Python Control Flow

Comparison Operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater Than
`<=`	Less than or Equal to
`>=`	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False

>>> 'dog' != 'cat'
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Boolean Operators

There are three Boolean operators: `and`, `or`, and `not`.

The `and` Operator's *Truth* Table:

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`

False and True	False
False and False	False

The `or` Operator's *Truth* Table:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The `not` Operator's *Truth* Table:

Expression	Evaluates to
not True	False
not False	True

Mixing Operators

You can mix boolean and comparison operators:

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

Also, you can mix use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

if Statements

The `if` statement evaluates an expression, and if that expression is `True`, it then executes the following indented code:

```
>>> name = 'Debora'

>>> if name == 'Debora':
...    print('Hi, Debora')
...
# Hi, Debora
```

```
>>> if name != 'George':
...     print('You are not George')
...
# You are not George
```

The `else` statement executes only if the evaluation of the `if` and all the `elif` expressions are `False`:

```
>>> name = 'Debora'

>>> if name == 'George':
...     print('Hi, George.')
... else:
...     print('You are not George')
...
# You are not George
```

Only after the `if` statement expression is `False`, the `elif` statement is evaluated and executed:

```
>>> name = 'George'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
...
# Hi George!
```

the `elif` and `else` parts are optional.

```
>>> name = 'Antony'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
... else:
...     print('Who are you?')
...
# Who are you?
```

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse, simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, and otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> # this if statement:
>>> if age < 18:
...    print('kid')
... else:
...    print('adult')
...
# output: kid

>>> # is equivalent to this ternary operator:
>>> print('kid' if age < 18 else 'adult')
# output: kid
```

Ternary operators can be chained:

```
>>> age = 15

>>> # this ternary operator:
>>> print('kid' if age < 13 else 'teen' if age < 18 else 'adult')

>>> # is equivalent to this if statement:
>>> if age < 18:
...     if age < 13:
...         print('kid')
...     else:
...         print('teen')
... else:
...     print('adult')
...
# output: teen
```

Switch-Case Statement

The *Switch-Case statements*, or **Structural Pattern Matching**, was firstly introduced in 2020 via PEP 622, and then officially released with **Python 3.10** in September 2022.

Matching single values

```
>>> response_code = 201
>>> match response_code:
...     case 200:
...         print("OK")
...     case 201:
...         print("Created")
...     case 300:
...         print("Multiple Choices")
...     case 307:
...         print("Temporary Redirect")
...     case 404:
...         print("404 Not Found")
...     case 500:
```

```
...         print("Internal Server Error")
...     case 502:
...         print("502 Bad Gateway")
...
# Created
```

Matching with the or Pattern

In this example, the pipe character (`|` or `or`) allows python to return the same response for two or more cases.

```
>>> response_code = 502
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...
# Internal Server Error
```

Matching by the length of an Iterable

```
>>> today_responses = [200, 300, 404, 500]
>>> match today_responses:
...     case [a]:
...             print(f"One response today: {a}")
...     case [a, b]:
...             print(f"Two responses today: {a} and {b}")
...     case [a, b, *rest]:
...             print(f"All responses: {a}, {b}, {rest}")
...
# All responses: 200, 300, [404, 500]
```

Default value

The underscore symbol (`_`) is used to define a default case:

```
>>> response_code = 800
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...     case _:
```

```
...         print("Invalid Code")
...
# Invalid Code
```

Matching Builtin Classes

```
>>> response_code = "300"
>>> match response_code:
...     case int():
...             print('Code is a number')
...     case str():
...             print('Code is a string')
...     case _:
...             print('Code is neither a string nor a number')
...
# Code is a string
```

Guarding Match-Case Statements

```
>>> response_code = 300
>>> match response_code:
...     case int():
...             if response_code > 99 and response_code < 500:
...                 print('Code is a valid number')
...     case _:
...             print('Code is an invalid number')
...
# Code is a valid number
```

while Loop Statements

The while statement is used for repeated execution as long as an expression is `True`:

```
>>> spam = 0
>>> while spam < 5:
...     print('Hello, world.')
...     spam = spam + 1
...
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
```

break Statements

If the execution reaches a `break` statement, it immediately exits the `while` loop’s clause:

```
>>> while True:
...     name = input('Please type your name: ')
...     if name == 'your name':
...         break
...
>>> print('Thank you!')
# Please type your name: your name
# Thank you!
```

continue Statements

When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop.

```
>>> while True:
...     name = input('Who are you? ')
...     if name != 'Joe':
...         continue
...     password = input('Password? (It is a fish.): ')
...     if password == 'swordfish':
...         break
...
>>> print('Access granted.')
# Who are you? Charles
# Who are you? Debora
# Who are you? Joe
# Password? (It is a fish.): swordfish
# Access granted.
```

For loop

The `for` loop iterates over a `list`, `tuple`, `dictionary`, `set` or `string`:

```
>>> pets = ['Bella', 'Milo', 'Loki']
>>> for pet in pets:
...     print(pet)
...
# Bella
# Milo
# Loki
```

The range() function

The `range()` function returns a sequence of numbers. It starts from 0, increments by 1, and stops before a specified number:

```
>>> for i in range(5):
...     print(f'Will stop at 5! or 4? ({i})')
...
# Will stop at 5! or 4? (0)
# Will stop at 5! or 4? (1)
# Will stop at 5! or 4? (2)
# Will stop at 5! or 4? (3)
# Will stop at 5! or 4? (4)
```

The `range()` function can also modify it's 3 defaults arguments. The first two will be the `start` and `stop` values, and the third will be the `step` argument. The step is the amount that the variable is increased by after each iteration.

```
# range(start, stop, step)
>>> for i in range(0, 10, 2):
...    print(i)
...
# 0
# 2
# 4
# 6
# 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
>>> for i in range(5, -1, -1):
...     print(i)
...
# 5
# 4
# 3
# 2
# 1
# 0
```

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
>>> for i in [1, 2, 3, 4, 5]:
...    if i == 3:
...        break
... else:
...    print("only executed when no item is equal to 3")
```

Ending a Program with sys.exit()

`exit()` function allows exiting Python.

```
>>> import sys

>>> while True:
...     feedback = input('Type exit to exit: ')
...     if feedback == 'exit':
...         print(f'You typed {feedback}.')
...         sys.exit()
...
# Type exit to exit: open
# Type exit to exit: close
# Type exit to exit: exit
# You typed exit
```