Python Basic Commands Structure Explained

# 1. Variables and Data Types

Variables store data that can be used later in the code.

# Integer  
age = 25  
# Explanation: `age` is a variable that stores an integer value `25`. The `=` operator assigns the value to the variable. Integers are whole numbers without a decimal point.

# Float  
height = 5.9  
# Explanation: `height` is a variable that stores a floating-point number `5.9`. Floats are numbers with a decimal point.

# String  
name = "John Doe"  
# Explanation: `name` is a variable that stores a string `"John Doe"`. Strings are sequences of characters enclosed in quotes.

# Boolean  
is\_student = True  
# Explanation: `is\_student` is a variable that stores a Boolean value `True`. Booleans represent truth values: `True` or `False`.

# List (Array)  
fruits = ["apple", "banana", "cherry"]  
# Explanation: `fruits` is a variable that stores a list of strings. Lists are ordered collections of items and can contain any data type.

# Dictionary (Map)  
person = {  
 "name": "Alice",  
 "age": 30,  
 "city": "New York"  
}  
# Explanation: `person` is a dictionary, which stores key-value pairs. Dictionaries allow you to associate values (e.g., `Alice`) with keys (e.g., `"name"`).

# 2. Conditional Statements

Conditional statements allow you to make decisions in your code.

if age > 18:  
 print("You are an adult.")  
# Explanation: The `if` statement checks whether `age` is greater than `18`. If the condition is `True`, the code block inside the `if` statement is executed, printing `"You are an adult."`.

elif age == 18:  
 print("You just turned an adult!")  
# Explanation: The `elif` (else if) statement checks another condition if the previous `if` condition was `False`. Here, it checks if `age` is exactly `18`.

else:  
 print("You are a minor.")  
# Explanation: The `else` statement runs if none of the previous conditions are `True`. It acts as a fallback, printing `"You are a minor."`.

# 3. Loops

Loops allow you to repeat a block of code multiple times.

# For Loop (iterates over a sequence)  
for fruit in fruits:  
 print(fruit)  
# Explanation: A `for` loop iterates over each item in the `fruits` list. The variable `fruit` takes the value of each item in the list on each iteration, and `print(fruit)` outputs the current `fruit`.

# While Loop (repeats as long as a condition is true)  
count = 0  
while count < 5:  
 print("Counting...", count)  
 count += 1  
# Explanation: A `while` loop runs as long as the condition `count < 5` is `True`. Inside the loop, it prints the current value of `count` and then increments `count` by `1` using `count += 1`.

# 4. Functions

Functions allow you to group code into reusable blocks.

def greet(name):  
 """  
 This function greets a person with their name.  
 """  
 return f"Hello, {name}!"  
# Explanation: The `def` keyword defines a function named `greet` that takes a parameter `name`. The triple quotes `"""` define a docstring, which is a comment describing what the function does. The `return` statement returns the greeting as a formatted string `f"Hello, {name}!"`.

# Using the function  
print(greet("Alice"))  
# Explanation: This line calls the `greet` function with the argument `"Alice"` and prints the returned greeting.

# 5. Input/Output

Getting input from the user and displaying output.

# Getting user input  
user\_name = input("Enter your name: ")  
# Explanation: The `input` function prompts the user to enter a value, which is then stored in the `user\_name` variable.

# Displaying output  
print(f"Welcome, {user\_name}!")  
# Explanation: The `print` function displays a formatted string that includes the user's name.

# 6. Exception Handling

Handling errors that might occur during code execution.

try:  
 number = int(input("Enter a number: "))  
 print(f"You entered: {number}")  
except ValueError:  
 print("That's not a valid number!")  
# Explanation: The `try` block contains code that might cause an error (e.g., converting user input to an integer). If an error occurs, the `except` block catches the `ValueError` and prints an error message. This prevents the program from crashing.

# 7. Classes and Objects

Object-oriented programming allows you to model real-world entities.

class Dog:  
 def \_\_init\_\_(self, name, breed):  
 self.name = name  
 self.breed = breed  
# Explanation: The `class` keyword defines a class named `Dog`. The `\_\_init\_\_` method is a special method called a constructor, which initializes the object's properties (`name` and `breed`). `self` refers to the instance of the class.

def bark(self):  
 return f"{self.name} says Woof!"  
# Explanation: The `bark` method defines a behavior of the `Dog` class, returning a string that simulates the dog barking.

# Creating an object of the Dog class  
my\_dog = Dog("Buddy", "Golden Retriever")  
# Explanation: This line creates an instance (object) of the `Dog` class, with the name `"Buddy"` and breed `"Golden Retriever"`, and stores it in the `my\_dog` variable.

# Using the object's method  
print(my\_dog.bark())  
# Explanation: This line calls the `bark` method on the `my\_dog` object and prints the result.

Please do this Survey Below (5min)  
https://forms.gle/8AJcCeXm7CdQgEfLA