tkinter skills from the previous day and introducing concepts for managing time-based events.

Project Architecture

A GUI application with a tomato image, a timer display, start/reset buttons, and a checkmark counter to track completed sessions.

Core Concepts

1. Adding Images with `PhotoImage`

Tkinter can't use JPG or other common formats directly. You typically use PNG or GIF files. The `PhotoImage` class is used to load an image, and it's then placed on a `Canvas` widget.

```

from tkinter import *

window = Tk()
canvas = Canvas(width=200, height=224, bg="yellow", highlightthickness=0)
tomato_img = PhotoImage(file="tomato.png")
canvas.create_image(100, 112, image=tomato_img)
canvas.pack()

window.mainloop()

```

2. Timers with .after()

The `.after()` method is a crucial part of the Tkinter event loop. It tells Tkinter to call a function after a specified delay (in milliseconds) without freezing the entire program (unlike `time.sleep()`).

Syntax: `widget.after(delay_ms, function_to_call, *args)`

This is perfect for creating a countdown timer. The countdown function calls itself every second using `.after()` .

```

# Example of a countdown function
def count_down(count):
    # Update the timer text on the canvas
    canvas.itemconfig(timer_text, text=f"{count}")
    if count > 0:
        # After 1000ms (1s), call count_down again with count - 1
        window.after(1000, count_down, count - 1)

```

Project Implementation Notes

- **Constants:** Define constants at the top of your script for work minutes, short break minutes, and long break minutes. This makes the code easier to read and modify.
- **State Management:** Use global variables or class attributes to keep track of the current number of reps and the timer instance (`reps` , `timer`).
- **Timer Logic:**
 - The `start_timer()` function determines whether the next session should be a work session, short break, or long break based on the `reps` count.
 - The `count_down()` function handles the second-by-second countdown and updates the UI.
 - When the countdown finishes, `start_timer()` is called again to begin the next session automatically.

- **Resetting the Timer:** The `reset_timer()` function needs to:
 - i. Stop the currently running timer using `window.after_cancel(timer)`.
 - ii. Reset the timer text to "00:00".
 - iii. Reset the title label to "Timer".
 - iv. Reset the `reps` count and checkmarks.

Day 29: Password Manager GUI Application

This project combines Tkinter with file I/O to create a practical desktop application for saving and retrieving passwords. It introduces message boxes for user feedback.

Project Architecture

A GUI with three `Entry` fields (Website, Email/Username, Password), three `Buttons` (Generate Password, Search, Add), and a logo.

Core Concepts

1. Tkinter messagebox

The `messagebox` module provides standard dialog boxes for displaying information, warnings, or asking yes/no questions. You must import it separately: `from tkinter import messagebox`.

```
from tkinter import messagebox

# Show simple information
messagebox.showinfo(title="Title", message="This is some info.")

# Show a warning
messagebox.showwarning(title="Warning", message="This is a warning.")

# Show an error
messagebox.showerror(title="Error", message="This is an error.")

# Ask a yes/no question
is_ok = messagebox.askokcancel(title="Confirm", message="Is it ok to proceed?")
# is_ok will be True if user clicks OK, False if Cancel
```

2. Managing User Input

- **Getting Text:** Use `.get()` on an `Entry` widget to retrieve what the user has typed.
- **Deleting Text:** Use `.delete(0, END)` to clear an `Entry` widget. `END` is a Tkinter constant.
- **Inserting Text:** Use `.insert(0, "text to insert")` to programmatically add text to an `Entry`.

3. Interacting with the Clipboard

The `pyperclip` library makes it easy to copy text to the system clipboard.

Installation: `pip install pyperclip`

```
import pyperclip
pyperclip.copy("Text to be copied")
```

Project Implementation Notes

- **Password Generation:** Create a function that generates a random password using letters, numbers, and symbols from lists, shuffles them, and returns the result. When the "Generate" button is clicked, this function is called, and the result is inserted into the password entry field and copied to the clipboard.
- **Saving Data:**
 - i. When the "Add" button is clicked, retrieve the data from all three entry fields.
 - ii. Perform validation: check if any of the fields are empty. If so, show a `messagebox.showinfo()` and do not proceed.
 - iii. Use `messagebox.askokcancel()` to confirm with the user before saving.
 - iv. Format the data into a single string (e.g., `"Website | Email | Password\n"`).
 - v. Open a `data.txt` file in **append mode** (`"a"`) and write the formatted string.
 - vi. Clear the website and password fields for the next entry.
- **Structure:** Using `.grid()` with `columnspan` is very useful here. For example, the website entry field and the add button might span multiple columns to create a clean layout.

Day 30: Errors, Exceptions, and JSON Data

This final day in the block covers how to handle errors gracefully in your code and introduces JSON, a superior format for storing and transferring structured data.

Core Concepts

1. Errors and Exception Handling

When Python encounters an error, it "raises" an exception and stops the program. We can "catch" these exceptions to prevent crashing and handle the error gracefully.

The `try...except...else...finally` block:

- **try** : Code that might cause an error is placed here.
- **except** : This block runs **only if** an exception occurred in the `try` block. You can specify the type of error (e.g., `FileNotFoundError` , `KeyError`).
- **else** : This block runs **only if** no exceptions occurred.
- **finally** : This block runs **no matter what**, whether an exception occurred or not. It's often used for cleanup operations.

```
try:
    file = open("a_file.txt")
    a_dictionary = {"key": "value"}
    print(a_dictionary["key"])
except FileNotFoundError:
    # This runs if a_file.txt doesn't exist
    file = open("a_file.txt", "w")
    file.write("Something")
except KeyError as error_message:
    # This runs if the key doesn't exist in the dictionary
    print(f"The key {error_message} does not exist.")
else:
    # This runs if the try block was successful
    content = file.read()
    print(content)
finally:
    # This runs no matter what
    file.close()
    print("File was closed.")
```

Raising your own exceptions with `raise` :

You can trigger your own exceptions if a certain condition isn't met.

```
if height > 3: raise ValueError("Human height should not be over 3 meters.")
```

2. JSON (JavaScript Object Notation)

JSON is a lightweight data-interchange format. It's human-readable and easy for machines to parse. It's the standard for APIs and configuration files. It looks very similar to Python dictionaries.

Python's built-in `json` module is used for this.

- `json.dump()` : Writing JSON data to a file. It takes two arguments: the data to write and the file object.
- `json.load()` : Reading JSON data from a file.
- `json.update()` : This isn't a single function. You first `load` the data into a Python dictionary, then use the dictionary's `.update()` method, and finally `dump` the modified dictionary back to the file.

```
import json

# --- WRITING to a JSON file ---
data = {
    "name": "Angela",
    "score": 100,
    "is_admin": True
}
with open("data.json", "w") as file:
    json.dump(data, file, indent=4) # indent makes it human-readable

# --- READING from a JSON file ---
with open("data.json", "r") as file:
    loaded_data = json.load(file)
    print(loaded_data["name"]) # Output: Angela

# --- UPDATING a JSON file ---
with open("data.json", "r") as file:
    # 1. Read existing data
    data_to_update = json.load(file)
    # 2. Update the Python dictionary
    data_to_update["score"] = 105
with open("data.json", "w") as file:
    # 3. Save the updated data
    json.dump(data_to_update, file, indent=4)
```

Project Implementation: Updating the Password Manager

The project is to refactor the Password Manager from Day 29 to handle errors and use JSON instead of a plain text file.

1. Search Functionality:

- Create a "Search" button.
- When clicked, get the website name from the entry field.
- Use a `try...except` block to open and read the `data.json` file. Handle the `FileNotFoundError` if the file doesn't exist yet.
- If the website exists as a key in the loaded JSON data, show the email and password in a `messagebox`. Handle the `KeyError` if the website is not found.

2. Refactor Saving Logic:

- Change the `save()` function to work with JSON.
- First, try to `load` existing data from `data.json`. If the file doesn't exist, start with an empty dictionary.
- `update` this dictionary with the new entry (e.g., `data.update(new_data)`).
- Finally, `dump` the entire updated dictionary back into `data.json`, overwriting the old file. This ensures the file is always a valid JSON object.

Python Days 31-40: Intermediate+ Python, APIs & Capstone Projects

This section covers intermediate-level projects using Tkinter, introduces the powerful world of Application Programming Interfaces (APIs), and culminates in two major capstone projects that integrate multiple advanced skills.

Day 31: Flash Card App Capstone Project

The goal is to build a language-learning flashcard app using **Tkinter** for the UI and **Pandas** for data management.

Key Concepts

- 1. Reading Data with Pandas:** Instead of manually handling CSV files, we use the Pandas library for robust and easy data manipulation.
 - `pandas.read_csv()` : Reads a CSV file into a DataFrame, a powerful table-like data structure.
 - `DataFrame.to_dict(orient="records")` : Converts the DataFrame into a list of dictionaries, where each dictionary represents a row (e.g., `[{'French': 'partie', 'English': 'part'}, ...]`). This format is ideal for working with individual flashcards.
- 2. Tkinter UI & Canvas:**
 - **Canvas Widget:** Used for drawing shapes and displaying images and text. It's perfect for creating the flashcard look.
 - `canvas.create_image()` : Places an image on the canvas.
 - `canvas.create_text()` : Places text on the canvas. We'll create text elements for the language title and the word itself.
 - `canvas.itemconfig()` : A crucial method used to change the properties of an item already on the canvas (e.g., change the text or color).
 - `canvas.config()` : Used to change the properties of the canvas widget itself (e.g., its background color).
- 3. Application Logic:**
 - **State Management:** The program needs to keep track of the `current_card`.
 - **Flipping Mechanism:** We use `window.after()` to schedule an action (like flipping the card) to happen after a certain delay without freezing the entire application.

- **Saving Progress:** When the user knows a word, it should be removed from the list of words to learn. The remaining words are saved to a new CSV file (words_to_learn.csv) so the user can pick up where they left off.

Code Implementation Snippets

1. Reading the Data:

```
import pandas

try:
    # Try to load progress first
    data = pandas.read_csv("data/words_to_learn.csv")
except FileNotFoundError:
    # If no progress file, start with the full list
    original_data = pandas.read_csv("data/french_words.csv")
    to_learn = original_data.to_dict(orient="records")
else:
    to_learn = data.to_dict(orient="records")

current_card = {}

def next_card():
    global current_card, flip_timer
    window.after_cancel(flip_timer) # Cancel previous timer
    current_card = random.choice(to_learn)
    canvas.itemconfig(card_title, text="French", fill="black")
    canvas.itemconfig(card_word, text=current_card["French"], fill="black")
    canvas.itemconfig(card_background, image=card_front_img)
    flip_timer = window.after(3000, func=flip_card) # Set a new timer
```

2. Flipping the Card:

```
def flip_card():
    canvas.itemconfig(card_title, text="English", fill="white")
    canvas.itemconfig(card_word, text=current_card["English"], fill="white")
    canvas.itemconfig(card_background, image=card_back_img)
```

3. Saving Known Words:

```
def is_known():
    to_learn.remove(current_card)
    # Create a new DataFrame with the remaining words and save it
    data = pandas.DataFrame(to_learn)
    data.to_csv("data/words_to_learn.csv", index=False) # index=False prevents pandas from writ:
    next_card()
```

Day 32: Automated Birthday Wisher

This project automates sending birthday emails. It introduces two crucial standard libraries: `smtplib` for sending emails and `datetime` for handling dates and times.

Key Concepts

1. `datetime` Module:

- A built-in Python module to work with dates and times.
- `datetime.now()` : Gets the current date and time.
- `now.weekday()` : Returns the day of the week as an integer (Monday is 0, Sunday is 6).
- `now.month` , `now.day` : Access specific parts of the date.
- This is essential for checking if today is someone's birthday.

2. `smtplib` Module:

- **S**imple **M**ail **T**ransfer **P**rotocol library. It defines how to send emails between servers.
- `smtplib.SMTP()` : Creates a connection object. You need the SMTP server address of your email provider (e.g., `smtp.gmail.com`).
- `connection.starttls()` : **T**ransport **L**ayer **S**ecurity. This encrypts the connection, making it secure. It's a mandatory step.
- `connection.login(user=my_email, password=my_password)` : Logs you into your email account.
Important: For services like Gmail, you must use an "App Password," not your regular login password, for security reasons.
- `connection.sendmail(from_addr, to_addrs, msg)` : Sends the email. The message should be formatted correctly, including `Subject:` and the body.
- `connection.close()` : Closes the connection.

Code Implementation Snippet

```
import smtplib
import datetime as dt
import random

MY_EMAIL = "your_email@gmail.com"
MY_PASSWORD = "your_app_password" # Use an app-specific password

now = dt.datetime.now()
today_tuple = (now.month, now.day)

# Assuming birthdays.csv has columns: name,email,year,month,day
import pandas
data = pandas.read_csv("birthdays.csv")
birthdays_dict = {(data_row["month"], data_row["day"]): data_row for (index, data_row) in data.}

if today_tuple in birthdays_dict:
    birthday_person = birthdays_dict[today_tuple]
    file_path = f"letter_templates/letter_{random.randint(1,3)}.txt"
    with open(file_path) as letter_file:
        contents = letter_file.read()
        contents = contents.replace("[NAME]", birthday_person["name"])

    with smtplib.SMTP("smtp.gmail.com", port=587) as connection:
        connection.starttls()
        connection.login(MY_EMAIL, MY_PASSWORD)
        connection.sendmail(
            from_addr=MY_EMAIL,
            to_addrs=birthday_person["email"],
            msg=f"Subject:Happy Birthday!\n\n{contents}"
        )
```

Day 33: API Endpoints & ISS Overhead Notifier

This day marks the transition to working with external data via APIs. An API is a way for different software applications to communicate with each other.

Key Concepts

1. What is an API?

- **Application Programming Interface.** It's a set of rules and definitions that allows one application to request services or data from another.
- Think of it like a waiter in a restaurant. You (your program) don't go into the kitchen (the server/database). You give your order (an API request) to the waiter (the API), who brings you your food (the data/response).

2. **API Endpoint:** A specific URL where an API can be accessed. For example,

`http://api.open-notify.org/iss-now.json` is the endpoint to get the current location of the International Space Station.

3. **requests Library:** The standard Python library for making HTTP requests.

- `requests.get(url)` : Makes a GET request to the specified URL to retrieve data.
- **Response Object:** The result of a request. It contains the status code, data, headers, etc.
- `response.status_code` : A number indicating the result. `200` means success. `404` means not found. `401` means not authorized.
- `response.raise_for_status()` : This is a great practice. It will automatically raise an exception if the request was unsuccessful (i.e., status code was not 200).
- `response.json()` : If the API returns data in JSON format (which is very common), this method will automatically parse it into a Python dictionary.

Code Implementation Snippet (ISS Notifier)

```
import requests
from datetime import datetime
import smtplib
import time

MY_LAT = 51.507351 # Your latitude
MY_LONG = -0.127758 # Your longitude

def is_iss_overhead():
    response = requests.get(url="http://api.open-notify.org/iss-now.json")
    response.raise_for_status()
    data = response.json()

    iss_latitude = float(data["iss_position"]["latitude"])
    iss_longitude = float(data["iss_position"]["longitude"])

    # Your position is within +5 or -5 degrees of the ISS position.
    if MY_LAT-5 <= iss_latitude <= MY_LAT+5 and MY_LONG-5 <= iss_longitude <= MY_LONG+5:
        return True
    return False

def is_night():
    parameters = {
        "lat": MY_LAT,
        "lng": MY_LONG,
        "formatted": 0, # Get time in 24h format
    }
    response = requests.get("https://api.sunrise-sunset.org/json", params=parameters)
    response.raise_for_status()
    data = response.json()
    sunrise = int(data["results"]["sunrise"].split("T")[1].split(":")[0])
    sunset = int(data["results"]["sunset"].split("T")[1].split(":")[0])

    time_now = datetime.now().hour

    if time_now >= sunset or time_now <= sunrise:
        return True # It's dark
    return False

# Main loop to run the check
while True:
```

```
time.sleep(60) # Wait 60 seconds between checks
if is_iss_overhead() and is_night():
    # Code to send an email (from Day 32)
    print("Look up!")
```

Day 34: API Practice - GUI Quiz App

This project combines the skills from Day 31 (Tkinter) and Day 33 (APIs) to create a quiz application that fetches questions from a public API.

Key Concepts

1. API Parameters:

- Many APIs allow you to customize the request by adding parameters to the URL. For example, you might request 10 questions of a specific category and difficulty.
- The `requests` library makes this easy: you pass a dictionary of parameters to the `params` argument in `requests.get()`.
- `parameters = {"amount": 10, "type": "boolean"}`
- `response = requests.get(url="https://opentdb.com/api.php", params=parameters)`

2. Type Hinting & Data Classes:

- As applications get more complex, using type hints (`question_text: str`) improves code readability and helps catch errors.
- This is a step towards using tools like **Pydantic** (mentioned in your NEETPrepGPT plan) which uses type hints to perform data validation, parsing, and serialization. It ensures the data you get from an API matches the structure you expect.

3. HTML Character Entities:

- Data from APIs often contains HTML character entities (e.g., `"` for a quote mark, `'` for an apostrophe).
- The built-in `html` module can be used to decode these: `html.unescape(text)`.

4. Structuring with Classes:

- The project is structured into multiple classes to separate concerns:
 - `Question` : A simple data model for a single question (text and answer).
 - `QuizBrain` : The logic engine. It keeps track of the score, the current question number, and checks answers.
 - `UI` : The Tkinter class responsible for displaying the question, score, and buttons. This separation makes the code much cleaner and easier to manage.

Code Implementation Snippet (UI Class)

```
from tkinter import *
from quiz_brain import QuizBrain
import html

THEME_COLOR = "#375362"

class QuizInterface:
    def __init__(self, quiz_brain: QuizBrain):
        self.quiz = quiz_brain
        self.window = Tk()
        self.window.title("Quizzler")
        self.window.config(padx=20, pady=20, bg=THEME_COLOR)

        # ... (score label setup) ...

        self.canvas = Canvas(width=300, height=250, bg="white")
        self.question_text = self.canvas.create_text(
            150, 125,
            width=280, # Text wrapping
            text="Some Question Text",
            fill=THEME_COLOR,
            font=("Arial", 20, "italic")
        )
        self.canvas.grid(row=1, column=0, columnspan=2, pady=50)

        # ... (button setup) ...

        self.get_next_question()
        self.window.mainloop()

    def get_next_question(self):
        self.canvas.config(bg="white")
        if self.quiz.still_has_questions():
            self.score_label.config(text=f"Score: {self.quiz.score}")
            q_text = self.quiz.next_question()
            self.canvas.itemconfig(self.question_text, text=html.unescape(q_text))
        else:
            self.canvas.itemconfig(self.question_text, text="You've reached the end of the quiz.")
            # Disable buttons

    def true_pressed(self):
```



```

        self.give_feedback(self.quiz.check_answer("True"))

def false_pressed(self):
    self.give_feedback(self.quiz.check_answer("False"))

def give_feedback(self, is_right):
    if is_right:
        self.canvas.config(bg="green")
    else:
        self.canvas.config(bg="red")
    self.window.after(1000, self.get_next_question)

```

Day 35: API Keys, Authentication & Environment Variables

This project, a "Rain Alert" app, introduces secure API practices by using authentication and environment variables to protect sensitive information like API keys.

Key Concepts

1. API Authentication & API Keys:

- Most powerful APIs require you to authenticate to track usage, prevent abuse, and sometimes charge for services.
- The most common method is an **API Key**, which is a unique string of characters you include in your request to identify yourself.
- This is often passed as a URL parameter (e.g., `&appid=YOUR_API_KEY`).

2. Environment Variables:

- **NEVER** hard-code sensitive information like API keys, passwords, or tokens directly into your source code. If you upload your code to a public repository like GitHub, your keys will be exposed.
- **Environment Variables** are variables stored outside your program, in the operating system. Your program can then read these variables at runtime.
- Python's `os` module is used to access them: `api_key = os.environ.get("OWM_API_KEY")`.
- This is a fundamental practice for production-grade applications like your NEETPrepGPT project.

3. Twilio API for Sending SMS:

- An example of a service API that allows you to programmatically send and receive SMS messages.
- It requires an Account SID and an Auth Token for authentication.

Code Implementation Snippet

```
import requests
import os
from twilio.rest import Client # Example SMS library

# --- Getting Environment Variables ---
# In your terminal (for Linux/Mac): export OWM_API_KEY="your_key_here"
# In your terminal (for Windows): set OWM_API_KEY="your_key_here"
# Or use a .env file and a library like python-dotenv

OWM_Endpoint = "https://api.openweathermap.org/data/2.5/onecall"
api_key = os.environ.get("OWM_API_KEY")
account_sid = os.environ.get("TWILIO_ACCOUNT_SID")
auth_token = os.environ.get("TWILIO_AUTH_TOKEN")

weather_params = {
    "lat": 28.6667, # Your location
    "lon": 77.2167,
    "appid": api_key,
    "exclude": "current,minutely,daily" # Get only hourly forecast
}

response = requests.get(OWM_Endpoint, params=weather_params)
response.raise_for_status()
weather_data = response.json()

# Slice to get the next 12 hours of weather data
weather_slice = weather_data["hourly"][0:12]

will_rain = False
for hour_data in weather_slice:
    condition_code = hour_data["weather"][0]["id"]
    if int(condition_code) < 700: # Codes below 700 indicate some form of precipitation
        will_rain = True

if will_rain:
    client = Client(account_sid, auth_token)
    message = client.messages.create(
        body="It's going to rain today. Remember to bring an ☔",
        from_='+15017122661', # Your Twilio phone number
        to='+911234567890' # Your verified phone number
    )
```

```
)  
print(message.status)
```

Day 36: Stock Trading News Alert Project

This project integrates two different APIs: one for stock price data (Alpha Vantage) and another for news (News API). It demonstrates how to combine and process data from multiple sources to create a useful alert.

Key Concepts

1. **Chaining API Calls:** The logic is sequential:

- **Step 1:** Call the Stock API to get recent price data.
- **Step 2:** Analyze the data. Calculate the percentage difference between the last two days.
- **Step 3 (Conditional):** If the percentage change is significant (e.g., > 5%), then proceed to call the News API.
- **Step 4:** Send an alert (SMS or email) with the stock change and the top 3 related news headlines.

2. **Data Slicing and Manipulation:**

- The stock API returns a time series of data. You need to access the most recent two days.
- Python's list slicing (`data_list[0:2]`) is perfect for this.
- Converting string values for prices to floats (`float()`) is necessary for calculations.

Code Implementation Snippet (Logic)

```
import requests
import os

STOCK_NAME = "TSLA"
COMPANY_NAME = "Tesla Inc"

STOCK_ENDPOINT = "https://www.alphavantage.co/query"
NEWS_ENDPOINT = "https://newsapi.org/v2/everything"

STOCK_API_KEY = os.environ.get("STOCK_API_KEY")
NEWS_API_KEY = os.environ.get("NEWS_API_KEY")

# --- Step 1 & 2: Get stock data and calculate difference ---
stock_params = {
    "function": "TIME_SERIES_DAILY",
    "symbol": STOCK_NAME,
    "apikey": STOCK_API_KEY
}
response = requests.get(STOCK_ENDPOINT, params=stock_params)
data = response.json()["Time Series (Daily)"]
data_list = [value for (key, value) in data.items()]
yesterday_data = data_list[0]
yesterday_closing_price = float(yesterday_data["4. close"])

day_before_yesterday_data = data_list[1]
day_before_yesterday_closing_price = float(day_before_yesterday_data["4. close"])

difference = yesterday_closing_price - day_before_yesterday_closing_price
up_down = " ▲ " if difference > 0 else " ▼ "
diff_percent = round((difference / yesterday_closing_price) * 100)

# --- Step 3: If difference is significant, get news ---
if abs(diff_percent) > 5: # Check for a 5% or greater change
    news_params = {
        "apiKey": NEWS_API_KEY,
        "qInTitle": COMPANY_NAME, # Search for company name in title
    }
    news_response = requests.get(NEWS_ENDPOINT, params=news_params)
    articles = news_response.json()["articles"]
    three_articles = articles[:3] # Get first 3 articles
```

```
# --- Step 4: Format and send alerts ---
formatted_articles = [f"{STOCK_NAME}: {up_down}{diff_percent}%\nHeadline: {article['title']}

# Send each article as a separate message/email
for article in formatted_articles:
    # Code to send SMS/email goes here
    print(article)
```

Day 37: Habit Tracker with Pixela API (POST, PUT, DELETE)

This project moves beyond just fetching data (GET) to modifying data on a server using more advanced HTTP requests: POST , PUT , and DELETE .

Key Concepts

1. More HTTP Verbs:

- **GET** : Retrieve data from a server.
- **POST** : Send data to a server to create a new resource (e.g., create a user, post a new data point).
- **PUT** : Update an existing resource on the server (e.g., change the color of a graph).
- **DELETE** : Remove a resource from the server.

2. requests Library for Advanced Requests:

- `requests.post(url, json=payload, headers=headers)`
- `requests.put(url, json=payload, headers=headers)`
- `requests.delete(url, headers=headers)`
- **json parameter:** Used to send a Python dictionary as the JSON body of the request.
- **headers parameter:** Used to send additional information, like authentication tokens. Some APIs require tokens to be sent in the request header for security.

Code Implementation Snippets

```
import requests
from datetime import datetime
import os

USERNAME = "your-username"
TOKEN = os.environ.get("PIXELA_TOKEN") # Stored as environment variable
GRAPH_ID = "graph1"

pixela_endpoint = "https://pixe.la/v1/users"
user_params = {
    "token": TOKEN,
    "username": USERNAME,
    "agreeTermsOfService": "yes",
    "notMinor": "yes",
}

# --- 1. POST - Create a user (only need to run once) ---
# response = requests.post(url=pixela_endpoint, json=user_params)
# print(response.text)

# --- 2. POST - Create a graph definition ---
graph_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs"
graph_config = {
    "id": GRAPH_ID,
    "name": "Cycling Graph",
    "unit": "Km",
    "type": "float",
    "color": "sora"
}
headers = {
    "X-USER-TOKEN": TOKEN
}
# response = requests.post(url=graph_endpoint, json=graph_config, headers=headers)
# print(response.text)

# --- 3. POST - Post a value (a pixel) to the graph ---
pixel_creation_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}"
today = datetime.now()
pixel_data = {
    "date": today.strftime("%Y%m%d"), # Format date as YYYYMMDD
    "quantity": "15.5",
```

```

}
response = requests.post(url=pixel_creation_endpoint, json=pixel_data, headers=headers)
print(response.text)

# --- 4. PUT - Update a pixel ---
update_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}/{today.strftime('%Y%m%d')}"
new_pixel_data = {
    "quantity": "10.2"
}
# response = requests.put(url=update_endpoint, json=new_pixel_data, headers=headers)
# print(response.text)

# --- 5. DELETE - Delete a pixel ---
delete_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}/{today.strftime('%Y%m%d')}"
# response = requests.delete(url=delete_endpoint, headers=headers)
# print(response.text)

```

Day 38: Workout Tracking with Google Sheets

This project uses a Natural Language Processing (NLP) API from Nutritionix to interpret plain English workout descriptions and then posts the structured data to a Google Sheet via the Sheety API.

Key Concepts

1. Natural Language Processing (NLP) APIs:

- These APIs are trained to understand and process human language.
- In this case, you send a sentence like "ran 3 miles and swam for 20 minutes" , and the API returns structured data like exercises, durations, and calories burned.

2. Sheety API:

- A simple service that turns any Google Sheet into an API.
- This is an example of an "API-as-a-service" that simplifies integration. You can use `GET` to read rows, `POST` to add a new row, `PUT` to update a row, and `DELETE` to remove one.

3. Authentication:

- **Nutritionix:** Uses an App ID and API Key in the request headers.
- **Sheety:** Can use "Basic Authentication" or a "Bearer Token," also sent in the headers. Bearer Token auth is common in modern APIs (like the ones used in your NEETPrepGPT project with JWT).

Code Implementation Snippet

```
import requests
from datetime import datetime
import os

# --- Nutritionix API Config ---
GENDER = "male"
WEIGHT_KG = 72.5
HEIGHT_CM = 167.6
AGE = 30
APP_ID = os.environ.get("NUTRITIONIX_APP_ID")
API_KEY = os.environ.get("NUTRITIONIX_API_KEY")
exercise_endpoint = "https://trackapi.nutritionix.com/v2/natural/exercise"

# --- Sheety API Config ---
sheet_endpoint = os.environ.get("SHEETY_ENDPOINT")
SHEETY_TOKEN = os.environ.get("SHEETY_TOKEN")

exercise_text = input("Tell me which exercises you did: ")

headers = {
    "x-app-id": APP_ID,
    "x-api-key": API_KEY,
}

parameters = {
    "query": exercise_text,
    "gender": GENDER,
    "weight_kg": WEIGHT_KG,
    "height_cm": HEIGHT_CM,
    "age": AGE,
}

# --- Step 1: Get structured data from Nutritionix ---
response = requests.post(exercise_endpoint, json=parameters, headers=headers)
result = response.json()

# --- Step 2: Post the data to Google Sheets via Sheety ---
today_date = datetime.now().strftime("%d/%m/%Y")
now_time = datetime.now().strftime("%X")

bearer_headers = {
```



```

    "Authorization": f"Bearer {SHEETY_TOKEN}"
}

for exercise in result["exercises"]:
    sheet_inputs = {
        "workout": {
            "date": today_date,
            "time": now_time,
            "exercise": exercise["name"].title(),
            "duration": exercise["duration_min"],
            "calories": exercise["nf_calories"]
        }
    }

    sheet_response = requests.post(
        sheet_endpoint,
        json=sheet_inputs,
        headers=bearer_headers
    )
    print(sheet_response.text)

```

Days 39 & 40: Capstone Project - Flight Deal Finder

This is a major two-part capstone project that brings together everything learned so far: APIs, authentication, data processing, and notifications. The goal is to build a service that checks for cheap flights to destinations you're interested in and notifies you when it finds a deal.

Key Concepts & Project Structure

The project is broken down into several classes, promoting good **Object-Oriented Programming (OOP)** practices, which is essential for large projects like NEETPrepGPT.

1. DataManager Class (Day 39):

- **Responsibility:** Manages all communication with the Google Sheet (via Sheety API).
- `get_destination_data()` : Reads the data from your sheet (which contains cities you want to fly to and your target price).
- `update_destination_codes()` : After finding the IATA codes for cities, this method writes them back to the sheet using `PUT` requests.

2. FlightSearch Class (Day 39):

- **Responsibility:** Manages all communication with the Flight Search API (Tequila by [Kiwi.com](https://kiwi.com/)).
- `get_destination_code(city_name)` : Takes a city name and queries the Tequila API to find its IATA (airport) code (e.g., "Paris" -> "PAR").
- `check_flights(...)` : The main search function. It takes an origin city code, destination city code, and date range, then queries the API for the cheapest flight.

3. `FlightData` Class (Day 40):

- **Responsibility:** A simple data structure class. Its purpose is to structure the flight data found by `FlightSearch` in a clean, organized way. This prevents passing around messy dictionaries.

4. `NotificationManager` Class (Day 40):

- **Responsibility:** Handles sending notifications (SMS via Twilio or Email via `smtplib`).
- `send_sms(message)` : Contains the logic to interact with the Twilio API.

Main Logic Flow (main.py) (Day 40)

```
from data_manager import DataManager
from flight_search import FlightSearch
from notification_manager import NotificationManager
from datetime import datetime, timedelta

ORIGIN_CITY_IATA = "LON" # Example: London

# Initialize all the manager classes
data_manager = DataManager()
flight_search = FlightSearch()
notification_manager = NotificationManager()

# 1. Get destination data from Google Sheet
sheet_data = data_manager.get_destination_data()

# 2. Fill in IATA Codes if they are missing in the sheet
if sheet_data[0]["iataCode"] == "":
    for row in sheet_data:
        row["iataCode"] = flight_search.get_destination_code(row["city"])
    data_manager.destination_data = sheet_data
    data_manager.update_destination_codes()

# 3. Search for flights for each destination
tomorrow = datetime.now() + timedelta(days=1)
six_month_from_today = datetime.now() + timedelta(days=(6 * 30))

for destination in sheet_data:
    flight = flight_search.check_flights(
        ORIGIN_CITY_IATA,
        destination["iataCode"],
        from_time=tomorrow,
        to_time=six_month_from_today
    )

# 4. If a cheap flight is found, send a notification
if flight is not None and flight.price < destination["lowestPrice"]:
    message = (
        f"Low price alert! Only £{flight.price} to fly from "
        f"{flight.origin_city}-{flight.origin_airport} to "
        f"{flight.destination_city}-{flight.destination_airport}, "
        f"from {flight.out_date} to {flight.return_date}."
```


```
)  
notification_manager.send_sms(message=message)
```

This structured, multi-class approach is highly scalable and maintainable, representing a significant step towards building professional-quality software.

Python Notes: Days 41-50 (Angela Yu's 100 Days of Code)

This section marks a shift from pure Python programming to web technologies. You'll learn the language of the web (HTML/CSS) to understand how websites are built, then use Python libraries like **Beautiful Soup** and **Selenium** to scrape data and automate browser actions.

Day 41: Introduction to Web Foundations with HTML

 **Goal:** Understand the basic structure and syntax of HTML (HyperText Markup Language). This is not Python, but knowing HTML is essential for web scraping.

Key Concepts

- **What is HTML?** It's the standard markup language used to create web pages. It describes the structure of a web page using elements.
- **HTML Boilerplate:** The essential structure every HTML file needs.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My Website</title>
</head>
<body>
  </body>
</html>
```

- **HTML Tags:** Keywords surrounded by angle brackets that define how your web browser will format and display the content.
 - `<h1>` to `<h6>` : Headings.
 - `<p>` : Paragraph.
 - `` or `<i>` : Emphasized or italicized text.
 - `` or `` : Important or bold text.
 - `
` : Line break.
 - `<hr>` : Horizontal rule (a thematic break).

- **Lists:**
 - `` : Unordered List (bullet points).
 - `` : Ordered List (numbered).
 - `` : List Item (used inside `` or ``).
- **Nesting:** Placing tags inside other tags. Proper nesting is crucial for a valid HTML structure.

```
<p>This is a <strong>bold</strong> paragraph.</p>
```

Project Focus: Personal Site

You'll start building a simple personal website using only HTML. This forces you to get comfortable with structuring content using different tags before you add any styling.

Day 42: Intermediate HTML

 **Goal:** Learn more advanced HTML elements for creating complex layouts and embedding content.

Key Concepts

- **Image Tag:** `` is a self-closing tag used to embed images.
 - `src` : The source attribute, which is the path to the image file.
 - `alt` : The alternate text attribute, which describes the image for screen readers and if the image fails to load.

```

```

- **Anchor Tag:** `<a>` creates a hyperlink to other web pages, files, or locations within the same page.
 - `href` : The hyperlink reference attribute, which specifies the URL.

```
<a href="https://www.google.com">Go to Google</a>
```

- **Tables:** Used to display data in rows and columns.
 - `<table>` : The main table container.
 - `<tr>` : Table Row.
 - `<th>` : Table Header (a cell with bold, centered text).
 - `<td>` : Table Data (a standard cell).

```

<table>
  <tr>
    <th>Name</th>
    <th>Role</th>
  </tr>
  <tr>
    <td>Angela</td>
    <td>Instructor</td>
  </tr>
</table>

```

- **Forms:** Collect user input.
 - `<form>` : The container for different types of input elements.
 - `<input>` : Can be text fields, checkboxes, password fields, radio buttons, and submit buttons.
 - `<label>` : A caption for an item in a user interface.

Project Focus: Structuring a CV

You'll use tables and other elements to structure your personal CV/resume in HTML, giving it a clear, organized layout.

Day 43: Introduction to CSS

 **Goal:** Learn how to style HTML elements using CSS (Cascading Style Sheets).

Key Concepts

- **What is CSS?** It's the language we use to style an HTML document. CSS describes how HTML elements should be displayed.
- **Ways to Add CSS:**
 - Inline CSS:** Using the `style` attribute directly inside an HTML tag (generally bad practice).

```
<h1 style="color:blue;">Blue Heading</h1>
```

- Internal CSS:** Placing `<style>` tags within the `<head>` section of the HTML file.

```
<head>
  <style>
    body {
      background-color: lightblue;
    }
  </style>
</head>
```

iii. **External CSS:** Linking to a separate `.css` file (best practice for organization).

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

- **CSS Selectors:** Patterns used to select the element(s) you want to style.
 - **Tag Selector:** Selects all elements with that tag name (e.g., `p`, `h1`, `body`).
 - **Class Selector:** Selects elements with a specific class attribute. Starts with a dot (`.`).

```
.highlight { color: yellow; }
```


- **ID Selector:** Selects one unique element with a specific id attribute. Starts with a hash (`#`).

```
#main-title { font-size: 40px; }
```

Project Focus: Styling Your Personal Site

You'll take the HTML site you built and apply CSS to change colors, fonts, and spacing, making it visually appealing.

Day 44: Intermediate CSS

 **Goal:** Dive deeper into CSS for creating professional layouts, focusing on the Box Model and positioning.

Key Concepts



- **The Box Model:** Every HTML element is essentially a box. This model describes how the different parts of the box (content, padding, border, margin) work together.
 - **Content:** The text, image, or other media in the element.
 - **Padding:** The transparent space around the content, inside the border.
 - **Border:** A line that goes around the padding and content.

- **Margin:** The transparent space around the border, separating it from other elements.
- **Positioning:** Controlling where elements appear on the page.
 - `static` : Default value. Elements render in order, as they appear in the document flow.
 - `relative` : The element is positioned relative to its normal position.
 - `absolute` : The element is positioned relative to its nearest positioned ancestor.
 - `fixed` : The element is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled.
 - `sticky` : A hybrid of `relative` and `fixed`.
- **Floats and Clears:** An older method for layout, often used to wrap text around images. (Modern layouts often use Flexbox or Grid instead).

Project Focus: Advanced Personal Site Layout

You'll refactor your personal site's CSS to use more advanced positioning and box model properties, creating a more complex and responsive layout.

Day 45: Web Scraping with BeautifulSoup

 **Goal:** Use Python to parse HTML from a live website and extract useful information. This is where your Python and new web knowledge combine! 

Key Concepts

- **Web Scraping:** The process of automatically extracting data from websites.
- **Libraries:**
 - `requests` : A simple and elegant Python library for making HTTP requests to get the HTML content from a URL.
 - `BeautifulSoup` : A library for pulling data out of HTML and XML files. It creates a parse tree that makes it easy to navigate, search, and modify the tree.
- **Workflow:**
 - Install libraries:** `pip install requests beautifulsoup4`
 - Fetch the page:** Use `requests.get(URL)` to download the HTML.
 - Create a "soup" object:** `soup = BeautifulSoup(html_content, "html.parser")`
 - Find elements:** Use `soup.find()`, `soup.find_all()`, or `soup.select()` to locate the data you need using tags, classes, or IDs.
 - Extract data:** Get the text or attributes from the found elements using `.getText()` or `element['attribute_name']`.

Code Example

```
import requests
from bs4 import BeautifulSoup

URL = "https://news.ycombinator.com/"

response = requests.get(URL)
website_html = response.text

soup = BeautifulSoup(website_html, "html.parser")


# Find the first article title
first_article_tag = soup.find(name="span", class_="titleline").find("a")
article_text = first_article_tag.getText()
article_link = first_article_tag.get("href")

print(f"Title: {article_text}")
print(f"Link: {article_link}")
```

Project Focus: 100 Movies to Watch

You'll scrape a website (like Empire's 100 Greatest Movies) to get a list of movie titles, then save them to a .txt file in order.

Day 46: Automating Spotify with Selenium

 **Goal:** Go beyond static scraping. Learn to control a web browser programmatically to perform actions on a dynamic website.

Key Concepts

- **Static vs. Dynamic Websites:**
 - **Static:** The content is fixed and delivered as-is (perfect for BeautifulSoup).
 - **Dynamic:** Content is loaded or changed with JavaScript after the initial page load (e.g., infinite scroll, pop-ups). BeautifulSoup can't handle this because it doesn't run JavaScript.
- **Selenium WebDriver:** A browser automation framework. It allows your Python script to open a browser, navigate to URLs, and interact with page elements just like a human would (clicking buttons, filling forms, etc.).

- **Setup:**

- i. Install Selenium: `pip install selenium`
- ii. Download a `WebDriver` for your browser (e.g., `chromedriver` for Chrome). Make sure its version matches your browser's version.

- **Basic Commands:**

- `driver = webdriver.Chrome()` : Opens a new Chrome browser window.
- `driver.get(URL)` : Navigates to a URL.
- `driver.find_element(By.NAME, "q")` : Finds a single element. You can find by `ID` , `NAME` , `CLASS_NAME` , `CSS_SELECTOR` , etc.
- `driver.quit()` : Closes the browser window.

Code Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By

# Path to your chromedriver executable
chrome_driver_path = "/path/to/your/chromedriver"
driver = webdriver.Chrome(executable_path=chrome_driver_path)

driver.get("https://www.python.org")


# Find an element by its CSS Selector
search_bar = driver.find_element(By.NAME, "q")
print(search_bar.get_attribute("placeholder")) # Prints "Search"

driver.quit()
```

Project Focus: Spotify Playlist Bot

You will use the `spotipy` library to authenticate with Spotify and Selenium to scrape the Billboard Hot 100 for a specific date, then create a new Spotify playlist with all those songs.

Day 47: Amazon Price Tracker Bot

 **Goal:** Combine web scraping and browser automation to build a useful tool that monitors a product price and notifies you.

Key Concepts


- **Finding Elements with Selenium:** Mastering different `By` strategies is key.
 - `By.ID` : Good for unique elements.
 - `By.NAME` : Often used for form inputs.
 - `By.XPATH` : A powerful way to navigate the HTML structure, even for complex paths.
 - `By.CSS_SELECTOR` : Very flexible and often more readable than XPath.
- **Extracting Text:** Once you have an element, use the `.text` attribute to get the visible text content.
- **Data Cleaning:** The text you scrape is often messy (e.g., `$19.99\n(Sale)`). You'll need to use Python string methods like `.strip()` , `.split()` , and type casting (`float()`) to clean it up into a usable number.
- **Sending Emails with Python:**
 - Use the built-in `smtplib` library to connect to an email server (like Gmail, Outlook) and send an email. This is how your bot will notify you.

Project Focus: Amazon Price Tracker

Your script will:

1. Use Selenium to navigate to an Amazon product page.
2. Scrape the current price of the item.
3. Clean the price data and convert it to a float.
4. Compare the current price to your target price.
5. If the price is at or below your target, use `smtplib` to send you an email alert.

Day 48: Advanced Selenium & Game Playing Bot

 **Goal:** Learn how to handle websites that take time to load content and how to perform more complex user actions.

Key Concepts

- **Waiting Strategies:** A major challenge in automation is that scripts can run faster than a webpage loads. If your script tries to find an element before it exists, it will crash.
 - **Implicit Wait:** `driver.implicitly_wait(5)` tells Selenium to wait up to 5 seconds for an element to appear before throwing an error. It's a global setting.

- **Explicit Wait (Preferred):** More precise. You tell Selenium to wait for a *specific condition* to be met. This is more reliable.

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Wait up to 10 seconds until the element with ID 'myElement' is clickable
element = WebDriverWait(driver, 10).until(
    EC.element_to_be_clickable((By.ID, "myElement"))
)
```


- **ActionChains:** For performing complex actions like hovering, drag-and-drop, or key presses.
- **Sending Keystrokes:** The `.send_keys()` method can be used to type into input fields or to send special keys like `Keys.ENTER`.

Project Focus: Cookie Clicker Bot

You'll create a bot that plays the Cookie Clicker game. It will:

1. Repeatedly click the giant cookie as fast as possible.
2. After a set interval (e.g., every 5 seconds), it will check the store for affordable upgrades.
3. It will buy the most expensive affordable upgrade to maximize cookie production.
4. This project heavily relies on managing time and checking for interactable elements.

Day 49: Automating Job Applications on LinkedIn

 **Goal:** Apply your Selenium skills to a real-world, multi-step workflow: automating job applications.

Key Concepts


- **Workflow Automation:** Breaking down a complex task into a series of discrete, repeatable steps that your script can execute.
 - Step 1: Go to LinkedIn jobs page.
 - Step 2: Sign in.
 - Step 3: Search for jobs with specific keywords and filters.
 - Step 4: Iterate through the search results.
 - Step 5: For each job, click the "Easy Apply" button.
 - Step 6: Fill out the application form (phone number, resume).
 - Step 7: Submit the application.

- **Handling Pop-ups and New Windows:** Sometimes clicking a link opens a new tab or a modal window. You need to learn how to switch the driver's focus to the new context.
- **Error Handling:** What happens if a job isn't "Easy Apply"? Or if a field is missing? Using `try...except` blocks is crucial for making your bot robust so it doesn't crash on the first unexpected event.

Project Focus: LinkedIn Job Application Bot

You will build a bot that can log in to LinkedIn, search for jobs using filters you define, and automatically complete the "Easy Apply" process for each listing it finds. **Disclaimer:** Use this ethically and sparingly. The goal is the learning exercise, not spamming recruiters.

Day 50: Auto Tinder Swiping Bot

 **Goal:** Build another complex automation bot that deals with pop-ups, dynamic content, and making decisions based on scraped information.

Key Concepts

- **Consolidating Skills:** This project combines everything you've learned about Selenium:
 - Logging into a site with credentials.
 - Using explicit waits to handle pop-ups (location access, notifications, etc.).
 - Finding elements using complex CSS selectors or XPath.
 - Sending clicks to perform actions (swiping left or right).
 - Creating a loop to perform the main action repeatedly.
 - Using `try...except` to gracefully handle interruptions like running out of likes or getting a match.
- **Decision Logic:** The bot needs a simple rule to follow, e.g., "swipe right on everyone" or a more complex (and likely ineffective) rule based on limited profile info. The focus is on the automation, not the decision-making.

Project Focus: Tinder Swiping Bot

The goal is to create a bot that logs into Tinder, dismisses all the initial pop-ups, and then enters a loop to automatically swipe right on profiles. It's a fun and challenging exercise in controlling a modern, dynamic web application.

Python Notes: Days 51-60 (Angela Yu's 100 Days of Code)

Here are the detailed notes for Days 51 through 60, covering advanced browser automation with Selenium and the fundamentals of web development with the Flask framework.

Day 51: Internet Speed Twitter Complaint Bot

Objective: Build a bot that checks your internet speed and, if it's below a promised threshold, automatically logs into Twitter and posts a complaint.

Key Concepts:

- **Selenium WebDriver:** An automation tool that allows your Python script to control a web browser. It simulates a real user clicking, typing, and navigating through web pages.
- **Locating Elements:** Finding specific HTML elements on a page to interact with them. Selenium provides various methods:
 - `find_element(By.ID, "value")`
 - `find_element(By.NAME, "value")`
 - `find_element(By.XPATH, "xpath_expression")`
 - `find_element(By.CSS_SELECTOR, "css_selector")`
 - `find_element(By.CLASS_NAME, "class_name")`
- **Explicit Waits:** Pausing your script until a certain condition is met (e.g., an element becomes clickable). This is crucial for dealing with modern websites that load content dynamically (asynchronously). Without waits, your script might try to click a button before it has appeared, causing an error.
 - Use `WebDriverWait` combined with `expected_conditions`.

Project Workflow:

1. Setup:

- Install Selenium: `pip install selenium`
- Download the appropriate `WebDriver` for your browser (e.g., `chromedriver` for Chrome).
- Store sensitive information (Twitter username, password, promised speeds) in variables or environment variables.

2. Part 1: Get Internet Speed

- The bot navigates to `https://www.speedtest.net/`.
- It waits for the "GO" button to be clickable and then clicks it.

- It waits for the results to appear (e.g., wait until the download/upload speed elements are visible). This can take a minute.
- It scrapes the download and upload speed text from the page.

3. Part 2: Tweet the Complaint

- The bot checks if the scraped speeds are below the promised speeds.
- If they are, it opens a new tab and navigates to Twitter.
- **Login:** It finds the username/email field, types the email, clicks next, finds the password field, types the password, and clicks the login button. *Note: Twitter's login flow can change; you may need to handle intermediate steps like entering a username.*
- **Compose Tweet:** It waits for the tweet input box to be visible, then types out a formatted complaint message including the actual download/upload speeds.
- It clicks the "Tweet" button to post.

Example Code Snippet (Waiting and Clicking):

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

# --- Inside a class method ---
# Wait for the "GO" button to be clickable for a maximum of 60 seconds
go_button = WebDriverWait(self.driver, 60).until(
    EC.element_to_be_clickable((By.CSS_SELECTOR, ".start-button a"))
)
go_button.click()

# Wait for download speed element to be present
time.sleep(60) # A simple but less robust wait
down_speed_element = self.driver.find_element(By.CSS_SELECTOR, ".download-speed")
self.down = float(down_speed_element.text)
```

Day 52: Instagram Follower Bot 🤖

Objective: Create a bot that logs into Instagram, finds a target account, and follows all of that account's followers.

Key Concepts:

- **Handling Pop-ups:** Websites often have pop-ups (e.g., "Save Login Info?", "Turn on Notifications"). The script must identify and dismiss them to continue.
- **Scrolling Inside an Element:** To load all followers, you need to scroll down within the followers list pop-up, not the main page. This requires JavaScript execution.
- **Exception Handling:** The script should be robust. Use `try-except` blocks to handle cases where an element isn't found or an unexpected pop-up appears.

Project Workflow:

1. Login to Instagram:

- Navigate to Instagram's login page.
- Use `time.sleep()` initially to give the page time to load (later, explicit waits are better).
- Find the username and password fields and `send_keys()`.
- Click the login button.
- Handle the "Save Info" and "Turn on Notifications" pop-ups by finding the "Not Now" buttons and clicking them.

2. Find Target Account's Followers:

- Navigate to the target account's profile (e.g., https://www.instagram.com/target_username/).
- Click on the "followers" link to open the followers list modal.

3. Scrape and Follow:

- The followers list is a scrollable pop-up.
- To scroll it, you need to get the modal element and then execute a JavaScript command.
- **JavaScript Execution:**

```
driver.execute_script("arguments[0].scrollTop = arguments[0].scrollHeight", modal_element)
```
- The bot enters a loop:
 - Find all the "Follow" buttons inside the modal.
 - Iterate through the buttons and click each one.
 - Use a `try-except` block because some buttons might change to "Following" or be blocked, causing errors.
 - After a cycle of following, scroll the modal down to load more followers.
 - Repeat until the desired number of users have been followed or the bottom of the list is reached.

Example Code Snippet (Scrolling a Modal):

```
# Assuming 'modal' is the WebElement for the follower list pop-up
# Find the modal first, e.g., by its XPATH
modal = self.driver.find_element(By.XPATH, '/html/body/div[6]/div[1]/div/div[2]/div/div/div/div/d

for i in range(10): # Scroll 10 times
    # In the browser, open dev tools and run the JS command to check it
    self.driver.execute_script("arguments[0].scrollTop = arguments[0].scrollHeight", modal)
    time.sleep(2) # Wait for new followers to load
```

Day 53: Data Entry Job Automation (Zillow Web Scraping) 🏠

Objective: Scrape rental property data (address, price, link) from Zillow and automatically fill out a Google Form with this information.

Key Concepts:

- **Combining Scraping Libraries:** Use **BeautifulSoup** for parsing static HTML content and **Selenium** for automating browser actions (data entry).
- **HTTP Headers:** Websites like Zillow may block simple `requests` calls. To appear like a real browser, you must provide HTTP headers, especially `User-Agent` and `Accept-Language`.
- **Web Scraping (Part 1 - BeautifulSoup):**
 - i. Use the `requests` library to get the HTML of the Zillow search results page. Provide the necessary headers.
 - ii. Create a BeautifulSoup `soup` object from the response text.
 - iii. Inspect the page's HTML to find the selectors for the listing card, price, address, and link for each property.
 - iv. Use `soup.select()` or `soup.find_all()` to extract all listings.
 - v. Loop through the results, cleaning the data (e.g., removing text like `"/mo"` from the price) and storing it in a list of dictionaries.
- **Data Entry (Part 2 - Selenium):**
 - i. Create a Google Form with fields for Address, Price, and Link.
 - ii. Use Selenium to open the live Google Form link.
 - iii. Loop through the list of scraped property data.
 - iv. For each property:
 - Find the input fields in the form using their XPATH or other selectors.
 - Use `send_keys()` to fill in the address, price, and link.
 - Click the "Submit" button.
 - To submit another response, you'll need to click the "Submit another response" link.

Example Code Snippet (Scraping with BeautifulSoup):

```
import requests
from bs4 import BeautifulSoup

ZILLOW_URL = "YOUR_ZILLOW_SEARCH_URL"
HEADERS = {
    "User-Agent": "Your User Agent String",
    "Accept-Language": "en-US,en;q=0.9"
}

response = requests.get(ZILLOW_URL, headers=HEADERS)
soup = BeautifulSoup(response.text, "html.parser")

# CSS selector might change, always inspect the page first!
all_link_elements = soup.select(".StyledPropertyCardDataWrapper a")
all_address_elements = soup.select(".StyledPropertyCardDataWrapper address")
all_price_elements = soup.select(".PropertyCardWrapper__StyledPriceLine")

# Process and store the data...
```

Day 54: Introduction to Web Development with Flask

Objective: Understand the basics of the Flask framework by creating a simple web server.

Key Concepts:

- **Web Framework:** A collection of libraries and modules that helps developers build web applications without having to handle low-level details like protocols, sockets, or thread management. Flask is a *micro-framework* because it's lightweight and provides only the essentials.
- **Flask Application:** The core of a Flask site is a `Flask` object instance.

```
from flask import Flask
app = Flask(__name__)
```

- **Routing:** The process of mapping URLs to Python functions. This is done with the `@app.route()` decorator. The function decorated by the route is called a **view function**.

```
@app.route('/') # This is the route for the homepage
def hello_world():
    return 'Hello, World!'
```

- **Running the Server:**

- The development server is run from the command line or within an
if __name__ == "__main__": block.
- Setting debug=True enables the debugger and automatically reloads the server when you make code changes.

Full Basic "Hello, World!" App:

```
from flask import Flask

# 1. Create a Flask app instance
app = Flask(__name__)

# 2. Define a route for the homepage URL ("/")
@app.route("/")
def hello_world():
    # 3. The function returns the content to be displayed in the browser
    return "<p>Hello, World!</p>"

# This block allows you to run the app directly from the Python script
if __name__ == "__main__":
    # debug=True enables auto-reloading and an interactive debugger in the browser
    app.run(debug=True)
```

Running from Terminal:

1. Set the environment variable: export FLASK_APP=main.py (on Mac/Linux) or set FLASK_APP=main.py (on Windows).
2. Run the server: flask run

Day 55: Higher Lower URL Game

Objective: Build a simple "guess the number" game in the browser using Flask, where the user guesses by changing the URL.

Key Concepts:

- **Python Decorators:** A function that takes another function as input, adds some functionality to it, and returns the modified function. `@app.route()` is a decorator. You can create your own to wrap multiple view functions with shared logic (e.g., making text bold).
- **Variable Rules / Dynamic URLs:** You can capture parts of a URL and pass them as arguments to your view function.

```
@app.route("/username/<name>")
def greet(name):
    return f"Hello there {name}!"
```

You can also specify a converter, like `<int:number>` , to automatically convert the URL part to an integer.

Project Workflow:

1. **Generate a Random Number:** When the server starts, pick a random number between 0 and 9. Store it in a global variable.
2. **Create the Homepage (/):**
 - This page should display a heading like "Guess a number between 0 and 9" and show a GIF.
3. **Create the Guessing Route (/<int:guess>):**
 - This route captures the user's guess from the URL.
 - The view function compares the `guess` to the randomly generated number.
 - It returns different HTML based on the comparison:
 - If `guess < random_number` , return "Too low!" with a "too low" GIF.
 - If `guess > random_number` , return "Too high!" with a "too high" GIF.
 - If `guess == random_number` , return "You found me!" with a "correct" GIF.

Example Code Snippet:

```

from flask import Flask
import random

app = Flask(__name__)

# Generate the number once when the app starts
random_number = random.randint(0, 9)
print(f"Pssst, the number is {random_number}")

@app.route('/')
def home():
    return '<h1>Guess a number between 0 and 9</h1>' \
        ''

@app.route('/<int:guess>')
def check_guess(guess):
    if guess < random_number:
        return '<h1 style="color: red;">Too low, try again!</h1>' \
            ''
    elif guess > random_number:
        return '<h1 style="color: purple;">Too high, try again!</h1>' \
            ''
    else:
        return '<h1 style="color: green;">You found me!</h1>' \
            ''

if __name__ == "__main__":
    app.run(debug=True)

```

Day 56: Rendering HTML/Static Files and Using Website Templates 🎨

Objective: Learn how to serve professional, multi-file websites with Flask by separating HTML, CSS, and JavaScript from your Python code.

Key Concepts:

- **Project Structure:** Flask expects a specific folder structure to find your files:

```
/project_folder
  /static
    /css
      style.css
    /js
      script.js
    /images
      image.png
  /templates
    index.html
    about.html
  main.py
```

- **render_template()** : This Flask function looks for a file in the `templates` folder, renders it, and sends it to the browser. You no longer return HTML strings from your view functions.

```
from flask import render_template

@app.route('/')
def home():
    return render_template('index.html')
```

- **url_for()** : This function generates a URL for a static file or another view function. **Crucially**, you should use `url_for()` to link to your CSS and JS files in your HTML. This prevents broken links if you change your URL structure.

Example Usage in `index.html` :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Website</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
  <h1>Hello from a Template!</h1>
</body>
</html>
```

The project for this day is to take a pre-built static HTML/CSS personal site and convert it into a running Flask application using this structure.

Day 57: Templating with Jinja in Flask Applications

Objective: Make websites dynamic by passing data from your Python backend to your HTML templates using the Jinja templating engine.

Key Concepts:

- **Jinja:** A templating language for Python. Flask uses it automatically when you call `render_template()`. Jinja allows you to embed code-like expressions directly into your HTML.
- **Expressions (`{{ ... }}`):** Used to print the value of a variable passed from the backend.

```
# In main.py
@app.route('/')
def home():
    current_year = 2025
    return render_template('index.html', year=current_year, name="Angela")
```

```
<footer>
    <p>Copyright {{ year }}. Built by {{ name }}.</p>
</footer>
```

- **Statements (`{% ... %}`):** Used for control flow, like `for` loops and `if` conditions.

```
{% for post in all_posts: %}
    <h2>{{ post.title }}</h2>
    <h3>{{ post.subtitle }}</h3>
{% endfor %}
```

- **Template Inheritance:** A powerful feature to avoid repeating code (like headers and footers).
 - i. **Create a `base.html`** : This file contains the common HTML structure (`<html>` , `<head>` , `<body>` , header, footer).
 - ii. **Define blocks** : In `base.html` , define placeholders using `{% block block_name %}{% endblock %}` .
 - iii. **Extend the base:** In other templates (e.g., `index.html`), use `{% extends "base.html" %}` at the top.
 - iv. **Fill the blocks:** Override the blocks with specific content for that page.

Project: Build a dynamic blog that fetches post data from an API (or a local Python list) and displays it on the homepage. Create a separate page (`post.html`) to display the full content of a single blog post when a user clicks on it.

Day 58: Bootstrap & Flask - Website Portfolio Project

Objective: Integrate the Bootstrap CSS framework into a Flask project to quickly build a responsive, professionally styled website.

Key Concepts:

- **Bootstrap:** A popular front-end toolkit that provides pre-styled components (like navbars, buttons, cards, and grids) and a powerful responsive grid system. It saves you from writing a lot of CSS from scratch.
- **Integration Methods:**
 - i. **CDN (Content Delivery Network):** The easiest way. Copy-paste the Bootstrap CSS `<link>` and JS `<script>` tags from the official Bootstrap website into your `base.html` template's `<head>` and `<body>` sections.
 - ii. **Local Files:** Download the Bootstrap files and place them in your `static` folder. Then link to them using `url_for()`.
- **Bootstrap Grid System:** A mobile-first system for creating layouts. It's based on a 12-column grid. You use classes like `.container`, `.row`, and `.col-md-4` to arrange your content. `md` stands for medium-sized screens; it changes for different screen sizes (e.g., `sm`, `lg`, `xl`).

Project: Rebuild the personal/portfolio website from Day 56 using Bootstrap components.

- Replace the custom CSS navbar with Bootstrap's responsive `navbar` component.
- Use the Bootstrap grid to create a multi-column layout for projects or skills.
- Style buttons and forms with Bootstrap classes (`.btn`, `.btn-primary`, `.form-control`).

Day 59: Blog Capstone Project Part 2 - Adding Styling

Objective: Apply the Bootstrap knowledge from Day 58 to the blog project from Day 57 to create a fully styled, multi-page blog website.

Project Workflow:

1. **Integrate Bootstrap:** Add the Bootstrap CDN links to your blog's `base.html`.
2. **Create a Consistent Layout:**
 - **Header:** Create a `header.html` partial template. Inside, use Bootstrap's `navbar` component to create a navigation bar with links to "Home" and "About" pages. Use `{% include "header.html" %}` in `base.html`.

- **Footer:** Create a `footer.html` partial. Add social media links and a copyright notice. Include it in `base.html`.
3. **Style the Homepage (`index.html`):**
- Wrap the content in a Bootstrap `.container`.
 - Display a clean "Jumbotron" or large header image.
 - List the blog post previews. Each preview should link to the full post page. Use Bootstrap classes to style the post titles, subtitles, and "Read More" links.
4. **Style the Post Page (`post.html`):**
- Inherit the header and footer from `base.html`.
 - Display the full post title, subtitle, and body content within a `.container`.
5. **Create About/Contact Pages:** Create simple static pages (`about.html` , `contact.html`) that extend `base.html` to ensure they share the same styling and navigation. Create routes for them in `main.py`.

Day 60: Blog Capstone Project Part 3 - POST Requests & Making Forms

Objective: Add a working contact form to the blog, allowing users to send messages that are processed by the Flask backend.

Key Concepts:

- **HTTP Methods:**
 - **GET** : Used to request data from a server (e.g., loading a webpage). This is the default method.
 - **POST** : Used to send data *to* a server to create or update a resource (e.g., submitting a form, logging in).
- **Handling POST in Flask:**
 - You must explicitly allow the `POST` method in your route decorator:
`@app.route('/contact', methods=["GET", "POST"])`.
 - Check which method was used with `if request.method == "POST":`.
- **HTML `<form>` Element:**
 - `action` : The URL where the form data should be sent.
 - `method` : The HTTP method to use (e.g., `post`).
 - `<input>` elements must have a `name` attribute. This `name` becomes the key for accessing the data in Flask.
- **request Object:** Imported from Flask (`from flask import request`), this object contains all the information about the current request.

- `request.form` : A dictionary-like object that holds the data from a submitted form. You can access values using keys that match the `name` attributes of your inputs (e.g., `request.form['username']`).

Project Workflow:

1. **Create the HTML Form:** In `contact.html` , build a form with inputs for name, email, and message. Set `action="/contact"` and `method="post"` .
2. **Update the Flask Route:**
 - Modify the `/contact` route to accept both `GET` and `POST` requests.
 - If the method is `GET` , just render the `contact.html` template as before.
 - If the method is `POST` :
 - Access the submitted data using `request.form.get("name")` , `request.form.get("email")` , etc.
 - (For this project) Print the received data to the console or implement logic to send an email with the data.
 - Change the heading on the page to "Successfully sent your message." to give the user feedback.

Example Code Snippet:

```
from flask import Flask, render_template, request

app = Flask(__name__)

# ... other routes ...

@app.route("/contact", methods=["GET", "POST"])
def contact():
    if request.method == "POST":
        # Get data from the form
        data = request.form
        name = data["name"]
        email = data["email"]
        message = data["message"]
        print(f"New message from {name} ({email}): {message}")
        return render_template("contact.html", msg_sent=True)
    # This is for the GET request
    return render_template("contact.html", msg_sent=False)
```

```
{% if msg_sent %}
    <h1>Successfully sent your message.</h1>
{% else %}
    <h1>Contact Me</h1>
    <form action="{ { url_for('contact') } }" method="post">
        <input name="name" type="text" placeholder="Name" required>
        <input name="email" type="email" placeholder="Email" required>
        <textarea name="message" placeholder="Your Message" required></textarea>
        <button type="submit">Send</button>
    </form>
{% endif %}
```

Day 61: Building Advanced Forms with Flask-WTF

Today's focus is on moving beyond basic HTML forms to create secure, validated, and easily manageable web forms using the **Flask-WTF** extension.

Key Concepts

- **CSRF Protection:** Cross-Site Request Forgery is an attack where a malicious site tricks a user into performing an unwanted action on a trusted site. Flask-WTF provides automatic CSRF protection.
- **Flask-WTF & WTForms:** Flask-WTF is a Flask extension that integrates the WTForms library, allowing you to define your forms as Python classes.
- **Form Validation:** Ensuring user input meets specific criteria (e.g., a field is not empty, an email is in the correct format).
- **Rendering Forms:** Injecting the form fields into your HTML templates cleanly.

Creating a Form Class

You define a form by creating a class that inherits from `FlaskForm`. Each field in the form is an object from `wtforms` (e.g., `StringField`, `PasswordField`, `SubmitField`).

Validators are arguments passed to a field to enforce rules.

```
# forms.py
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, Length

class LoginForm(FlaskForm):
    # DataRequired validator ensures the field is not submitted empty.
    email = StringField(label='Email', validators=[DataRequired(), Email()])

    # We can chain multiple validators.
    password = PasswordField(label='Password', validators=[DataRequired(), Length(min=8)])

    submit = SubmitField(label="Log In")
```

Handling Forms in Your Flask App

1. **Secret Key:** Flask-WTF requires a `SECRET_KEY` in your app configuration for CSRF protection.
2. **Instantiation:** Create an instance of your form class in your route.
3. **Validation:** Use `form.validate_on_submit()` to check if the form was submitted (POST request) and if all validators passed.
4. **Data Access:** If validation is successful, access the data with `form.field_name.data`.

```
# main.py
from flask import Flask, render_template
from forms import LoginForm

app = Flask(__name__)
# This key should be a long, random, secret string in a real app.
app.config['SECRET_KEY'] = 'a-super-secret-key'

@app.route("/login", methods=["GET", "POST"])
def login():
    login_form = LoginForm()
    # This block runs only on a POST request when all validators pass.
    if login_form.validate_on_submit():
        # Access the validated data
        email = login_form.email.data
        password = login_form.password.data
        if email == "admin@email.com" and password == "12345678":
            return render_template("success.html")
        else:
            return render_template("denied.html")
    # This runs on a GET request or if validation fails.
    return render_template('login.html', form=login_form)
```

Rendering the Form in HTML

In your Jinja2 template, you can render the form fields easily.

- `form.hidden_tag()` : This is crucial! It adds the hidden CSRF token field.
- `form.<field_name>.label` : Renders the `<label>` tag.
- `form.<field_name>()` : Renders the input field itself (e.g., `<input type="text">`).
- `form.<field_name>.errors` : A list of validation errors for a specific field.

```

<form method="POST" novalidate>
    {{ form.hidden_tag() }}
    <p>
        {{ form.email.label }}<br>
        {{ form.email(size=30) }}
        {% for err in form.email.errors %}
        <span style="color: red;">{{ err }}</span>
        {% endfor %}
    </p>
    <p>
        {{ form.password.label }}<br>
        {{ form.password(size=30) }}
        {% for err in form.password.errors %}
        <span style="color: red;">{{ err }}</span>
        {% endfor %}
    </p>
    <p>{{ form.submit() }}</p>
</form>

```

Day's Project: A Simple Login Page

The project for the day is to create a simple website with a login page that uses a `LoginForm` created with Flask-WTF. The form should validate that the email and password fields are filled correctly and then check them against hardcoded values to grant or deny access.

Day 62: Coffee & Wifi Project

This day integrates multiple technologies—Flask, WTForms, Bootstrap, and CSV data handling—to build a practical web application.

Key Concepts

- **Flask-Bootstrap:** An extension that simplifies integrating the Bootstrap CSS framework into Flask apps, making styling much faster.
- **Data Persistence with CSV:** Using Python's built-in `csv` module to read from and append to a CSV file, which acts as a simple database.
- **Advanced WTForms Fields:** Using different field types like `SelectField` for dropdowns and `URLField` for URLs.

Integrating Flask-Bootstrap

After `pip install flask-bootstrap`, you initialize it in your app.

```
from flask import Flask
from flask_bootstrap import Bootstrap

app = Flask(__name__)
app.config['SECRET_KEY'] = 'some-secret'
Bootstrap(app) # Initialize Bootstrap
```

Then, in your base template (`base.html`), you can inherit from Bootstrap's base template.

```
{% extends "bootstrap/base.html" %}

{% block title %}Coffee & Wifi{% endblock %}

{% block content %}
<div class="container">
    </div>
{% endblock %}
```

The Project: "Coffee & Wifi" Website

The goal is to build a website that lists cafes and their amenities (like WiFi strength and power socket availability). Users can also add new cafes to the list.

1. Display Cafes (`/cafes`):

- Read the `cafe-data.csv` file using `csv.reader`.
- Store the data in a list of lists.
- Pass this list to the `cafes.html` template and use a Jinja2 `for` loop to render the data in an HTML table.

2. Add a New Cafe (`/add`):

- Create a `CafeForm` using Flask-WTF with fields for the cafe name, location (Google Maps URL), opening/closing times, and `SelectField`s for ratings.
- The `SelectField` requires a `choices` argument, which is a list of tuples (value, label).


```

class CafeForm(FlaskForm):
    cafe = StringField('Cafe name', validators=[DataRequired()])
    location = URLField('Cafe Location on Google Maps (URL)', validators=[DataRequired()], U
    # ... other fields
    wifi_rating = SelectField('Wifi Strength Rating', choices=["X", "👍", "👍👍", "👍👍👍"]
    # ... more fields
    submit = SubmitField('Submit')

```

- When the form is submitted and validated, open the `cafe-data.csv` file in **append mode** (`'a'`).
- Create a new row with the data from `form.field.data` and write it to the CSV.
- Redirect the user back to the `/cafes` page.

```

# main.py
@app.route('/add', methods=["GET", "POST"])
def add_cafe():
    form = CafeForm()
    if form.validate_on_submit():
        # Create a new row as a string
        new_row = f"\n{form.cafe.data},{form.location.data},{...}"
        with open('cafe-data.csv', mode='a', encoding='utf-8') as csv_file:
            csv_file.write(new_row)
        return redirect(url_for('cafes'))
    return render_template('add.html', form=form)

```

Day 63: Databases with SQLite & SQLAlchemy

This is a pivotal day where we move from fragile CSV files to robust relational databases.

Key Concepts

- **Relational Databases:** Data is stored in tables with rows and columns. Relationships can be defined between tables. SQLite is a lightweight, file-based database engine perfect for development and small apps.
- **SQLAlchemy:** An Object Relational Mapper (ORM). It allows you to interact with your database using Python objects and methods instead of writing raw SQL queries. This makes your code more readable, portable, and less prone to SQL injection attacks.

- **CRUD Operations:** The four fundamental functions of database management: **C**reate, **R**ead, **U**ppdate, and **D**eleate.

Setting up the Database with Flask-SQLAlchemy

1. Install: `pip install flask-sqlalchemy`
2. Configure and initialize in your app:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

##CREATE DATABASE
app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///new-books-collection.db"
#Optional: But it will silence the deprecation warning in the console.
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

Defining a Model (Table)

A model is a Python class that inherits from `db.Model`. Each attribute of the class represents a column in the table.

```
##CREATE TABLE
class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(250), unique=True, nullable=False)
    author = db.Column(db.String(250), nullable=False)
    rating = db.Column(db.Float, nullable=False)

# This line is needed to actually create the table in the database file.
# You only need to run this once.
db.create_all()
```

- `db.Column` : Defines a column.
- `db.Integer` , `db.String` , `db.Float` : Define the data type.
- `primary_key=True` : Marks this column as the unique identifier for each row.
- `unique=True` : Ensures all values in this column are unique.
- `nullable=False` : Makes this a required field.

CRUD Operations with SQLAlchemy

```
# --- CREATE a new record ---
new_book = Book(title="Harry Potter", author="J. K. Rowling", rating=9.3)
db.session.add(new_book)
db.session.commit() # Don't forget to commit!

# --- READ all records ---
all_books = db.session.query(Book).all()

# --- READ a specific record by query ---
book = Book.query.filter_by(title="Harry Potter").first()

# --- READ a specific record by PRIMARY KEY ---
book_to_update = Book.query.get(1) # Gets the book with id=1

# --- UPDATE a record ---
book_to_update.title = "Harry Potter and the Chamber of Secrets"
db.session.commit()

# --- DELETE a record ---
book_to_delete = Book.query.get(1)
db.session.delete(book_to_delete)
db.session.commit()
```

Day's Project: A Virtual Bookshelf

The project is to create a web application that manages a book collection.

- The homepage displays all books from the database in a list.
- There's an `add.html` page with a form to add new books.
- When the form is submitted, a new `Book` object is created and saved to the database.

Day 64: My Top 10 Movies Website

This project builds on Day 63, applying SQLAlchemy CRUD operations in a more complex web application that interacts with an external API.

Key Concepts

- **Applying CRUD in Flask:** Integrating Create, Read, Update, and Delete operations into Flask routes.
- **External API Integration:** Using the `requests` library to fetch data from an external API (like The Movie Database - TMDb).
- **Dynamic Routing for Updates:** Using routes like `/edit/<int:movie_id>` to target specific database records for modification.

The Project: "Top 10 Movies" Website

The application allows a user to build a personal list of their top movies.

1. Database Model (`Movie`):

```
class Movie(db.Model):  
    id = db.Column(db.Integer, primary_key=True)  
    title = db.Column(db.String(250), unique=True, nullable=False)  
    year = db.Column(db.Integer, nullable=False)  
    description = db.Column(db.String(500), nullable=False)  
    rating = db.Column(db.Float, nullable=True) # User's rating  
    ranking = db.Column(db.Integer, nullable=True) # User's ranking  
    review = db.Column(db.String(250), nullable=True) # User's review  
    img_url = db.Column(db.String(250), nullable=False)
```

2. Main Page (`/`):

- **READ:** Fetches all movies from the DB:
`all_movies = Movie.query.order_by(Movie.rating).all()` .
- **Logic for Ranking:** After fetching, loop through the sorted movies and assign a rank.
- Displays the movies on `index.html` .

3. Editing a Movie (`/edit`):

- This route handles both GET (to show the form) and POST (to update the data).
- It uses a dynamic URL: `@app.route("/edit", methods=["GET", "POST"])` .
- The movie's `id` is passed via a query parameter (e.g., `/edit?id=1`).
- **UPDATE:**

```

@app.route("/edit", methods=["GET", "POST"])
def edit_movie():
    movie_id = request.args.get('id')
    movie_to_update = Movie.query.get(movie_id)
    if request.method == "POST":
        # Get data from the submitted form
        movie_to_update.rating = request.form["rating"]
        movie_to_update.review = request.form["review"]
        db.session.commit()
        return redirect(url_for('home'))
    return render_template("edit.html", movie=movie_to_update)

```

4. Deleting a Movie (/delete):

- **DELETE:** Similar to edit, it gets the movie `id` from the URL.
- `movie_to_delete = Movie.query.get(movie_id)`
- `db.session.delete(movie_to_delete)`
- `db.session.commit()`

5. Adding a Movie (/add):

- This is a two-step process.
- First, the user searches for a movie title on an `add.html` form.
- Your Flask app takes this query, calls the TMDb API, and gets a list of matching movies.
- It then renders a `select.html` page showing the results.
- When the user clicks a movie from the list, you get the TMDb movie ID. You then make a *second* API call to get detailed data for that specific movie.
- **CREATE:** You use this detailed data to create a new `Movie` object and add it to your own database.

Day 65: Web Design School - Create a Website that People will Love

This day is a brief detour from backend coding to focus on the principles of **User Interface (UI)** and **User Experience (UX)** design. The goal is to make your websites not just functional, but also beautiful and intuitive.

Key Concepts

- **Color Theory:** Understanding the color wheel, complementary colors, and triadic colors to create a visually appealing palette. Tools like [Coolors.co](https://coolors.co) are introduced for generating palettes.
- **Typography:** The art of arranging text to make it legible, readable, and appealing. Key ideas include choosing font pairings (one for headings, one for body text), font sizing, and line spacing. Google Fonts is the go-to resource.
- **UI Patterns:** Common, reusable solutions to design problems. Examples include navbars, cards, carousels, and forms. Frameworks like Bootstrap provide pre-built components for these patterns.
- **User Experience (UX):** Focuses on the overall feel of the application. Is it easy to use? Is the flow logical? Does it solve the user's problem without frustration?

Day's Project: Redesign an Existing Project

There's no new coding project today. Instead, the task is to take one of the previous projects (like the Top 10 Movies website or the Coffee & Wifi site) and apply the design principles learned to improve its appearance and usability.

Actionable Steps:

1. **Choose a Color Palette:** Go to [Coolors.co](https://coolors.co) and generate a palette of 3-5 colors. Apply these colors consistently throughout your site (e.g., primary color for buttons, background color, text color).
2. **Select Fonts:** Go to Google Fonts and pick two complementary fonts. Use one for all `<h1>`, `<h2>`, etc., and the other for all `<p>` tags and body text.
3. **Improve Layout:** Use CSS (or Bootstrap's grid system) to create a more organized and visually balanced layout. Instead of a simple list, maybe display movies or cafes as "cards".
4. **Add a Favicon:** Create a small icon for your website that appears in the browser tab.
5. **Refine with CSS:** Use CSS properties like `padding`, `margin`, `border-radius`, and `box-shadow` to give elements breathing room and a modern feel.

The goal is to transform a purely functional site into a professional-looking one.

Day 66: Building Your Own API with RESTful Routing

Today, you switch from being a *consumer* of APIs to a *creator*. You will learn to build your own API that allows other programs to interact with your data.

Key Concepts

- **API (Application Programming Interface):** A way for two or more computer programs to communicate with each other. A Web API allows communication over the internet, typically using HTTP.
- **REST (REpresentational State Transfer):** An architectural style for designing networked applications. It's not a strict protocol but a set of principles.
- **RESTful Routing:** A convention for structuring API endpoints based on the resources they manage and the HTTP verbs used to interact with them.

HTTP Verb	Endpoint	Action
GET	/cafes	Read - Get all cafes
GET	/cafes/<id>	Read - Get a specific cafe
POST	/cafes	Create - Add a new cafe
PUT / PATCH	/cafes/<id>	Update - Update a specific cafe
DELETE	/cafes/<id>	Delete - Delete a specific cafe

- **JSON (JavaScript Object Notation):** A lightweight data-interchange format. It's the standard format for sending and receiving data in modern web APIs. Flask has a `jsonify` function to easily convert Python dictionaries to JSON responses.

Day's Project: Cafe & Wifi API

The goal is to take the cafe data (now stored in a an SQLAlchemy database) and build a RESTful API for it.

Setup:

- Migrate the cafe data from the CSV file into an SQLite database with an SQLAlchemy `Cafe` model.
- Add a `to_dict()` method to your `Cafe` model to easily convert a `Cafe` object into a dictionary, which can then be converted to JSON.

```
class Cafe(db.Model):  
    # ... columns ...  
  
    #Create a method to convert this object into a dictionary.  
    def to_dict(self):  
        return {column.name: getattr(self, column.name) for column in self.__table__.columns}
```

Implementing the Endpoints:


```

from flask import Flask, jsonify, render_template, request

# ... App and DB setup ...

# Get all cafes
@app.route("/all")
def get_all_cafes():
    cafes = db.session.query(Cafe).all()
    # Use a list comprehension to convert each cafe object to a dictionary
    return jsonify(cafes=[cafe.to_dict() for cafe in cafes])

# Get a random cafe
@app.route("/random")
def get_random_cafe():
    # ... logic to get a random cafe ...
    return jsonify(cafe=random_cafe.to_dict())

# Add a new cafe
@app.route("/add", methods=["POST"])
def post_new_cafe():
    new_cafe = Cafe(
        name=request.form.get("name"),
        # ... get other data from request.form ...
    )
    db.session.add(new_cafe)
    db.session.commit()
    return jsonify(response={"success": "Successfully added the new cafe."})

# Update a cafe's price
@app.route("/update-price/<int:cafe_id>", methods=["PATCH"])
def patch_new_price(cafe_id):
    new_price = request.args.get("new_price")
    cafe = db.session.query(Cafe).get(cafe_id)
    if cafe:
        cafe.coffee_price = new_price
        db.session.commit()
        return jsonify(response={"success": "Successfully updated the price."})
    else:
        return jsonify(error={"Not Found": "Sorry a cafe with that id was not found."})

# ... Implement DELETE endpoint ...

```

You would test these endpoints using a tool like **Postman** or **Insomnia**, not a web browser (except for GET requests).

Day 67: Blog Capstone Project Part 3 - RESTful Routing

This day applies the RESTful principles learned on Day 66 to the ongoing Blog Capstone project. The goal is to add functionality to create, edit, and delete blog posts.

Key Concepts

- **Integrating WTForms with SQLAlchemy:** Creating forms that map directly to the fields of a database model.
- **Pre-populating Forms:** When editing a post, the form fields should be filled with the existing data from the database.
- **Full-stack CRUD:** Connecting a front-end (HTML templates with forms) to a back-end (Flask routes) that performs CRUD operations on a database (SQLAlchemy).

Project Goal: Add Full Post Management to the Blog

1. Create a New Post (/new-post):

- Create a `CreatePostForm` with WTForms for the title, subtitle, author, image URL, and body content. The body should use a `CKEditorField` for rich text editing (requires `Flask-CKEditor` extension).
- The route handles GET (to show the form) and POST (to process it).
- On POST, create a new `BlogPost` object using the form data, add it to the DB session, commit, and redirect to the main page.

2. Edit an Existing Post (/edit-post/<int:post_id>):

- This route uses a dynamic URL to identify the post to edit.
- **GET Request:**
 - Fetch the `BlogPost` object from the database using the `post_id`.
 - Create an instance of the `CreatePostForm`, but pre-populate it with the data from the fetched post object:

```

# In the /edit-post/<post_id> route
post = BlogPost.query.get(post_id)
edit_form = CreatePostForm(
    title=post.title,
    subtitle=post.subtitle,
    img_url=post.img_url,
    author=post.author,
    body=post.body
)

```

- Render the form template, passing in `edit_form`.

- **POST Request:**

- When the populated form is submitted, fetch the post from the DB again.
- Update its attributes with the new data from `edit_form.field.data`.
- Commit the changes and redirect to the post's page.

3. Delete a Post (`/delete/<int:post_id>`):

- This is a simpler route that only needs to handle GET requests (triggered by a link/button).
- Fetch the post to delete using the `post_id`.
- Use `db.session.delete()` and `db.session.commit()`.
- Redirect to the home page.

```

# In main.py
@app.route("/edit-post/<int:post_id>", methods=["GET", "POST"])
def edit_post(post_id):
    post = BlogPost.query.get(post_id)
    # Pre-populate the form
    edit_form = CreatePostForm(obj=post) # A shortcut for pre-populating

    if edit_form.validate_on_submit():
        post.title = edit_form.title.data
        post.subtitle = edit_form.subtitle.data
        post.img_url = edit_form.img_url.data
        post.author = edit_form.author.data
        post.body = edit_form.body.data
        db.session.commit()
        return redirect(url_for("show_post", post_id=post.id))

    return render_template("make-post.html", form=edit_form, is_edit=True)

```

Day 68: Authentication with Flask-Login

This day introduces one of the most critical features of any web application: user authentication (registration and login).

Key Concepts

- **Authentication vs. Authorization:**
 - **Authentication:** Verifying who a user is (e.g., with a username and password). Are you who you say you are?
 - **Authorization:** Determining what an authenticated user is allowed to do (e.g., an admin can delete posts, but a regular user cannot). What are you allowed to do?
- **Password Hashing & Salting: NEVER** store passwords in plain text. A password should be run through a one-way hashing algorithm (like SHA-256) to create a hash. A "salt" (a random string) is added before hashing to prevent rainbow table attacks. The `werkzeug` library (a dependency of Flask) has helpers for this.
- **Flask-Login:** A Flask extension that handles the common tasks of user session management: logging users in, logging them out, and remembering them between sessions.

Implementation Steps

1. Create a User Model:

- This model must inherit from `db.Model` and `UserMixin`. `UserMixin` provides default implementations for the properties that Flask-Login expects user objects to have (e.g., `is_authenticated`).

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

2. Set up LoginManager :

```

from flask_login import LoginManager

# ... after app creation ...
login_manager = LoginManager()
login_manager.init_app(app)

# This user_loader callback is used to reload the user object
# from the user ID stored in the session.
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

```

3. Registration Route (/register):

- Create a registration form.
- When the form is submitted, hash and salt the password from the form.
- Create a new `User` object with the hashed password and save it to the database.

```

from werkzeug.security import generate_password_hash, check_password_hash

# In the register route after form validation
hashed_and_salted_password = generate_password_hash(
    form.password.data,
    method='pbkdf2:sha256',
    salt_length=8
)
new_user = User(
    email=form.email.data,
    password=hashed_and_salted_password,
    name=form.name.data
)
db.session.add(new_user)
db.session.commit()

```

4. Login Route (/login):

- Create a login form.
- When submitted, first find the user in the database by their email.
- Then, use `check_password_hash()` to compare the stored hashed password with the password the user just typed in.
- If they match, use the `login_user()` function from Flask-Login to create the session.

```

from flask_login import login_user

# In the login route after form validation
user = User.query.filter_by(email=email).first()
if user and check_password_hash(user.password, password):
    login_user(user) # This handles the session.
    return redirect(url_for('secrets'))

```

5. Protecting Routes:

- Use the `@login_required` decorator on any route that should only be accessible to logged-in users. Flask-Login will automatically redirect unauthenticated users to the login page.

```

from flask_login import login_required

@app.route('/secrets')
@login_required
def secrets():
    return render_template("secrets.html")

```

6. Logout Route (/logout):

- This is a simple route that calls the `logout_user()` function.

Day 69: Blog Capstone Project Part 4 - Adding Users

This day fully integrates the authentication system from Day 68 into the blog project, creating relationships between users and their posts.

Key Concepts

- **Database Relationships (One-to-Many):** A core concept in relational databases. In our blog, one `User` can have many `BlogPosts`. This is modeled in SQLAlchemy using `db.relationship()` and foreign keys.
- **Foreign Keys:** A column (or a set of columns) in a table that establishes a link between the data in two tables. The `blog_posts` table will have an `author_id` column that links to the `id` of the `users` table.
- **Authorization Logic:** Implementing checks to ensure that only the author of a post can edit or delete it.

Implementation Steps

1. Update Database Models:

- **User Model:** Add a relationship to `BlogPost`. This is the "one" side of the one-to-many relationship. The `back_populates` argument links this relationship to the one in the `BlogPost` model.

```
# In User model
posts = db.relationship("BlogPost", back_populates="author")
```

- **BlogPost Model:** Add a foreign key to the `User` table and the corresponding relationship. This is the "many" side.

```
# In BlogPost model
author_id = db.Column(db.Integer, db.ForeignKey("users.id")) # Note: "users" is table name
author = db.relationship("User", back_populates="posts")
```

You will need to delete your old database file and recreate it for these changes to take effect.

2. Tie Posts to the Logged-in User:

- When creating a new post, you no longer need an "author" text field in the form. The author is the `current_user` provided by Flask-Login.

```
from flask_login import current_user

# In /new-post route
new_post = BlogPost(
    title=form.title.data,
    # ... other fields ...
    author=current_user # Assign the user object itself
)
```

3. Display Post Author Information:

- In your templates (like `index.html` and `post.html`), you can now access the author's information through the post object: `post.author.name`.

4. Implement Authorization:

- In the routes for editing and deleting posts, you must check if the logged-in user is the actual author of the post.
- A good way to do this is to create an "admin-only" decorator.

```

from functools import wraps
from flask import abort

def admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # If id is not 1 then return abort with 403 error
        if not current_user.is_authenticated or current_user.id != 1:
            return abort(403) # Forbidden
        # Otherwise continue with the route function
        return f(*args, **kwargs)
    return decorated_function

```

- Then, apply this decorator to the routes for creating, editing, and deleting posts. (Note: A more robust solution checks `current_user.id == post.author_id` instead of hardcoding an admin ID of 1).

5. Conditional Rendering in Templates:

- Use Jinja2 `if` statements in your templates to show "Edit" and "Delete" buttons only if the `current_user` is the post's author.

```

{% if current_user.id == post.author.id %}
<div class="clearfix">
    <a class="btn btn-primary float-right" href="{{url_for('edit_post', post_id=post.id)}}">
</div>
{% endif %}

```

Day 70: Deploying Your Web Application

The final step: taking the application you've built on your local machine and making it accessible to the world on the internet.

Key Concepts

- **Production vs. Development Server:** The Flask development server (`app.run()`) is not suitable for production. It's not secure, stable, or scalable. A production-grade Web Server Gateway Interface (WSGI) server like **Gunicorn** is used instead.
- **Cloud Hosting Platforms:** Services like Render, PythonAnywhere, Railway, or Google Cloud that provide the infrastructure to run your web application. (Historically, Heroku was the focus of this

lesson, but its free tier was discontinued).

- **Environment Variables:** A way to manage configuration settings (like secret keys, database URLs, API keys) outside of your code. This is crucial for security. You should never commit secrets to version control (Git).
- **requirements.txt** : A file that lists all the Python packages your project depends on. The hosting platform uses this file to install the necessary libraries. You can generate it with
`pip freeze > requirements.txt` .
- **Procfile** : A file (specific to platforms like Heroku/Render) that tells the hosting service what command to run to start your application. For a Flask app, this will be a Gunicorn command.

General Deployment Steps (using a platform like Render)

1. Prepare Your Application:

- Make sure your `SQLALCHEMY_DATABASE_URI` is configured to use an environment variable. The platform will provide a URL for its own PostgreSQL database.

```
# Use the provided DATABASE_URL or fall back to a local sqlite DB
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get("DATABASE_URL", "sqlite:///blog.db")
```

- Generate `requirements.txt` : `pip freeze > requirements.txt` . Make sure `gunicorn` is included in this file (`pip install gunicorn`).
- Create a `Procfile` (literally a file named `Procfile` with no extension) in your root directory:

```
web: gunicorn main:app
```

(This tells the server: "This is a web process. Run the Gunicorn server on the `app` object found inside the `main.py` file.")

2. Use Git:

- Your project must be a Git repository.
- Commit all your code, `requirements.txt` , and `Procfile` .
- Create a `.gitignore` file to exclude files you don't want to upload, like your virtual environment folder (`venv/`) and local database files (`.db`).

3. Sign up for a Hosting Platform (e.g., Render):

- Create a new "Web Service" and connect it to your GitHub account.
- Select the repository for your blog project.

4. Configure the Service:

- The platform will likely detect that it's a Python project from your `requirements.txt` file.
- Set the **Start Command** to `gunicorn main:app` .

- Go to the "Environment" section and add your **Environment Variables** (e.g., `SECRET_KEY` , `DATABASE_URL`). The platform provides the `DATABASE_URL` for you when you create a database add-on.

5. Deploy:

- Click the "Create Web Service" or "Deploy" button.
- The platform will pull your code from GitHub, install the dependencies from `requirements.txt` , and run the start command from your `Procfile` .
- You can watch the build logs for any errors. If everything is successful, you will be given a public URL where your application is now live! ✨

Python Notes: Days 71-80 (Data Science with Pandas, Matplotlib & Scikit-Learn)

This block marks a significant shift into the world of **Data Science**. You'll move from building applications to analyzing and interpreting data. These skills are fundamental for any data-driven project, including the AI/ML aspects of your NEETPrepGPT plan.

Day 71: Introduction to Pandas and Data Exploration

Objective

Learn to use the Pandas library to load, inspect, and perform initial analysis on tabular data. The core data structure you'll master is the **DataFrame**.

Key Concepts

- What is Pandas?
- The DataFrame and Series data structures.
- Loading data from a CSV file.
- Inspecting a DataFrame: `.head()` , `.tail()` , `.shape` , `.columns` .
- Accessing columns and rows.
- Performing basic calculations: `.mean()` , `.max()` , `.value_counts()` .

Detailed Notes & Code Snippets

1. Introduction to Pandas

Pandas is the most popular Python library for data manipulation and analysis. It provides two primary data structures:

- **Series**: A one-dimensional labeled array, like a single column of a spreadsheet.

- **DataFrame:** A two-dimensional labeled data structure with columns of potentially different types, like a full spreadsheet or an SQL table.

2. Loading Data and Initial Inspection

You almost always start by loading data from a source, most commonly a `.csv` file.

```
import pandas as pd

# Load the dataset into a DataFrame
df = pd.read_csv('salaries_by_college_major.csv')

# --- Inspection Methods ---

# See the first 5 rows
print(df.head())

# See the last 5 rows
print(df.tail())

# Get the dimensions (rows, columns)
print(f"DataFrame shape: {df.shape}")

# Get the column names
print(f"Column names: {df.columns}")
```

3. Accessing Data

You can access columns like a dictionary and rows using indexing.

```

# Access a single column (returns a Pandas Series)
starting_salary_col = df['Starting Median Salary']
print(starting_salary_col.head())

# Get the maximum value in a column
print(f"Max starting salary: {starting_salary_col.max()}")

# Find the index of the row with the max value
max_salary_index = starting_salary_col.idxmax()
print(f"Index for max salary: {max_salary_index}") # Outputs an integer index

# Access a specific row by its index label
row_data = df.loc[max_salary_index]
print(row_data)

# Access a specific cell (row index, column name)
major_with_max_salary = df.loc[max_salary_index, 'Undergraduate Major']
print(f"Major with highest starting salary: {major_with_max_salary}")

```

Project Context

Today's project involves analyzing a dataset of salaries by college major. You'll use these inspection techniques to answer questions like "Which major has the highest starting salary?" and "Which major has the highest mid-career salary?".



Day 72: Data Cleaning and Manipulation with Pandas

Objective

Learn how to handle common data quality issues, such as missing values (NaNs) and incorrect data types. Clean data is essential for accurate analysis.

Key Concepts

- Identifying missing values: `.isna()` .
- Removing rows/columns with missing values: `.dropna()` .

- Filling missing values: `.fillna()` .
- Changing column data types: `.astype()` .
- Working with dates and times.

Detailed Notes & Code Snippets

1. Finding and Handling Missing Values (NaN)

Real-world data is messy. `NaN` (Not a Number) is Pandas' way of representing missing data.

```
import pandas as pd

df = pd.read_csv('salaries_by_college_major.csv')

# Check for missing values in the entire DataFrame
print(df.isna().sum())

# Let's assume the last row has missing data
# You can see this by inspecting df.tail()

# Drop rows with any NaN values
clean_df = df.dropna()
print(f"Original shape: {df.shape}")
print(f"Shape after dropna: {clean_df.shape}")

# If you wanted to fill missing values instead (e.g., with 0)
# df_filled = df.fillna(0)
```

2. Manipulating Columns

You can create new columns from existing ones or modify them.

```
# Create a new column by performing an operation on existing columns
# Calculate the spread between mid-career 90th percentile and 10th percentile salaries
spread_col = clean_df['Mid-Career 90th Percentile Salary'] - clean_df['Mid-Career 10th Percentile Salary']

# Insert the new column into the DataFrame at a specific position
clean_df.insert(1, 'Spread', spread_col)
print(clean_df.head())

# To sort the DataFrame by this new column
low_risk_majors = clean_df.sort_values('Spread')
print(low_risk_majors[['Undergraduate Major', 'Spread']].head())
```

3. Grouping Data with `.groupby()`

This is one of the most powerful features in Pandas. It allows you to split data into groups, apply a function, and combine the results.

```
# Group by 'Group' column and find the average salaries for each group
grouped_data = clean_df.groupby('Group').mean(numeric_only=True)
print(grouped_data)
```

Project Context

The project continues with the salary dataset, focusing on cleaning it and then using `.groupby()` to find the majors with the highest potential and lowest risk (smallest salary spread).



Day 73: Aggregation, Merging, and Pivoting DataFrames

Objective

Learn advanced data manipulation techniques, including how to combine multiple datasets and reshape data for better analysis.

Key Concepts

- Advanced `.groupby()` with `.agg()`.

- Combining DataFrames with `.merge()` (like SQL joins).
- Reshaping DataFrames with `.pivot_table()` .

Detailed Notes & Code Snippets

1. Combining DataFrames (`.merge()`)

Often, your data is split across multiple files. `pd.merge()` allows you to combine them based on a common column.

```
import pandas as pd

# Imagine two dataframes
df1 = pd.DataFrame({'student_id': [1, 2, 3], 'subject': ['Math', 'Science', 'History']})
df2 = pd.DataFrame({'student_id': [1, 2, 3], 'grade': [85, 92, 78]})

# Merge them on the common 'student_id' column
merged_df = pd.merge(df1, df2, on='student_id')
print(merged_df)
```

2. Reshaping with `.pivot_table()`

Pivoting is useful for summarizing data in a matrix-like format.

```
# Using the Nobel Prize dataset from the course
df = pd.read_csv('nobel_prize_data.csv')

# Create a pivot table to see the number of prizes per year and category
prize_counts = df.pivot_table(index='year', columns='category', values='prize', aggfunc='count')
print(prize_counts.tail())
```

Project Context

Today's project involves analyzing the **Nobel Prize dataset**. You'll need to clean it, manipulate date formats, and use `.pivot_table()` and other aggregations to uncover trends, like which countries or categories have won the most prizes over time.



Day 74: Introduction to Data Visualization with Matplotlib

Objective

Learn to create basic plots and charts to visually represent data using the **Matplotlib** library. A picture is worth a thousand rows of data.

Key Concepts

- Creating basic plots: `plt.plot()` , `plt.scatter()` .
- Customizing plots: labels, titles, colors, line styles, figure size.
- Working with subplots.

Detailed Notes & Code Snippets

1. Basic Plotting

Matplotlib is the foundational plotting library in Python.

```

import matplotlib.pyplot as plt
import pandas as pd

# Using the grouped data from Day 72
clean_df = pd.read_csv('salaries_by_college_major.csv').dropna()
spread_col = clean_df['Mid-Career 90th Percentile Salary'] - clean_df['Mid-Career 10th Percentile Salary']
clean_df.insert(1, 'Spread', spread_col)
highest_spread = clean_df.sort_values('Spread', ascending=False)

# Plotting Starting Median Salary vs. Spread
plt.figure(figsize=(10, 6)) # Set the figure size (width, height in inches)

plt.scatter(highest_spread['Starting Median Salary'], highest_spread['Spread'])

# Adding labels and title for clarity
plt.xlabel('Starting Median Salary ($)')
plt.ylabel('Salary Spread ($)')
plt.title('Starting Salary vs. Salary Spread for College Majors')

plt.grid(True) # Add a grid
plt.show() # Display the plot

```

Project Context

You will use the **LEGO dataset** for this project. The goal is to create various plots to answer questions like: "How has the average number of parts in a LEGO set changed over the years?" and "Which LEGO themes have the most sets?".



Day 75: Advanced Visualization with Seaborn and Plotly

Objective

Build upon Matplotlib by using **Seaborn** for more aesthetically pleasing statistical plots and get an introduction to **Plotly** for interactive charts.

Key Concepts

- Seaborn for statistical plots: `regplot` , `boxplot` , `histplot` .
- Creating interactive plots with Plotly Express.

Detailed Notes & Code Snippets

1. Seaborn for Regression Plots

Seaborn is built on top of Matplotlib and makes creating complex statistical plots much easier.

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Load Google trends data about programming languages
df = pd.read_csv('programming_languages_search_trends.csv')
df.date = pd.to_datetime(df.date) # Convert date column to datetime objects

# Create a regression plot to see the trend of Python searches over time
plt.figure(figsize=(12, 7))
sns.regplot(x=df.index, y=df['python'], scatter_kws={'alpha':0.3}, line_kws={'color':'red'})

plt.title('Search Trend for "Python" Over Time')
plt.xlabel('Date')
plt.ylabel('Search Interest')
plt.show()
```

2. Plotly for Interactive Visualizations

Plotly creates interactive, web-based visualizations that you can zoom, pan, and hover over for more information.

```
import plotly.express as px

# Using the same programming languages data
fig = px.line(df, x='date', y=['python', 'java', 'javascript'],
              title='Interactive Search Trends for Programming Languages')

fig.show() # This will open an interactive plot in your browser or notebook
```

Project Context

The main project uses Google Trends data to analyze the popularity of different programming languages over time. You will use Seaborn and Plotly to visualize these trends and compare them.



Day 76: Advanced Pandas & NumPy

Objective

Dive deeper into advanced Pandas features and get an introduction to **NumPy**, the fundamental package for numerical computing in Python.

Key Concepts

- Pandas: Multi-level indexing.
- NumPy: The `ndarray` object.
- Vectorization and broadcasting in NumPy for high-performance computation.

Detailed Notes & Code Snippets

1. NumPy Arrays

NumPy's core object is the `ndarray` (n-dimensional array). It's faster and more memory-efficient than Python lists for numerical operations.

```
import numpy as np

# Create a NumPy array from a list
my_list = [1, 2, 3, 4, 5]
my_array = np.array(my_list)

# Perform vectorized operations (no loops needed!)
print(my_array * 2) # Output: [ 2  4  6  8 10]
print(my_array + 5) # Output: [ 6  7  8  9 10]

# Generate arrays
zeros = np.zeros(5) # [0. 0. 0. 0. 0.]
ones = np.ones((2, 3)) # 2x3 array of 1s
```

2. The Power of Vectorization

Vectorization allows you to perform operations on entire arrays at once, which is executed in optimized, pre-compiled C code. This is *much* faster than iterating with a Python `for` loop.

```
# Using NumPy is significantly faster for large datasets
large_array = np.arange(1_000_000)

# NumPy vectorized operation (very fast)
%timeit large_array ** 2

# Python list comprehension (slower)
large_list = list(range(1_000_000))
%timeit [x**2 for x in large_list]
```

Project Context

You will revisit the **LEGO dataset** and apply NumPy for faster calculations. You will also tackle more complex data manipulation tasks that require a deeper understanding of Pandas indexing and grouping.



Day 77: Linear Regression with Scikit-Learn

Objective

Build your first machine learning model! Understand the theory and practical application of **Linear Regression** using the **Scikit-Learn** library.

Key Concepts

- What is Linear Regression? (Finding the best-fit line).
- The Scikit-Learn library.
- Splitting data into training and testing sets.
- Creating, training (`.fit()`), and using a model (`.predict()`).
- Evaluating model performance (`.score()`).

Detailed Notes & Code Snippets

1. Theory of Linear Regression

Linear regression models the relationship between a dependent variable (what you want to predict) and one or more independent variables (features) by fitting a straight line to the data. The equation is $y = mx + b$, where the model learns the best values for the slope (m) and the intercept (b).

2. Implementation with Scikit-Learn

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# 1. Load and prepare the data
df = pd.read_csv('boston_housing.csv')
X = df[['RM', 'LSTAT']] # Features (e.g., avg rooms, % lower status)
y = df['PRICE']          # Target (what we want to predict)

# 2. Split data into training and testing sets
# 80% for training, 20% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Create and train the model
model = LinearRegression()
model.fit(X_train, y_train) # The "learning" happens here

# 4. Evaluate the model
train_score = model.score(X_train, y_train) # R-squared value for training data
test_score = model.score(X_test, y_test)    # R-squared value for testing data

print(f"Training R^2 Score: {train_score:.4f}")
print(f"Testing R^2 Score: {test_score:.4f}")

# 5. Make predictions on new, unseen data
new_house_features = [[6.5, 5.0]] # e.g., 6.5 rooms, 5% lower status population
predicted_price = model.predict(new_house_features)
print(f"Predicted price for the new house: ${predicted_price[0]*1000:.2f}")
```

Project Context

You'll use a dataset like the **Boston Housing dataset** to predict house prices based on features like the number of rooms, crime rate, etc. This is a classic "hello world" project for machine learning.



Day 78: Logistic Regression and Classification

Objective

Learn about **Logistic Regression**, a powerful algorithm for solving classification problems (i.e., predicting a category, not a continuous number).

Key Concepts

- Classification vs. Regression.
- The Sigmoid function.
- Building a Logistic Regression model in Scikit-Learn.
- Evaluating classification models: Confusion Matrix, Accuracy, Precision, Recall.

Detailed Notes & Code Snippets

1. Theory of Logistic Regression

While Linear Regression outputs a continuous value, Logistic Regression outputs a probability (between 0 and 1). It uses the sigmoid function to map any real-valued number into this range. We can then set a threshold (e.g., 0.5) to classify the outcome into one of two categories (e.g., "Yes" or "No", "Survived" or "Died").

2. Implementation with Scikit-Learn

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# 1. Load and prepare the data (e.g., Titanic dataset)
df = pd.read_csv('titanic.csv')
# (Data cleaning and feature engineering steps would go here)
# ... for simplicity, assume we have features X and target y
X = df[['Pclass', 'Age', 'Fare']].fillna(df['Age'].mean()) # Simplified features
y = df['Survived']

# 2. Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Create and train the model
log_model = LogisticRegression(max_iter=1000)
log_model.fit(X_train, y_train)

# 4. Make predictions
predictions = log_model.predict(X_test)

# 5. Evaluate
accuracy = accuracy_score(y_test, predictions)
conf_matrix = confusion_matrix(y_test, predictions)

print(f"Model Accuracy: {accuracy:.4f}")
print("Confusion Matrix:")
print(conf_matrix)
```

Project Context

The quintessential project for this day is predicting survival on the **Titanic**. You will use passenger data (age, class, sex, etc.) to build a model that classifies whether a passenger likely survived or not.



Day 79: Advanced Machine Learning Models

Objective

Get a high-level overview of more advanced and powerful machine learning models beyond linear/logistic regression.

Key Concepts

- **Decision Trees:** A flow-chart-like structure.
- **Random Forests:** An ensemble of many decision trees.
- **Support Vector Machines (SVM):** A model that finds the optimal hyperplane to separate data points.
- The concept of **Overfitting** and **Underfitting**.

Detailed Notes

This day is more conceptual, introducing you to the existence and general idea of more complex models.

- **Decision Tree:** Easy to interpret. It splits the data based on feature values, creating a tree of decisions. Prone to overfitting (learning the training data too well).
- **Random Forest:** The "wisdom of the crowd" for models. It builds many different decision trees on random subsets of the data and averages their predictions. This reduces overfitting and generally improves performance. It's often a great "go-to" model.
- **Overfitting:** Your model learns the training data, including its noise and quirks, so perfectly that it fails to generalize to new, unseen data. It will have a high training score but a low testing score.
- **Underfitting:** Your model is too simple to capture the underlying patterns in the data. It will perform poorly on both the training and testing sets.

The Scikit-Learn API is consistent, so using these models is syntactically similar to what you've already done:

```
from sklearn.ensemble import RandomForestClassifier

# Same workflow: create, fit, predict, score
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_accuracy = rf_model.score(X_test, y_test)
print(f"Random Forest Accuracy: {rf_accuracy:.4f}")
```

Project Context

You might apply a Random Forest model to the same Titanic or Boston Housing dataset to see if you can achieve better performance than with the simpler models from the previous days.

Day 80: Capstone Project - LEGO Dataset Analysis

Objective

Combine everything you've learned in the data science section (Days 71-79) to perform a complete, end-to-end data analysis project.

Project Steps

- 1. Ask Interesting Questions:** Start by formulating questions you want to answer with the data.
 - What is the relationship between the number of parts and the price of a LEGO set?
 - How many unique themes are there? Which are the most popular?
 - How has the color palette of LEGO bricks changed over time?
- 2. Data Wrangling & Cleaning (Pandas):**
 - Load multiple CSV files (`sets.csv` , `colors.csv` , `themes.csv`).
 - Handle missing values in the datasets.
 - Merge the different DataFrames together to create one master DataFrame for analysis.
- 3. Data Exploration & Visualization (Pandas, Matplotlib, Seaborn):**
 - Use `.groupby()` and `.agg()` to summarize the data.
 - Create plots (bar charts, line charts, scatter plots) to visualize your findings and answer the questions you formulated. For example, plot the number of sets released per year.
- 4. (Optional) Machine Learning (Scikit-Learn):**

- Build a simple linear regression model to predict the number of parts in a set based on the year it was released.

Example Code Snippet (Merging and Plotting)

```
import pandas as pd
import matplotlib.pyplot as plt

# Load datasets
sets = pd.read_csv('sets.csv')
themes = pd.read_csv('themes.csv')

# Merge to get theme names
sets_with_themes = pd.merge(sets, themes, left_on='theme_id', right_on='id', suffixes=('_set',

# Explore sets per year
sets_by_year = sets_with_themes.groupby('year').count()['set_num']

# Plotting
plt.figure(figsize=(14, 8))
plt.plot(sets_by_year.index[:-2], sets_by_year.values[:-2]) # Exclude last 2 years for complete
plt.title('Number of LEGO Sets Released Per Year')
plt.xlabel('Year')
plt.ylabel('Number of Sets')
plt.grid(True)
plt.show()
```

This capstone solidifies your new data science skills, preparing you for the next phase of the course, which often involves building full-stack web applications that can serve up these kinds of analyses.

Days 81-84: Portfolio Project - Text-to-Morse Code Converter

This project focuses on building a desktop application that converts plain text into its Morse code equivalent.

Project Goal

Create a GUI application using Tkinter where a user can type in English text, press a button, and see the corresponding Morse code translation.

Core Concepts

- **GUI with Tkinter:** Solidifying skills in creating windows, labels, buttons, and text entry widgets.
- **Dictionaries for Mapping:** Using a Python dictionary is the most efficient way to map characters to their Morse code representation.
- **String Manipulation:** Iterating through the user's input string and building the output string.
- **Handling User Events:** Connecting a button click to a function that performs the conversion.

Implementation Steps & Code Snippets

1. Setup the UI:

- Create the main window, a title label, a `Text` widget for user input, a `Button` to trigger the conversion, and a `Label` or another `Text` widget to display the result.

```

import tkinter as tk

# --- Morse Code Dictionary ---
MORSE_CODE_DICT = { 'A':'.-.', 'B':'-...', 'C':'-.-.', 'D':'-..', 'E':'.',
                    'F':'.-.-.', 'G':'--.', 'H':'....', 'I':'. .', 'J':'.---',
                    'K':'.-.-', 'L':'.-..', 'M':'--', 'N':'-.', 'O':'---',
                    'P':'.-.-.', 'Q':'--.-', 'R':'.-.', 'S':'. . .', 'T':'-.',
                    'U':'. . -', 'V':'. . . -', 'W':'.-.-', 'X':'-.-.', 'Y':'-.-.-',
                    'Z':'--..', '1':'.-----', '2':'.----', '3':'.---',
                    '4':'.---.', '5':'.----', '6':'-....', '7':'---...',
                    '8':'----.', '9':'-----', '0':'-----', ',', ':'-.-.-.-',
                    '.':'.-.-.-', '?':'.-.-.-.', '/':'-.-.-.', '-':'-.-.-.-',
                    '(':'-.-.-.', ')':'-.-.-.-', ' ':'/' }

# --- UI Setup ---
window = tk.Tk()
window.title("Text to Morse Code Converter")
window.config(padx=50, pady=50)

# Widgets (Labels, Text, Button) go here
# ...

```

2. Implement Conversion Logic:

- Create a function that will be called when the button is pressed.
- This function should get the text from the input widget (`text_widget.get("1.0", tk.END)`).
- Iterate through each character of the input string (converted to uppercase).
- Use a `try-except` block to handle characters that are not in your `MORSE_CODE_DICT`.
- Look up the Morse code for each character in the dictionary and append it to a result string.
- Display the result in the output widget.

```
def convert_to_morse():
    text_to_convert = input_text.get("1.0", tk.END).upper()
    morse_result = ""
    for char in text_to_convert:
        try:
            morse_result += MORSE_CODE_DICT[char] + " "
        except KeyError:
            # Handle characters not in the dictionary, e.g., ignore or add a special symbol
            pass
    output_label.config(text=morse_result)

# --- Button ---
convert_button = tk.Button(text="Convert", command=convert_to_morse)
convert_button.grid(row=2, column=0)
```

Days 85-86: Portfolio Project - Image Watermarking App

This project involves building a desktop application to add a watermark (text or a logo) to images.

Project Goal

Create a GUI application that allows a user to open an image file and automatically add a predefined watermark to it.

Core Concepts

- **Pillow (PIL) Library:** The cornerstone of this project. Used for opening, manipulating, and saving images. Key functions include `Image.open()`, `ImageDraw.Draw()`, and `image.save()`.
- **Tkinter GUI:** Used to create the user interface, including buttons for opening files and triggering the watermarking process.
- **File Dialogs:** Using `tkinter.filedialog` to let the user browse their computer and select an image file. This is a crucial concept for any application that interacts with the user's filesystem.

Implementation Steps & Code Snippets

1. **Install Pillow:**

- Ensure you have the Pillow library installed: `pip install Pillow` .

2. UI Setup:

- Create a simple Tkinter window with two buttons: "Open Image" and "Add Watermark". You can also add a canvas to display the loaded image.

3. File Opening Logic:

- The "Open Image" button should trigger a function that uses `filedialog.askopenfilename()` .
- Store the path of the selected file in a global variable or class attribute.

```
from tkinter import filedialog
from PIL import Image, ImageDraw, ImageFont

filepath = ""

def open_image():
    global filepath
    filepath = filedialog.askopenfilename(
        title="Select an Image",
        filetypes=(("JPEG files", "*.jpg"), ("PNG files", "*.png"), ("All files", "*.*"))
    )
    if filepath:
        # Optionally, display the image in the UI
        print(f"Image selected: {filepath}")
```

4. Watermarking Logic:

- The "Add Watermark" button calls the main watermarking function.
- This function opens the selected image using `Image.open()` .
- It creates a drawing context with `ImageDraw.Draw(image)` .
- It defines the text, font, and color for the watermark.
- It uses `draw.text()` to write the watermark onto the image.
- Finally, it saves the modified image, usually with a new name like `watermarked_image.jpg` .

```

def add_watermark():
    if not filepath:
        print("Please open an image first.")
        return

    with Image.open(filepath) as im:
        # Create a drawing context
        draw = ImageDraw.Draw(im)

        # Define watermark text and font
        text = "@ Your Name 2025"
        font = ImageFont.truetype("arial.ttf", 36) # Make sure you have the font file
        text_width, text_height = draw.textsize(text, font)

        # Position the watermark (e.g., bottom right)
        width, height = im.size
        x = width - text_width - 10
        y = height - text_height - 10

        # Add the text
        draw.text((x, y), text, font=font, fill=(255, 255, 255, 128)) # White with some tra

        # Save the new image
        im.save("watermarked_output.jpg")
        print("Watermark added successfully!")

```

Days 87-88: Portfolio Project - Cafe & WiFi Website

This project marks a shift from desktop applications to web development using the Flask framework.

Project Goal

Build a simple website that displays a list of cafes from a CSV file. The website should show information like cafe name, location (as a Google Maps link), and WiFi strength.

Core Concepts

- **Flask Framework:** A lightweight web framework for Python. You'll learn the basics:
 - Routing (`@app.route('/')`)

- Rendering templates (`render_template()`)
- Running a development server.
- **HTML & CSS (with Bootstrap):** Using HTML to structure the web pages and Bootstrap to quickly style them and make them responsive.
- **Templating with Jinja2:** Flask uses Jinja to embed Python-like code directly into HTML files (e.g., for loops to display data).
- **Working with CSV Data:** Using Python's built-in `csv` module to read data from `cafe-data.csv` and pass it to the web templates.

Implementation Steps & Code Snippets

1. Project Setup:

- Install Flask: `pip install Flask` .
- Create a project structure:

```
/cafe-project
|-- main.py
|-- cafe-data.csv
|-- /templates
    |-- index.html
    |-- cafes.html
|-- /static
    |-- /css
        |-- styles.css
```

2. Basic Flask App (`main.py`):

- Set up the main application file.

```
from flask import Flask, render_template
import csv

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

3. Reading CSV and Rendering Data:

- Create a route `/cafes` that reads the `cafe-data.csv` file.
- Convert the CSV data into a list of lists or a list of dictionaries.
- Pass this data to the `cafes.html` template.

```
@app.route('/cafes')
def cafes():
    with open('cafe-data.csv', newline='', encoding='utf-8') as csv_file:
        csv_data = csv.reader(csv_file, delimiter=',')
        list_of_rows = []
        for row in csv_data:
            list_of_rows.append(row)
    return render_template('cafes.html', cafes=list_of_rows)
```

4. Displaying Data with Jinja (`cafes.html`):

- Use a Jinja `for` loop to iterate through the `cafes` data passed from `main.py` and display it in an HTML table.

```
<div class="container">
  <h1>Cafes with WiFi</h1>
  <table class="table table-dark table-striped">
    <thead>
      <tr>
        {% for header in cafes[0] %}
          <th scope="col">{{ header }}</th>
        {% endfor %}
      </tr>
    </thead>
    <tbody>
      {% for row in cafes[1:] %}
        <tr>
          <td>{{ row[0] }}</td>
          <td><a href="{{ row[1] }}" target="_blank">Map Link</a></td>
          <td>{{ row[2] }}</td>
          <td>{{ row[3] }}</td>
          <td>{{ row[4] }}</td>
          <td>{{ row[5] }}</td>
          <td>{{ row[6] }}</td>
        </tr>
      {% endfor %}
    </tbody>
  </table>
</div>
```

Days 89-90: Portfolio Project - Disappearing Text Writing App

Back to desktop applications, this project is a creative writing tool that forces you to keep typing or else your work disappears.

Project Goal

Create a simple text editor where, if the user stops typing for a set amount of time (e.g., 5 seconds), all the text they've written is deleted.

Core Concepts

- **Tkinter Event Handling:** Specifically, binding a key press event (`<Key>`) to a function.
- **Scheduling with `window.after()`** : This is the core mechanism of the app. `after()` schedules a function to be called after a certain number of milliseconds. This is how we'll check for inactivity.
- **Canceling Scheduled Events with `after_cancel()`** : To prevent the text from being deleted while the user is typing, we need to cancel the previously scheduled "delete" task every time a new key is pressed.

Implementation Steps & Code Snippets

1. UI Setup:

- A simple Tkinter window with a large `Text` widget is all that's needed.

2. Core Logic:

- Create a global variable or class attribute to hold the ID of the `after()` job.
- Create a function `delete_text()` that clears the text widget.
- Create a function `key_pressed()` that is triggered on every keystroke.
- Inside `key_pressed()` :
 - Cancel the previous `after()` job using `window.after_cancel(timer_id)` .
 - Schedule a new `after()` job to call `delete_text()` in 5000 milliseconds (5 seconds).
 - Store the new job ID.

```
import tkinter as tk

# --- Constants ---
INACTIVITY_TIME = 5000 # 5 seconds in milliseconds
timer = None

# --- Functions ---
def delete_text():
    text_widget.delete("1.0", tk.END)
    print("Text deleted due to inactivity!")

def on_key_press(event):
    global timer
    # If a timer is already running, cancel it
    if timer:
        window.after_cancel(timer)

    # Start a new timer
    timer = window.after(INACTIVITY_TIME, func=delete_text)

# --- UI Setup ---
window = tk.Tk()
window.title("Disappearing Text App")
window.config(padx=20, pady=20)

text_widget = tk.Text(window, height=20, width=80, font=("Arial", 14))
text_widget.pack()

# Bind the key press event to the text widget
text_widget.bind("<KeyPress>", on_key_press)

# Start the first timer when the app loads
timer = window.after(INACTIVITY_TIME, func=delete_text)

window.mainloop()
```

Days 91-92: Portfolio Project - Image Colour Palette Generator

This project combines web scraping and the Pillow library to extract a color palette from an image found online.

Project Goal

Build a script that scrapes a website for an image and then uses a library to identify the 10 most dominant colors in that image, creating a color palette.

Core Concepts

- **Web Scraping:** Using libraries like **BeautifulSoup** and **Requests** to fetch a web page and parse its HTML to find an image URL.
- **colorgram.py library:** A specialized library for extracting colors from images. It's much simpler than trying to do this manually with Pillow.
- **Data Structures:** Storing the extracted colors (which are often RGB tuples) in a list.

Implementation Steps & Code Snippets

1. Install Libraries:

- `pip install colorgram.py`
- `pip install beautifulsoup4`
- `pip install requests`

2. Scrape for an Image URL (Example):

- This part is highly dependent on the target website. The goal is to isolate the `src` attribute of an `` tag.
- *Note: This is a conceptual example. Scraping is fragile and site-specific.*

3. Extract Colors:

- Use `colorgram.extract()` to get the colors. This function can take a file path or a URL.
- The result is a list of `color` objects. Each object has an `rgb` attribute.

```
import colorgram

# This can be a local file path or a URL to an image
image_source = "image.jpg"
number_of_colors = 10

# Extract colors
colors = colorgram.extract(image_source, number_of_colors)

# Store them in a list of RGB tuples
rgb_palette = []
for color in colors:
    r = color.rgb.r
    g = color.rgb.g
    b = color.rgb.b
    new_color = (r, g, b)
    rgb_palette.append(new_color)

print(rgb_palette)
# Output might be: [(236, 224, 212), (198, 13, 32), (247, 237, 227), ...]
```

4. Application (Optional):

- You could use these colors in another project, for example, using the Turtle graphics library to draw a grid of dots representing the color palette (similar to a Damien Hirst spot painting).

Days 93-94: Portfolio Project - Google Dino Game Automation

This is a fun automation project where you write a Python script that plays the Google Chrome "No Internet" dinosaur game.

Project Goal

Use screen capture and browser automation to detect obstacles (cacti) in the game and trigger the dinosaur to jump automatically.

Core Concepts

- **Selenium:** A powerful browser automation tool. It's used to open the game (`chrome://dino`) and send commands to the browser (like pressing the space bar).
- **Pillow (PIL):** Used for taking screenshots of a specific region of the screen where the obstacles appear.
- **Pixel Analysis:** The core logic involves checking the color of specific pixels in the screenshot. If a pixel color matches the color of a cactus, it means an obstacle is approaching.
- **Timing and Control:** Using the `time` module to create loops and pauses to continuously check the screen.

Implementation Steps & Code Snippets

1. Install Libraries:

- `pip install selenium`
- `pip install pillow`
- You'll also need to download the correct `WebDriver` for your browser (e.g., `chromedriver`).

2. Setup Selenium:

- Open the Chrome browser and navigate to the game.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time

chrome_driver_path = "path/to/your/chromedriver"
driver = webdriver.Chrome(executable_path=chrome_driver_path)
driver.get("chrome://dino")

# Wait for page to load and get the body element to send keys
body = driver.find_element(By.TAG_NAME, "body")
time.sleep(1)
body.send_keys(Keys.SPACE) # Start the game
```

3. Game Loop and Screen Capture:

- Create a loop that runs as long as the game is active.
- Inside the loop, use Pillow's `ImageGrab.grab()` to take a screenshot of the area in front of the dinosaur. The `bbox` (bounding box) parameter is crucial here.

4. Obstacle Detection:

- Iterate over the pixels in the screenshot.

- The background color of the game is typically white or light gray. Obstacles (cacti, birds) are dark gray or black.
- If you detect a dark pixel, it signifies an obstacle.

```
from PIL import ImageGrab

# Coordinates for the box in front of the dino to check for obstacles
# These values need to be found by trial and error for your screen resolution.
OBSTACLE_CHECK_BOX = (250, 400, 300, 450) # (left, top, right, bottom)

def jump():
    body.send_keys(Keys.SPACE)

game_on = True
while game_on:
    # Take a screenshot of the detection area
    image = ImageGrab.grab(bbox=OBSTACLE_CHECK_BOX)

    # Check for non-white pixels
    for x in range(image.width):
        for y in range(image.height):
            pixel_color = image.getpixel((x, y))
            # The game's background is typically (247, 247, 247)
            if pixel_color != (247, 247, 247):
                jump()
                time.sleep(0.1) # Brief pause to avoid multiple jumps
                break # Exit inner loop
        else:
            continue # Only executed if the inner loop did NOT break
    break # Exit outer loop
```

Days 95-96: Portfolio Project - Custom Web Scraper API

This project brings together web scraping and API development. Instead of just running a scraper, you'll build an API that can be called to trigger the scraper and return the results.

Project Goal

Create a Flask API with a specific endpoint (e.g., `/api/v1/price`). When this endpoint is called, the server runs a web scraper (e.g., to get the price of a product on Amazon) and returns the scraped data in JSON format.

Core Concepts

- **API Development with Flask:** Going beyond rendering HTML to returning structured data (JSON).
- **JSON (JavaScript Object Notation):** The standard format for data exchange on the web. Python dictionaries are easily converted to JSON.
- **Flask's `jsonify`:** The function used to properly format a Python dictionary into a JSON response with the correct headers.
- **Scraping Logic:** Using BeautifulSoup and Requests to get the target data from a website.

Implementation Steps & Code Snippets

1. Project Setup:

- Install Flask, BeautifulSoup, and Requests.
- Create a `main.py` file.

2. Scraper Function:

- Write a standalone function that takes a URL, scrapes it, and returns the desired data (e.g., product price).
- **Crucially**, add `headers` to your request to mimic a real browser and avoid being blocked.

```

import requests
from bs4 import BeautifulSoup

def get_product_price(url):
    HEADERS = {
        "User-Agent": "Your User Agent String",
        "Accept-Language": "en-US,en;q=0.9"
    }
    response = requests.get(url, headers=HEADERS)
    response.raise_for_status()
    soup = BeautifulSoup(response.text, 'html.parser')

    # NOTE: These selectors are examples and WILL change.
    price_tag = soup.find(name="span", class_="a-price-whole")
    if price_tag:
        return price_tag.getText().strip(".")
    return "Price not found"

```

3. Flask API Endpoint:

- Create a Flask app.
- Define a route, for example, /get-price .
- This route will call your scraper function and return the result using jsonify .

```

from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def home():
    return "<h1>Price Scraper API</h1><p>Usage: /get-price</p>"

@app.route("/get-price")
def get_price():
    # Example URL
    product_url = "https://www.amazon.com/dp/B0756CYWWD/"
    price = get_product_price(product_url)
    return jsonify(product_name="Sample Product", price=price)

if __name__ == '__main__':
    app.run(debug=True)

```



Day 97: Portfolio Project - Deploying a Website

This day is about taking one of your web projects (like the Cafe & WiFi website) and putting it on the internet for everyone to see.

Project Goal

Deploy a Flask web application to a hosting service so that it's live on the web with a public URL.

Core Concepts

- **Web Hosting Platforms:** Understanding the role of services like **PythonAnywhere**, **Heroku**, or **Replit**. PythonAnywhere is often recommended for beginners.
- **WSGI (Web Server Gateway Interface):** The standard that allows a web server (like Apache or Nginx) to communicate with your Python Flask application. You don't need to code this, but you'll configure it on the hosting platform.
- **Environment Variables:** Best practice for managing sensitive information (like API keys) instead of hardcoding them.
- **File Management:** Uploading your project files (`.py` , `templates` , `static`) to the server.
- **Dependency Management:** Using a `requirements.txt` file to tell the server which Python packages to install.

Deployment Steps (Example with PythonAnywhere)

1. **Sign up for PythonAnywhere.**
2. **Upload Files:** Use the "Files" tab to upload your `main.py` , your CSV file, and your `templates` and `static` directories.
3. **Create a `requirements.txt` :** On your local machine, run `pip freeze > requirements.txt` and upload this file.
4. **Open a "Bash console"** on PythonAnywhere and run `pip install -r requirements.txt` to install your project's dependencies (like Flask).
5. **Configure the Web App:**
 - Go to the "Web" tab and create a new web app.
 - Choose the "Flask" framework and the Python version you used.
 - PythonAnywhere will create a `flask_app.py` file for you. You need to edit this file to import your app from `main.py` .
 - In the "Code" section on the Web tab, your `flask_app.py` should look like this:

```
# This file contains the WSGI configuration required to serve up your
# web application at http://<your-username>.pythonanywhere.com/
# ...
import sys

# add your project directory to the sys.path
project_home = '/home/<your-username>/mysite' # Change mysite to your project folder na
if project_home not in sys.path:
    sys.path = [project_home] + sys.path

# import the Flask app object
from main import app as application # noqa
```

6. **Reload the App:** Click the big "Reload" button on the Web tab. Your site should now be live at <your-username>.pythonanywhere.com .

Day 98: Data Science - Analyzing Space Race Data

The final days of the course pivot to data analysis and machine learning, starting with a fun dataset about the Space Race.

Project Goal

Use **Pandas** and **Matplotlib** to explore a dataset about space missions. Answer questions like: Which country has launched the most missions? How has the number of launches changed over time?

Core Concepts

- **Jupyter Notebooks / Google Colab:** The ideal environment for interactive data analysis.
- **Pandas DataFrame:** The primary data structure for working with tabular data in Python.
 - Loading data: `pd.read_csv()`
 - Inspecting data: `.head()` , `.info()` , `.describe()`
 - Cleaning data: `.isnull().sum()` , `.dropna()` , changing data types with `.to_datetime()` .
 - Querying & Grouping: `.groupby()` , `.value_counts()` .
- **Matplotlib:** The fundamental library for creating static visualizations in Python.
 - Creating plots: `plt.figure()` , `plt.bar()` , `plt.plot()` , `plt.title()` , `plt.xlabel()` .

Analysis Workflow & Code Snippets

1. Load and Inspect Data:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('mission_launches.csv')
print(df.shape)
print(df.isnull().sum()) # Check for missing values
```

2. Data Cleaning:

- Drop unnecessary columns with `df.drop()` .
- Handle missing values, for example, by removing rows with `df.dropna()` .
- Convert date columns to datetime objects for easier analysis:
`df['Date'] = pd.to_datetime(df['Date'])` .

3. Analysis and Visualization:

- **How many missions were successful vs. failed?**

```
status_counts = df['Status Mission'].value_counts()
plt.figure(figsize=(8, 6))
plt.bar(status_counts.index, status_counts.values)
plt.title('Mission Status Counts')
plt.show()
```

- **How many launches per country?** (Requires cleaning the 'Location' column to extract country names).
- **How many launches per year?**

```
df['Year'] = df['Date'].dt.year
launches_per_year = df['Year'].value_counts().sort_index()

plt.figure(figsize=(14, 7))
plt.plot(launches_per_year.index, launches_per_year.values)
plt.title('Number of Space Launches Per Year')
plt.xlabel('Year')
plt.ylabel('Number of Launches')
plt.grid(True)
plt.show()
```

Day 99: Data Science - Analyzing Deaths by Police in the US

This project involves working with a more complex and sensitive dataset to practice more advanced Pandas skills.

Project Goal

Use Pandas to analyze a dataset on police killings in the United States. Answer questions like: What is the age distribution of the deceased? Which states have the highest numbers of incidents? What are the racial demographics of the incidents?

Core Concepts

- **Advanced Pandas:**
 - `.groupby()` with multiple columns.
 - `.agg()` to apply multiple aggregation functions at once.
 - Merging and joining DataFrames (if using multiple datasets).
 - Working with string methods (`.str`) on columns.
- **Seaborn:** A higher-level plotting library built on top of Matplotlib that makes creating beautiful statistical plots easier. `sns.countplot()` , `sns.boxplot()` .

Analysis Workflow & Code Snippets

1. Load and Clean:

- Load the CSV file into a Pandas DataFrame.
- Check for missing values (`.isnull().sum()`) and decide on a strategy (e.g., drop rows where age is missing).
- Check data types with `.info()` and convert columns if necessary (e.g., `age` to numeric).

2. Exploratory Data Analysis (EDA):

- **Age Distribution:**

```
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.histplot(df['age'], bins=30, kde=True)
plt.title('Age Distribution of Deceased')
plt.xlabel('Age')
plt.show()
```

- **Deaths by Race:**

```
plt.figure(figsize=(12, 7))
sns.countplot(y=df['race'], order=df['race'].value_counts().index)
plt.title('Number of Incidents by Race')
plt.show()
```

- **Incidents by State:**

```
state_counts = df['state'].value_counts().head(10)
plt.figure(figsize=(12, 7))
sns.barplot(x=state_counts.index, y=state_counts.values)
plt.title('Top 10 States by Number of Incidents')
plt.show()
```

Day 100: Capstone Project - Predicting App Store Ratings

The final day introduces the fundamentals of machine learning by building a model to predict mobile app ratings.

Project Goal

Using a dataset of app store data, clean the data, select relevant features, and build a simple linear regression model with Scikit-learn to predict app ratings.

Core Concepts

- **Machine Learning Fundamentals:**
 - **Features vs. Target:** Identifying which columns are input variables (features, e.g., app size, price) and which is the output variable we want to predict (target, e.g., user rating).
 - **Train-Test Split:** The crucial process of splitting data into a training set (to build the model) and a testing set (to evaluate its performance on unseen data).
- **Scikit-learn:** The go-to library for machine learning in Python.
 - `train_test_split()` for splitting data.
 - `LinearRegression()` for the model itself.
 - Model training with `.fit()`.
 - Making predictions with `.predict()`.

- Evaluating performance with `.score()` (R-squared).
- **Feature Engineering:** The process of creating new features or transforming existing ones to improve model performance. For this project, it's mostly about selecting numeric features and dropping non-numeric ones.

ML Workflow & Code Snippets

1. Load and Preprocess Data:

- Load the data into a Pandas DataFrame.
- This is the most critical step. You'll need to clean the data extensively:
 - Drop rows with missing values.
 - Convert columns like 'Size', 'Installs', and 'Price' from strings (e.g., '1.9M', '1,000,000+', '\$2.99') into numerical formats (e.g., 1.9, 1000000, 2.99). This requires significant string manipulation.
 - Select only the numeric columns to use as features.

2. Define Features (X) and Target (y):

```
# After cleaning...
features = ['Reviews', 'Size', 'Installs', 'Price']
X = df[features]
y = df['Rating']
```

3. Train-Test Split:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. Train the Model:

```
from sklearn.linear_model import LinearRegression

# Create a regression model object
model = LinearRegression()

# Train the model using the training sets
model.fit(X_train, y_train)
```

5. Evaluate the Model:

- Use the trained model to make predictions on the *test* data.

- Check the R-squared score, which measures how well the model explains the variance in the data (a value closer to 1 is better).

```
# Make predictions using the testing set
y_pred = model.predict(X_test)

# The score (R-squared)
score = model.score(X_test, y_test)
print(f"Model R-squared score: {score:.2f}")

# You can also look at the model's learned coefficients
print("Coefficients: \n", model.coef_)
```

This final project serves as a launchpad into the world of machine learning, demonstrating how all the data cleaning and manipulation skills learned with Pandas are essential prerequisites for building predictive models.

Congratulations on completing 100 Days of Code! 🎉