Python Fundamentals

Print Statement

The print statement displays a message

```
print("Helo World!")
```

The basic usage involves passing one or more arguments to print. These arguments are separated by commas and are concatenated with spaces in the output.

Using Format Strings

You can use format strings to control the formatting of the output. Format strings are specified using curly braces {} as placeholders for values. The format function is then used to replace these placeholders with actual values.

```
print('Name: {}, Age: {}'.format("Python", "20"))
```

F-Strings

Python introduced F-strings as a convenient way to format strings. You can directly embed expressions and variables within the string using curly braces preceded by an 'f' character.

```
name = 'Bob'
age = 25
print(f'Name: {name}, Age: {age}')
```

Formatting Options

You can specify formatting options within the placeholders to control how values are displayed. For example:

```
pi = 3.14159
print(f'Value of pi: {pi:.2f}')
```

n this case, :.2f specifies that the pi value should be formatted as a floating-point number with two decimal places.

Escape Sequences

You can use escape sequences to include special characters in your output. For example:

```
print('This is a new line\nThis is on the next line')
print('This is a tab\tThis is after the tab')
```

Printing Multiple Lines

To print multiple lines, you can use triple-quoted strings (single or double quotes) to preserve line breaks:

```
multiline_text = '''This is line 1.
This is line 2.
This is line 3.'''
print(multiline_text)
```

Comments

Comments in Python are used to add explanations or notes within your code that are not executed by the Python interpreter. They are essential for making your code more understandable to both yourself and others who may read your code. Here are examples of single-line and multi-line comments along with explanations:

```
# This is a single-line comment
# Single-line comments begin with a hash symbol (#)
# They are used to add explanations or notes on a single line
# Anything after the hash symbol on the same line is considered a
comment

print("Hello, world!")  # This is a comment

# Example of a single-line comment within code

x = 5  # Initialize x with the value 5



# Multi-line comments are often used for more extensive explanations
or documentation.
# You can use triple-quotes (single or double) to create multi-line
comments.
# Anything enclosed between the triple-quotes is considered a comment.

"""
```

```
This is a multi-line comment.
It spans across multiple lines.
Useful for explaining complex code sections.
"""

'''
This is another way to create a multi-line comment.
You can use single or double triple-quotes.
'''


# Multi-line comments can also be used to temporarily "comment out"
blocks of code
# by enclosing the code within triple-quotes.
# This is useful for debugging or testing different code sections.

"""
print("This line is commented out")
x = 10  # This line is also commented out
"""

# Comments are not executed by Python and do not affect the program's
behavior.
# They are solely for human readability and should be used to make
your code
# easier to understand and maintain.
```

Variables and DataTypes

Variables and data types are fundamental concepts in programming. They are used to store and manipulate data in a program. Let's break down what variables are and explore the different data types in Python:

Variables

In Python, a variable is like a container or a named storage location used to store data values. You can think of a variable as a label attached to a particular value. Variables allow you to work with data in a flexible and dynamic way.

Key points about variables:

Naming Conventions: Variable names in Python must follow certain naming conventions. They must start with a letter (a-z, A-Z) or an underscore (_) and can be followed by letters, digits (0-9), or underscores. Variable names are case-sensitive, meaning name and Name are considered different variables.

Assignment: You can assign values to variables using the assignment operator =. The value on the right side is assigned to the variable on the left side.

Dynamic Typing: Python uses dynamic typing, which means you don't need to declare the data type of a variable explicitly. Python determines the data type of a variable based on the value it holds.

Here are examples of each variable and data type in Python:

Integer

Represents whole numbers, both positive and negative, and is used for counting and mathematical operations.

```
# Integer variable
x = 5
y = -10

# Mathematical operations with integers
sum_result = x + y
product_result = x * y

print("Sum:", sum_result)
print("Product:", product_result)
```

Floating Point Number

Represents numbers with decimal points, allowing for precise calculations, especially in scientific and engineering contexts.

```
# Floating-point variables
pi = 3.14159
temperature = -0.5

# Mathematical operations with floats
circumference = 2 * pi * 5  # Calculate the circumference of a circle
with radius 5
half_temperature = temperature / 2

print("Circumference:", circumference)
print("Half Temperature:", half_temperature)
```

String

Represents text data enclosed in quotes, used for working with textual information, such as names and messages.

```
# String variables
name = "Alice"
message = 'Hello, Python!'

# Concatenating strings
greeting = "Hi, " + name + "!"

print(greeting)
print("Message:", message)
```

Boolean

Represents text data enclosed in quotes, used for working with textual information, such as names and messages.

```
# Boolean variables
is_python_fun = True
is_learning = False

# Conditional statements
if is_python_fun:
    print("Python is fun!")

if not is_learning:
    print("Not currently learning.")
```

Date & Time

In Python, the datetime module is used to work with dates and times. It provides various classes and functions for creating, manipulating, and formatting date and time values. Here's an explanation of the datetime module and some examples of how to use it:

datetime Module: Explanation: The datetime module in Python allows you to work with date and time values. It includes several classes, such as datetime, date, time, and timedelta, which provide functionality for handling different aspects of date and time. You can use it for tasks like representing dates, calculating time differences, and formatting date and time strings.

Examples

Get the current date and time

```
from datetime import datetime  # Import the datetime module

# Get the current date and time
current_datetime = datetime.now()  # Create a datetime object representing the current date and time
print("Current Date and Time:", current_datetime)  # Print the current date and time
```

Formatting Date and Time as Strings:

```
from datetime import datetime  # Import the datetime module

# Format a datetime object as a string
current_datetime = datetime.now()  # Create a datetime object representing the current date and time
formatted_date = current_datetime.strftime("%Y-%m-%d %H:%M:%S")  # Format the datetime object as a string
print("Formatted Date and Time:", formatted_date)  # Print the formatted date and time
```

Calculating Time Differences:

```
from datetime import datetime, timedelta  # Import the datetime and timedelta classes

# Calculate the difference between two dates
date1 = datetime(2023, 9, 27)  # Create a datetime object for the first date
date2 = datetime(2023, 10, 2)  # Create a datetime object for the second date
time_difference = date2 - date1  # Calculate the time difference
print("Time Difference:", time_difference)  # Print the time difference
```

Parsing Dates from Strings:

```
from datetime import datetime  # Import the datetime module

# Parse a date from a string
date_str = "2023-09-27"  # Define a date string
parsed_date = datetime.strptime(date_str, "%Y-%m-%d")  # Parse the string into a datetime object
print("Parsed Date:", parsed_date)  # Print the parsed date
```

Adding Time Intervals:

```
from datetime import datetime, timedelta  # Import the datetime and timedelta classes

# Add a time interval to a date
current_date = datetime.now()  # Create a datetime object for the current date and time
one_week = timedelta(weeks=1)  # Create a timedelta object representing one week
new_date = current_date + one_week  # Add the time interval to the current date
print("One Week Later:", new_date)  # Print the resulting date
```

Exception Handling

Exception handling is a crucial concept in programming that allows you to gracefully manage and respond to errors or exceptional situations that may occur during the execution of a program. Python, like many programming languages, provides built-in support for handling exceptions. Here, we'll explain what exceptions are, discuss different types of exceptions, and introduce the concept of exception handling.

What Are Exceptions?

Explanation: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exception occurs, Python generates an exception object that contains information about the error, such as its type and context. These exceptions can be caused by various factors, such as invalid input, file not found, or division by zero.

Types of Exceptions:

SyntaxError:

```
    Occurs when there is a syntax error in your code, making it invalid Python code.
```

NameError:

```
    Occurs when you try to access a variable or function that is not defined.
```

TypeError:

```
    Occurs when you perform an operation on a data type that is not supported.
```

ValueError:

```
    Occurs when you try to convert a value to a different data type, but the conversion is not valid.
```

FileNotFoundError:

```
    Occurs when you attempt to open or access a file that does not exist.
```

ZeroDivisionError:

```
    Occurs when you try to divide a number by zero.
```

Exception Handling:

Explanation:

```
    Exception handling is the process of anticipating and handling exceptions in your code to prevent crashes and allow for graceful error recovery. In Python, you can use try, except, else, and finally blocks to handle exceptions. The try block contains the code that may raise an exception, while the except block specifies how to handle the exception if it occurs.
```

```python
try:
    # Code that may raise an exception
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError as e:
    # Handle the exception
    print("Error:", e)  # Print the error message
```

The try block attempts to execute the code inside it. If an exception occurs, Python searches for a matching except block. If a matching except block is found, it executes the code within that block. You can specify the type of exception to catch (e.g., ZeroDivisionError) and assign it to a

variable (e) for further handling. You can also have multiple except blocks to handle different types of exceptions.

Additionally, you can use else to execute code when no exceptions occur and finally to specify code that always executes, regardless of whether an exception occurs or not.

```
try:
    result = 10 / 2  # This will not raise an exception
except ZeroDivisionError as e:
    print("Error:", e)
else:
    print("No exceptions occurred. Result:", result)
finally:
    print("This will always run.")
```

Exception handling allows your program to continue running even when errors occur, providing a way to gracefully recover from unexpected situations and maintain program stability.