🐍 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.

```python
# 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.