8 Operators

Python operators are special symbols or keywords that perform operations on variables and values. They are essential for performing arithmetic, comparison, logical, and other types of operations in Python. Following different types of operators are used in Python:

8.1 Arithmetic Operators

These operators are used to perform basic mathematical operations.

- Addition (+): Adds two operands.

```
a = 5
b = 3
print(a + b)
```

```
Output:
8
```

- Subtraction (-): Subtracts the second operand from the first.

```
print(a - b)
```

```
Output:
2
```

- Multiplication (*): Multiplies two operands.

```
print(a * b)
```

```
Output:
15
```

- Division (/): Divides the first operand by the second.

```
print(a / b)
```

```
Output:
1.6666666666666667
```

- Floor Division (//): Divides the first operand by the second and returns the largest integer less than or equal to the result.

```
print(a // b)
```

```
Output:
1
```

- Modulus (%): Returns the remainder of the division.

```
print(a % b)
```

```
Output:
2
```

- Exponentiation (**): Raises the first operand to the power of the second.

```
print(a ** b)
```

```
Output:
125
```

8.2 Comparison Operators

These operators compare two values and return a Boolean result (True or False).

- Equal to (==): Checks if two operands are equal.

```
print(a == b)
```

```
Output:
False
```

- Not equal to (!=): Checks if two operands are not equal.

```
print(a != b)
```

```
Output:
True
```

- Greater than (>): Checks if the first operand is greater than the second.

```
print(a > b)
```

```
Output:
True
```

- Less than (<): Checks if the first operand is less than the second.

```
print(a < b)
```

```
Output:
False
```

- Greater than or equal to (>=): Checks if the first operand is greater than or equal to the second.

```
print(a >= b)
```

```
Output:
True
```

- Less than or equal to (<=): Checks if the first operand is less than or equal to the second.

```
print(a <= b)
```

```
Output:
False
```

8.3 Logical Operators

These operators are used to combine conditional statements.

- AND (and): Returns True if both statements are true.

```
print(a > 2 and b < 5)
```

```
Output:
True
```

- OR (or): Returns True if one of the statements is true.

```
print(a > 2 or b > 5)
```

```
Output:
True
```

- NOT (not): Reverses the result, returns False if the result is true.

```
print(not(a > 2))

Output:
False
```

8.4 Assignment Operators

These operators are used to assign values to variables.

- Assignment (=): Assigns a value to a variable.

```
c = a + b
```

- Add and assign (+=): Adds and assigns the result.

```
a += b  # Equivalent to a = a + b
```

- Subtract and assign (-=): Subtracts and assigns the result.

```
a -= b  # Equivalent to a = a - b
```

- Multiply and assign (*=): Multiplies and assigns the result.

```
a *= b  # Equivalent to a = a * b
```

- Divide and assign (/=): Divides and assigns the result.

```
a /= b  # Equivalent to a = a / b
```

- Floor divide and assign (//=): Floor divides and assigns the result.

```
a //= b  # Equivalent to a = a // b
```

- Modulus and assign (%=): Takes modulus and assigns the result.

```
a %= b  # Equivalent to a = a % b
```

- Exponent and assign (**=): Raises to the power and assigns the result.

```
a **= b  # Equivalent to a = a ** b
```

8.5 Membership Operators

These operators test for membership in a sequence (like lists, tuples, or strings).

- IN (in): Returns True if a value is found in the sequence.

```
my_list = [1, 2, 3, 4]
print(2 in my_list)

Output:
True
```

- NOT IN (not in): Returns True if a value is not found in the sequence.

```
print(5 not in my_list)
```

```
Output:
True
```

8.6 Identity Operators

These operators compare the memory locations of two objects.

- IS (is): Returns True if both variables point to the same object.

```
x = [1, 2, 3]
y = x
print(x is y)
```

```
Output:
True
```

- IS NOT (is not): Returns True if both variables do not point to the same object.

```
x = [1, 2, 3]
y = x
print(x is y)
```

```
Output:
True
```

8.7 Order of Operations

The order of operations in Python follows the standard mathematical rules (PEMDAS: Parentheses, Exponents, Multiplication and Division, Addition and Subtraction). One can use parentheses to clarify the order of operations.

```
x = 1
y = 3
z = 7
result = (x + y) * z
print(result)
```

```
Output:
28
```

8.8 Combining Operators

Operators can be combined to create complex expressions.

Example:

```
age = 26
has_permission = True  # Example value, can be True or False
is_employee = False    # Example value, can be True or False

is_valid = (age >= 18) and (has_permission or is_employee)
print(is_valid)
```

```
Output:
True
```

9 Control Flow

9.1 Introduction to Control Flow

Control flow structures in Python enable the execution of specific code blocks based on conditions and the repetition of code through loops. This chapter explores conditional statements and loops, fundamental for decision-making and iterative processes.

9.2 Conditional Statements (if, elif, else)

if Statement: The `if` statement allows the execution of a block of code only if a specified condition is true.

```
# if condition: Code to execute if condition is true
var1 = 100
if var1 == 100:
   print("1 - Got a true expression value")
   print("2 - The value is:", var1)
```

```
Output:
1 - Got a true expression value
2 - The value is: 100
```

else Statement: The `else` statement provides a block of code to execute when none of the preceding conditions are true.

```
if condition:
   # Code to execute if condition is true
else:
   # Code to execute if condition is false
```

An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value. The else statement is an optional statement, and there could be at most only one else statement following if. Syntax

The syntax of the if...else statement is –

```
var1 = 100
if var1 > 10:
   print("1 - The value is greater than 10")
   print("2 - The value is:", var1)
else:
   print("1 - The value is smaller than 10")
   print("2 - The value is:", var1)
```

```
Output:
1 - The value is greater than 10
2 - The value is: 100
```

elif Statement

The `elif` (else if) statement is used to check additional conditions if the preceding `if` or `elif` conditions are false.

```
if condition1:
    # Code to execute if condition1 is true
```

```
elif condition2:
    # Code to execute if condition2 is true
else:
        # Code to execute if condition is false
```

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if. Syntax. Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows:

```
var1 = 100
if var1 < 10:
  print("1 - The value is smaller than 10")
  print("2 - The value is:", var1)
elif var1 < 25:
  print("1 - The value is smaller than 25")
  print("2 - The value is:", var1)
elif var1 < 50:
  print("1 - The value is smaller than 50")
  print("2 - The value is:", var1)
else:
  print("1 - The value is greater than 50")
  print("2 - The value is:", var1)
```

```
Output:
1 - The value is greater than 50
2 - The value is: 100
```

```
# if-else-if ladder Example
a = 67
b = 76
c = 99
if(a > b & b > c):
    print("condition a > b > c is TRUE")
elif a < b & b > c:
    print("condition a < b > c is TRUE")
elif a < b & b < c:
    print("condition a < b < c  is TRUE")
```

```
Output:
condition a < b < c  is TRUE
```

9.3 Nested statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct. In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

The syntax of the nested if...elif...else construct may be –

```
if expression1:
statement(s)
if expression2:
statement(s)
elif expression3:
statement(s)
elif expression4:
statement(s)
else: statement(s)
else:
statement(s)
```

```
# nested loop example 1

var = 100
if var < 200:
   print("The value is less than 200")
   if var == 150:
      print("And is 150")
   elif var == 100:
      print("And is 100")
   elif var == 50:
      print("And is 50")
   elif var < 50:
      print("The value is less than 50")
else:
   print("Could not find true condition")
```

```
Output:
The value is less than 200
And is 100
```

```
#Nested loop example 2
a = 10
b = 11
if a == 10:
    if b == 10:
        print("a:10 b:10")
    else:
        print("a:10 b:11")
else:
    if a == 11:
        print("a:11 b:10")
    else:
        print("a:11 b:11")
```

```
Output:
a:10 b:11
```

9.4 Python - Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times. Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to

execute a statement or group of statements multiple times. Python programming language provides following types of loops to handle looping requirements.

while loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops

You can use one or more loop inside any another while, for or do..while loop.

9.5 Loops (for, while)

for Loop:

The `for` loop is used for iterating over a sequence (e.g., a list, tuple, string).

```
for item in iterable:
    # Code to execute for each item in the iterable
```

It has the ability to iterate over the items of any sequence, such as a list or a string. A for-loop in Python executes a block of code- once for each element of an iterable data type: one which can be accessed one element at a time, in order. Both strings and lists are such iterable types in Python. A *block* is a set of lines of code that are grouped as a unit; in many cases, they are executed as a unit as well, perhaps more than once. Blocks in Python are indicated by being indented an additional level (usually with four spaces—remember to be consistent with this indentation practice).

When using a for-loop to iterate over a list, we need to specify a variable name that will reference each element of the list in turn.

Syntax

for iterating_var in sequence:
statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

```
gene_ids = ["abc1", "abc2", "abc3"]
for i in gene_ids:
  print("Gene ID is " + i)
```

```
Output:
Gene ID is abc1
Gene ID is abc2
Gene ID is abc3
```

Using for-loops in Python often confuses beginners, because a variable (e.g., i) is being assigned without using the standard = assignment operator. If it helps, you can think of the first loop through the block as executing i = gene_ids[0], the next time around as executing i = gene_ids[1], and so on, until all elements of gene_ids have been used.

```
gene_ids = ["abc1", "abc2", "abc3"]
counter = 1
for i in gene ids:
  print(counter)
  print("Gene ID is " + i)
  counter = counter + 1
```

```
Output:
1
Gene ID is abc1
2
Gene ID is abc2
3
Gene ID is abc3
```

```
# Example 1
for letter in 'Python':
   print('Current Letter :', letter)
```

```
Output:
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
```

```
# Example 2
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:
   print('Current fruit :', fruit)
```

```
Output:
Current fruit : banana
Current fruit : apple
Current fruit : mango
```

```
# Example 3
for i in range(1,4):
  print(i**2)
  print(i)
```

```
Output:
1
1
4
2
9
3
```

```
# Example 4
x = [2,5,3,9,8,11,6]
count = 0
for i in x:
```

```
  if i % 2 == 0:
    count = count+1
print(count)
```

```
Output:
3
```

9.5.1 Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example –

```
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
  print('Current fruit :', fruits[index])
```

```
Output:
Current fruit : banana
Current fruit : apple
Current fruit : mango
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

9.5.2 Using else Statement with For Loop

Python supports to have an else statement associated with a loop statement. If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list. The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
1 for num in range(10,20): #to iterate between 10 to 20
2   for i in range(2,num): #to iterate on the factors of the number
3     if num%i == 0: #to determine the first factor
4       j=num/i #to calculate the second factor
5       print('%d equals %d * %d' % (num,i,j))
6       break #to move to the next number, the #first FOR
7   else: # else part of the loop
8     print(num, 'is a prime number')
```

```
Output:
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

while Loop:

A while loop statement in Python programming language repeatedly executes a block of code as long as a given condition is true.

```
Syntax
while condition:
    # Code to execute while condition is true
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop. Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

```
# Example 1
b=1
while b <= 5:
  print(b)
  b = b + 1
```

```
Output:
1
2
3
4
5
```

```
# Example 2
count = 0
while count < 9:
  print('The count is:', count)
  count = count + 1
```

```
Output:
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

9.5.3 The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop. An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
var = 1
while var == 1 : # This constructs an infinite loop
  num = raw input("Enter a number :")
  print("You entered: ", num)
```

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

9.5.4 Using else Statement with While Loop

Python supports to have an else statement associated with a loop statement. If the else statement is used with a while loop, the else statement is executed when the condition becomes false. The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
count = 0
while count < 5:
   print(count, " is  less than 5")
   count = count + 1
else:
   print(count, " is not less than 5")
```

```
Output:
0  is  less than 5
1  is  less than 5
2  is  less than 5
3  is  less than 5
4  is  less than 5
5  is not less than 5
```

9.5.5 loop inside loops (nested loops)

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

```
for iterating_var in sequence:
  for iterating_var in sequence:
    statements(s)
    statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows

```
while expression:
  while expression:
    statement(s)
    statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa. See the following examples.

```
# Example 1
i = 2
while(i < 20):
```

```
  j = 2
  while(j <= (i/j)):
    if not(i%j): break
    j = j + 1
  if (j > i/j) :
    print(i, " is prime")
  i = i + 1
```

```
Output:
2  is prime
3  is prime
5  is prime
7  is prime
11  is prime
13  is prime
17  is prime
19  is prime
```

```
# Example 2: Counting Stop Codons using nested for loop
seq = "ATGCCTGTAAGCTTAGCTGTGA"
stop_counter = 0
for index in range(0, len(seq)-3+1):
  codon = seq[index:index+3]
  if codon == "TAG" or codon == "TAA" or codon == "TGA":
    stop_counter = stop_counter + 1
print(stop_counter)
```

```
Output:
3
```

```
# Example 3: Counting Stop Codons using nested while loop
seq = "ATGCCTGTAAGCTTAGCTGTGA"
stop_counter = 0
index = 0
while index <= len(seq)-3:
  codon = seq[index:index+3]
  if codon == "TAG" or codon == "TAA" or codon == "TGA":
    stop_counter = stop_counter + 1
  index = index + 1
print(stop_counter)
```

```
Output:
3
```

9.6 Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements. Click the following links to check their detail.

break statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

9.6.1 Break and Continue Statements:

Break Statement:

The `break` statement is used to exit a loop prematurely, bypassing the remaining iterations.

```
for item in iterable:
    if condition:
        break
    # Code to execute for each item until the condition is met
```

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C. The most common use for break is when some external condition is triggered, requiring a hasty exit from a loop. The break statement can be used in both while and for loops. If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of code after the block.

```
# Example 1
for letter in 'Python':
   if letter == 'h':
      break
   print('Current Letter :', letter)
```

```
Output:
Current Letter : P
Current Letter : y
Current Letter : t
```

```
# Example 2
var = 10
while var > 0:
   print('Current variable value :', var)
   var = var -1
   if var == 5:
      break
```

```
Output:
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
```

Continue Statement:

The `continue` statement skips the rest of the code inside a loop for the current iteration and moves to the next iteration.

```
for item in iterable:
    if condition:
        continue
```

```
    # Code to execute for each item, skipping the rest for a specific
condition
```

It returns the control to the beginning of the while loop.. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

```
# Example 1
for letter in 'Python':
   if letter == 'h':
      continue
   print('Current Letter :', letter)
```

```
Output:
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

```
# Example 2
var = 10
while var > 0:
   var = var -1
   if var == 5:
      continue
   print('Current variable value :', var)
```

```
Output:
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
```

9.6.2 Pass

It is used when a statement is required syntactically but you do not want any command or code to execute. The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet.

```
for letter in 'Python':
   if letter == 'h':
      pass
      print('This is pass block')
   print('Current Letter :', letter)
```

```
Output:
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
```

```
Current Letter : o
Current Letter : n
```

Understanding control flow structures is essential for creating dynamic and responsive programs. Conditional statements allow for decision-making, while loops enable the repetition of tasks. Break and continue statements provide additional control within loops, enhancing the flexibility and efficiency of code. These concepts are crucial for implementing logic in various programming scenarios, including those encountered in biological data analysis and processing.