PYTHON O LEVEL

FUNCTIONS

AKSHAR
COMPUTER
CENTRE

CONTACT US:

DOEACCAKSHARCOMPUTERCENTRE.COM

Functions

- Functions are blocks of organized, reusable code that perform a specific task.
- They allow you to break down a problem into smaller tasks, making your code more modular and easier to manage.
- Functions can take inputs, process them, and return outputs.
- They follow a top-down approach of problem-solving, where you solve the main problem by breaking it down into smaller sub-problems.
- Functions promote modular programming, where you write small, independent modules that can be combined to solve complex problems.

```python
# Example of a simple function
def greet(name):
    return f"Hello, {name}!"

# Calling the function
print(greet("Alice"))  # Output: Hello, Alice!
```

Function Parameter

- Parameters are variables that are defined in the function definition and are used to pass values into a function.
- They allow functions to accept inputs and work with them.

```python
# Example of function with parameters
def add(a, b):
    return a + b

# Calling the function with arguments
result = add(3, 5)
print(result)  # Output: 8
```

Functions

Local Variables

- Variables defined inside a function are called local variables.
- They are only accessible within the function where they are defined.

```python
# Example of local variables
def my_func():
    x = 10  # Local variable
    print(x)

my_func()  # Output: 10
# print(x) will result in an error because x is a local variable and is not accessible outside the function.
```

Return Statement

- The return statement is used to exit a function and return a value back to the caller.
- It can also be used to return multiple values.

```python
# Example of return statement
def square(x):
    return x * x

result = square(4)
print(result)  # Output: 16
```

Functions

Default argument values

- Default argument values are specified in the function definition.
- If the caller does not provide a value for that argument, the default value is used.

```python
# Example of default argument values
def greet(name="World"):
    return f"Hello, {name}!"

print(greet())  # Output: Hello, World!
print(greet("Alice"))  # Output: Hello, Alice!
```

Keyword arguments

- Keyword arguments are passed to a function with the syntax keyword=value.
- They allow you to specify arguments in any order by explicitly naming the parameter.

```python
# Example of keyword arguments
def greet(greeting, name):
    return f"{greeting}, {name}!"

print(greet(name="Alice", greeting="Hi"))  # Output: Hi, Alice!
```

Functions

VarArgs Parameters

- VarArgs parameters allow you to pass a variable number of arguments to a function.
- They are denoted by *args and can be used when the number of arguments that a function needs to accept is unknown.

```python
# Example of VarArgs parameters
def sum_values(*args):
    total = 0
    for num in args:
        total += num
    return total

print(sum_values(1, 2, 3, 4))  # Output: 10
```

Now, let's put this all together with a layman example:

Imagine you're baking a cake. Each step of the recipe can be likened to a function. Mixing the ingredients could be one function, baking the cake could be another. You pass ingredients (parameters) to these functions and they perform specific tasks. If you need to adjust the recipe, you can change just one part (function) without rewriting the entire recipe. So, functions help you break down a complex task (like baking a cake) into smaller, manageable steps, making your baking process more efficient and flexible.

Explanation

[doubt clearance]

VarArgs

Using *args in Python allows you to pass a variable number of arguments to a function. It's quite flexible and handy when you're unsure about the exact number of arguments that will be passed.
However, if you prefer not to use *args for some reason, you can still achieve similar functionality by explicitly defining the parameters you expect. Instead of accepting a variable number of arguments, you define each parameter separately.

```python
def values(num1, num2, num3, num4):
    total = num1 + num2 + num3 + num4
    return total

print(values(1, 2, 3, 4))
```

This approach explicitly specifies the parameters num1, num2, num3, and num4, instead of using *args to handle a variable number of arguments.

Explanation

[doubt clearance]

How functions work

Let's create a simple Python program that calculates the area of different shapes: rectangle, triangle, and circle. We'll explain each concept as we go along.

```python
# Global variable
PI = 3.14

# Function to calculate the area of a rectangle
def rectangle_area(length, width):
    # Parameters: length and width are parameters
    area = length * width  # Local variable: area
    return area

# Function to calculate the area of a triangle
def triangle_area(base, height):
    # Parameters: base and height are parameters
    area = 0.5 * base * height
    return area

# Function to calculate the area of a circle
def circle_area(radius):
    # Parameter: radius
    area = PI * radius * radius
    return area

# Main function
def main():
    # Function calls with argument values
    rect_area = rectangle_area(5, 4)
    print("Area of rectangle:", rect_area)

    tri_area = triangle_area(3, 6)
    print("Area of triangle:", tri_area)

    circ_area = circle_area(2)
    print("Area of circle:", circ_area)

# Entry point of the program
if __name__ == "__main__":
    main()
```

Explanation

[doubt clearance]

How functions work

Explanation:

1. Parameters: These are variables in a function definition. In our example, length, width, base, height, and radius are parameters.
2. Local variables: These are variables defined within a function and can only be accessed within that function. In each function (rectangle_area, triangle_area, circle_area), area is a local variable.
3. Global variables: These are variables defined outside of any function and can be accessed throughout the program. Here, PI is a global variable.
4. Return statement: This is used to return a value from a function. In our functions, we return the calculated area.
5. Function calls: These are made by using the function name followed by parentheses containing the arguments. For example, rectangle_area(5, 4) calls the rectangle_area function with arguments 5 and 4.
6. Argument values: These are the values passed to a function when it's called. In our example, 5 and 4 are argument values for rectangle_area.
7. Main function: This is the entry point of the program where execution begins. It typically calls other functions and orchestrates the flow of the program.
8. if name == "main": This conditional statement ensures that main() is only called if the script is executed directly, not if it's imported as a module into another script. It's a common Python idiom.

These concepts are fundamental to understanding how functions work in programming.

Top Down Approach

Top Down Approach in problem solving:

Top-down approach of problem-solving is a systematic method where a complex problem is broken down into smaller, more manageable sub-problems. Each sub-problem is solved individually, and their solutions are then combined to solve the original problem. This approach promotes modular programming, where code is organized into smaller, reusable functions, making it easier to understand, debug, and maintain.

Problem Decomposition:

- Identify the main problem and break it down into smaller, more manageable tasks or sub-problems.
- Each sub-problem should be independent and address a specific aspect of the overall problem.

Function Design:

- Design functions to solve each sub-problem identified during problem decomposition.
- Functions should have well-defined inputs, outputs, and responsibilities, making them reusable and modular.

Step-by-Step Implementation:

- Implement each function, focusing on solving its respective sub-problem.
- Test each function independently to ensure correctness before integrating it into the main solution.

Integration:

- Integrate the individual functions into the main solution, ensuring proper coordination and communication between them.
- Test the integrated solution to verify that it solves the original problem correctly.

Top Down Approach

Let's say we want to create a program that calculates the area of a rectangle.

- Problem Analysis: We need to calculate the area of a rectangle.
- Define Main Functionality: We'll start with a main function calculate_area that takes length and width as inputs and returns the area.
- Identify Subtasks: We need to write a function to calculate the area based on the length and width provided.
- Implement Subtasks:

```
def calculate_area(length, width):
    return length * width
```

- Integration:

```
def main():
    length = float(input("Enter the length of the rectangle: "))
    width = float(input("Enter the width of the rectangle: "))
    area = calculate_area(length, width)
    print("The area of the rectangle is:", area)

if __name__ == "__main__":
    main()
```

- Testing and Refinement: Test the program with different inputs to ensure it works correctly. Refine as needed.
- By following the top-down approach, we've designed and implemented the program in a structured manner, making it easier to understand, maintain, and extend.

Modular Programming

Modular programming in Python refers to breaking down a large program into smaller, self-contained modules that focus on specific tasks or functionalities. Each module encapsulates a set of related functions, classes, or variables, promoting code reusability, maintainability, and readability. Here are some advantages of modular programming in Python:

1. **Code Reusability**: Modules enable reuse of code across projects, reducing redundancy and saving development time.
2. **Maintainability**: Modular structure simplifies debugging, updating, and modifying code without affecting other parts of the program.
3. **Readability**: Organizing functionalities into modules enhances code readability and comprehension for developers.
4. **Scalability**: Adding new features is easier as modular programming allows for seamless integration and extension of existing modules.
5. **Collaboration**: Facilitates teamwork as developers can work on separate modules simultaneously, minimizing conflicts.
6. **Testing**: Independent testing of modules ensures better code quality and reliability through effective unit testing.
7. **Encapsulation**: Modules hide implementation details, exposing only necessary interfaces, reducing complexity and dependencies.
8. **Abstraction**: Modules abstract away implementation details, providing a simplified interface for interacting with complex functionalities.

Library Functions

A library function, in the context of programming, refers to a pre-defined function that is provided by a library or module. Libraries in programming contain collections of functions and routines that can be used to perform specific tasks without having to write the code for those tasks from scratch. These functions are designed to be reusable and can be called upon in different parts of a program to perform their specific tasks.

There are three main library functions: input () , eval () and print () .

input() :

The input() function is a built-in library function in Python that allows the user to enter input from the keyboard. It prompts the user with a message (if provided) and waits for the user to type something. Once the user presses Enter, it returns the entered value as a string.

```
name = input("Enter your name: ")
print("Hello", name)
```

eval() :

The eval() function is a built-in library function in Python that evaluates a string as a Python expression. It takes a string as input and executes it as a Python expression. This function is powerful but should be used with caution as it can execute arbitrary code.

```
result = eval("2 + 3")
print(result)  # Output: 5
```

Library Functions

print() :

The print() function is a built-in library function in Python used to display output to the console. It takes one or more arguments and prints them to the standard output (usually the console).

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

String Functions

A string function in Python is a built-in operation or method specifically designed to manipulate strings. These functions allow you to perform various tasks such as searching for substrings, modifying case, splitting strings into lists, and more. They help in efficiently processing and transforming string data within Python programs.

count() :

Returns the number of occurrences of a substring in the given string.

```
# Define the string
my_string = "Hello, World!"

# 1. count()
print("Count:", my_string.count("l"))  # Output: 3
```

find() :

Returns the lowest index of the substring if found in the given string, otherwise returns -1.

```
# 2. find()
print("Find:", my_string.find("World"))  # Output: 7
```

rfind() :

Returns the highest index of the substring if found in the given string, otherwise returns -1.

```
# 3. rfind()
print("rFind:", my_string.rfind("l"))  # Output: 10
```

String Functions

capitalize() :

Capitalizes the first character of the string.

```
# 4. capitalize()
print("Capitalize:", my_string.capitalize())  # Output: Hello, world!
```

title() :

Capitalizes the first character of each word in the string.

```
# 5. title()
print("Title:", my_string.title())  # Output: Hello, World!
```

lower() :

Converts all characters in the string to lowercase.

```
# 6. lower()
print("Lower:", my_string.lower())  # Output: hello, world!
```

upper() :

Converts all characters in the string to uppercase.

```
# 7. upper()
print("Upper:", my_string.upper())  # Output: HELLO, WORLD!
```

String Functions

swapcase():

Swaps the case of all characters in the string.

```
# 8. swapcase()
print("Swapcase:", my_string.swapcase())  # Output: hELLO, wORLD!
```

islower():

Returns True if all characters in the string are lowercase, otherwise returns False.

```
# 9. islower()
print("Is Lower:", my_string.islower())  # Output: False
```

isupper() :

Returns True if all characters in the string are uppercase, otherwise returns False.

```
# 10. isupper()
print("Is Upper:", my_string.isupper())  # Output: False
```

istitle() :

Returns True if the string is titlecased, otherwise returns False.

```
# 11. istitle()
print("Is Title:", my_string.istitle())  # Output: True
```

replace() :

Replaces occurrences of a substring with another substring.

```
# 12. replace()
print("Replace:", my_string.replace("Hello", "Hi"))  # Output: Hi, World!
```

String Functions

strip():

Removes leading and trailing whitespace characters from the string.

```
# 13. strip()
my_string = "  hello, world  "
print("Strip:", my_string.strip())  # Output: hello, world
```

lstrip():

Removes leading whitespace characters from the string.

```
# 14. lstrip()
print("LStrip:", my_string.lstrip())  # Output: hello, world
```

rstrip():

Removes trailing whitespace characters from the string.

```
# 15. rstrip()
print("RStrip:", my_string.rstrip())  # Output:   hello, world
```

split():

Splits the string into a list of substrings based on a delimiter.

```
# 16. split()
print("Split:", my_string.split(","))  # Output: ['  hello', ' world  ']
```

partition() :

Splits the string at the first occurrence of a specified separator and returns a tuple containing the part before the separator, the separator itself, and the part after the separator.

```
# 17. partition()
print("Partition:", my_string.partition(","))  # Output: ('  hello', ',', '
world  ')
```

String Functions

join():

Joins elements of an iterable with a specified separator.

```
# 18. join()
my_list = ['hello', 'world']
print("Join:", ", ".join(my_list))  # Output: hello, world
```

isspace():

Returns True if all characters in the string are whitespace characters, otherwise returns False.

```
# 19. isspace()
my_string = "   "
print("Is Space:", my_string.isspace())  # Output: True
```

isalpha():

Returns True if all characters in the string are alphabetic, otherwise returns False.

```
# 20. isalpha()
my_string = "hello"
print("Is Alpha:", my_string.isalpha())  # Output: True
```

isdigit():

Returns True if all characters in the string are digits, otherwise returns False.

```
# 21. isdigit()
my_string = "123"
print("Is Digit:", my_string.isdigit())  # Output: True
```

String Functions

isalnum():

Returns True if all characters in the string are alphanumeric, otherwise returns False.

```
# 22. isalnum()
my_string = "hello123"
print("Is Alnum:", my_string.isalnum())  # Output: True
```

startswith():

Returns True if the string starts with the specified prefix, otherwise returns False.

```
# 23. startswith()
print("Starts With:", my_string.startswith("hello"))  # Output: True
```

endswith():

Returns True if the string ends with the specified suffix, otherwise returns False.

```
# 24. endswith()
print("Ends With:", my_string.endswith("123"))  # Output: True
```

encode():

Returns the encoded version of the string using the specified encoding.

```
# 25. encode()
my_string = "hello, world"
print("Encode:", my_string.encode(encoding="utf-8"))  # Output: b'hello,
world'
```

decode():

Decodes the encoded string using the specified encoding.

```
# 26. decode()
my_bytes = b'hello, world'
print("Decode:", my_bytes.decode(encoding="utf-8"))  # Output: hello, world
```

String

A string in Python is a sequence of characters, enclosed within either single (' ') or double (" ") quotes, allowing manipulation and storage of textual data. It supports various operations like slicing, concatenation, and formatting, making it a fundamental data type for representing text-based information in Python programs.

Slicing:

- Slicing is a technique in Python that allows you to extract a portion of a string (or any sequence) by specifying a range of indices.
- The syntax for slicing is string *[start_index : end_index : step]* , where start_index is inclusive and end_index is exclusive.
- If start_index is not specified, it defaults to 0. If end_index is not specified, it defaults to the end of the string. The step parameter specifies the increment between each index and defaults to 1.

```python
my_string = "Hello, World!"

# Extracting "Hello"
print("Slicing 1:", my_string[0:5])  # Output: Hello

# Extracting "World"
print("Slicing 2:", my_string[7:12])  # Output: World

# Using negative indices to slice from the end
print("Slicing 3:", my_string[-6:-1])  # Output: World

# Skipping characters with step
print("Slicing 4:", my_string[::2])  # Output: Hlo ol!
```

String

Membership:

- Membership operators (*in* and *not in*) are used to test whether a value or variable is found in a sequence.
- For strings, membership testing is often used to check if a substring exists within a larger string.

```
my_string = "Hello, World!"

# Checking if 'Hello' is present in the string
print("Membership 1:", 'Hello' in my_string)  # Output: True

# Checking if 'Python' is present in the string
print("Membership 2:", 'Python' in my_string)  # Output: False

# Checking if 'Python' is present in the string
print("Membership 2:", 'Man' not in my_string)  # Output: True
```

Pattern Matching:

- Pattern matching involves finding a specific pattern within a string.
- In Python, the '*re*' module provides support for regular expressions, which are powerful tools for pattern matching.

```
import re
pattern = r'\bcat\b'
if re.search(pattern, 'The cat is black'):
    print('Found')
```

```
# Pattern to match a word starting with 'H'
pattern = r'\bH\w+'
my_string = "Hello, World! How are you?"

# Finding all words starting with 'H'
matches = re.findall(pattern, my_string)
print("Pattern Matching:", matches)  # Output: ['Hello', 'How']
```

Regex

[just for context]

Regex:

Regular Expression (regex) is a sequence of characters that forms a search pattern. It's used for pattern matching within strings, providing a powerful and flexible way to search, match, and manipulate text data based on specific patterns.

Why Use Regex in Pattern Matching:

Regex allows for precise and efficient pattern matching within strings, enabling tasks such as finding specific words or patterns, validating input formats (like email addresses or phone numbers), extracting data, and more. It provides a concise and flexible syntax to express complex search patterns, making it a valuable tool in text processing and data extraction tasks.

Now, let's move on to a simple pattern matching program and gradually increase the complexity:

```python
import re

# Define the pattern
pattern = r'hello'

# Define the string
my_string = "Hello, world! Say hello to everyone hello."

# Find all matches
matches = re.findall(pattern, my_string)

# Print the matches
print("Pattern Matching:", matches)  # Output: ['hello', 'hello']
```

In this simple program, we're searching for the word 'hello' in the given string. The *findall()* method returns a list containing all occurrences of the pattern in the string.

Regex

[just for context]

Intermediate Pattern Matching Program:

In this intermediate program, we're using a more complex pattern \b[A-Z]\w+ to match words starting with an uppercase letter. The \b ensures word boundary, [A-Z] matches any uppercase letter, and \w+ matches one or more word characters.

```
import re

# Define the pattern
pattern = r'\b[A-Z]\w+'

# Define the string
my_string = "Hello, World! How are you?"

# Find all matches
matches = re.findall(pattern, my_string)

# Print the matches
print("Pattern Matching:", matches)  # Output: ['Hello', 'World']
```

Numeric Functions

Numeric functions in Python are built-in functions that operate on numeric data types, such as integers and floating-point numbers. They provide various operations for mathematical calculations and manipulation of numeric values.

eval():

eval() function evaluates a string as a Python expression and returns the result.

```python
import math
import random

# eval()
expression = "2 + 3 * 4"
result = eval(expression)
print("eval() Result:", result)  # Output: 14
```

max() & min():

max() function returns the largest item from an iterable or the largest of two or more arguments.

min() function returns the smallest item from an iterable or the smallest of two or more arguments.

```python
# max() and min()
numbers = [5, 8, 2, 10]
max_num = max(numbers)
min_num = min(numbers)
print("max() Result:", max_num)  # Output: 10
print("min() Result:", min_num)  # Output: 2
```

Numeric Functions

pow():

pow() function returns the value of x to the power of y.

```
# pow()
base = 2
exponent = 3
result = pow(base, exponent)
print("pow() Result:", result)  # Output: 8
```

round():

round() function returns the floating-point number rounded to the specified number of decimals.

```
# round()
number = 3.14159
rounded = round(number, 2)
print("round() Result:", rounded)  # Output: 3.14
```

int():

int() function converts a number or string to an integer.

```
# int()
number_str = "10"
integer = int(number_str)
print("int() Result:", integer)  # Output: 10
```

random():

random() function returns a random floating-point number between 0 and 1.

```
# random()
random_number = random.random()
print("random() Result:", random_number)  # Output: Random floating-point
number between 0 and 1
```

Numeric Functions

ceil() & floor():

ceil() function returns the smallest integer greater than or equal to a given number.

floor() function returns the largest integer less than or equal to a given number.

```
# ceil() and floor()
num = 3.7
ceiling = math.ceil(num)
floor_value = math.floor(num)
print("ceil() Result:", ceiling)  # Output: 4
print("floor() Result:", floor_value)  # Output: 3
```

sqrt():

sqrt() function returns the square root of a given number.

```
# sqrt()
number = 16
square_root = math.sqrt(number)
print("sqrt() Result:", square_root)  # Output: 4.0
```

These programs demonstrate the usage of various numeric functions in Python for performing mathematical operations and manipulating numeric values.

Date & Time

Date and time functions:

Date & time functions in Python are provided by various modules such as datetime and time. These functions enable manipulation, formatting, and calculation of dates and times, allowing for effective handling of temporal data in Python programs.

```
import datetime

# Get the current date and time
current_datetime = datetime.datetime.now()
print("Current Date & Time:", current_datetime)

# Format the current date and time
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date & Time:", formatted_datetime)

# Get the current date
current_date = datetime.date.today()
print("Current Date:", current_date)

# Get the current time
current_time = datetime.datetime.now().time()
print("Current Time:", current_time)
```

In the above program, we demonstrate various date & time functions such as getting the current date and time, formatting dates and times, and obtaining the current date and time separately.

Recursion

Recursion is a programming technique where a function calls itself in order to solve a problem. In recursive functions, the function repeatedly invokes itself with smaller instances of the problem until it reaches a base case, where the solution can be computed directly without further recursion.

```python
# Example of factorial calculation using recursion
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Calculate factorial of 5
result = factorial(5)
print("Factorial of 5:", result)  # Output: 120
```

In the above program, we implement a recursive function to calculate the factorial of a number. The function calls itself with smaller instances of the problem (n-1) until it reaches the base case (n = 0), where it returns 1. Then, it recursively computes the factorial by multiplying n with the factorial of (n-1) until it reaches the initial call.

Thank you

Contact us for further inquiries

Akshar Computer Centre | +0191-2585266 | doeaccaksharcomputercentre.com