Python Course Introduction: Key Concepts and Projects

- Python is a popular programming language with applications in automation, AI, building websites and apps, and machine learning.
- Course objectives:
    - Learn core Python concepts
    - Build three projects using the Django framework
    - Explore machine learning applications of Python

Project 1: Grocery Store Website

- Develop a website for an imaginary grocery store with a homepage displaying products and an admin area for managing stock.
- Use the Django framework to build this project from scratch, even if you have no prior experience in web development.

Machine Learning Applications of Python

- Learn how to write Python programs that predict music preferences based on user profiles.
- Automate repetitive tasks using Python by processing thousands of spreadsheets in seconds.

Accessibility and Skill Level

- The course is designed for beginners, regardless of age or experience. It offers exercises to build confidence and hands-on guidance from a seasoned software engineer.

Instructor Bio

- Mosh Hamedani, the instructor, has over two decades of experience in software engineering and has taught coding to more than 3 million people.

Notes:

1. The Python programming language is versatile and widely used across various domains such as automation, artificial intelligence (AI), building websites and applications, and machine learning.
2. In this course, the main objective is to learn the core concepts of Python and apply them in three projects using the Django framework.
3. One of the projects involves creating a website for an imaginary grocery store with features like product display on the homepage and an admin area for managing stock. This project will be built from scratch using the Django framework, making it accessible to beginners with no prior experience in web development.
4. Another aspect covered in this course is machine learning applications of Python. Students will learn how to write programs that predict music preferences based on user profiles and automate repetitive tasks by processing thousands of spreadsheets in seconds.
5. The course is designed for beginners of all ages and skill levels, offering hands-on guidance from a seasoned software engineer and exercises to build confidence in coding.

6. The instructor, Mosh Hamedani, has over two decades of experience in software engineering and has taught coding to more than 3 million people. He will lead the course, ensuring that learners acquire practical skills and knowledge in Python programming.

Installing Python 3

Key Points:

- Open browser & head to Python.org
- Download latest version of Python (at the time of recording: Python 3.7.2)
- Windows users must check "Add Python to PATH"
- Mac users install via setup wizard, agree to license agreement, and install

Code Editor:

- PyCharm: Integrated Development Environment (IDE) for writing Python code
  - Download from jetbrains.com/pycharm
  - Choose Community edition (free) or Professional addition (paid)
  - Windows users follow installation wizard; Mac users drag & drop, then launch

Project Creation:

- PyCharm's main page: Create new project -> Name project "Hello World" -> Select Python 3 interpreter -> Click "Create"

Precise Notes:

```
1. Open browser and navigate to python.org.
2. Download the latest version of Python (at the time of recording, it was 3.7.2).
3. For Windows users: Check the box that says "Add Python to PATH" during
installation.
4. For Mac users: Install Python via a setup wizard and agree to the license
agreement.
```

Code Editor Installation:

```
1. Go to jetbrains.com/pycharm.
2. Choose either the Community edition (free) or Professional addition (paid) for
PyCharm.
3. Follow the installation wizard for Windows users, and drag & drop for Mac
users, then launch the application.
```

Project Creation:

```
1. In PyCharm's main page: Create a new project by clicking on "Create New
Project".
```

```
2. Name the project "Hello World" in the dialog box that appears.
3. Ensure the Python 3 interpreter is selected from the list, and then click
"Create" to create the project.
```

Numerical Values:

- None

Markdown Syntax:

- None

Latex Equations:

- None

Advanced Bullet Points:

- None

Summarizing Important Parts of Reading or Topic:

- Install Python 3 from python.org (latest version)
- Add Python to PATH for Windows users during installation
- Install PyCharm IDE (Community edition free, Professional addition paid)
- Set the base interpreter to Python 3 in PyCharm's project settings

```
1. Go to python.org and download the latest version of Python.
2. For Windows users, make sure to check "Add Python to PATH" during installation.
3. Download PyCharm from jetbrains.com/pycharm (Community edition free or
Professional addition paid).
4. In PyCharm, set the base interpreter to Python 3 in the project settings.
```

Essential Information:

- Install Python from python.org
- Add Python to PATH for Windows users
- Download and install PyCharm IDE (Community edition free or Professional addition paid)
- Set the base interpreter to Python 3 in PyCharm's project settings

Title: Python Hello World Program

Objective

The objective of this tutorial is to create a simple "Hello World" program in Python using the print() function and save it as an `.py` file. The program will be executed through the terminal, allowing us to see the output on screen.

Prerequisites

1. A computer with Python installed or access to a cloud-based Python environment.
2. Basic understanding of how to open and edit text files using a code editor or IDE.
3. Familiarity with terminal commands (if not, please refer to online resources for an introductory tutorial on the command line).

Creating the Hello World Program

1. Open your preferred code editor or IDE.
2. Create a new Python file by going to File > New > Python File (or similar menu options depending on your editor). This will create a blank file, which you can rename as desired. For example, let's call it `app.py`.
3. In the Python file, type or copy-paste the following code:

```
print("Hello World!")
```

4. Save the file using File > Save (or similar menu options) and ensure that the file extension is `.py`.
5. Open your terminal and navigate to the folder where you saved the `app.py` file by entering the appropriate command, such as:

```
cd /path/to/your/folder
```

6. Once in the correct directory, run the Python program using the following command:

```
python app.py
```

7. The terminal will execute the program and display the output "Hello World!" on the screen.

Conclusion

Congratulations, you have just created your first Python program! This simple example demonstrates how to print text to the console using the `print()` function. As you continue learning Python, you will discover more advanced features and functionality that can be used in combination with this basic building block.

How Python Code Gets Executed:

```
* A Python program consists of one or more statements which are executed
sequentially from top to bottom by the Python interpreter.
* Each line of code represents a single statement in the Python programming
language.
* The first line is executed, then the second, and so on until all lines have been
processed.
```

Python Code Execution Example:

```
# Draws an imaginary dog with hair
print("Name:")  # Print statement to ask for name
print()          # Blank line
print("The Dog:") # Print statement to identify the dog
print("o" + "-" * 4)  # Print statement to draw head and body of the dog
print("  | | |")      # Print statement to draw legs of the dog
```

Execution Process:

1. The Python interpreter reads and interprets the first line `print("Name:")`.
2. It then prints "Name:" on the console.
3. The next line `print()` is executed, which simply prints an empty line.
4. The third line `print("The Dog:")` is interpreted by the interpreter and printed as "The Dog:" on the console.
5. Finally, the fourth line `print("o" + "-" * 4)` is processed, where the string "o" is concatenated with four hyphens using the multiplication operator. The resulting string "o----" is then printed to represent the head and body of the imaginary dog.
6. The final print statement `print("  | | |")` is interpreted by the interpreter and printed on the console to draw the legs of the imaginary dog.

```
* Expressions are pieces of code that produce values. For example, in line 4, `"o"
+ "-" * 4` produces a string with four hyphens following an 'o'.
* The multiplication operator is used for repetition in Python strings and lists.
It multiplies the number to the right by the number of elements or characters to
its left.
```

Homework Exercise:

Use print statements to draw another shape, such as a heart, ball, or other shapes of your choice.

```
* Consider using different techniques and operators, like concatenation (`+`),
multiplication (`*`), or even mathematical operations (e.g. `sqrt()`) for drawing
more complex shapes.
```

Python Learning Time Summary:

- Python learning time depends on individual commitment and goals.
- 2 hours/day of consistent practice for 3 months leads to basic programming proficiency.
- Specialization in a specific area, such as web applications or AI, requires additional courses (e.g. HTML, CSS, JavaScript, Django).
- Total time to become job-ready: 9-12 months with a combination of Python and specialized skills.
- Junior developer salary expectations: $50-60K/year initially, potentially rising to $100-120K/year as experience grows.
- Encouragement for aspiring learners to commit to daily practice and share their aspirations in the comments.

Keywords: Python, learning time, job readiness, specialization, web development, salary expectations, commitment, comment box

Variables are essential components in programming that store data temporarily in a computer's memory. They help us organize our code by attaching labels to specific pieces of information. This allows us to use these labels anywhere in our program to access the corresponding values. In Python, variables are created using an identifier (name) followed by an equals sign and then the value assigned to that variable.

Here's a detailed summary of the key points from this content:

1. Variables store temporary data in computer memory.
2. Identifiers (names) are used to label the stored values, allowing them to be referenced later in the program.
3. The process of creating and using variables involves allocating memory, storing a value, attaching a label, and then accessing that value through the label.
4. Variables can store various types of data: integers (whole numbers), floats (numbers with decimal points), strings (sequences of characters), and booleans (true/false values).
5. Python uses an underscore to separate multiple words in variable names for better readability and maintainability.
6. Variables should be defined using lowercase letters, except for built-in functions like `True` or `False`.
7. In this context, a hospital program requires three variables: one for the patient's name, another for their age, and a boolean to indicate whether they are a new or existing patient.

To summarize in bullet points:

- Variables store data in computer memory.
- Identifiers (names) label stored values.
- Process involves allocating memory, storing value, attaching label, and accessing via label.
- Variables can hold integers, floats, strings, and booleans.
- Python uses underscores to separate words in variable names.
- Define variables using lowercase letters, except for built-ins.
- Hospital program requires three variables: name, age, new/existing patient boolean.

Title: Receiving Input in Python

- To receive input from a user in Python, you can use the `input()` function. This function will wait for the user to enter a value and then return that value.
- The input() function takes a string as an argument which is displayed on the terminal window as a prompt message for the user to type their answer.
- You can store the returned value from the input function in a variable using assignment operator `=`, like this: `name = input("What is your name? ")`.
- To concatenate strings together and print them dynamically in Python, you use the `+` operator to combine strings. For example, `print("Hi " + name)`.
- The space after the question mark on the terminal window helps keep the cursor separated from the prompt message, which makes it easier for users to type their answer.
- You can modify your program to ask multiple questions by simply adding more input statements and storing each returned value in a separate variable. For example:

```
name = input("What is your name? ")
color = input("What is your favorite color? ")
print(f"{name} likes {color}")
```

Python Cheat Sheet (Notes)

1. Type Conversion (Variables)

- Input function always returns a string, so it is necessary to convert strings into numbers for arithmetic operations.
    - int() function converts a string to an integer.
    - float() function converts a string to a floating point number.

2. Type Conversion (Functions)

---

1. Python has built-in functions to convert data types:

- int()
- float()
- bool()

2. Example Program - Age Calculator

```
# Prompt user for year of birth
birth_year = input("Enter your birth year (e.g. 1982): ")
age = 2019 - int(birth_year)
print(f"Your age is {age} years old.")
```

- When using `input()` function, the value entered by the user is stored as a string.

- To perform arithmetic operations, like subtracting the current year from the birth year, the input must be converted to an integer using `int()`.
- The resulting age will also be of type int.

3. Additional Notes

- In Python, when working with numbers, it is generally best practice to convert strings into integers or floats as soon as possible for clarity and ease of reading code.

Strings in Python

- Strings are sequences of characters enclosed within quotes (either single `''` or double `""`).
- To include a quote character within the string, use an opening quote that is different from the closing one:
  - If the string starts with a single quote, end it with a single quote.
  - If the string starts with a double quote, end it with a double quote.
- When defining strings that span multiple lines or contain special characters, use triple quotes `'''` or `"""` to indicate the start and end of the string:

```
message = """
Hi Jon!
This is a multi-line
string with lots of spaces.
"""
print(message)
```

- Access individual characters in a string using indexing with square brackets `[]`. The first character has index 0, the second character has index 1, and so on:

```
name = "Jennifer"
print(name[0]) # prints 'J'
print(name[5]) # prints 'r'
print(name[-2]) # prints 'n' (indexing starts from 0)
```

- Python strings are immutable, which means that they cannot be modified once created:

```
message = "Hello"
message[0] = "H"    # This will cause an error since strings are immutable.
```

- Slicing with square brackets and colons can be used to extract parts of a string, including substrings from the start index to the end index:
  - Start index is inclusive (included) in the substring being returned.
  - End index is exclusive (not included) in the substring being returned.

```
name = "Jennifer"
print(name[1:])   # prints 'ennifer'
print(name[:-2]) # prints 'Jen'
```

Note that if no start or end index is specified, the default values are 0 for the start index and the length of the string for the end index.

Exercise: Define a variable named `surname` with value "Williams". Print the characters from index 1 to index -2 (last character) without using any string functions or methods. What do you expect to see on the terminal?

Formatted Strings in Python

- Use when dynamically generating text with variables
- Prefix string with `f` (formatted)
- Placeholders within curly braces are filled with variable values

Example:

```
first_name = "John"
last_name = "Smith"

message = f"{first_name}, in [{last_name}] is a coder."
print(message)  # Outputs: John, in [Smith] is a coder
```

Note: The `f` prefix and curly braces syntax can also be used with other string formatting functions such as `format()` or `format_map()`

```
String Methods
--------------

1. Length: `len(course)` - returns number of characters in string (e.g., 20 for
"Python for beginners").
2. Capitalization:
   - `upper()` converts all lower case to upper case.
   - `lower()` converts all upper case to lower case.
3. String manipulation methods:
   - `replace(old, new[, max])`: replaces occurrences of `old` with `new`,
returning a copy (e.g., "Python for absolute beginners").
4. Character search and check:
   - `in`: checks if a substring exists in the string (e.g., `'python' in
course`).
5. General notes on methods vs functions:
   - Methods are built into specific objects, like strings or lists, and belong to
them.
   - Functions are general purpose, not limited to one object type.
6. Common mistakes when using Python string methods:
```

```
    - `upper()` or `lower()` do not modify the original string, they create a new
copy.

Examples of Python code for each method/function:
```python
course = "Python for beginners"
print(len(course))  # 20
print(course.upper())
print(course)
print(course.lower())
print(course)
print(course.replace("for", "with"))
print(course)
print("python" in course)
```

Note: Some methods and functions may require string input, as shown above with the `course` variable. Always check documentation for specific method/function usage and examples.

Python supports various arithmetic operations that are commonly used in mathematics. These include addition, subtraction, multiplication, division, and modulus (remainder). Python also has a concept called "augmented assignment" operators which allow for more concise code by combining an operation with an assignment statement. The following table summarizes the arithmetic operations and their corresponding mathematical symbols:

Python Operation	**Mathematical Symbol**	**Description**
`+` (addition)	Plus sign	Adds two numbers together
`-` (subtraction)	Minus sign	Subtracts one number from another
`*` (multiplication)	Asterisk	Multiplies two numbers together
`/` (division)	Slash	Divides a number by another number
`%` (modulus)	Percent sign	Returns the remainder of the division operation
`**` (exponentiation)	Two asterisks	Raises one number to the power of another number
`x += y`	Augmented addition	Adds `y` to `x` and assigns the result back to `x`
`x -= y`	Augmented subtraction	Subtracts `y` from `x` and assigns the result back to `x`
`x *= y`	Augmented multiplication	Multiplies `x` by `y` and assigns the result back to `x`
`x /= y`	Augmented division	Divides `x` by `y` and assigns the result back to `x` (returns a float)
```

Python Operation	Mathematical Symbol	Description
`x //= y`	Augmented floor division	Divides `x` by `y`, truncating any remainder, and assigns the result back to `x` (returns an integer)
`x %= y`	Augmented modulus	Calculates the remainder of `x` divided by `y` and assigns the result back to `x`

It's important to note that in Python, when performing division using the forward slash (`/`) operator, the result is always a floating-point number (float). If you want an integer result for division, you can use either the floor division operator (`//`) or the modulus operator (`%`) followed by dividing the remainder by `y`.

Python also supports bitwise operations such as bitwise and (`&`), bitwise or (`|`), and bitwise xor (`^`), which can be used to perform arithmetic operations on integers at a bit-level. These will not be discussed further in this summary.

Example code:

```
# Arithmetic operations
print(10 + 3)  # Outputs: 13 (addition operator `+`)
print(10 - 5)  # Outputs: 5 (subtraction operator `-`)
print(10 * 2)  # Outputs: 20 (multiplication operator `*`)
print(10 / 2)  # Outputs: 5.0 (division operator `/`)
print(10 % 3)  # Outputs: 1 (modulus operator `%`)
print(10 ** 3)  # Outputs: 1000 (exponentiation operator `**`)
```

```
# Augmented assignment operators
x = 10
print(x + 3)  # Outputs: 13

x += 3  # Increments `x` by 3 and assigns the result back to `x`
print(x)  # Outputs: 16
```

Operator Precedence

Operator precedence refers to the order of operations used when evaluating expressions. The general rule is that multiplication, division, and addition have higher precedence than exponentiation.

Examples:

1. 10 + 3 * 2 = 16 because the multiplication has a higher precedence than the addition.
2. x = 2 ** 2 * 3 = 22 because the exponentiation has the highest precedence and is applied first.
3. x = (10 + 3) * 10 - 3 = 47 because parentheses take priority over all other operators, so the expression within them is evaluated first, followed by multiplication and subtraction in order of precedence.

Summary:

- Operator precedence determines the order of operations when evaluating expressions.
- Multiplication, division, and addition have lower precedence than exponentiation.
- Parentheses can be used to override operator precedence.

Summary of Math Functions Tutorial

- To work with numbers, use built-in round and abs functions:
  - `round(x)` rounds to nearest integer (e.g. 3 for 2.9)
  - `abs(y)` returns absolute value (e.g. 2.9 for -2.9)
- For complex mathematical calculations, import the `math` module:

```
import math
```

- Access functions within the `math` module using the dot operator:

```
math.sin(angle)  # sine of angle
math.cos(angle)  # cosine of angle
math.tan(angle)  # tangent of angle
math.sqrt(x)      # square root of x
math.exp(x)       # e raised to the power of x
```

- Documentation for more functions: Python 3 math module

Notes on Math Functions Tutorial

- The tutorial discusses useful built-in functions in Python, such as `round()` and `abs()`.
  - These functions allow users to manipulate numbers more effectively.
  - For instance, the `round()` function can be used to round a number to the nearest integer (e.g. 3 for 2.9).
- Additionally, the tutorial introduces the concept of modules and demonstrates how to import the `math` module in Python.
  - Modules are separate files containing reusable code and allow users to organize their programs more efficiently.
- The tutorial emphasizes the need to import the `math` module when working with complex mathematical calculations, as it contains a variety of built-in functions for performing such operations.
  - By importing the `math` module, users can access various useful functions through the dot operator (e.g. `math.sin()`).
- The tutorial encourages learners to explore and learn more about the available functions in the `math` module by referring to its documentation.
  - This resource provides a comprehensive list of all the functions within the module, along with explanations for each one.

Conclusion The Math Functions Tutorial teaches users how to use built-in Python functions such as `round()` and `abs()` for working with numbers. It also introduces modules and demonstrates the importance of importing the `math` module when performing complex mathematical calculations. Lastly, it encourages

learners to explore and learn more about the available functions in the `math` module by referring to its documentation.

If Statements In Python, an if statement is used to execute code only if a condition is true. Here is an example of an if statement in Python:

```
if some_condition:  # The condition must be a boolean value (True or False)
    # Execute this block of code if the condition is true
else:  # If the condition is false, execute this block of code
    # This block will only be executed if the condition is false
```

Here are some key points about if statements in Python:

- The condition must evaluate to either True or False.
- Only one of the blocks (either the "if" block or the "else" block) will be executed, depending on whether the condition is true or false.
- If multiple conditions need to be checked, a nested `elif` statement can be used instead of using another `if` statement.

```
# Here's an example with two conditions and an 'else' clause:
if some_condition:
    # Execute this block if the first condition is true
elif some_other_condition:  # If the first condition is false, check this second
condition instead
    # Execute this block if the second condition is true
else:  # If both conditions are false, execute this block
    # This block will only be executed if both previous blocks are not executed
```

Here's an example of a program that uses an `if` statement to calculate and display a down payment for a house based on the buyer's creditworthiness:

```
price = 1_000_000  # The price is set to $1 million
has_good_credit = True  # Assume the buyer has good credit
down_payment = 0.2 * price if not has_good_credit else 0.1 * price  # Calculate
the down payment based on creditworthiness
```

You can use formatted strings (`f`-strings) in Python to display the down payment with a dollar sign:

```
print(f"Down payment for a buyer with good credit is ${down_payment} dollars.")
```

This will output a string that displays the down payment amount with a dollar sign and units (dollars):

```
Down payment for a buyer with good credit is $100000 dollars.
```

Sure! I can help you understand the topic of logical operators in Python. Here are some notes summarizing the important parts:

- Logical Operators in Python: We use these operators to combine multiple conditions within if statements.
  - `and`: Both conditions must be true for the statement to execute.
  - `or`: At least one condition must be true for the statement to execute.
  - `not`: Inverts a boolean value (true becomes false, and vice versa).
- Example: Consider an application processing loans. If the applicant has high income and good credit, they are eligible for a loan. This involves two conditions that must both be met.
- To combine these conditions, we use the `and` operator within an if statement:

```
if has_high_ [income] and has_good_credit:
    print("Eligible for loan")
else:
    print("Not eligible for loan")
```

- If either condition is false, the applicant will not be eligible.
- The `or` operator can also be used when we want to execute a statement if at least one of the conditions is true:

```
if has_high_income or has_good_credit:
    print("Eligible for loan")
else:
    print("Not eligible for loan")
```

- The `not` operator can be used to invert a boolean value. For example, if the applicant doesn't have a criminal record and has good credit, they are eligible for a loan. Here's an implementation of this rule:

```
if not has_criminal_record and has_good_credit:
    print("Eligible for loan")
else:
    print("Not eligible for loan")
```

- Note that the `not` operator inverts a boolean value. So if we give it a true value, it converts it to false, and vice versa.

In summary, logical operators allow us to combine multiple conditions within if statements in Python. The `and` operator requires both conditions to be true for the statement to execute, while the `or` operator requires at least one condition to be true. The `not` operator inverts a boolean value.

Python Comparison Operators

In Python, comparison operators are used to compare values or expressions with each other. They help us in making decisions based on these comparisons within the program. The common comparison operators in Python include:

```
<     less than
<=    less than or equal to
==    equal to
!=    not equal to
>   greater than
>=    greater than or equal to
```

Example Usage

```
print(5 > 3)          # True
print(5 >= 4)         # True
print("Hello" == "Goodbye")  # False
print(10 < 20)        # False
```

Weight Converter Program

The Weight Converter program is a simple example of how to convert weight between pounds and kilograms. The program prompts the user for their weight in either pounds or kilograms, converts the weight as needed, and then displays the converted weight in the desired unit. Here's an example implementation:

```
weight = float(input("Enter your weight: "))  # Input weight from user
unit = input("Is it in lbs or kgs? ")

if unit == 'lbs':
    converted_weight = weight * 0.4536
    print(f"You are {converted_weight} kilograms.")
elif unit == 'kgs':
    converted_weight = weight / 0.4536
    print(f"You are {converted_weight:.2f} pounds.")
else:
    print("Invalid input. Please enter either lbs or kgs.")
```

This program uses the `float()` function to convert the user's input from a string to a floating-point number before performing any calculations. The decision on whether the weight is in pounds or kilograms is made based on the value entered by the user, and then the appropriate conversion factor is applied to calculate the converted weight in the desired unit.

While Loops

A while loop in Python allows you to execute a block of code multiple times as long as a specific condition remains true. It is commonly used in building interactive programs and games. Here's how to create a while loop:

```
i = 1
while i <= 5:
    print(i)
    i += 1
print('done')
```

This code will print the numbers 1-5 followed by "done" on the terminal. The condition `i <= 5` is evaluated before each iteration, and as long as it remains true, the loop will continue to execute. Once `i` becomes greater than 5 (due to the increment after each print), the loop will end.

Example with String Repeating

```
while i <= 5:
    print('*' * i)
    i += 1
print('done')
```

This code will print a triangle of asterisks on the terminal. The expression `'*' * i` repeats the string with the number of asterisks equal to `i`, resulting in a pattern that resembles a triangle.

Notes:

- The loop variable (`i`) is incremented after each iteration using the statement `i += 1`.
- The condition for the while loop should be placed on a single line with no indentation.
- It's important to ensure that the loop will eventually terminate, either by changing the value of the loop variable or by introducing a break statement if necessary.

Key Takeaways:

- While loops are used to execute a block of code multiple times as long as a specific condition remains true.
- The structure of a while loop consists of a loop variable, a loop condition, and an increment statement (often called the "loop body").
- Ensure that the loop will eventually terminate or introduce a break statement if necessary to avoid infinite loops.

Additional Information: While loops can be combined with other control structures, such as if statements, to create more complex programs. For example, you could use an if statement within a while loop to perform different actions based on certain conditions.

```
while i <= 5:
    if i % 2 == 0:  # If the number is even
```

```
        print('even')
    else:  # If the number is odd
        print('odd')
    i += 1
print('done')
```

This code will print "even" for each even number between 1 and 5, and "odd" for any odd numbers. It demonstrates how you can use a while loop in combination with an if statement to perform different actions based on certain conditions.

```
Topic: Building a Guessing Game

Summary of Key Points:
1. Use a variable (`secret_number`) to store the secret number (e.g. 9).
2. Create a while loop with a condition that checks if `guess_count` is less than
`guess_limit` (e.g. 3).
3. Inside the while loop:
   - Use the `input()` function to get a user's guess as a string, then convert it
to an integer using the `int()` function.
   - Compare the user's guess with `secret_number` using an if statement.
     - If they guessed correctly (i.e. `guess == secret_number`), print "You win!"
and use a break statement to exit the loop.
   - If the user's guess was incorrect:
     - Increment `guess_count` by 1.
     - Print "Sorry, you failed." if `guess_count` is equal to `guess_limit` (i.e.
they ran out of tries).
   4. Use meaningful and descriptive variable names to improve code readability.
   5. Include an else statement at the end of a while loop to execute some code
when the loop completes without breaking.

Notes:
- Avoid using "less than or equal to" operator (i.e. `<`) in your condition if you
only want the loop to run a set number of times. Use "less than" operator (i.e.
`<`) instead.
- Make sure to convert user input from string to integer before comparing it with
the secret number.
```

```
# Example Example Example Example Example Example Example Example Example Example
Example Example Example Example Example Example Example Example Example Example
Example Example Example Example Example Example Example Example Example Example
Example

In Python, you have a `while` loop that reads commands from standard input and
responds accordingly based on the command. If the user types "help", the program
lists the available commands: "start" to start the car, "stop" to stop the car,
and "quit" to quit the program. If any other command is entered, it tells the user
that it doesn't understand what they are saying.

To enhance this program as requested, you need to store whether or not the car is
```

```
currently started in a variable. This can be done by adding a Boolean variable
called `started` at the beginning of your script and initializing it with a value
of `False`:
```python
started = False
```

Then, when the user enters "start", you need to check if the car is already started before starting it again. If it's not started, set `started` to True and respond accordingly. Otherwise, print an error message:

```
if command == 'start':
    if started:
        print("The car is already running! What are you doing?")
    else:
        started = True
        print("Car is started.")
```

Similarly, when the user enters "stop", check if the car is currently stopped before stopping it again. If it's not currently running, print an error message; otherwise, set `started` to False and respond accordingly:

```
if command == 'stop':
    if started:
        started = False
        print("Car is stopped.")
    else:
        print("The car is already stopped! What are you doing?")
```

With these modifications, your program will provide the expected responses when a user attempts to start or stop the car multiple times.

Python for Loops

A Python for loop is used to iterate over a sequence (like a string, list, or tuple) and execute a block of code once for each item in the sequence. Here's how you can use a for loop in Python:

```
# Example 1 - Iterating Over Strings
for char in "Python":
    print(char)

# Output:
# P
# y
# t
# h
# o
# n
```
```

```
# Example 2 - Iterating Over Lists
prices = [10, 20, 30]
total = 0
for price in prices:
    total += price
print("Total:", total)

# Output:
# Total: 60
```

In the first example, we iterate over each character in a string and print it. In the second example, we define a list of prices and use a for loop to calculate the total cost by adding up each price.

You can also use the built-in range function to create a range of numbers that you can iterate over:

```
# Example 3 - Iterating Over Ranges
for num in range(1, 10):
    print(num)

# Output:
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

In this example, we create a range of numbers from 1 to 10 (inclusive) and iterate over each number in the range.

Remember that when you use a for loop to iterate over an object, Python automatically creates a new variable in each iteration to hold the current item in the sequence. You can give this variable any name you want (like "item" or "price"), but it's usually best to choose a name that clearly indicates what the variable represents.

And that's all there is to using for loops in Python!

Nested loops are a way of executing one loop inside another loop. They can be used for various purposes such as generating lists of coordinates or creating shapes using strings of characters. Here's how nested loops can be used to solve these types of problems:

```
# Generate list of numbers
numbers = [5, 2, 5, 2, 5]

# Use nested for loop to generate the shape
for item in numbers:
```

```
    output = ""
    for count in range(item):
        output += "x"
    print(output)
```

In this example, we first iterate over each item in the `numbers` list. In each iteration, we then use an inner loop to generate a string of characters that represents a row of the shape. We do this by setting an output variable to an empty string and then using another loop to append the appropriate number of 'x' characters based on the value of `item` in the outer loop. After the inner loop has finished executing, we print out the resulting string of characters that represents that row of the shape.

By alternating between these two loops (the outer for loop and the inner for loop), we can generate a complex structure like a shape using only strings of characters.

Lists in Python:

- Define a list of names (John, Bob, Mosh, Sarah, Mary).

```
names = ["John", "Bob", "Mosh", "Sarah", "Mary"]
```

- Print the list using square brackets and comma separation between elements.

```
print(names)
```

- Access individual elements by index (first element is 0th index, second element is 1st index, etc.).

```
print(names[2]) # prints "Mosh"
print(names[-1]) # prints "Mary"
```

- Use negative indices to access elements from the end of the list.
- Access a range of elements using square brackets and colon, with optional start and/or end index values.

```
print(names[2:5]) # prints ["Mosh", "Sarah", "Mary"]
print(names[:4]) # prints ["John", "Bob", "Mosh", "Sarah"]
print(names[-3:]) # prints ["Sarah", "Mary"]
```

- Set default values for start and end indices if omitted.

```
print(names[:2]) # prints ["John", "Bob"]
```

- Modify elements in the list by indexing them with square brackets.

```
names[0] = "Data"
print(names) # prints ["Data", "Bob", "Mosh", "Sarah", "Mary"]
```

- Find the largest number in a list of numbers using a for loop to iterate through the list and compare each element with the current maximum.

```
numbers = [3, 6, 2, 8, 4, 10]
max_number = numbers[0]
for number in numbers:
  if number > max_number:
    max_number = number
print(max_number) # prints 10
```

  Note that the largest number will be at any index within the list, not just the end or beginning.

2D Lists In Python:

- Two-dimensional lists (2DL) are commonly referred to as matrices in mathematics and programming languages.
- A 2DL is a list where each item in the list is another list. This structure allows for easy representation of data with multiple rows and columns.
- The syntax for defining a 2DL in Python is to use square brackets within square brackets, where each inner bracket represents a row, and items within each inner bracket are values in that particular row.

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

- To access an individual item in a 2DL, use square brackets within the outer square bracket. The indices of both the row and column are used to reference the specific value. For example:

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

# Accessing item (row=0, column=0)
item = matrix[0][0]
print(item)  # Output: 1
```

- To modify an individual item in a 2DL, use the same square bracket notation as for accessing items. Update the value at that specific index position. For example:

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

# Modifying item (row=0, column=0) to 20
matrix[0][0] = 20
print(matrix)  # Output: [[20, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- To iterate over all the items in a 2DL, use nested loops. One loop iterates over the rows (outer list) and another loop iterates over the columns (inner lists) within each row. For example:

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

for row in matrix:  # Iterate over rows
    for item in row:  # Iterate over items within each row
        print(item)
```

- 2DLs are commonly used in data science and machine learning applications for representing complex datasets with multiple attributes. They can be easily manipulated and processed using various algorithms.

Welcome to my Python tutorial! In this video, we're going to cover tuples in Python. Tuples are similar to lists because they both store multiple items, but unlike lists, tuples are immutable. This means that once you create a tuple, its values cannot be changed. Let me show you an example of how to create a tuple.

```
numbers = (1, 2, 3)
print(numbers)
```

The output will be `(1, 2, 3)`, indicating that we have created a tuple with three values: 1, 2, and 3. To access individual items within the tuple, you can use square brackets just like you would for lists. For example, to get the first item in our tuple, we can write `numbers[0]`, which will return `1`.

One important thing to note about tuples is that they are not mutable. This means that once a tuple has been created, its values cannot be changed. If you try to assign a new value to an item in the tuple, Python will raise a TypeError because it's expecting an immutable object like a tuple. Here's an example:

```
numbers = (1, 2, 3)
numbers[0] = 5
```

This code will raise a `TypeError` because tuples are not supposed to be modified once they have been created.

Another thing to note about tuples is that they can be used in place of lists when you don't need to modify the values stored within them. This can make your code more efficient, as it avoids creating new objects every time you want to add or remove items from a list. For example, if we wanted to create a set of all even numbers between 1 and 20 using tuples, we could do something like this:

```
even_numbers = tuple(x for x in range(1, 21) if x % 2 == 0)
print(even_numbers)
```

This code will create a tuple of all even numbers between 1 and 20. The output would be `(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)`.

In summary, tuples are similar to lists in that they can store multiple items, but unlike lists, they are immutable and cannot be modified once they have been created. This makes them useful for situations where you don't need to modify the values stored within a data structure. If you want to learn more about Python, I recommend checking out my course on Udemy. Thanks for watching!

Notes on Dictionaries in Python:

- **Dictionaries**: A dictionary is a data structure that stores information as key-value pairs. It's similar to a real-world dictionary where you have words (keys) and their definitions (values).

```
# Example of a customer with attributes like name, email, phone number, address,
etc.
customer = {
    "name": "John Smith",
    "email": "john@example.com",
    "phone": "123-456-7890"
}
```

- **Keys**: Keys are used to access the values stored in a dictionary. Keys can be any immutable type such as strings, numbers or tuples.

```
# Accessing values using keys
print(customer["name"]))  # Output: "John Smith"
```

- **Values**: Values can be of any data type. To update a value in the dictionary, you just assign a new value to its key.

```
# Updating values in the dictionary
customer["phone"] = "+123-456-7890"
print(customer)  # Output: {'name': 'John Smith', 'email': 'john@example.com',
'phone': '+123-456-7890'}
```

- **Accessing Values**: You can access values in a dictionary using keys in square brackets (`[]`) or by calling the `get` method with a key and optionally a default value.

```
# Accessing values using keys in square brackets
print(customer["email"])  # Output: 'john@example.com'
```

- **Adding New Key-Value Pairs**: You can add new key-value pairs to a dictionary either by assigning a value directly to the key or by using the `update` method.

```
# Adding new key-value pair
customer["address"] = "123 Example St, Anytown"
print(customer)  # Output: {'name': 'John Smith', 'email': 'john@example.com',
'phone': '+123-456-7890', 'address': '123 Example St, Anytown'}
```

- **Deleting Key-Value Pairs**: You can delete a key-value pair from a dictionary either by using the `del` statement or by calling the `pop` method.

```
# Deleting key-value pairs
del customer["email"]
print(customer)  # Output: {'name': 'John Smith', 'phone': '+123-456-7890',
'address': '123 Example St, Anytown'}
```

- **Exercise**: Create a dictionary to translate numbers from 1 to 10 to words.

```
# Dictionary for translating numbers from 1 to 10 to words
digits_mapping = {
    "1": "one",
    "2": "two",
    "3": "three",
    "4": "four",
    "5": "five",
    "6": "six",
    "7": "seven",
    "8": "eight",
    "9": "nine",
```

```
    "10": "ten"
}
```

- **Example Usage**: Ask the user to enter a phone number and then use the dictionary to translate it into words.

```
# Get the users phone number
phone_number = input("Enter your phone number: ")

# Translate the digits to words
output = ""
for digit in phone_number:
    output += digits_mapping[digit] + " "
print(output.strip())  # Output: one two three four five six seven eight nine ten
```

Topic: Emoji Converter using Dictionaries in Python

Summary:

- The program converts typed messages into emojis based on a dictionary of mapped words and smiley faces.
- It uses the input function to prompt for a message, then splits it into individual words by spaces.
- A dictionary is created with keys representing special characters or words and values corresponding to smiley faces as strings.
- The program checks each word in the list against the dictionary using the `get` method. If found, the value (smiley face) is added to an output string; if not, the original word is used.
- Finally, the output string is printed with spaces between words and the smiley face at the end.

Code Excerpt:

```
words = message.split()  # Split input message by spaces into individual words
output = ""
for word in words:
    if word in emojis:  # Check if the current word is a key in the dictionary
        output += emojis[word]  # Add value (smiley face) to output string if
found
    else:
        output += word  # Otherwise, add original word to output string

print(output)
```

Functions are essential in programming as they allow for the reuse of code across different applications. They also help organize large programs into smaller, more manageable pieces of code, making it easier to maintain and debug. In Python, a function is defined using the `def` keyword followed by the function name and a block of code enclosed within curly braces (or indentation in the case of Python). When the function is called, the code inside the block is executed. Functions can accept arguments or inputs which allow for more flexibility

and customization of the behavior of the function. It's important to always end your function definition with a blank line as it is one of the PEP8 style guidelines for formatting Python code. Functions help in reducing code duplication, making it easier to maintain and debug large programs.

- Difference between calling a function with parameters and the `print` function in Python: The `print` function takes information, but our greet function doesn't. Parameters allow functions to receive information and act like local variables that can be used within the function.
- Adding parameters to a function definition by placing them inside parentheses after the function name.
- Passing values to function parameters when calling the function. These values become local variables in the context of the function.
- Using formatted strings (`{}`) to insert the value of a variable into a string.
- Multiple arguments can be passed to functions with multiple parameters, separated by commas.
- Parameters are placeholders that define where information can be supplied when calling a function; arguments are the actual pieces of information provided when calling a function.
- Defining functions allows for code reuse and reduces the need for repetition.

```
def greet_ print("Hi there, nice to meet you!")
```

```
greet print("Hi {name}, welcome aboard!".format(name=user)) # Using a formatted
string
print("Hi {first name} {last name}, welcome
aboard!".format(first_name=user["name"], last_name=user["last_name"]))
```

```
def greet_user(first_name, last_name): # Defining a function with multiple
parameters
    print(f"Hi {first_name} {last_name}, welcome aboard!")

greet_user("John", "Smith")
```

- Keyword Arguments: When defining parameters for Python functions, it is important to always supply values or an error will occur.

  Positional Arguments: Positions matter and changing the order of arguments can lead to different results.

  Keyword Arguments: Positions don't matter because the combination of a parameter name followed by its value creates a keyword argument, improving code readability.

  Ordering Rules: Keyword arguments should always come after positional arguments. Mixing both types of arguments requires following specific order rules to prevent errors.

- Example: A function for calculating the total cost of an order (total, shipping, discount) can improve readability by using keyword arguments, which helps clarify what each value represents.

Return Statement:

- Allows a function to return a value to its caller.
- Useful for returning results of calculations.
- In Python, the default return value is `None` if no `return` statement is provided.
- To improve readability and clarity, use keyword arguments over positional ones.

Returning Values:

```
def square(number):
    result = number * number
    return result
```

Example Function (`square()`) that returns the square of a given `number`.

Calling Functions with Return Statements:

```
square_of_three = square(3)  # Call function and store result in variable
print(square_of_three)      # Print result
```

Example where function is called and its result stored in a variable, then printed.

Using `return` in Functions without Positional Arguments:

```
def print_hello():
    print("Hello World!")
    return  # Add return statement to end of function

print_hello()              # Call function and don't store result
```

Example where a function is called, but its result isn't stored in a variable.

Using `return` with Keyword Arguments:

```
def square(number, power=2):
    result = number ** power
    return result
```

Function (`square()`) that allows for both positional and keyword arguments.

By using `return` statements in functions, you can provide meaningful values to the caller, improving code readability and functionality. Remember that if a function does not have a return statement, it will return `None` by default in Python.

```
# Reusable Function Returning None
```

````
Create a reusable Python function that can be used in multiple applications. The
function should take a string as input and convert it into emoji format. This code
is from an emoji converter program and the task is to extract the main
functionality into a separate function.

```python
# Original Code: Emoji Converter Program
def emoji_converter(message):
    # Function definition for converting text to emoji format
    # Input: message (string)
    # Output: output (string)
    output = ""
    for char in message:
        if char == ":)":
            output += "😄"
        elif char == ":D":
            output += "🤣"
        elif char == ":(":
            output += "😦"
        else:
            output += char
    return output

# Main Program
message = input()  # Receive input from the terminal
result = emoji_converter(message)  # Call the function with the message as input
print(result)  # Print the resulting string in emoji format
````

The extracted reusable function is named `emoji_converter`, which clearly explains its purpose. It takes a single parameter, `message`, which is of type string. The function converts all occurrences of specific smiley faces ("😃", "😄", and "😮") into corresponding emoji characters using a for loop.

```python
def emoji_converter(message):
    output = ""
    for char in message:
        if char == ":)":
            output += "😄"
        elif char == ":D":
            output += "🤣"
        elif char == ":(":
            output += "😦"
        else:
            output += char
    return output
```

The `emoji_converter` function is called with the input message and returns a string in emoji format. This string is then stored in the variable `result` and finally printed using the `print()` statement.

Key points to note:
```

1. The main functionality of converting smiley faces into emojis has been extracted into a separate function, making it reusable for different applications.
2. The input is not included within the function as it can come in various forms and should be handled separately by each application.
3. The output handling (printing) is also kept outside the function since the final use of the output may vary between different applications.
4. The code follows Python coding conventions with proper indentation, variable naming, and spacing for readability.

```
```

Exceptions emoji converter:

- Function takes parameter called message.
- Returns the output.

Handling errors in Python programs:

1. Write program to get user age from terminal.
2. Use input variable and label like "age", then store result in a variable named "age".
3. Print "age" to check implementation.
4. Check exit code of the program for success (0) or crash (1).

Error handling:

- Use try/except construct.
- Type "try" and define code block inside it, including input variable and age calculation.
- Add an "except" clause with the type of error you may encounter, such as "ValueError".
- Define what should happen if program encounters a specific type of error within the except block.
- Print proper error message (e.g., "Invalid value").

Different kinds of exceptions:

1. Add another "except" clause for different types of errors or exceptions.
2. In this case, use "except ZeroDivisionError" to handle 0 division errors.
3. Print a different error message (e.g., "Age cannot be 0") within the except block.
4. Check exit code again; it should still be 0 if program didn't crash.

Example code:

```
try:
    age = int(input("Enter your age: "))
    income = 20000 / age
except ValueError:
    print("Invalid value")
except ZeroDivisionError:
    print("Age cannot be 0")
```

Comments are used in Python programming language to add notes or explanations to the program's source code, which are not executed and hence ignored by the compiler during execution. They help in explaining complex parts of a code, providing context, adding reminders for future updates, and communicating with other developers reading the code. However, it is important to avoid using comments that explain what the code does because they can become outdated as changes are made to the code over time. It's also advisable not to use comments in front of functions or methods if their purpose is obvious from their names. Instead, use comments to explain why certain assumptions have been made and how to improve the code.

Classes in Python

In Python, a class is used to create a blueprint for objects. It is similar to creating new data types. Classes are important because they allow us to model complex concepts that cannot be represented by basic built-in types like numbers and strings.

To define a class in Python, we use the `class` keyword followed by the name of the class. The naming convention for classes is to capitalize the first letter of each word. For example, if we were defining a class called "Point", it would be written as `class Point:`.

Inside the class definition, we can define methods and attributes for our objects. Methods are functions that operate on instances of the class, while attributes are variables that belong to an instance of the class.

For example, consider a simple class called `Point` with two attributes, `x` and `y`:

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

We can then create instances of the `Point` class by calling it like a function:

```
p1 = Point(3, 4)
p2 = Point(-5, 8)
```

These are instances or objects of our `Point` class. We can access their attributes using dot notation:

```
print(p1.x)  # Output: 3
print(p1.y)  # Output: 4
```

We can also define methods for our objects that operate on the instance data. For example, we could define a `draw` method in our `Point` class like this:

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

```
    def draw(self):
        print(f"Drawing point at ({self.x}, {self.y})")
```

We can then call this method on an instance of the `Point` class:

```
p1 = Point(3, 4)
p1.draw()  # Output: Drawing point at (3, 4)
```

In summary:

- A class is used to define a blueprint for objects in Python.
- Classes allow us to model complex concepts that cannot be represented by basic built-in types.
- To define a class, we use the `class` keyword followed by the name of the class and a colon.
- Inside the class definition, we can define methods and attributes for our objects using the `self` keyword.
- Instances of a class are created by calling the class like a function with optional arguments for its attributes.
- We can access instance data using dot notation and call methods on an instance using the same notation.

Constructors and Attributes in Python Objects

- Problem with initializing attributes: A point object may not have x or y coordinates set.
- Solution: Use a constructor to initialize these attributes when creating an object.

```
class Point:
    def __init__(self, x=0, y=0):  # Constructor function for the class
        self.x = x
        self.y = y

# Create a point object with initial values set in constructor
p1 = Point(3, 4)
print(p1.x)  # Output: 3
print(p1.y)  # Output: 4
```

Example of Person Class with Name Attribute and Talk Method

```
class Person:
    def __init__(self, name=None):  # Constructor function for the class
        self.name = name

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

```
p1 = Person()
p2 = Person("Jon Smith")

# Print message using the Talk method
p1.talk()  # Output: Hi, I am None.
p2.talk()  # Output: Hi, I am Jon Smith.
```

Inheritance (Python)

- In Python, each object is an instance of a class.
- Inheritance is a mechanism for using code that has already been written and tested.
- It allows for the creation of new classes that are built on top of existing ones.
- This can help to avoid duplication of code and maintain consistency across similar objects.
- To use inheritance, you define a new class called the parent class or base class, which contains the methods that will be inherited by other classes.
- You then define one or more subclasses (or child classes) that inherit from the parent class.
- The subclasses can add their own unique methods and attributes without duplicating code.
- When you create an instance of a subclass, it automatically inherits all the attributes and methods defined in its parent class(es).
- Inheritance is especially useful when working with classes that have common features or behaviors.
- It allows for greater flexibility and maintainability in your codebase.

The image shows a code editor with Python script open, where a user has created a function named `find_max` to find the largest number in a list. The function is defined within a module called "utils" which is imported into the main script file. This organization follows good coding practices by separating utility functions from the main application logic.

In addition to defining and calling the `find_max` function, there's also an example of how not to do it: using built-in names as variables. In this case, the variable name "max" is used, which shadows the built-in `max` function. This can lead to confusion and bugs, especially when trying to use the original `max` function within the same scope.

The image also includes a warning message from PyCharm IDE indicating that using built-in names as variables can be dangerous because it may cause unexpected behavior or hide existing features of the language. The suggestion is made to avoid overriding these names if possible, and to choose more descriptive variable names that do not conflict with Python's built-ins.

Overall, this image illustrates how important it is to follow good coding practices when working with programming languages like Python, especially in terms of naming variables and avoiding conflicts with built-in functions or modules.

Python Packages

- Organize related modules into directories (packages)
  - A package is essentially a directory with `__init__.py` file, which identifies the directory as a package
- Importing from a package:
  - Import entire module using the package name and dot notation (`import ecommerce.shipping`)
  - Import specific function or class from a module within a package

```
from ecommerce.shipping import calculate_shipping
```

- Import all functions/classes in a module by importing the module itself (useful when working with multiple functions)

```
from ecommerce.shipping import *
```

Python Package Structure Example

```
e-commerce/
  __init__.py
  sales.py
  customer_service.py
```

Importing from a Package in Python

Import entire module:

```
from ecommerce import sales
```

Import specific function or class:

```
from ecommerce.customer_service import handle_ticket
```

Import all functions/classes within the module:

```
from ecommerce.sales import *
```

Python `__init__.py` File

- Identifies a directory as a package
- Allows access to modules and their contents from other packages
- Can contain initialization code for the package

Python Import Syntax

```
import <module_name>
from <package_name> import <module_name>
from <package_name>.<module_name> import *  # Import all functions/classes in a
module
from <package_name>.<module_name> import <function_or_class>
```

Python Import vs. From Import

- `import ecommerce` loads the entire package into memory
- `from ecommerce import shipping` imports only specific parts of the package into memory, which is more efficient for large packages with many modules

Python Package Usage Example: Django Framework

- Django framework uses packages to organize its components

```
mysite/
  __init__.py
  settings.py
  urls.py
  wsgi.py
  manage.py
myproject/
  __init__.py
  models.py
  views.py
  forms.py
  tests.py
  migrations/
    __init__.py
```

- Each Django application is a separate Python package, making it easier to manage and organize the codebase.

Python has a standard library that contains several modules for common tasks such as sending emails, working with date and time, generating random values and passwords. The built-in `random` module provides methods for generating random numbers. One method is the `randomint()` method, which generates an integer in a specified range. Another method is `randint()` , which also generates an integer in a specified range but requires two arguments to specify the range. Additionally, there are methods such as `choice()` and `shuffle()` that can be used for randomly picking an item from a list or shuffling the items in a list, respectively. The `random` module is very powerful and has many applications.

Working with Directories Using Python's `pathlib` Module

Python provides an object-oriented file system module called `pathlib` that allows us to work with directories and files. The following are some best practices when working with this module in PyCharm:

1. Import the `Path` class from the `pathlib` module, as shown below:

```
from pathlib import Path
```

2. Create a `Path` object to reference a file or directory on your computer. There are two ways to do this:
    - Absolute paths (starting from the root of your hard disk)
    - Relative paths (starting from the current directory)

3. Use methods provided by the `Path` class to interact with the files and directories:

```
# Check if a path exists
print(path.exists())

# Create a new directory
path.mkdir()

# Delete an existing directory (if empty)
path.rmdir()

# Get all files/directories in a given path
for item in path.glob('*'):
    print(item)
```

4. Examples:
    - To check if a path exists, use the `exists()` method and print its result.
    - To create a new directory, call the `mkdir()` method on your `Path` object.
    - To delete an existing directory (if it's empty), call the `rmdir()` method on your `Path` object.
    - To get all files/directories in a given path, use the `glob()` method with a search pattern and iterate over the results using a `for` loop.

```
# Example usage of the methods
path = Path('path_to/directory')  # Replace 'path_to' with an absolute or relative
path to your directory
print(path.exists())
if not path.is_dir():
    path.mkdir()
else:
    for item in path.glob('*'):
        print(item)
```

5. Important key words and concepts related to working with directories using `pathlib`:
    - `Path` class: an object representing a file or directory path
    - `exists()` method: checks if a path exists on the filesystem
    - `mkdir()` method: creates a new directory at the given path
    - `rmdir()` method: removes (deletes) an existing empty directory

    - `glob()` method: searches for files/directories matching a specified pattern and returns them as `Path` objects.

```
# Example usage of methods
path = Path('path_to/directory')  # Replace 'path_to' with an absolute or relative path to your directory
print(f"{path} exists: {path.exists()}")
if not path.is_dir():
    path.mkdir()
else:
    for item in path.glob('*'):
        print(item)
```

Pypi and Pip e-commerce directory Notes

- Python's standard library provides comprehensive modules for common tasks but is not exhaustive.
- The Python Package Index (PyPi) contains hundreds of packages created by community developers, published for reuse in programs.
- PyPi offers packages for various purposes, including sending SMS messages and accessing Yelp information.
- Not all packages are complete or bug-free; some may still be in development.
- To search and access PyPi packages:
    1. Go to pypi.org (e.g., search for "sms").
- Many Python developers contribute their projects to PyPi.
- The Python community maintains a vast array of reliable, well-documented packages.
- A complete Python course can cover some of the most useful PyPi packages, such as those used to access Yelp information and automate web applications' testing.
- To install packages from PyPi:
    1. Go to pypi.org (e.g., search for "openpyxl").
    2. Locate the package in question.
    3. Note its version number.
- Use pip command to install the package on your system.

Installation Instructions:

```
pip install <package_name>
```

Notes on Importing Packages:

1. Python imports modules and packages by default, from the current directory or standard library path.
2. To import a package:
    - Use its name (e.g., `openpyxl` in the case of `openpyxl`).
3. If you have multiple versions of the same module installed, use the `importlib` module to load the desired version.

Additional Resources:

- PyPi's website: https://pypi.org/
- Python's documentation on packages and modules: https://docs.python.org/3/tutorial/modules.html
- Python Packaging User Guide: https://packaging.python.org/en/latest/user/introduction.html

Note: This summary is based on the provided information, focusing on key points related to PyPi and Pip installation. It does not include specific details about web scraping or other Python-related tasks. The content of notes.txt is not available for this summary.

It seems like there might be some missing steps or context here. Could you provide more information about what you are trying to achieve with your Python code? It looks like you're working on updating values in a spreadsheet and adding charts, but the code snippet is incomplete and there may be an issue with the way you are saving the workbook. Additionally, it would be helpful if you could share the specific error messages or issues that have arisen when running your code. This way I can provide more targeted assistance.

Hello! It looks like you're ready to start a project using Jupiter Notebook for machine learning. The image shows an empty Jupiter notebook with only one cell active, which is ready for writing code or inserting text. The interface elements and keyboard shortcuts mentioned are typical of the Jupiter environment, where users can execute Python code and display results in real-time.

In this particular scenario, you're setting up a project to recommend music albums based on user profiles, which include age and gender. This is a common use case for collaborative filtering techniques, where the model learns from the patterns of similar users (collaboration) to make predictions about new users.

To get started, you would usually begin by importing libraries, loading your dataset, cleaning and preprocessing data, creating features, splitting the data into training and testing sets, and then fitting a machine learning model using an algorithm like Random Forest or K-Means Clustering (depending on whether it's a classification or clustering problem).

Once you have trained your model, you would evaluate its performance and fine-tune as needed. Finally, you would use the model to make recommendations for new users by providing their age and gender information.

Remember that Jupiter Notebook is an interactive environment where you can write code, analyze data, and create visualizations all in one place. It's a powerful tool for data scientists and programmers alike. Good luck with your project!

In this image, you're shown the process of visualizing a decision tree from a machine learning model using Python and the scikit-learn library. Here are the steps taken in the image:

1. Importing the necessary libraries: `import pandas as pd` for data manipulation, `import numpy as np` for numerical computations, `from sklearn import tree` to access decision trees from scikit-learn, and `import os` to work with file paths if needed.

2. Loading a dataset into a Pandas DataFrame: The `df = pd.read_csv('music-recommender.csv')` line suggests that the data is stored in a CSV file named "music-recommender.csv", which will be loaded into a DataFrame called 'df' for processing.

3. Splitting the dataset: The `X = df.iloc[:, :-1]` and `y = df.iloc[:, -1:]` lines split the DataFrame into input features (X) and the target variable (y). This is a common pattern in machine learning where

the last column of the data often contains labels or targets for classification problems like predicting music genres.

4. Training the decision tree model: The `model = tree.DecisionTreeClassifier()` line creates an instance of the DecisionTreeClassifier, which will be used to build a decision tree based on the input data X and output label y.

5. Visualizing the decision tree: The `tree.export_graphviz(model, out_file='music-recommender.dot', feature_names=X.columns, class_names=y.unique(), filled=True, rounded=True, special_characters=True)` line exports the model in GraphViz format (dot file), which can be visualized to understand how the decision tree works.

6. Opening and viewing the dot file: The image shows a graphical representation of the decision tree in Visual Studio Code using an extension for graph visualization. This is where we can see that based on certain conditions like age and gender, different music genres are recommended. For example, if someone is young (age < 25.5) and female (gender == 0.5), they might be recommended hip hop music.

The image also shows the command `!dot -Tpng music-recommender.dot -o music-recommender.png` in a terminal window, which suggests that this command is being used to convert the dot file into an image format (PNG) that can be easily shared or viewed without requiring the GraphViz software to be installed on the machine.

This visualization is a valuable tool for understanding how machine learning models make decisions and can help with model interpretability, debugging, and feature selection.

The content provided appears to be discussing the topic of building a website using Django, which is a Python web framework. The text mentions class names for genres and feature names for age and gender, suggesting the use of machine learning algorithms in creating a personalized experience for users of the website. There's also a mention of creating a project called "pyshop" with Django.

Based on this information, here are some notes summarizing key points:

- Project goal: Build a personalized website using Django.
- Genre and age/gender features to be used for personalization.
- Using machine learning algorithms for user experience.
- Creating a project called "pyshop" with Python and Django.

Note that this summary does not include any detailed or technical information about the actual implementation steps, such as setting up a development environment, installing dependencies, or writing code in Python or Django. These are topics typically covered in more depth in tutorials or documentation related to Django web development.