Operators and Control Flow

Contents

- Operators
- Precedence and Order of Evaluation
- Expressions
- Decision Structures
- The while Loop
- The for Loop
- The break Statement
- The continue Statement

Operators

- Operators do something on operands

```
>>> 3 + 5  # Plus
8
>>> 3 + 5.0
8.0
```

```
>>> 50 - 24  # Minus
26
>>> -5.2
-5.2
```

```
>>> 2 * 3  # Multiply
6
>>> 2.0 * 3
6.0
```

```
>>> 3 ** 4  # Power
81
```

Operators

```
>>> 15 / 3   # Float division
5.0
>>> 13 // 3  # Integer division resulting in an integer
4
>>> 3.5 // 2 # Integer division resulting in a float
1.0
```

```
>>> 13 % 3  # Modulo (returns the remainder of the division)
1
>>> -25.5 % 2.25
1.5
```

* $-25.5 = 2.25 \times (-12) + 1.5$

```
>>> 2 << 2  # Left Shift
8
```

* Left shifting 10 by 2 bits gives 1000

Operators

```
>>> 11 >> 1  # Right Shift
5
```

* Right shifting 1011 by 1 bit gives 101

```
>>> 5 & 3  # Bitwise AND
1
```

* 101 AND 011 = 001

```
>>> 5 | 3  # Bitwise OR
7
```

* 101 OR 011 = 111

```
>>> 5 ^ 3  # Bitwise XOR
6
```

* 101 XOR 011 = 110

```
>>> ~ 5  # Bitwise Invert (bit-wise inversion of x is -(x+1))
-6
```

* Bit-wise inversion of 0101 is 1010, which is a two's complement representation of –6

Operators

```
>>> 5 < 3  # Less Than
False
>>> 3 < 5
True
>>> 3 < 5 < 7
True
```

```
>>> 5 > 3  # Greater Than
True
```

```
>>> x = 3
>>> y = 6
>>> x <= y  # Less Than or Equal To
True
```

```
>>> x = 4
>>> y = 3
>>> x >= y  # Greater Than or Equal To
True
```

Operators

```
>>> 2 == 2  # Equal To
True
>>> 'str' == 'stR'
False
>>> 'str' == 'str'
True
```

```
>>> 2 != 3  # Not Equal To
True
```

```
>>> x = True
>>> not x  # Boolean Not
False
```

```
>>> x = False
>>> y = True
>>> x and y  # Boolean AND
False
```

* **`x and y`** returns **`False`** if **`x`** is **`False`**, else it returns the evaluation of **`y`**
(**`y`** is not evaluated if **`x`** is **`False`** -- short-circuit evaluation)

Operators

```
>>> x = False
>>> y = True
>>> x or y  # Boolean OR
True
```

* **x or y** returns **True** if **x** is **True**, else it returns the evaluation of **y**
(**y** is not evaluated if **x** is **True** -- short-circuit evaluation)

```
>>> l = 10
>>> m = 5
>>> n = 0
>>> (n != 0) and (l == (m / n))
False
```

* An error will occur if both parts of the compound condition are evaluated

Operators

- Relational operators

Python Notation	Numeric Meaning	String Meaning
==	equal to	identical to
!=	not equal to	different from
<	less than	precedes lexicographically
>	greater than	follows lexicographically
<=	less than or equal to	precedes lexicographically or is identical to
>=	greater than or equal to	follows lexicographically or is identical to
in		substring of
not in		not a substring of

Operators

- Methods that return either Boolean values

Method	Returns True when
str1.isdigit()	all of *str1*'s characters are digits
str1.isalpha()	all of *str1*'s characters are letters of the alphabet
str1.isalnum()	all of *str1*'s characters are letters of the alphabet or digits
str1.islower()	*str1* has at least 1 alphabetic character and all of its alphabetic characters are lowercase
str1.isupper()	*str1* has at least 1 alphabetic character and all of its alphabetic characters are uppercase
str1.isspace()	*str1* contains only whitespace characters

Shortcut for Math Operation and Assignment

- The shortcut expression for **`var = var operation expression`** is **`var operation= expression`**
  - The following are equivalent

```
>>> a = 2
>>> a = a + 3
```

```
>>> a = 2
>>> a += 3
```

Precedence and Order of Evaluation

- The following table summarizes the rules for precedence of all operators, including those we have not yet discussed
  - Rows are in order of increasing precedence
  - Operators on the same line have the same precedence

`lambda`	Lambda Expression
`if - else`	Conditional Expression
`or`	Boolean OR
`and`	Boolean AND
`not`	Boolean NOT
`in, not in, is, is not, <, <=, >, >=, !=, ==`	Comparisons
`|`	Bitwise OR
`^`	Bitwise XOR

Precedence and Order of Evaluation

`&`	Bitwise AND
`<<, >>`	Shifts
`+, -`	Addition and Subtraction
`*, /, //, %`	Multiplication, Division, Integer Division, and Modulus
`+x, -x, ~x`	Positive, Negative, bitwise NOT
`**`	Exponentiation
`x[index], x[index:index], x(arguments…), x.attribute`	Subscription, slicing, call, attribute reference
`(expressions…), [expressions…], {key: value…}, {expressions…}`	Binding or tuple display, list display, dictionary display, set display

- It is better to use parentheses to group operators and operands appropriately in order to explicitly specify the precedence

Associativity

- Operators are usually associated from left to right
  - E.g., `2 + 3 + 4` is evaluated as `(2 + 3) + 4`

- Some operators like assignment operators have right to left associativity
  - E.g., `a = b = c` is treated as `a = (b = c)`

Expressions

- An expression is a combination of values, variables, and operators
  - A value all by itself is considered an expression, and so is a variable
- A statement is a unit of code that the Python interpreter can execute
  - We have seen two kinds of statement: print and assignment
- Technically an expression is also a statement
  - The important difference is that an expression has a value; a statement does not

```
>>> x = 7
>>> 17
17
>>> x
7
>>> x + 17
24
```

Expressions

```
length = 5
breadth = 2

area = length * breadth

print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

[RUN]

```
Area is 10
Perimeter is 14
```

– Notice that Python puts a space between `'Area is'` and the variable `area` in the output

Decision Structures

- **`if`** and nested **`if-else`** statement

```
number = 23
guess = int(input('Enter an integer: '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
else:
    if guess < number:
        print('No, it is a little higher than that')
    else:
        print('No, it is a little lower than that')

print('Done')
# This last statement is always executed
# after the if statement is executed.
```

Decision Structures

- The string entered by the user is converted to an integer using `int()` and then stored in the variable `guess` (assuming that the string contains a valid integer in the text)
- A colon at the end of `if` statement indicates to Python that a block of statements follows
- Notice that we use indentation levels to tell Python which statements belong to which block

Decision Structures

```
[RUN]

Enter an integer : 50
No, it is a little lower than that
Done

[RUN]

Enter an integer : 22
No, it is a little higher than that
Done

[RUN]

Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

Decision Structures

- The following code is equivalent to that of the previous example

```
number = 23
guess = int(input('Enter an integer: '))

if guess == number:
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
elif guess < number:
    print('No, it is a little higher than that')
else:
    print('No, it is a little lower than that')

print('Done')
```

- This extension of the **`if-else`** statement allows for more than two possible alternatives with the inclusion of **`elif`** clauses
  - Reduces the amount of indentation required

Decision Structures

- The **`elif`** and **`else`** parts are optional

```
## Find the largest of three numbers.
firstNumber = eval(input('Enter the first number: '))
secondNumber = eval(input('Enter the second number: '))
thirdNumber = eval(input('Enter the third number: '))

# Determine and display the largest value.
largest = firstNumber
if secondNumber > largest:
    largest = secondNumber
if thirdNumber > largest:
    largest = thirdNumber
print('The largest number is', str(largest) + '.')
```

[Run]

```
Enter the first number: 3
Enter the second number: 7
Enter the third number: 4
The largest number is 7.
```

Decision Structures

– The integer stored in **`largest`** is converted to a string by **`str()`** before it is concatenated with another string (a period) by **`+`**

- A space will be printed before the period if we change the print statement as follows

```
. . . . . .
print('The largest number is', largest, '.')
```

[Run]

```
. . . . . .
The largest number is 7 .
```

The while Loop

- A while loop has the form

```
while condition:
    indented block of statements
```

  - Python first checks the truth value of ***condition***
  - If the condition evaluates to **`False`**, Python skips over the body of the loop and continues to execute the optional else-block and then continues to the line (if any) after the loop
  - If the continuation condition evaluates to **`True`**, the body of the loop is executed
  - After each pass through the loop, the continuation condition is rechecked and the body will be continually executed until the condition evaluates to **`False`**

The while Loop

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer: '))
    if guess == number:
        print('Congratulations, you guessed it.')
        running = False  # this causes the while loop to stop
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

The while Loop

[Run]

```
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

  - There is no need to repeatedly run the program for each guess, as we have done with the **`if-else`** example at the beginning

- If there is an **`else`** clause for a while loop, it is always executed unless you break out of the loop with a **`break`** statement
- The **`True`** and **`False`** are called Boolean types and they are equivalent to the values `1` and `0`, respectively

The while Loop

```
## Find the minimum, maximum, and average of a sequence of
## integers.

count = 0  # number of nonnegative integers input
total = 0  # sum of the nonnegative integers input

# Obtain numbers and determine count, min, and max.
print('(Enter -1 to terminate entering numbers.)')
num = int(input('Enter a nonnegative integer: '))
min = num
max = num

while num != -1:
    count += 1
    total += num
    if num < min:
        min = num
    if num > max:
        max = num
    num = int(input('Enter a nonnegative integer: '))
```

The while Loop

```
# Display results.
if count > 0:
    print('Minimum:', min)
    print('Maximum:', max)
    print('Average:', total / count)
else:
    print('No nonnegative integers were entered.')
```

[Run]

```
(Enter -1 to terminate entering numbers.)
Enter a nonnegative integer: 3
Enter a nonnegative integer: 7
Enter a nonnegative integer: 4
Enter a nonnegative integer: -1
Minimum: 3
Maximum: 7
Average: 4.66666666667
```

The for Loop

- The for loop is used to iterate through a sequence of objects

```
for var in sequence:
    indented block of statements
```

  - The loop variable ***var*** is successively assigned each value in the sequence and the loop body is executed after each assignment
  - ***sequence*** might be an arithmetic progression of numbers, a string, a list, a tuple, or a file object
    - When *m* and *n* are integers such that $m < n$, we can use the function `list(range(m, n))` to generate

      `[m, m + 1, m + 2, . . . , n − 1]`

    - `range(m, n, s)` takes the step count of *s* instead of 1 (When $m > n$, *s* can be a negative integer)
    - `range(0, n)` can be abbreviated to `range(n)`

The for Loop

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

[Run]

```
1
2
3
4
The for loop is over
```

- The **`else`** part is optional
  - When included, it is always executed once after the for loop is over unless a **`break`** statement is encountered

The for Loop

- Nested for loops:

```
## Display a triangle of asterisks.
numberOfRows = int(input('Enter a number from 1 to 20: '))

for i in range(numberOfRows):
    for j in range(i + 1):
        print('*', end='')
    print()
```

[Run]

```
Enter a number from 1 to 20: 5
*
* *
* * *
* * * *
* * * * *
```

The for Loop

- Looping through the characters of a string:

```
## Reverse the letters in a word.
word = input('Enter a word: ')
reversedWord = ''

for ch in word:
    reversedWord = ch + reversedWord
print('The reversed word is ' + reversedWord + '.')
```

[Run]

```
Enter a word: zeus
The reversed word is suez.
```

The for Loop

- Looping through the lines of a text file:

```
infile = open('fileName.txt', 'r')
for line in infile:
    indented block of statements
infile.close()
```

  - First statement establishes connection between program and file (assuming the file is in the same folder as the program)
  - **`for`** loop reads each line of the file in succession
    - **`line`** contains each line as a string
    - Executes indented block of statement(s) for each line
  - Last statement terminates the connection

The for Loop

- Looping through the lines of a text file:

```
## Display presidents with a specified first name.
firstName = input('Enter a first name: ')
foundFlag = False
infile = open('USPresidents.txt', 'r')
for line in infile:
    if line.startswith(firstName + ' '):
        print(line.rstrip())
        foundFlag = True
infile.close()
if not foundFlag:
    print('No president had the first name', firstName + '.')
```

[Run]

```
Enter a first name: John
John Adams
John Quincy Adams
John Tyler
John F. Kennedy
```

The for Loop

- Looping through the lines of a text file:
  - **`startswith`** is a string method
  - The variable **`foundFlag`** tells us if at least one president had the requested first name
  - The **`rstrip`** method removes the newline character at the end of each line of the text file
    - No matter how many there are, whitespaces (spaces, new line characters, tabs) are removed from the right side of a string

The for Loop

- The **`pass`** statement:
  - The header of a for loop must be followed by an indented block of at least one statement
  - The **`pass`** statement is a do-nothing placeholder statement

```
## Display the last line of a text file.
infile = open('USPresidents.txt', 'r')
for line in infile:
    pass
print(line.rstrip())
infile.close()
```

[Run]

```
Barack Obama
```

The break Statement

- The break statement causes an exit from anywhere in the body of a loop terminating the loop, even if the loop condition has not become **`False`** or the sequence of items has not been completely iterated over
- If you break out of a **`for`** or **`while`** loop, any corresponding loop **`else`** block is not executed

The break Statement

```
print('Enter QUIT to terminate entering something')
while True:
    s = input('Enter something : ')
    if s == 'QUIT':
        break
    print('Length of the string is', len(s))
print('Done')
```

[Run]

```
Enter QUIT to terminate entering something
Enter something : Programming is difficult
Length of the string is 24
Enter something : Use Python!
Length of the string is 11
Enter something : QUIT
Done
```

– The built-in `len()` function returns the length of the input string

The continue Statement

- The continue statement causes Python to skip the rest of the statements in the current loop body and to continue to the next iteration of the loop

The continue Statement

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too short')
        continue
    print('Input is of sufficient length')
```

[Run]

```
Enter something : a
Too short
Enter something : 12
Too short
Enter something : abc
Input is of sufficient length
Enter something : quit
```