

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.