03. Control Flow

September 30, 2025

1 3. Control Flow

Control flow refers to the order in which the statements in a program are executed. It determines how the program's instructions are processed based on certain conditions or decision points. In essence, control flow dictates the path that the program takes during its execution.

There are several key elements of control flow:

- **Sequential Execution**: This is the default behavior of a program, where statements are executed one after the other in the order they appear in the code.
- **Conditional Execution**: Conditional statements, such as `if` statements, allow the program to execute certain blocks of code only if specific conditions are met. Depending on whether a condition evaluates to true or false, the program follows different paths.
- **Looping**: Looping constructs, such as `for` loops and `while` loops, enable the program to execute a block of code repeatedly as long as certain conditions are true. This allows for efficient repetition of tasks without the need for duplicate code.
- **Branching**: Branching statements, like `break` and `continue`, provide mechanisms to alter the flow of control within loops. They allow for early termination of loops or skipping iterations.

1.1 3.1. Conditional Statement (`if-elif-else`)

In Python, conditional statements are used to execute certain blocks of code based on whether a specific *condition* is true or false. The condition can be as simple as a variable, or it can be a more complex expression that can be evaluated as true or false.

The syntax of a simple conditional statement in Python is as follows:

```
if condition:
    statement_1
    ...
    statement_n
```

Notice the whitespace at the beginning of `statement_1 ... statement_n`. This is called indentation. While other programming languages use curly brackets to group together statements in the same block of statements, Python uses indentation. This contributes to a higher readability of the program. In order to end the `if` statement, you simply return to the previous indentation level.

What can be used as a *condition* in an `if` statement? - comparison between values - boolean values - boolean expressions - expressions containing other operators (e.g. `in` and `not in`) - function/method calls - more generally, anything that can be evaluated as true or false

1.1.1 Comparison operators

The table below contains comparison operations in Python.

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal

The result of a comparison operations is always a boolean value, making them another common appearance with conditionals.

[1]:
```
5 < 7.5  # numbers of different types can be compared
```

[1]: `True`

[2]:
```
"abc" >= "aaa"  # strings can also be compared
```

[2]: `True`

[3]:
```
grade = 7
0 < grade <= 10  # comparisons can be chained
```

[3]: `True`

[4]:
```
grade == 7  # checking if two values are equal
```

[4]: `True`

1.1.2 Exercises 3.1.1

1. Ask the user to input a number and store it in a float variable. Write expressions that check the following:
   - if the number is equal to 5
   - if the number is different than 3
   - if the number is greater than or equal to 0
   - if the number is strictly greater than 4.3 and smaller or equal to 7.9
2. Ask the user to input their name. Write an expression that is `True` if the name is longer than 5 characters.
3. Ask the user to input a word. Check if that word is a palindrome (the word is the same if read the same backwards as forwards, e.g. “madam”).

1.1.3 Boolean values

`bool` is a built-in data type that represents Boolean values, which can be either true or false. Boolean values are used to represent the truthfulness or falsehood of an expression or condition.

The `bool` type in Python has two values: `True` and `False`. Let's see an example:

[5]:
```
name = "Felix"
is_cat = True
```

[6]:
```
type(is_cat)
```

[6]:
```
bool
```

Boolean values are the perfect candidates to be used as a condition in `if` statements:

[7]:
```
print(f"{name} is my pet.")
if is_cat:
    print("He is a cat.")
```

```
Felix is my pet.
He is a cat.
```

1.1.4 Using objects of different types as a condition

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of a boolean operation. Here are most of the built-in objects considered false. These values are also called *falsy* values (as opposed to *truthy* values) because they evaluate as `False` without being exactly `False`:

- constants defined to be false: `None`, `False`
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

[8]:
```
times_played = 3
if times_played:
    print(f"You have played the game {times_played} times.")
```

```
You have played the game 3 times.
```

1.1.5 Exercises 3.1.2

1. In the program that asks for the name of the user, print *Hello, <name>!* if the user's name is not empty.

1.1.6 Boolean operations (`and`, `or`, `not`)

Boolean operations in Python involve logical operations performed on Boolean values (`True` or `False`). These operations allow you to combine, negate, or modify Boolean values to determine the truth value of complex expressions. These are the Boolean operations, ordered by ascending priority:

Operation	Result
`x or y`	if x is false, then y, else x
`x and y`	if x is false, then x, else y

Operation	Result
`not x`	if x is false, then `True`, else `False`

`and` and `or` are short-circuit operators, meaning that the second operand won't be evaluated if the first one is enough to decide the truth value of the expression.

[9]:
```
True and False
```

[9]:
```
False
```

[10]:
```
True or False
```

[10]:
```
True
```

[11]:
```
True or undefined
```

[11]:
```
True
```

1.1.7 Exercises 3.1.3

1. Given two integers, `x` and `y`, create a boolean which is `True` if one of them is 100 or if their sum is 100.
2. Given a string, print `True` if it contains both `a` and `e`.
3. Given a string, print `True` if it does not contain the character `t`.

1.1.8 Comparison to `True`, `False` and `None`

`True` and `False` are Python's boolean values.

`None` is a special constant that represents the absence of a value or a null value. It is often used to signify that a variable or expression does not have a meaningful value or that a function does not return anything.

Comparison to `True`, `False` and `None` is done with `is` and `is not` operators.

[12]:
```
name = None
if name is None:
    print("No value assigned to name")
```

```
No value assigned to name
```

[13]:
```
name = "Ana"
if name is not None:
    print(f"Hello, {name}!")
```

```
Hello, Ana!
```

If we want to check that a value is specifically `True` or `False`, and not just any truthy or falsy value, we will compare it with identity operators:

[14]:
```
name = "Felix"
is_cat = True
if is_cat is True:
    print(f"{name} is a cat.")
```

```
Felix is a cat.
```

1.1.9 Exercises 3.1.4

1. Initialize a variable `age` with the value `None`. Check if `age` is `None`, and, if it is, ask the user to input his age. Print the value of variable `age`.
2. In the code above, change `age` value to a numeric value. Re-run the code. What happens?

1.1.10 The `else` clause

In the code we've written by now, we have used simple `if` statements to decide if a block of code gets executed or not. The `else` clause is used in conjunction with the `if` statement to specify a block of code to be executed when the condition of the `if` statement evaluates to `False`. It provides an alternative path of execution when the condition is not met.

Here's the general syntax of the `if` statement with the `else` clause:

```
if condition:
    # Code block to execute if condition is True
else:
    # Code block to execute if condition is False
```

[5]:
```
x = 100

if x > 10:
    print("x is greater than 10")
else:
    print("x is not greater than 10")
```

```
x is greater than 10
```

1.1.11 Exercises 3.1.5

1. Create a Python program that takes an integer input from the user and determines whether it is even or odd. Output "Even" if the number is divisible by 2, and "Odd" if it is not divisible by 2.
2. Ask the user to enter a password. If the input from the user is equal to "pass123" display "Access granted", otherwise display "Access denied".
3. Write a Python program that determines whether a given year entered by the user is a leap year or not. A leap year is a year that is divisible by 4, but not divisible by 100 or if it is divisible by 400. Display the appropriate message, e.g. $>$ 2024 is a leap year $>>$ 1900 is not a leap year

1.1.12 The `elif` clause

The `elif` clause (short for "else if") is used in conjunction with the `if` statement to specify additional conditions to be tested if the original `if` condition evaluates to False. It allows for the testing of multiple conditions sequentially, providing an alternative path of execution for each condition that is met.

Here's the general syntax of the `if` statement with the `elif` clause:

```
if condition1:
    # Code block to execute if condition1 is True
elif condition2:
    # Code block to execute if condition1 is False and condition2 is True
elif condition3:
    # Code block to execute if both condition1 and condition2 are False, and condition3 is Tru
else:
    # Code block to execute if none of the conditions are True
```

[11]:
```
x = 1

if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is equal to 10")
else:
    print("x is less than 10")
```

```
x is less than 10
```

1.1.13 Exercises 3.1.6

1. Create a program that takes the current hour as an integer value (in 24-hour format) as input from the user and greets them with an appropriate message based on the time of day: `Morning: [5 - 12)` `Afternoon: [12 - 17)` `Evening: [17 - 21)` `Night: [21 - 5)`
2. Create a program that asks the user to enter two numbers. Display the larger number or the message "The numbers are equal" if they are equal.

1.1.14 Nesting `if` statements

`if` statements can be nested inside each other. This allows for more complex decision-making processes where multiple conditions need to be evaluated in a hierarchical manner.

As with all Python control structures, proper indentation is crucial. The code blocks within nested if statements must be indented to the appropriate level to indicate their hierarchical relationship.

[14]:
```
x = 10
y = 20

if x > 5:
    if y > 15:
```

```
        print("Both x and y are greater than their respective thresholds.")
    else:
        print("x is greater than 5, but y is not greater than 15.")
else:
    print("x is not greater than 5.")
```

```
Both x and y are greater than their respective thresholds.
```

1.1.15 Exercises 3.1.7

1. Create a Python program that calculates the delivery fee for an online order based on the distance of the delivery and the weight of the package. The pricing rules are as follows:

   - For distances under 10 km:
     - Packages weighing less than 5 kg: $5 delivery fee.
     - Packages weighing 5 kg or more: $8 delivery fee.
   - For distances of 10 km or more:
     - Packages weighing less than 5 kg: $10 delivery fee.
     - Packages weighing 5 kg or more: $15 delivery fee.

   Prompt the user to enter the distance and the weight of the package and output the delivery fee.

2. Write a Python program for checking the speed of drivers.

   - If speed is less than or equal to 50, it should print `"OK"`.
   - Otherwise, for every 5 km above the speed limit (50), it should give the driver one demerit point and print the total number of demerit points. For example, if the speed is 60, it should print: `"Points: 2"`.
   - If the driver gets more than 12 points, display `"License suspended"` instead of the number of demerit points.

   Prompt the user to enter the speed and output the appropriate message.

1.2 3.2. Repetitive statements - `while`

The while statement is used to repeatedly execute a block of code as long as a specified condition is true. It provides a way to create loops in Python, allowing for the execution of a set of statements multiple times until the condition evaluates to false.

Here's the general syntax of the while statement:

```
while condition:
    # Code block to execute while the condition is true
```

The condition can be any expression that evaluates to either `True` or `False`, just like we've already seen for conditionals.

It's important to ensure that the condition specified in the while statement eventually becomes false to prevent an infinite loop. If the condition never becomes false, the loop will continue indefinitely, potentially causing the program to hang or crash.

Inside the loop, you can include statements that modify variables or conditions, potentially affecting the loop's termination condition. This allows for dynamic control over how many times the loop is executed.

[6]:
```
x = 0
while x < 5:
    print(x)
    x = x + 1
```

```
0
1
2
3
4
```

In the example above, the while loop continues executing as long as the condition `x < 5` is true. The value of `x` is printed in each iteration, and `x` is incremented by 1 in each iteration. The loop terminates when `x` becomes 5, as 5 is not less than 5.

`x = x + 1` can be replaced with `x += 1`, which uses the *in-place* operator `+=`. All arithmetic operators (and also some other operators) have their *in-place* correspondent, which does the operation and assigns the result back to the same variable:

[19]:
```
x = 0
while x < 5:
    print(x)
    x += 1
```

```
0
1
2
3
4
```

[20]:
```
greeting = "hello"
greeting += "!"
print(greeting)
```

```
hello!
```

1.2.1 Exercises 3.2

1. Write a program that prints all integers between 500 and 525 (both included) using a `while` loop.
2. Write a program that prints all numbers between 0 (included) and 100 (not included) which are divisible by 7 but not divisible by 5.
3. Write a Python program that calculates the factorial of a given number entered by the user. The factorial of a non-negative integer `n` is the product of all positive integers less than or equal to `n`.
4. Create a Python program that asks the user to enter a password and continues prompting until they enter the correct password. The correct password is "password123".

1.3 3.3 Repetitive statements - `for`

A `for` statement is used to iterate over an iterable object and do something for each element. It allows you to execute a block of code repeatedly for each item in the sequence.

Here's the general syntax of the for loop:

```
for item in sequence:
    # Code block to execute for each item in the sequence
```

The `for` loop iterates over each item in the sequence one by one. It automatically assigns each item in the sequence to the variable specified in the loop header (`item` in the example above), allowing you to access the current item within the loop's code block.

It is similar to `foreach` in other programming languages. `for` is often the preferred looping statement because *iterable* objects are widely used in Python. For the moment, we can experiment with `str` - the iterable type we have studied so far:

[21]:
```
greeting = 'hello'
for character in greeting:
    print(character)
```

```
h
e
l
l
o
```

1.3.1 Exercises 3.3

1. Prompt the user to enter a word. Iterate over the word and display, for each letter, if it is a vowel or not (use `vowels = "aeiouAEIOU"`), e.g. > w is not a vowel

   o is a vowel

1.3.2 3.3.1 The `range()` function

In order to emulate the behavior of `for` statements as we know them from other languages:

```
for (i=0; i<10; i++) {
  do_something(i);
}
```

in Python, we use the `range()` built-in function:

[22]:
```
for i in range(5):  # Generating a sequence of numbers from 0 to 4 with a step␣
↪of 1
    print(i)
```

```
0
1
2
```

```
3
4
```

Here's the general syntax of the `range()` function:

`range(start, stop, step)`

- `start`: Optional. The starting value of the sequence (inclusive). If omitted, the default value is 0.
- `stop`: Required. The stopping value of the sequence (exclusive). The sequence will stop before reaching this value.
- `step`: Optional. The increment (or decrement if negative) between each number in the sequence. If omitted, the default step is 1.

The `range()` function returns a sequence of integers starting from `start` (inclusive) up to, but not including, `stop`, with each number incremented by `step`. If `start` is omitted, it defaults to 0. If `step` is omitted, it defaults to 1.

The `range()` function does not generate the entire sequence at once. Instead, it generates the numbers on demand as they are needed, which makes it memory-efficient for generating large sequences.

[23]:
```
for num in range(1, 11, 2):  # Generating a sequence of numbers from 1 to 10␣
↪with a step of 2
    print(num)
```

```
1
3
5
7
9
```

[24]:
```
for num in range(10, 0, -1):  # Generating a sequence of numbers from 10 to 1␣
↪with a step of -1
    print(num)
```

```
10
9
8
7
6
5
4
3
2
1
```

1.3.3 Exercises 3.3.1

1. Write a program that prints all integers between 500 and 525 (both included) using a `for` loop.

2. Write a program that computes the sum of all numbers between 100 and 150 using a `for` loop.
3. Write a Python program which iterates the integers from 1 to 50 and prints their value. For multiples of three print `"Fizz"` instead of the number and for the multiples of five print `"Buzz"` instead of the number. For numbers which are multiples of both three and five print `"FizzBuzz"` instead of the number.

1.4 3.4 Loop alteration (`break` and `continue` statements)

`break` and `continue` are control flow statements that are used within loops to alter their behavior.

1.4.1 3.4.1. The `break` statement

The `break` statement is used to immediately terminate the execution of a loop (such as `for` or `while`) when a certain condition is met.

When the `break` statement is encountered within a loop, the loop is exited immediately, and the program continues executing the code that follows the loop.

`break` is commonly used to exit a loop prematurely when a specific condition is fulfilled, without iterating through the remaining elements or steps in the loop.

[25]:
```
for i in range(5):
    if i == 3:
        break
    print(i)
```

```
0
1
2
```

1.4.2 Exercises 3.4.1

1. Write a Python program that prompts the user to enter a positive integer and determines whether it is a prime number. Use a `for` loop to check if the number is divisible by numbers between 2 and itself. If the number is divisible by any other number, break out of the loop and conclude that the number is not prime.
2. Create a Python program that prompts the user to enter a sentence and checks whether it meets certain criteria. Use a `while` loop to repeatedly prompt the user until a valid sentence is entered. A sentence is valid if it is longer than 5 characters and contains at least one space. If the input meets the criteria, break out of the loop.

1.4.3 3.4.2. The `continue` statement

The `continue` statement is used to skip the current iteration of a loop and continue with the next iteration.

When the `continue` statement is encountered within a loop, the remaining code within the loop for the current iteration is skipped, and the loop proceeds with the next iteration.

`continue` is commonly used to skip certain iterations of a loop based on a specific condition, without exiting the loop entirely.

[26]:
```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

```
0
1
3
4
```

1.4.4 Exercises 3.4.2

1. Write a Python program that prompts the user to enter a positive integer and prints all odd numbers up to that integer. Use a `for` loop to iterate over the range of numbers and use the `continue` statement to skip even numbers.
2. Create a Python program that prompts the user to enter a sentence and displays the non-vowel characters in the sentence. Use a `for` loop to iterate over the characters in the sentence and use the `continue` statement to skip vowels.