Chapter 2: Control Flow

Conditional statements

we will cover the following topics:

1. if statements
2. if-else statements
3. if-elif-else statements
4. ternary operator

Before we begin, let's define what a conditional statement is. A conditional statement is a programming construct that allows you to execute a certain block of code only if a certain condition is met.

if statements

The most basic form of a conditional statement is the **`if`** statement. It has the following syntax:

```
if condition:
    # code to be executed if condition is True
```

Here, **`condition`** is an expression that evaluates to a boolean value (either True or False). If the condition is True, the code inside the **`if`** block will be executed. Otherwise, it will be skipped.

Here's an example:

```

x = 5

if x > 0:
    print("x is positive")

# The output of this code will be: "x is positive"
```

if-else statements

Sometimes, you might want to specify different code blocks to be executed depending on whether the condition is True or False. In such cases, you can use the **`if-else`** statement. It has the following syntax:

```
if condition:
    # code to be executed if condition is True
else:
    # code to be executed if condition is False
```

Here's an example:

```
x = -5
```

```
if x > 0:
    print("x is positive")
else:
    print("x is not positive")

# The output of this code will be: "x is not positive"
```

If-elif-else statements

Sometimes, you might want to specify multiple conditions and execute different code blocks depending on which condition is met. In such cases, you can use the **`if-elif-else`** statement. It has the following syntax:

```
if condition1:
    # code to be executed if condition1 is True
elif condition2:
    # code to be executed if condition1 is False and condition2 is True
elif condition3:
    # code to be executed if condition1 and condition2 are False and condition3 is True
...
else:
    # code to be executed if all conditions are False
```

Here's an example:

```
x = 0

if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")

# The output of this code will be: "x is zero"
```

Loops

Loops are an important control structure, as they allow you to repeat a block of code multiple times. There are two main types of loops:

- For loops and
- While loops.

For Loops

For loops are used to iterate over a sequence of elements, such as a list, tuple, or string. The syntax for a for loop is:

```
for element in sequence:
    # code to be executed
```

Here's an example of a for loop that iterates over a list of numbers and prints out each number:

```
numbers = [1, 2, 3, 4, 5]

for number in numbers:
    print(number)

# Output :

1
2
3
4
5
```

You can also use the range() function to specify the number of iterations for the loop. For example:

```
for i in range(5):
    print(i)

# Output :

0
1
2
3
4
```

You can also specify a start and end value for the range function, as well as a step value:

```
for i in range(2, 6, 2):
    print(i)

# Output:
2
4
```

While Loops

While loops are used to repeat a block of code as long as a certain condition is met. The syntax for a while loop is:

```
while condition:
    # code to be executed
```

Here's an example of a while loop that prints out the numbers 1 to 5:

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

# Output:
1
2
3
4
5
```

It's important to include a way to update the condition inside the while loop, or else the loop will run indefinitely and create an infinite loop.

You can also use the break and continue statements to control the flow of the loop. The break statement will exit the loop completely, while the continue statement will skip the rest of the current iteration and move on to the next one.

```
i = 1
while True:
    if i > 5:
        break
    elif i % 2 == 0:
        i += 1
        continue
    print(i)
    i += 1

# Output:
1
3
5
```

Functions

Defining and calling functions

Defining a Function

To define a function, you can use the **`def`** keyword followed by the function name and a set of parentheses containing the function's parameters:

```
def greet(name):
  print("Hello, " + name)
```

Calling a Function

To call a function, you can simply use the function name followed by a set of parentheses containing the arguments you want to pass to the function:

```
greet("John") # prints "Hello, John"
```

Parameters

The function definition above defines a function called `greet` that takes a single parameter called `name`. The function prints a greeting message using the `name` parameter.

Arguments

In the example above, the `greet` function is called with the argument "John", which is passed to the `name` parameter of the function. This causes the function to print the greeting message "Hello, John". On the other hand, arguments are the values passed to a function when it is called. In the example above, "John" is the argument passed to the `greet`function.

Returning a Value

Functions can also return a value to the caller using the `return` keyword:

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

result = add(3, 4) # result is 7
```

In the example above, the `add` function takes two arguments, `x` and `y`, and returns their sum. When the function is called with the arguments 3 and 4, it returns the value 7, which is assigned to the `result` variable.

Scope and global variables

The scope of a variable is the region of the code where the variable is defined and can be accessed. There are two types of scope: global scope and local scope.

Global variables are variables that are defined outside of any function and are available to all functions in the program. They can be accessed and modified from anywhere in the code.

For example:

```
x = 10 # global variable

def func1():
  print(x) # prints 10

def func2():
  x = 20 # local variable
  print(x) # prints 20

func1()
```

```python
func2()
print(x) # prints 10
```

In the example above, the variable **`x`** is defined as a global variable and is initially set to 10. The **`func1`** function can access and print the value of **`x`**, because it is in the global scope.

The **`func2`** function defines a local variable called **`x`**, which shadows the global variable with the same name. This means that within the function, the local variable **`x`** takes precedence over the global variable **`x`**, and any operations on **`x`** within the function will affect the local variable instead of the global variable.

However, outside of the **`func2`** function, the global variable **`x`** is still accessible and has the value 10.