Control flow

Control flow statements

Control flow statements are a fundamental aspect of programming languages that enable you to control the execution of your code based on certain conditions, loops, and branching. They allow you to make decisions and repeat actions, making your programs more dynamic and responsive. Common control flow statements in most programming languages include:

- **Conditional Statements**:
- **if**: Allows you to execute a block of code only if a specified condition is true. There can also be an optional "else" block for an alternative action.
- **else if (elif)**: Used to check multiple conditions in a sequence.
- **else**: Executed when none of the previous conditions are true.
- **switch (or case statements)**: Allows you to choose one of many code blocks to be executed.
- **Looping Statements**:
- **for**: A loop that iterates a specific number of times.
- **while**: A loop that continues as long as a specified condition is true.
- **do-while (or repeat-until)**: A loop that executes a block of code at least once, and then continues to execute while a condition is true.
- **Control Flow Keywords**:
- **break**: Used to exit the current loop or switch statement.
- **continue**: Skips the current iteration of a loop and continues with the next iteration.
- **return**: Exits a function and returns a value (if required).
- **goto** (in some languages): Allows jumping to a labeled point in the code. Note that "goto" is generally discouraged and considered bad practice because it can make code harder to understand and maintain.

Introduction to functions

In Python, a function is a reusable block of code that performs a specific task or a set of tasks. Functions are essential for breaking down a program into smaller, more manageable parts, making your code more organized and easier to maintain. Functions are defined using the `def` keyword, followed by a function name and a set of parentheses that may contain input parameters, and a colon.

Here's a basic introduction to creating and using functions in Python:

- **Defining a Function:**

- You can define a function using the **`def`** keyword, followed by the function name, a pair of parentheses, and a colon. Any input parameters go inside the parentheses. The function body is indented and contains the code to be executed when the function is called.

```
def greet(name):
    print(f"Hello, {name}!")
```

Calling a Function:

To execute the code inside a function, you need to call the function by its name and provide the required arguments if any.

```
greet("Alice")  # Calling the 'greet' function with the argument "Alice"
```

Return Statement:

Functions can return values using the **`return`** statement. The return value can be assigned to a variable or used directly.

```
def add(x, y):
    result = x + y
    return result


sum_result = add(5, 3)
print(sum_result)  # This will print 8
```

Function Parameters:

Functions can accept zero or more parameters, which act as placeholders for the values you pass when you call the function. You can also provide default values for parameters.

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")


greet("Bob")  # This will print "Hello, Bob!"
greet("Alice", "Hi")  # This will print "Hi, Alice!"
```

Docstrings:

It's a good practice to provide a docstring, a multi-line string that explains what the function does. This helps other developers (and yourself) understand the purpose of the function.

```
def add(x, y):
    """
    This function adds two numbers and returns the result.
    """
    result = x + y
    return result
```

Scope:

Functions can access variables defined outside of them, but they cannot modify those variables unless they are explicitly declared as global or nonlocal. This is due to the concept of variable scope.

```
x = 10


def modify_x():
    global x  # Access the global 'x' variable
    x = 20


modify_x()
print(x)  # This will print 20
```

The random module

The `random` module in Python is a built-in module that provides functions for generating random numbers and making random selections. This module is commonly used in various applications, such as games, simulations, cryptography, and any situation where randomness is required. To use the `random` module, you need to import it first

```
import random
```

Here are some of the most commonly used functions and methods provided by the `random` module:

- **Generating Random Numbers:**
- **`random.random()`**: Returns a random floating-point number between 0 and 1 (inclusive of 0, exclusive of 1).

```
random_number = random.random()
```

`random.randint(a, b)`: Returns a random integer between **`a`** and **`b`**, inclusive of both endpoints.

```
random_integer = random.randint(1, 10)
```

`random.uniform(a, b)`: Returns a random floating-point number between **`a`** and **`b`**.

```
random_float = random.uniform(2.5, 5.5)
```

Generating Random Sequences:

- **`random.choice(sequence)`**: Returns a random element from the given sequence (e.g., a list or a string).

```
fruits = ['apple', 'banana', 'cherry', 'date']

random_fruit = random.choice(fruits)
```

Recursive functions

A recursive function in Python is a function that calls itself to solve a problem. Recursive functions are useful when a problem can be broken down into smaller, similar sub-problems. Each recursive call works on a smaller instance of the problem until a base case is reached, at which point the function returns a result without making further recursive calls. Recursive functions are characterized by two components:

- **Base Case:** The base case defines the condition under which the recursion should terminate. It provides the exit condition, and when this condition is met, the function returns a value or performs a specific action without making further recursive calls.
- **Recursive Case:** The recursive case defines how the function calls itself with modified arguments to make progress toward the base case. Each recursive call should move closer to the base case.

Here's an example of a recursive function that calculates the factorial of a number:

```
def factorial(n):

  # Base case: factorial of 0 is 1

  if n == 0:

    return 1

  # Recursive case: factorial of n is n times factorial of (n-1)

  else:

    return n * factorial(n - 1)


result = factorial(5)

print(result)  # Output: 120
```

In this example:

- The base case is when `n` is 0. In this case, the function returns 1.
- In the recursive case, the function calculates the factorial of `n` by multiplying `n` with the factorial of `n-1`.

The function keeps calling itself with a smaller value of `n` until `n` becomes 0 (the base case), at which point the recursion stops, and the function returns 1. Then, the results propagate back up through the stack of recursive calls to calculate the final result.

It's important to have a well-defined base case and ensure that the recursive calls make progress toward the base case to prevent infinite recursion.

Introduction to modules

In Python, a module is a file that contains Python code, including variables, functions, and classes. Modules allow you to organize and reuse your code by breaking it into separate files, making your codebase more manageable and maintainable. Python provides a wide range of built-in modules, and you can also create your own custom modules. Here's an introduction to working with modules in Python:

Built-in Modules:

Python comes with a large standard library of modules that provide a wide range of functionality, from working with files and math operations to web development and data manipulation. To use a built-in module, you need to import it using the `import` statement. For example, to use the `math` module:

```
import math


print(math.sqrt(16))  # This will print the square root of 16, which is 4.0
```

You can explore the Python Standard Library to see the extensive list of built-in modules: Python Standard Library.

Creating Custom Modules:

You can create your own custom modules by organizing related functions, variables, and classes in a Python file with a `.py` extension. For example, you can create a file named `my_module.py` and define functions and variables within it:

```
# my_module.py


def greet(name):
    return f"Hello, {name}!"
```

```
my_variable = 42
```

Importing Specific Items:

You can also import specific functions, variables, or classes from a module, which can be useful if you only need a subset of the module's contents. For example:

```
from my_module import greet


print(greet("Bob"))  # This will print "Hello, Bob!"
```

Module Aliases:

You can provide an alias for a module when importing it, which can make your code more concise and readable. Common aliases include **`import module as alias`**. For example:

```
import my_module as mm


print(mm.greet("Charlie"))  # This will print "Hello, Charlie!"
```