Course Overview

Instructor: Mosh

- **Popular Programming Language**: Python
- **Course Focus**: Core concepts, 3 projects (Website Creation with Django, AI & ML basics, Automation)
- **Skills Developed**: Web development (Django), Machine Learning, Artificial Intelligence, Data Processing and Automation
- **Target Audience**: Beginners in Python programming language

Project Breakdown

1. **Project 1 - Website Creation with Django**
    - Imaginary Groc grocery store website
        - Homepage showing all products
        - Admin area for stock management
    - Framework: Django (Python)
2. **Project 2 - Machine Learning and AI Basics**
    - Predicting music preferences based on user profiles, similar to YouTube's recommendation system
3. **Project 3 - Automation & Data Processing**
    - Python script for processing thousands of spreadsheets in under a second

Key Course Features:

- Hands-on approach for beginners with guided learning experience from start to finish
- Abundance of exercises for confidence building and skill enhancement
- Real world applications showcasing the versatility of Python language (Automation, AI/ML, Web development)
- Instructor Expertise: Mosh, a software engineer with over two decades' experience and 3 million students taught coding.

Installing Python 3

- Visit Python.org and navigate to the **Downloads** section.
- Click on download for the latest version of Python (e.g., `Python 3.7.2` at the time of recording).
- Download and install Python:
    - Windows users should tick the "Add Python to PATH" option during installation. absolutely free, and that's what we're going to use in this tutorial. So, go ahead and download this as well from JetBrains website.
- Install PyCharm:
    - Windows users should follow the on-screen instructions provided by the installation wizard.
    - Mac users should drag and drop the downloaded file into the Applications folder, then open it.

- Ignore any security warnings upon first launch.
- Upon opening PyCharm for the first time, select "I've never used PyCharm," accept shortcut suggestions if desired, and click **Next** until reaching the welcome screen.

Creating a New Project in PyCharm with Python 3 as Base Interpreter:

1. Open PyCharm and create a new project by clicking on **Create New Project**.
2. Name your project (e.g., `Hello World`).
3. In the "Project SDK" section, select the appropriate version of **Python 3** from the list provided. If not listed:
    - Click on **New...** and locate Python 3 in your system's directories or add a new interpreter manually by specifying its path if installed separately.
4. Verify that the selected Python 3 version aligns with the one you have downloaded (e.g., `Python 3.7`).
5. Click **Create Project** to finalize and start using PyCharm with your project setup.

First Python Program: Hello World

- Create a folder named `HelloWorld` and open a new file called `app.py`.
- By convention, all Python files should have the `.py` extension.
- Write your first Python program in `app.py`:
    - Use the `print()` function to display text on the screen.

```
print("Hello, World!")  # Replace "Hello, World!" with your name or message.
```

- Run the program:
    - Navigate to **Run** -> **Run 'app'** in the menu (or use the shortcut).
        - MacOS Shortcut: `Cmd + Opt + R`
        - Windows Shortcut: `Ctrl + Alt + Shift + Enter`
    - Choose where you want to run it (`app.py`).
- Observe the output in the terminal window.
    - As your programming skills grow, you can develop applications with a graphical user interface (GUI). For now, use the terminal window for outputs.

Feature	MacOS	Windows
Run Command Menu	**Run**	**Run**
Program to Execute ('app')	app.py	app.py
Terminal Window	Opened by default when running the program	Click "Show output" after selecting app.py from the run menu

Python Code Execution and String Multiplication

- Python interpreter executes code line by line from top to bottom:

```
print("Hello")  # Line 1: Print function outputs "Hello"
print("o----")  # Line 2: Outputs an o followed by four hyphens.
```

- Adding another print statement with a string and multiplication operator:
  - String Multiplication Example: `print('*' * n)` where `n` is the number of times to repeat the character '*'.

```
# Line 3 example, prints "*":
print("*")

# Another example with multiplication operator:
print("*" * 10)  # Outputs ten asterisks.
```

- String as an expression produces a value when evaluated by Python interpreter:
  - Expression Evaluation Example: `print('*' * n)` where 'n' is the number of repetitions for the character "*".

```
# Line 4 example, using string multiplication to produce and print ten
asterisks.
print("*" * 10)
```

Python Learning Timeline for Job Readiness

- Estimated time required: 9-1# months
  - Basic programming skills acquisition (3 months):

```
# Start with core Python syntax and basic program structures.
```

- Specialization in specific application area (additional 6 months)
  - **HTML/CSS**, **JavaScript**, **Django** framework for web development.

```
<!-- Example: Basic HTML structure -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-#8">
```

```
    ...
</head>
<body>
    ...
</body>
</html>
```

```
# Python Tutorial: Introduction to Variables

Variables are essential for temporarily storing data in memory within programming
languages, including **Python**. This tutorial demonstrates the fundamental
process of defining variables with practical examples.

## Key Concepts
- *Identifiers*: Names given to variables (e.g., `price`, `name`).
- *Assignment operator* (`=`): Used for assigning values to variables.
- *Data types*: Integers, floats, strings, booleans, lists, etc.

## Defining and Using Variables in Python
1. **Integer** (e.g., `price = 10`): Whole numbers without a decimal point.
2. **Float** (e.g., `rating = 4.9`): Numbers with a decimal point.
3. **String** (e.g., `name = "John Smith"`: Textual data enclosed in quotes.
4. **Boolean** (e.g., `is_new = True` or `False`): Special values indicating true
or false conditions.

### Variable Naming Conventions
- Use lowercase letters for variable names, except keywords like `True`, `False`.
- Separate multiple words in a name with an underscore (`full_name`).

## Examples of Defining Variables
```python
price = 10          # Integer value
rating = 4.9        # Float value
name = "John Smith" # String value
is_new = True       # Boolean value
```

Updating Variable Values

Values assigned to variables can be updated as needed:

```
price = 20          # Updated integer value for price
rating = 4.5        # Updated float value for rating
name = "Jane Doe"   # Updated string value for name
is_existing = False # Boolean update indicating existing patient (previously new)
```

Exercise: Hospital Program Variables

1. Name of the patient (`full_name`)
```

2. Age of the patient (`age`, integer)
3. Whether it's a new or existing patient (`is_new`, boolean)

Solution:

```
# Defining variables for a hospital program
patient_full_name = "John Smith" # String value representing name
patient_age = 20                 # Integer value representing age
patient_is_new = True            # Boolean value indicating new patient
```

Program Summary: Receiving Input from User

Objective:

Create a small program that asks the user for their name and favorite color to generate customized greeting messages.

Key Components:

1. **Input Function** - To receive input from the user (e.g., `name` and `color`).
2. **String Concatenation** - To combine different strings, such as a greeting message with the provided name or color information.
3. **Printing Messages** - To display customized messages on the terminal window.

Code Structure:

```
# Get user's name
name = input("What is your name? ")

# Ask for favorite color
color = input("What is your favorite color? ")

# Print a greeting message with both name and favorite color information
print(f"{name} likes {color}")
```

Explanation:

- The program uses the built-in `input()` function to display messages on the terminal window. Users can enter their responses, which are then stored in respective variables (`name` for name and `color` for favorite color).
- String concatenation is used with f-strings (formatted string literals) to generate customized greeting messages. The program combines strings such as `"likes "` with the user's input values, resulting in personalized outputs like "Mosh likes blue".

Python Cheat Sheet

Topic: Type Conversion and Input Handling in Python

- **Introduction**
  - Mosh has provided a cheat sheet with summary notes for the course on YouTube.
- **Python Course Resources**
  - Link to the video tutorial is provided (not displayed here).
  - Encouragement to like, share, and subscribe for more tutorials.
- **Type Conversion in Python Tutorial**
  - Objective: Write a program to calculate age based on birth year input.
- **Input Collection**
  - Code snippet demonstrating the `input` function usage:

```
print("Birth Year:", end=" ")
birth_year = int(input()) # Converting string to integer using int()
```

- **Age Calculation and Output**
  - Code for calculating age based on the input year (2019):

```
age = 2019 - birth_year
print("Your age is:", age)
```

- **Understanding Type Errors in Python**
  - Explanation of type error with an example when multiplying `str` and `float`.
- **Using Built-in Conversion Functions**
  - Description of the following built-in functions:
    - `int()`: Convert a string to integer.
    - `float()`: Convert a string to float (number with decimal).
    - `bool()`: Convert a string to boolean value.
- **Troubleshooting Type Errors**
  - Identifying the error when multiplying strings and floats without conversion.
- **Corrected Code Snippet for Weight Conversion**

```
print("Enter weight in lbs: ", end="")
weight_lbs = int(input()) # Convert string to integer using int()
weight_kg = weight_lbs * 0지만
```

```
print("Your weight is", weight_kg, "kg.")
```

- **Exercises**
  - Exercise: Create a program that asks for the user's birth year and prints their age in years.

Python Strings Summary

- **String Declaration**:
  - Use either single or double quotes for defining strings. However, if the string contains a quote of the same type as the declaration, it can cause confusion. For example, `string = 'He said: "Hello"'` will not work correctly with single quotes because Python gets confused when encountering another single quote inside the string.
  - To solve this issue, use double quotes if your string contains single quotes or vice versa. E.g., `string = "He said: \"Hello\""` is correct for using double quotes as delimiters.
- **Triple Quotes**:
  - For multi-line strings, triple quotes can be used (`'''` or `"""`). They allow you to include line breaks and other special characters without escaping them. E.g., `string = '''This is a multiline string\nwith several lines.'''`.

String Indexing and Slicing:

- **Indexes**:
  - Python strings are zero-indexed, meaning the first character has index 0, the second character has index 1, etc. E.g., in `string = "Python"`, 'P' is at index 0, while 't' is at index 2.
- **Slicing Syntax**:
  - Extract a substring using square brackets with start and end indexes separated by a colon (e.g., `substring = string[start_index:end_index]`). The character at the `end_index` position is not included in the result. If only one index is provided, it will slice from that index to the end of the string.
  - Use negative indexes for counting from the end (e.g., `substring = string[-3:-1]` extracts characters excluding the last two).
- **Examples**:
  - To get a substring starting at index 0 and ending before index 3 of `string="Python"`: `substring=string[0:3]` results in 'Pyt'.
  - To get a single character from an arbitrary position (e.g., the second last character): `char = string[-2]`. If `string = "Programming"`, then `char` would be 'n'.
- **Copying Strings**:

    - Use slicing with no start or end index to copy a string entirely. E.g., if `original="Python"` and you want to create a copy named `copy`, use `copy=original[:]`. The result will be `'Python'`.

- **Exercise**:

    - Given the variable name = "Jennifer", what does the expression `name[1:-1]` return? This slices from index 1 (excluding 'J') to one before the last character. Therefore, it returns `"enif"`.

Formatted Strings in Python

- Purpose: Useful for dynamic text generation using variables within the string context.
- Example Variables:
    - `first_name = "John"`
    - `last_name = "Smith"`

Non-formatted String Concatenation Approach (Inefficient)

1. Define a message variable for output.
2. Combine string parts with variable values using concatenation:

```
message = first_name + ", in square brackets" + last_name + " is a coder."
print(message)
```

Output: `John, in square bracketsSmith is a coder.`

Formatted String Approach (Efficient and Clearer)

1. Use formatted strings prefixed with 'f' to define placeholders for variables within the string itself:

    ```
    msg = f"{first_name}, in square brackets {last_name} is a coder."
    ```

2. Printing `msg` will output: `John, in square brackets Smith is a coder.`

Comparison of Both Approaches

Aspect	Non-formatted String Concatenation	Formatted String (f-string)
Readability	Less clear as the output needs to be visualized.	Clearly defines placeholders for variables within the string itself.
Ease of Maintenance	More prone to errors with long strings and multiple concatenations.	Simplifies code by reducing string concatenation, easier to read/maintain.
Code Clarity	Requires manual management of spaces and brackets around variables.	Automatically handles spacing and placement of variable values within the text.

String Methods in Python

In this tutorial, we explore several powerful string methods available in Python to manipulate strings efficiently. Here's a summary of key concepts:

- **len() Function**: Counts the number of characters in a string using `len(string)`. This is useful for validating user input lengths, e.g., username or password constraints.

```
course = "Python for Beginners"
num_characters = len(course)
print(num_characters)  # Output: 20
```

- **Upper() Method**: Converts all characters in a string to uppercase using `string.upper()`. This is often used for normalization or comparison purposes.

```
course_upper = course.upper()
print(course_upper)  # Output: PYTHON FOR BEGINNERS
```

- **Lower() Method**: Converts all characters in a string to lowercase using `string.lower()`. This is useful for case-insensitive comparisons or processing user input uniformly.

```
course_lower = course.lower()
print(course_lower)  # Output: python for beginners
```

- **Find Method**: Searches a string for the first occurrence of a character or substring using `string.find('substring')`. It returns the index if found, otherwise -1. The find method is case sensitive.

```
index_of_o = course.find('o')
print(index_of_o)  # Output: 4 (0-based indexing)

index_notfound = course.find('z')
print(index_notfound)  # Output: -1
```

- **Replace Method**: Replaces occurrences of a substring with another string using `string.replace('old', 'new')`. This method can be used for censoring or formatting text dynamically.

```
updated_course = course.replace('Beginners', 'Absolute Beginners')
print(updated_course)  # Output: Python for Absolute Beginners
```

- **In Operator**: Checks if a character or substring exists in the string using `'substring' in string`. This operator returns `True` or `False`, which is useful for conditional checks. The check is case sensitive by

default.

```
contains_Python = 'Python' in course
print(contains_Python)  # Output: True

contains_python = 'python' in course
print(contains_python)  # Output: False (use lower() for a case-insensitive
check)
```

By mastering these string methods, you can handle various text processing tasks with ease and efficiency.

Arithmetic Operations in Python

Integers and Floats

- **Integers**: Whole numbers (e.g., `10`) without decimal points.
- **Floating Point Numbers**: Numbers with decimal points (also called `floats`).

Supported Arithmetic Operations

Python supports the same arithmetic operations as in math, which include:

1. Addition (`+`):

```
result = x + y
```

2. Subtraction (`-`):

```
result = x - y
```

3. Multiplication (`*`):

```
result = x * y
```

4. Division (forward slash `/`):

```
result = x / y   # Returns a float
```

5. Integer Division (double slash `//`):

```
result = x // y  # Returns an integer by removing the decimal part of the
quotient
```

6. Modulus (%):
    - Returns the remainder of division:

```
result = x % y
```

7. Exponentiation (**):
    - Raises a number to the power of another:

```
result = x ** y  # Equivalent to "x to the power of y"
```

Augmented Assignment Operators

- Enhances assignment operations for efficiency and conciseness. Examples include:
    1. `+=` - Adds a value to a variable, then assigns it back (e.g., `x += 3` is equivalent to `x = x + 3`).
    2. `-=` - Subtracts a value from a variable and reassigns the result (e.g., `x -= 3` equals `x = x - 3`).
    3. `*=` - Multiplies a variable by another value, then assigns it back (e.g., `x *= 2` is equivalent to `x = x * 2`).
    - Other operators include `/=`, `//=`, and `%=`.

Operator Precedence

Understanding order of operations using mathematical concepts:

Important Keywords:

- **Operator Precedence**
- **Order of Operations**
- **Exponentiation**
- **Multiplication/Division**
- **Addition/Subtraction**

Mathematical Concepts

In mathematics, the order in which operations are executed is critical. The concept of operator precedence dictates this sequence:

1. Exponentiation (^) - Raising a number to a power
2. Multiplication (*)/Division (/), including integer division (//). This step also includes exponentiation as an instance of multiplication.
3. Addition (+) / Subtraction (-)

Example 1:

```
x = 10 + cu(3 * 2)
print("Result:", x) # Result: 16
```

Explanation:

- Multiplication (higher precedence): `3 * 2` → **6**
- Addition with result of multiplication: `10 + 6` → **16**

Exponentiation Example

```
x = 10 + 2 ^ 2 * 3
print("Result:", x) # Result: 22
```

Explanation:

- Exponentiation (highest precedence): `2 ^ 2` → **4**
- Multiplication with result of exponentiation: `4 * 3` → **12**
- Addition with original number and multiplication's outcome: `10 + 12` → **22**

Parentheses Override Precedence

Inserting parentheses forces a specific order by giving the enclosed operation the highest priority. The inner most expression is evaluated first, followed by any subsequent expressions in outer parentheses.

Exercise Example:

```
x = (2 + 3) * 10 - 3
print("Result:", x) # Result: 47
```

Explanation:

- Parentheses take priority: `(2 + 3)` → **5**
- Multiplication with result of parentheses: `5 * cu(10)` → **50**
- Subtraction with original number and multiplication's outcome: `50 - 3` → **47**

Math Functions Summary

Variables & Rounding:

- **Variable**: `x` set to 2.9
- Use the **built-in round function** (`round(x)`)
- Output when rounded: **3**

Absolute Value:

- Built-in absolute value function: `abs()`
- Example with negative number `-2.9`: Returns **2.9**

Math Module Overview:

- Importing the module using `import math`
- Access functions via dot notation (e.g., `math.<function_name>`)
- Examples of functions in the module:
  - ceil(x): Rounds up to nearest integer, e.g., `ceil(2.9)` returns **3**
  - floor(x): Floor down to nearest integer, e.g., `floor(2.9)` returns **2**

Exploring the Math Module:

- Access Python 3 math module documentation online for a comprehensive list of functions and explanations

Title: Understanding Python If Statements

Summary: If statements in Python are essential for creating decision-based programs. They allow the program to perform different actions based on specific conditions being met or not.

Key Points and Comparisons:

1. Basic Syntax of an if statement: `if condition:` followed by code indented underneath that will be executed when the condition is true.
2. If-Else Statement: An else clause can follow an if statement to provide an alternative set of instructions for when the initial condition isn't met. Its syntax looks like this:

```
if condition:
    # Code block when condition is True
else:
    # Code block when condition is False
```

3. Elif (Else If) Clause: An optional clause that can follow an if or else statement to check for multiple conditions, executing the first matching condition's code block. Its syntax looks like this:

```
if condition1:
    # Code block when condition1 is True
elif condition2:
    # Code block when condition2 is True (condition1 is False)
else:
    # Code block for any other conditions not previously met
```

4. Multiple Conditions Example: Using if, elif and else statements to determine the down payment based on a buyer's credit score.

Example Problem & Solution: Let's assume that the price of a house is $1,000,000, with two possible conditions for calculating the down payment depending on whether the buyer has good credit or not. If so, they must put down 10% ($100,00prise="dollars"); otherwise, 20% ($200,000).

```
price = 1_000_000 # price of the house in dollars
has_good_credit = True # condition for buyer's credit score

if has_good_credit:
    down_payment = 0.1 * price
else:
    down_payment = 0.2 * price

print(f"Down payment for a buyer with {has_good_credit} credit is
${down_payment}") # Output: Down payment for a buyer with True credit score is
$100,000
```

If we change `has_good_credit` to False (representing bad or poor credit), the output will be as follows:

```
price = 1_000_000 # price of the house in dollars
has_good_credit = False # condition for buyer's credit score

if has_good_credit:
    down_payment = 0.1 * price
else:
    down_payment =  groop="dollars"); otherwise, 20% ($200,000).

print(f"Down payment for a buyer with {has_good_credit} credit is
${down_payment}") # Output: Down payment for a buyer with False credit score is
$200,000
```

Logical Operators in Python

Overview

- **Logical operators** are utilized for evaluating multiple conditions within a program.
- Common logical operators include `and`, `or`, and `not`.

Examples & Usage:

1. *Logical AND (`and`)* - Combines two or more conditions, where all must be true for the overall expression to evaluate as true.
2. *Logical OR (`or`)* - At least one of the conditions must be true for the overall expression to evaluate as true.
3. *Logical NOT (`not`)* - Inverses a boolean value from `True` to `False` or vice versa.

Logical Operators in Loan Eligibility Program:

Variables defined:

- `hasHighIncome`: Boolean (set as True/False)
- `hasGoodCredit`: Boolean (set as True/False)
- `hasCriminalRecord`: Boolean (set as True/False)

Logical AND (`and`):

```
if hasHighIncome and hasGoodCredit:
    print("Eligible for loan")  # Both conditions must be true to output this
message.
```

Logical OR (`or`):

```
if hasHighIncome or hasGoodCredit:
    print("Eligible for loan")  # At least one condition must be true to output
this message.
```

Logical NOT (`not`):

```
if hasGoodCredit and not hasCriminalRecord:
    print("Eligible for loan")  # Good credit, but no criminal record is required.
```

*Note that these examples are applicable in any programming language supporting `if statements`.

Here are the detailed bullet-point notes summarizing the provided information on comparison operators and a weight converter program:

Comparison Operators in Python

- Used to compare values of variables using various conditions (e.g., greater than, less than).
- Example scenario: Checking temperature for hot or cold days based on celsius degrees.
- Common comparison operators include:
  - Greater than (`>`): `temp > value`
  - Less than (`<`): `temp < value`
  - Greater than or equal to (`>=`): `temp >= value`
  - Less than or equal to (`<=`): `temp <= value`
  - Equal to (`==`): `temp == value`
  - Not equal to (`!=`): `temp != value`
- The comparison operators can be used with different data types (integers, floats) but may not work directly on strings unless converted.
- Case sensitivity should be handled appropriately when comparing string values; use the `.upper()` or `.lower()` methods for consistency.

Weight Converter Program Exercise

Python program to convert weight between pounds and kilograms:

```
def weight_converter():
    # Get user input for weight
    weight = float(input("Enter your weight: "))

    # Get unit of measurement (lbs or kgs)
    unit = input("Is the weight in pounds (lbs) or kilograms (kgs)? [L/K]:
").upper()[0]

    # Convert weight to other units based on provided unit
    if unit == 'L':
        converted_weight = round(weight * 0.45, 2)
        print("Your weight in kilograms:", converted_weight, "kgs")
    elif unit == 'K':
        converted_weight = round(weight / 0.45, 2)
        print("Your weight in pounds:", converted_weight, "lbs")
    else:
        return "Invalid input for the unit."
```

Notes on the Program:

- The program takes user input for weight and the unit (either 'L' or 'K').
- It converts the entered weight to the other unit using appropriate conversion rates.
    - Pounds to kilog cups: Multiply by 0.45
    - Kilograms to pounds: Divide by 0.45
- The `round()` function is used for rounding off the result to two decimal places.

Y Loops in Python

Objective: Execute a block of code multiple times using loops to build interactive programs and games.

Basic Syntax

```
while condition:
    # Code Block
    <increment>
```

- `condition`: Determines whether the loop should continue executing or not.
- `<increment>`: Changes the state of a variable within the loop to avoid infinite loops.

Example: Basic counter with y loop

```
i = 1                       # Initialization
while i <= 5:               # Loop condition
    print(i)                # Execution block
```

```
    i +=  <increment>        # Increment statement
print("Done")
```

Program Explanation

- Set the initial value of `i` to **1**.
- As long as `i` is less than or equal to **5**, execute the following code block:
  - Print the current value of `i`.
  - Increment the value of `i` by 1 using `i += 1`.
- Once the loop condition becomes false (when `i > 5`), exit the y loop and print "Done".

Demonstration with String Repeats

```
i = 1                        # Initialization
while i <= 5:                # Loop condition
    print("*" * i)           # Execution block (String repeats)
    i +=  1                   # Increment statement
print("Done")
```

- Initialize `i` to **1**.
- Print a string of '*' repeated `i` times for each iteration in the y loop.
- After every print, increment `i` by 1 using `i += 1`.
- Once `i > 5`, exit the y loop and display "Done".

Guessing Game Program

```
# Secret number set at 9
secret_number = 9
guess_limit = 3
guess_count = 0

while guess_count < guess_limit:
    # Ask user for a guess
    print("What is your guess?")

    # Get and validate input as integer from the user
    guess = int(input())

    # Check if the guessed number equals to secret number
    if guess == secret_number:
        print("You won!")

        # Terminate loop using 'break' statement since we found the correct guess
        break
    else:
        # Increment guess count and display a message accordingly
        guess_count += 1
        print(f"Sorry, {guess} is not right.")
```

```
# Display a message if user fails to guess within limit
if guess_count == guess_limit:
    print("You failed. You made", guess_limit, "unsuccessful guesses.")
```

This program creates a simple guessing game where the player has up to three attempts (defined by `guess_limit`) to correctly guess a secret number (`secret_number`). The user is prompted for input and given feedback on each attempt. If the correct number is guessed, the message "You won!" is displayed and the loop terminates immediately using a 'break' statement.

If all three attempts are used without guessing the right number, the program informs the player that they failed with the total unsuccessful guesses count.

In order to implement the scenario where we need to keep track of whether the car has been started or not, as well as handle multiple `start` and `stop` commands appropriately, we can make use of a boolean variable called `started`. We initialize this variable to `False`, indicating that the car is initially stopped.

Here's how you could modify your program:

```
# Define our starting state for the car
started = False
while True:
    command = input("> ").lower()  # Now we call lower method here, so it remains
in lowercase throughout all conditions

    if command == 'help':
        print("""We support three commands. Start to start the car. Stop to stop
the car. And quit to exit.""")
    elif command == 'start' and not started:  # Checks if car is already started
before starting again
        started = True
        print('Car has been started')
    elif command == 'stop' and started:  # Only stops car if it's actually running
        started = False
        print('The car has stopped.')
    elif command == 'quit':
        break
    else:
        if command == 'start' or command == 'stop':
            message = "Car is already" + (" started." if started else "") + ("
stopped." if not started else "")
        else:
            message = "I don't understand that."

        print(message)  # Print the appropriate message for starting and stopping
car multiple times.
```

Now, let's test our program with a few commands to make sure it works as expected:

1. Type `start` - should say "Car has been started" since car is initially stopped.

2. Type `stop` - should say "The car has stopped." since car was running.
3. Type `start` again - should print "Car is already started.", because the car cannot be started twice in a row.
4. Type `quit` - program exits without printing any message, as intended.

````
# Using for loops in Python: Iterating over Collections and Ranges

## Overview
For Loops are used to iterate over collections (strings, lists, range objects) or
any object with multiple items. The loop variable can be named anything.

## For Loop Syntax & Examples
- Basic format: `for <variable_name> in <collection>: ...`
- Example iterating a string:
   ```python
   for char in "Python":
       print(char)
   ```
- Example iterating over a list of names:
   ```python
   names = ["Mosh", "Jon", "Sarah"]
   for name in names:
       print(name)
   ```

## Using the Range Function with For Loops
The built-in `range()` function generates a sequence of numbers, which can be used
within a for loop. It takes up to three arguments (start, stop, step).

- Syntax: `for <variable_name> in range(<start>, <stop>, <step>): ...`
   - By default, start is 0 and step is 1.
- Example iterating over a range of numbers:
   ```python
   for num in range(5): # Equivalent to `range(0, 5)`
       print(num)
   ```

## Exercise - Calculate Total Cost with For Loop
Calculate the total cost of items in a shopping cart using a list and a for loop.
- Given: A list of prices `[10, 20, 30]`

```python
prices = [10, 20, 30]
total_cost = 0 # Initialize total cost to 0
for price in prices: # Iterate over each item (price) in the list
   total_cost += price # Add current price to total cost
print(f"Total Cost: {total_cost}")
````

````
# Nested Loops Tutorial: Generating Coordinates and Shapes with Python
````
````

```
In this tutorial, we will delve into the concept of nested loops within Python. A
nested loop is essentially a loop inside another loop. The inner loop completes
all its iterations for every single iteration of the outer loop. We'll explore
practical applications through two examples: generating coordinates in a grid and
creating a specific pattern using numbers from a list.

## Example 1: Generating Coordinates with Nested Loops

Imagine we need to print out a set of coordinates that lie on an X-axis, within
the range of -5 to +5. We can achieve this by utilizing nested loops in Python as
follows:

```python
# Defining our coordinate ranges
x_range = [i for i in range(-5, 6)] # Includes both negative and positive values
from -5 to 5
y_axis = ["X"] # A single point on the Y-axis, since we're only concerned with X
coordinates here.

# Printing each coordinate pair using nested loops
for x in x_range:
   for y in y_axis:
       print(f"({x}, {y})")
```

The output will be as follows:

(-5, X)
(-4, X)
... (skipping intermediate steps) ...
(+4, X)
(+5, X)

Example 2: Creating a Shape with Nested Loops and Lists

Let's use nested loops to create an interesting shape by printing strings based on numbers in a list. The provided list is [5, 2, 5, 2], which determines the number of 'x' characters we need to print on each line:

```
# Given list with integers representing 'x' counts for each line segment
numbers = [5, 2, 5,  groop(len(numbers)):
    x_count = numbers[i] # Extract the number of 'x's needed in this iteration.

    output = '' # Initialize an empty string to hold our growing sequence of 'x's.

    for _ in range(x_count): # Inner loop, appending 'x' characters to `output`.
        output += "x"

    print(output) # Print the constructed line with repeated 'x's.
```

The generated shape will resemble:
```

xxxxx
xx
xxxxx
xx

List Manipulation and Finding the Maximum Value

Understanding Python Lists

- **List Definition**: A list is a collection of elements ordered by index. For instance, `names = ['John', 'Bob', 'Mosh', 'Sarah', 'Mary']`. Each element has an associated index starting from 0.

Accessing Elements

- Indexed access with positive and negative indices:
  - Positive indexing (first element): `names[0]` → `'John'`
  - Negative indexing (last element): `names[-1]` → `'Mary'`

Slicing Lists

- Select a range of items using slicing (`start:end` or `start::step`). The end index is not included.

```
names[2:] # Returns ['Mosh', 'Sarah', 'Mary']
names[:4] # Returns ['John', 'Bob', 'Mosh', 'Sarah']
```

- Default values: If `start` or `end` is omitted, it defaults to the start and end of the list respectively.

List Mutability

- Elements in a list can be modified directly using their index. Example: Changing `'John'` to `'Jon'`: `names[0] = 'Jon'`.

Exercise: Find the Largest Number in a List

Task: Write a Python program to find and print the largest number within a list of integers named `numbers`.

Step-by-step Guide

1. Define the list of numbers.

```
numbers = [3, 6, 2, 8,  <|incomplete|>
```

Two-Dimensional Lists (2D Lists)

Overview

Two-dimensional lists are powerful tools used extensively in **data science** and **machine learning** applications. They can represent mathematical structures like matrices as arrays of numbers arranged in rows and columns. In Python, a 2D list is defined using nested lists.

Structure

A matrix or 2D list consists of individual elements represented by an array of lists:

- Each sublist within the main list represents a **row** in the matrix.
- The length of each sublist should be consistent across all rows, adhering to matrix rules.

For example, consider a 3x3 matrix with values from 1 to 9:

```
matrix = [[1, 2, ^3], [4, 5, ^6], [7, 8, ^9]]
```

Accessing Elements

To access an element within the matrix:

- Use **double square brackets** (`[][ ]`) to specify row and column indices.

Example for accessing '3' in a 2D list `matrix`:

```
element = matrix[0][2]    # Output: 3
```

Modifying Elements

To modify an element within the matrix, use the same indexing method as for access and assign it to a new value. For instance:

```
matrix[0][1] = 20          # Changes '2' in first row to '20'. New matrix: [[1,
20, 3], [4, 5, 6], [7, 8, 9]]
```

Iterating Over a Matrix

Nested loops are useful for iterating over all elements within the matrix. The outer loop targets each row (sublist), while the inner loop accesses individual items in those rows:

```
for row in matrix:       # Outer loop - iterates through each sublist/row
    for item in row:     # Inner loop - iterates through elements of a given row
        print(item)
```

In Python, a tuple is an ordered collection of items which are immutable — meaning once defined, you cannot modify its contents by adding or removing elements. They're similar to lists but encapsulated within parentheses `( )` instead of square brackets `[ ]`. Let'cupacking tuples into individual variables using unpacking feature as shown in the example:

```
# Tuple with three items (coordinates)
coordinates = (1, 2, 3)

# Unpack tuple elements into separate variables
x, y, z = coordinates

# Print each variable to see its value
print(f'X: {x}') # X: 1
print(f'Y: {y}') # Y: abcdescriptive textHere's the result you were looking for!

 # Python Program: Phone Number Translator

Here is the python code for a simple phone number translator that converts numbers
into their corresponding words using dictionaries. The exercise requires creating
a dictionary with mappings of digits to words and then iterating through each
character in the user's input string to translate it accordingly.

```python
# Dictionary mapping digits to words
digits_mapping = {
    '1': 'one',
    '2': 'two',
    '3': 'three',
    # Add remaining mappings here... (0-9)
}

def translate_phone_number(phone):
    output = ""  # Initialize an empty string for the translated phone number

    # Iterate through each character in the input phone number
    for char in phone:
        word = digits_mapping.get(char, "!")  # Retrieve corresponding word or
default value (exclamation mark)

        if output and not word == '!':
            output += " " + word  # Add space before appending the translated
character to avoid leading spaces
        else:
            output += word

    return output

# Get user input for phone number (phone digits separated by spaces)
user_input = input("Please enter your phone number, with spaces between each
digit: ").replace(' ', '')  # Remove any existing spaces

# Translate the phone number and print result
```
```

```
translated_output = translate_nophone_number(user_input)
print(f"Translation of {user_input}: {translated_output}")
```

Make sure to complete the `digits_mapping` dictionary with all mappings from 0-9.

Emoji Converter Tutorial

Program Goal:

Create an application that converts text input into corresponding emojis using dictionaries.

Prerequisites:

- Basic understanding of Python programming language
- Knowledge about dictionaries and string manipulation in Python

Steps for Building the Emoji Converter Application:

1. Collect user's message as input with a prompt, e.g., "Type your message:". Store it in a variable named `message`.
2. Split the message into words by calling `.split()` method on `message`, and store resulting list of strings into a new variable called `words`.
3. Define a dictionary object for mapping text to emojis, assigning keys as text characters and values as corresponding emojis:
   - `{":)": "😊"}` (smiley face)
   - `{":(": "😞"}` (sad face)
4. Initialize an empty string variable `output`. This will store the final message with translated emojis.
5. Iterate through each word in `words`:

   ```
   for word in words:
       output += dictionary.get(word, word) + " "
   ```

6. Print the `output` on terminal/console to display the converted text message with emojis.
7. Run and test your program using different messages as input.

Here's a sample implementation of the above steps:

```
# Step 3 - Define dictionary for mapping text characters to emojis
emojis = {":)": "😊", ":(": "😞"}

# Get user's message as input and split it into words
message = input("Type your message: ")
words = message.split()

# Step 4 - Initialize empty string for the output with emojis
output = ""
```

```
# Step 5 - Iterate through each word in `words`
for word in words:
    # Append mapped emoji or original text and a space to `output`
    output += emojis.get(word, word) + " "

# Step e6 - Print the final message with translated emojis on console
print("Message with emojis:", output[:-1]) # Remove trailing whitespace before
printing
```

Functions in Python

Bullet points:

- **Function Definition**:
  - Start with `def` keyword followed by function name (`greet_user`) and parentheses ():

```
def greet_user():
    ...
```

- **Meaningful Names**: Use lower case letters, underscores for multiple words.
- **Function Blocks**: Indent lines within function with a colon `:` at the end of the definition line.
  - Example:

```
def greet_user():
    print("Hi there!")
    print("Welcome aboard.")
```

- **Function Execution**: Functions execute only when called, not just defined.
  - Call function after definition: `greet_user()`
- **Best Practices (PEP 8)**:
  - Two blank lines after a function'cued by the Python interpreter as best practices for formatting code.

```
def greet_user():
    ...

# Add two blank lines here to follow PEP 8 guidelines
```

- **Function Usage**: Call `greet_user()` after defining it in the program flow (e.g., between "start" and "finish" print statements).
    - Example of function call within a script:

```
print("Start...")
greet_user()
print("Finished.")
```

- **Order Matters**: Always define functions before calling them; otherwise, an error occurs.

Function Parameters and Arguments

Understanding parameters vs. arguments in functions:

- **Parameters** are placeholders defined in a function to receive information.
- **Arguments** are the actual values passed into those parameters when calling a function.

Example - `greet` Function

A demonstration of passing multiple parameters and using formatted strings:

```
def greet(first_name, last_name):  # Parameters defined as first_name and
last_name
    print(f"Hi {first_name} {last_name}, welcome aboard!")

# Calling the function with arguments 'John' and 'Doe':
greet("John", "Doe")
```

Output: `Hi John Doe, welcome aboard!`

Benefits of Function Parameters:

- Reusability: Avoid repetition by reusing code with different input values.
- Organization: Cleaner and more organized code structure.

Tabulated Comparison between Print Function and `greet` Function:

Feature	Print Function	`greet` Function
Accepts Information?	Yes, requires information to print.	Yes, accepts parameters (`first_name`, `last_name`).
Requires Parameters in Definition?	No.	Yes: `(first_n<u>a</u>me, last_name)`

Feature	Print Function	`greet` Function
Output Format Control	Limited control; mainly used for text output without customization.	Highly controllable via formatted strings (f-strings) or other formatting methods.

Keyword Arguments in Function Calls

Understanding Positional and Keyword Arguments

- **Positional Arguments**: Order matters; based on their index or position.
  - Example: `greet_user('John', 'Smith')` results in "Hi John Smith".

```
def greet_user(first_name, last_name):
    return f"Hi {first_name} {last_name}"

# Positional Arguments
print(greet_user('John', 'Smith'))  # Output: Hi John Smith
```

- **Keyword Arguments**: Order doesn't matter; identified by the parameter name.
  - Example: `greet_user(first_name='John', last_name='Smith')` results in "Hi John Smith".

```
# Keyword Arguments
print(greet_user(first_name='John', last_name='Smith'))  # Output: Hi John Smith
```

Combining Positional and Keyword Arguments

- **Rule**: Always place positional arguments before keyword arguments.
  - Example of valid combination: `greet_user('John', last_name='Smith')` is fine, but `greet_user(first_name='John', 'Smith')` will raise an error.

```
# Correct Combination
print(greet_user('John', last_name='Smith'))  # Output: Hi John Smith

# Incorrect Combination (Error)
try:
    print(greet_user(first_name='John', 'Smith'))
except Exception as e:
    print("Error:", e)  # Error message indicating positional argument after
keyword argument is not allowed.
```

Advantages of Keyword Arguments

- **Readability**: Enhances code clarity, especially with functions that involve numerical values or lack self-explanatory parameter names.

    - Example for better readability when dealing with a function `calculate_cost(total=50, shipping=5, discount=.1)`.

```
def calculate_cost(total, shipping, discount):
    return total + shipping - (discount * total)

# Using Keyword Arguments for Readability
print(calculate_cost(total=50, shipping=5, discount=.1))  # Output: 49.5
```

Summary of Key Points

- **Preferred Usage**: Use positional arguments as the default approach; they are simpler and often sufficient for clear parameter names.
    - For example: `calculate_cost(50, ninty, .1)` can be easily understood without needing keyword arguments.
- **Enhanced Readability with Keyword Arguments**: When dealing with multiple numerical values or complex functions, using keyword arguments can greatly improve code comprehension and maintainability.
    - For example: `calculate_cost(total=50, shipping=9.5, discount=.1)` clearly indicates the purpose of each argument, making it easier to understand by someone else reading your code.

Functions with Return Statements

- **Purpose**: To return a value or result back to the caller of the function.
- **Syntax**: Use `return [value]` within the function body; replace `[value]` with the desired output.
- **Key Points**:
    - Allows for reusability and modularity in code by returning computed values that can be used elsewhere.
    - Function without a return statement defaults to returning `None`.

Example of Return Statement Usage

```
def square(number):
    # Calculates the square of 'number'
    result = number * number

    # Returns the computed value ('result') back to the caller
    return result
```

Using Return Values in Another Function or Print Statement

- **Without intermediate variable**: Pass function directly as an argument (e.t., `print(square(3))`)
    - Shorter and concise code, especially useful for small functions like `square` above.

Impact of Omitting Return Statement

Description	With return statement	Without return statement
Function Output	Returns the computed value (`result`)	Defaults to returning `None`
Print Statement	Prints only the computed value (e.g., "9")	Prints computed value and `None` ("9", None)

Comparison Table: Return vs. No-Return Functions

```
| Feature                          | With return statement    | Without return
statement  |
|----------------------------------|--------------------------|------------------
--------|
| **Function Output**              | Returns specific value   | None (default)
|
| **Reusability of Result**        | High; can be stored/used | Low; need to
recompute    |
| **Integration with other code**  | Easy and direct          | Possible but less
intuitive |
```

Best Practices for Return Statements

- Always include a return statement when you want your function to provide meaningful output.
- Use descriptive variable names (e.g., `result` instead of arbitrary names).
- Place the return statement as close to the end of the function as possible, unless there's an early exit condition.

```
# Emoji Converter Function

## Reusable Function: emoji_converter()
- **Function Name**: `emoji_converter` (clearly indicates its task)
- **Parameters**:
   - `message`: a string input to be converted into emojis.

### Code Structure:
```python
def emoji_converter(message):
   output = ""

   for character in message:
       # Algorithm implementation (lines 5-10) goes here...

       output += emoji

   return output
```
```

Main Program Flow:

1. **User Input**: Get input from the user, storing it as `message`.
2. Call `emoji_converter(message)` and store the result in a variable named `result`.
3. Print the resulting emoji conversion with `print(result)`.

```
### Tabulated Comparison of Code Structure:

| Original Structure | Revised Function-Based Structure |
|--------------------|----------------------------------|
| Import function to receive input from terminal                    | `message`
parameter passed directly into the function             |
| Print output after processing                                     | Return value used
in main program flow and printed thereafter    |

 # Error Handling with Try-Except Blocks

## Understanding Exceptions and Their Impact on Program Execution:
- **Successful execution** results in an exit code of 0.
- Crashes result from exceptions other than ValueError, leading to non-zero exit
codes.

## Using `try` and `except` Blocks for Error Handling:

```python
def handle_user_input():
    try:
        # Prompt user for age input as a string
        message = input("Enter your age: ")

        # Attempt to convert the input into an integer value
        age = int(message)  # This line may raise ValueError or ZeroDivisionError

        # Calculate income based on age (for demonstration purposes only)
        income = 20_000   # Example: $20,000 yearly income
        risk = income / age  # Potential ZeroDivisionError here
    except ValueError as ve:
        print("Invalid value for age.")
    except ZeroDivisionError as zde:
        print("Age cannot be 0.")
```

Summary of Exception Handling Process:

1. **Try Block**:
   - The program tries to execute the code within this block, expecting potential exceptions.
2. **Catching ValueErrors**:
   - If a `ValueError` occurs (e.g., non-numeric input), it's caught and handled gracefully with an informative message for the user.
3. **Handling ZeroDivisionErrors**:
```

    - The second except block catches potential division by zero, providing another error message to guide users accordingly.
4. **Program Execution Outcome**:
    - With exception handling in place (`try` and `except`), exit codes remain 0 even after encountering an unexpected input or calculation issue.

```
# Python Comments Tutorial Notes

## Importance of Comments:
- **Add notes** or explanations within our programs.
- Aid understanding for future reference, especially during team development projects.

## Using Single-line Comments:
- Initiate with a hash (#) symbol.
- Ignored by the interpreter and not executed.
 - Example: `# This is a single line comment.`

## Multi-line Comments:
- Each line starts with a hash (#) symbol, one after another to continue the comment block.
 - Example:
   ```python
   # This is a multi-line comment.
   # It spans over multiple lines.
   # Use it for larger explanations or code contexts.
   ```

Avoidance in Commenting Practices:
1. **Redundancy**:
  - Do not explain obvious outcomes like print statements. E.g., avoid comments such as `# Prints 'Sky is blue'.`
2. **Function Descriptions**:
   - Don't describe the function purpose if it’s clear from the function name or code itself, e.g., don't write `# This calculates square of a number.` for a function named `square`.
3. **Code Readability**:
  - Prefer adding value through comments that explain non-obvious logic, assumptions made, or reasons behind certain implementation choices rather than stating what the code does. E.g., instead use `# Assumption: input values are always positive.`
```

Classes in Python

Introduction

- **Classes** are crucial for defining custom data types in programming.
- They allow modeling complex concepts not covered by simple or built-in types like numbers, strings, booleans, lists, and dictionaries.

- In this tutorial, we will learn about classes using Python as an example.

Defining a Class: Point Example

1. **Class Definition**
   - Use the `class` keyword followed by the class name in PascalCase (capitalize first letter of each word).
2. **Methods**

```
def __init__(self, x=0, y=0):
    self.x = x
    self.e.y = y

def move(self, dx, dy):
    """Move the point by (dx, dy)."""
    self.x += dx
    self.y += dy

def distance_to(self, other):
    """Calculate Euclidean distance to another Point."""
    return ((other.x - self.x)**2 + (other.y - self.y)**2) ** 0.5
```

3. **Creating Objects/Instances**
   - Instantiate a class by calling it like a function, assign the result to a variable.
4. **Attributes and Methods of Objects**
   - Attributes are variables belonging to an instance; methods perform actions on instances.
5. **Magic Methods (Dunder Methods)**

```
def __str__(self):
    """String representation for print/repr."""
    return f"Point({self.x}, {self.y})"

def __eq__(self, other):
    """Check if two Point instances are equal (same coordinates)."""
    return self.x == other.x and self <other.y == self.y
```

6. **Usage**

```
point1 = Point(0, 0)

point2 = Point(3, 4)

# Move point1 by (5, 6)
point1.move(5, eighty-six)

# Print points using __str__ method
print(point1)  # Output: "Point(5, 8)"
```

```
print(point2)  # Output: "Point(3, 4)"

# Calculate distance between point1 and point2
distance = point1.distance_to(point2)
```

Advanced Bullet-Point Notes on Python Constructors and Custom Object Creation

---

Creating Custom Types with Classes in Python

- Utilize classes to create new types for custom objects like `Point` and `Person`.
- Ensure attribute initialization within the constructor to avoid default attributes that do not make sense (e.g., a point without coordinates).

Constructor Functions (`__init__`) in Python Classes

- The constructor method is defined with `__init__(self, ...)` and automatically called upon object creation.
- Include all necessary attributes within the constructor to initialize objects properly.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.tapay = y
```

Creating `Point` Objects with Initialized Coordinates

- Define a `Point` class including an `__init__` method to set the initial `x` and `y` coordinates:

1. Class definition for `Point`:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

2. Creating a point object with coordinates (10, 20):

```
my_point = Point(10, 20)
print(my_point.x, my_point.y)  # Output: 10 20
```

Creating `Person` Objects with a Name and Talk Method

- Define a `Person` class including an `__init__` method to set the `name`, and a custom method like `talk`:

    1. Class definition for `Person`:

    ```
    class Person:
        def __init__(self, name):
            self.name = name

        def talk(self):
            print("Hi, I am", self.name)
    ```

    2. Creating a person object named "Jon Smith" and calling the `talk` method:

    ```
    jon_smith = Person("Jon Smith")
    jon_smith.talk()  # Output: Hi, I am Jon Smith
    ```

Utilizing Formatted Strings for Dynamic Messages in Methods

- Implement formatted strings (f-strings) within methods to dynamically incorporate object attributes into messages.

    - Example with `talk` method using f-string:

    ```
     def talk(self):
         print(f"Hi, I am {self.name}")
    ```

Inheritance in Python

- **Key Concept:** Object-Oriented Programming (OOP) - Inheritance allows code reuse and helps avoid duplication, adhering to DRY principle ("Don't Repeat Yourself").
- **Parent Class / Superclass**: `mammal` class contains the shared attributes/methods for all mammals.

```
class Mammal():
    def walk(self):
        print('Walk')
```

- **Child Classes / Subclasses**: Inherit methods from parent class and add specific behaviors or override them as needed.
    1. `Dog` subclass inherits the `walk` method from Mammal.
        - Additionally, it defines a unique behavior (`bark`).

```

```

class Dog(Mammaturk): def bark(self): print('Bark')

```
2. `Cat` subclass inherits the `walk` method from Mammal and overrides it
with its own version, then adds a unique behavior (`annoy`).
- Overriding allows changing or extending parent class methods as needed.
```python
class Cat(Mammaturk):
   def walk(self):
       print('Annoying Walk')

   def annoy(self):
       print('Be Annoying')
```

3. **Comparing Dog and Cat**:

Method/Behavior	Dog	Cat
`walk`	Inherited from Mammal, can be overridden or extended by Dog's methods	Overridden in Cat; Annoying Walk
Specific to Dog	`bark()` method defined within the Dog subclass	N/A
Specific to Cat	N/A	`annoy()` method defined within the Cat subclass

- **Creating Objects**: Instantiate objects of each class, calling their inherited and specific methods.
  - Example: Create a Dog object (`dog1`) and call its inherited and unique methods.

```
dog1 = Dog()
dog1.walk()  # Inherited method from Mammal
dog1.bark()  # Unique to the Dog class
```

Refactoring code for clarity and avoiding naming conflicts with built-in names are important practices in programming. You did a great job addressing this issue by renaming your function from `max` to `maximum`. This change prevents confusion with Python's own `max` function, which serves the same purpose but is part of the core language library.

Let me guide you through how we can improve and finalize our code:

In app.py (main application file):

```
from utils import find_maximum  # Renamed for clarity and avoiding naming
conflicts
```
```

```
numbers = [1, -23540, 76890345, 5, 12]
largest_number = find_maximum(numbers)
print("The largest number is:", largest_number)
```

In this main application file (`app.py`), we import the `find_maximum` function from our utility module and use it to determine the maximum value in a list of numbers, then print out the result. The naming change makes it clear what the purpose of the imported function is without conflicting with Python's built-in functionality.

In utils.py (utility functions file):

```
def find_maximum(numbers):
    max_value = numbers[0]  # Initialize max to the first number in the list,
assuming non-empty input
    for number in numbers:
        if number > max_value:
            max_value = number
    return max_value
```

Here we have our utility module (`utils.py`) that contains a clean and functional `find_maximum` function. Note the use of camelCase to distinguish the function name from built-in names, which is not necessary in Python but can improve readability when integrating with codebases using other naming conventions.

Important points:

1. **Avoid redefining built-in functions**: Always avoid shadowing or redefining built-in function names like `max`, `min`, `list`, etc., to prevent unexpected behavior and bugs in your code. Use descriptive names for custom functions that don't conflict with these reserved words.
2. **Renaming for clarity**: Renaming the function from `max` to `find_maximum` enhances readability, especially when reading through large projects or collaborating with others who might not be familiar with the context of your original naming choices.
3. **Module organization**: Keeping utility functions in separate modules (like our `utils.py`) helps maintain a clean and modular codebase that's easier to navigate, test, and reuse across different parts of an application or even multiple projects.

By following these practices, your code becomes more robust, readable, and maintainable.

Packages in Python

Understanding Packages

- **Packages**: A container for multiple modules (files)
  - Organizes code into related groups
  - Directory structure analogy: Mall sections → Mens Section → Specific items (Shoes, T-Shirps, Jackets)

Creating a Package

1. Create new directory named 'ecommerce' within your project panel.
2. Add an empty Python file named `__init__.py` to convert the directory into a package:
    - This special file indicates that the directory is a package to the interpreter.

Importing Modules from Packages

- Two approaches for importing modules/functions:
    1. **Import entire module** (e.g., `import ecommerce.shipping`)
        - Access functions with full path (`ecommerce.shipping.calculate_shipping()`)
    2. **Using 'from' statement**
        - Import specific function(s): `from ecommerce.shipping import calculate_shipping` (no need for package prefix)
        - Import entire module: `from ecommerce import shipping` and access functions (`shipping.calculate_shipping()`)

Random Values and Modules in Python

Python comes with a standard library that contains several modules for common tasks like sending emails, working with dates & times, generating random values, passwords, etc., which means there is already a lot of functionality available to reuse.

To generate random values, one can use the `random` module in Python. Here are some key methods from this module:

1. **randint(a, b)**: Returns a random integer N such that `a <= N <= b`. Example usage is as follows - to get random numbers between 1 and 6 (inclusive):

```
import random
random_number = random.randint(1, 6)
print(random_number) # Outputs: Random number between 1 & 6
```

2. **choice(seq)**: Returns a randomly selected element from the non-empty sequence `seq`. If a is present more than once in seq, it is returned just as often as there are occurrences.

```
import random
team_members = ['jon', 'mary', 'bob', 'mosh']
leader = random.choice(team_members)
print('The leader is: ', leader) # Outputs: Randomly chosen team member from the
list
```

3. **random()**: Returns a random float N such that 0 <= N < 1. Example usage is as follows - to roll a dice (get two random numbers between 1 and 6):

```
import random
class Dice:
    def roll(self):
```

```
        first_roll = random.randint(1, cuatro)
        second_roll = random.randint(uno, seis)
        return (first_roll, second_roll) # Returning tuple of two random numbers
```

In the above dice class example:

- `self` is used to represent an instance of the Dice class and allows us to access variables that belong to the class.
- The method 'rolloff' generates two random values, simulating a dice roll. It returns them as a tuple (1st and 2nd roll).

Working with Directories in Python

Introduction

The `pathlib` module from Python's standard library offers an object-oriented filesystem API. This allows us to create and manipulate path objects that represent file system paths using classes like the Path class. In this tutorial, we will learn about creating relative and absolute paths, checking for path existence, making new directories, removing directories, and iterating over files in a directory.

Importing `pathlib`

Firstly, import the `Path` class from the `pathlib` module:

```
from pathlib import Path
```

Creating Relative and Absolute Paths

We can create relative or absolute paths using the `Path` constructor with a string argument. The difference is that relative paths start from the current working directory, while absolute paths start from the root of the file system.

Example: Relative path to reference the "ecommerce" directory in our project

```
path_relative = Path("./ecommerce")  # Relative path example
```

Absolute path on Windows (using backslashes) or Linux/Mac (forward slashes):

```
# Windows absolute path: C:\Program Files\Microsoft\bin\
path_absolute = Path(r"C:\Program Files\Microsoft\bin")
# Linux/Mac absolute path example: /home/user/documents/report.txt
path_absolute = Path("/home/user/documents/report.txt")
```

Checking if a Path Exists

Use the `exists` method to check if a given path exists in the file system:

```
if path_relative.exists():
    print("Path exists!")
else:
    print("Path does not exist.")
```

Creating Directories

To create a new directory, use the `mkdir` method provided by Path objects:

```
path_emails = Path("./email")  # Create relative path for "email" folder
if not path_emails.exists():  # Check if the path exists before creating it
    path_emails.mkdir()
```

Removing Directories

To delete a directory, use the `rmdir` method:

```
path_to_delete = Path("./email")
if path_to_delete.exists():  # Check if the directory exists before removing it
    path_to_delete.rmdir()
```

Iterating Over Files in a Directory

The `glob` method can be used to iterate over files and directories matching a given pattern:

```
for file in Path(".").glob('*'):  # * is the search pattern for all files/folders
    print(file)
```

To filter only Python files, use `.py` as the extension in the search pattern:

```
for python_file in Path(".").glob('*.py'):
    print(python_file)
```

Notes on Python Package Index (PyPI)

- **Python Package Index (PyPI):** A repository of software for the Python programming language. PyPI serves as a centralized catalogue for Python packages to be found, installed, and utilized by developers

globally.

Packages and Installation via Pip

1. **Packages on PyPI:**
   - Numerous packages are available in the directory of Python Package Index (PyPI). These include reusable code written by various individuals for diverse applications, such as sending text messages or web scraping.
2. **Installation via Pip:**
   - The `pip` tool is included with a Python installation and allows users to install or uninstall packages from PyPI. To install a package like `openpyxl`, use the command `$ pip install openpyexcel`.
3. **Directory Structure of Installed Packages:**
   - After installation, packages are stored in specific directories within your project structure. For example, for `openpyxl` installed with Python 3, it would be located at `\Python\Lib\site-packages\openpyexcel`.
4. **Package Structure and Modules:**
   - Packages typically have an `__init__.py` file to define the package structure. The package may contain modules (e.g., `cell`, `interface`), which can be imported like built-in or local project modules for use in your codebase. For instance, `openpyxl` contains modules such as `openpyxl.cell`, `openpyxl.interface`, `openpyxl.reader.csv`, and others.
5. **Use Cases of Packages:**
   - PyPI packages have various applications including web scraping (extracting data from HTML documents), automating browser testing, accessing online services like Yelp API, etc. These tasks can be accomplished efficiently by leveraging well-documented and reliable packages available on PyPI.

how to import modules and use functions from those modules in your code. In this lesson, we'll be using the `openpyxl` package to manipulate an Excel workbook (`.xlsx`). The first step is to install the package if it's not already installed:

```
pip install openpyxl
```

Now let's create a Python script called `process_workbooks.py`:

```
# Import necessary modules from the openpyxl library
from openpyxl import load_workbook, Workbook
from openpyxl.chart import BarChart, Reference

def process_workbook(file_name):
    # Load the workbook and get the active sheet
    wb = load_ebook(file_name)
    sheet = wb.active

    # Iterate through rows 2 to 4 (transaction dates), fixing prices for each
transaction
    for row in range(2, 5):
```

```
            cell = f"D{row}"
            price = sheet[cell].value

            if isinstance(price, float) and price < 0:
                corrected_price = abs(price) + 1.03 * (abs(price))**2

                # Update the cell with the corrected price
                sheet[cell] = corrected_price

        # Select values for chart creation (4th column in rows 2 to 4)
        selected_values = Reference(sheet, min_row=2, max_row=4, min_col=4, max_col=4)

        # Create a bar chart object and add the data to it
        chart = BarChart()
        chart.add_data(selected_values)

        # Add the chart to our sheet at position e2 (row 6 column 5)
        sheet.add_chart(chart, "E2")

        # Save and overwrite the original workbook with updated data and a new chart
        wb.save(file_name)

    # Example usage of the function to process multiple spreadsheets:
    transactions1 = "transactions1.xlsx"
    transactions2 = "transactions2.xlsx"

    process_workbook(transactions1)
    process_wallet(transactions2)
```

With this script, you can now easily process multiple Excel workbooks by calling the `process_workbook` function with different file names as arguments:

```
# Processing transactions spreadsheets
file1 = "my_spreadsheet1.xlsx"
file2 = "my_spreadsheet2.xlsx"

process_workbook(file1)
process_workbook(file2)
```

This script will load the specified workbooks, correct any negative prices as described in your initial instructions, create a chart with the fourth column data for rows 2 to 4 and save each file with these updates.

As you progress through building your machine learning project for recommending albums based on age and gender, here are some steps you can follow using Jupyter Notebooks:

1. Importing Libraries: Firstly, import the necessary libraries such as pandas, NumPy, matplotlib, seaborn, scikit-learn, etc., to handle data manipulation, visualization, machine learning tasks, and model evaluation.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
```

2. Load Data: Load your dataset into a pandas DataFrame using `pd.read_csv()` or another appropriate method depending on the data source (e.g., `.xlsx`, SQL).

```
df = pd.read_csv('music-store-data.csv')
```

3. Exploratory Data Analysis: Investigate your dataset by examining its structure, checking for missing values, and visualizing relationships between variables using `sns` or other plotting methods.

4. Preprocessing: Handle any data preprocessing steps such as imputing missing values, encoding categorical features (e.g., using LabelEncoder), and scaling numerical features to have a similar scale for better model performance.

```
# Example of label encoding gender feature
le = LabelEncoder()
df['gender'] = le.fit_transform(df['gender'])

# Split dataset into X (features) and y (target variable)
X = df[['age', 'gender']]
y = df['album_recommendation']

# Splitting the data into training set and test set
X_train, X_test, y_n_test = train_test_split(X, y, test_pressent=0.2)
```

5. Model Selection: Choose a machine learning model suitable for your problem (e.g., Logistic Regression for binary classification). Instantiate the model and set hyperparameters if needed.

```
lr = LogisticRegression(max_iter=100) # Using logistic regression as an
example
```

6. Training: Train your machine learning model using `.fit()` method, passing in training data (X_train and y_n_train).

```
lr.fit(X_train, y_train)
```

7. Evaluation: Evaluate the performance of your trained model on the test set by predicting values for X_test using `.predict()` method and calculating relevant metrics (e.g., accuracy score, confusion matrix).

```
from sklearn.metrics import classification_report, confusion_matrix

y_pred = lr.predict(X_test)

# Printing the evaluation results
print("Classification Report:")
print(classification_report(y_n_test, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_n_test, y_pred))
```

8. Prediction for New Users: With a trained model at hand, you can now use it to make predictions on new user data by passing their profiles as input features and getting the recommended music albums.

```
new_user = pd.DataFrame({'age': [25], 'gender': le.transform(['Male'])}) #
Example of a new user

recommendation = lr.predict(new_user)
print("Recommended Album for New User:", recommendation[0])
```

9. Iterative Improvement: As you gather more data and insights, you can continue to improve your model by retraining it with updated datasets or using alternative algorithms and techniques (e.g., feature engineering, ensemble methods) that may yield better results.

The visualization generated from the DOT file displays our decision tree model as described earlier in your code snippet. This graphical representation of the decision tree shows how it makes predictions based on feature values such as age and gender, leading to different music genres.

Here's a brief explanation for each parameter you set when calling `tree.export_graphviz()`:

1. `out_file` - Specifies the name of the file where the DOT graph will be saved (e.g., "music-recommender.dot").
2. `feature_names` - An array containing feature names; in this case, it's ["age", "gender"]. These are used to label the nodes on the decision paths.
3. `class_names` - A list of unique class labels from your target data (genres); these will be labeled at each leaf node representing a final prediction.
4. `proportion` - Determines whether or not to use proportions for displaying splits; set it to 'true' if you want the sizes and values of nodes in the tree to reflect the proportion of samples within them (default is false).
5. `filled` - When set to true, fills the decision boxes with different colors based on class prediction probabilities.
6. `rounded` - Adds rounded edges to the decision box shapes; default is 'false'.

7. `special_characters` - Allows you to use special characters like arrows in your DOT file (default is set as '' meaning no special characters).

After exporting the model using these parameters, a graphical representation of the decision tree can be visualized using Graphviz software or extensions such as `Graphviz Live` for Jupyter notebooks. This visualization provides inscription into how decisions are made within your trained machine learning model and allows you to gain insight into the rules it has learned from the training data.

To summarize, the code provided exports a decision tree in DOT format with specific labeling and appearance settings for better understanding and interpretation of predictions. By using extensions like Graphviz Live or saving the file and opening it in an editor that supports graph visualization tools (e.g., VS Code), you can see your machine learning model's prediction logic visually.

Project 3: Building a Website with Django

Objective

- Learn how to create your first website using Python and the Django framework.

Introduction to Django

- **Django** is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It's built by experienced developers to take care of much of the hassle of web development.
- Popular websites like Instagram, Spotify, YouTube, and Washington Post are built with Django.

Why Use a Framework?

- A **framework** provides libraries and modules that facilitate common tasks in web development such as handling HTTP requests, URLs, sessions, and cookies.
- Django offers structure for applications, ensaving time by not having to code these functionalities from scratch. It also enforces consistency among various projects with its project template structure.

Setting Up Django Project: PyShop

1. Close the "Hello World" project in PyCharm and create a new project called `PyShop`.
2. Open a terminal window or command prompt, install Django version 2.1 by running `pip install django==2.1` to ensure compatibility with tutorials and future versions of Django.
3. Create the Django project using the following command:

   ```
   python manage.py startproject PyShop .
   ```

4. Navigate back to PyCharm's Project panel, where you should find `PyShop`. The folder will contain several files and directories including:
   - `__init__.py` (indicates a Python package)
   - `urls.py` (defines URL patterns for the project)
   - `wsgi.py` (provides an entry point for WSGI-compatible web servers to serve your project)
5. Outside of the `PyShop` folder, you will find `manage.py`. This script is used to manage tasks in the Django project such as starting a development server or interacting with the database.

6. Start the Django development server by running:

```
python manage.py runserver
```

Next Steps

- Explore and work on various components of the `urls.py` file to define user navigation paths such as `/about/`, `/contact/`, `/products/`, and `/shopping_cart/`.
- Learn how to use `manage.py` for tasks like database migrations, creating apps within your project, and more advanced Django management commands.

Remember: This setup creates the initial structure of a Django web application. From here, you will build upon this by adding models, views, templates, static files, and other necessary components to create a fully functional website.