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').lo

if choice1 == "left":
  choice2 = input('You come to a lake. There is an island in the middle of the lake. Type "wait'
  if choice2 == "wait":
    choice3 = input("You arrive at the island unharmed. There is a house with 3 doors. One red,
    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 'dis
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 calcul
      num1 = answer
    else:
      should_continue = False
      calculator() # Recursion to start over

calculator()
```