Modulo Operator In Python (Simplified Examples)

Shahbaz KhanPublished: April 19, 2021Last updated: April 19, 2021

There are two ways in which we can do arithmetic division of two numbers. One of them is the floating-point division.
In this type of division, we get a single result after dividing two numbers, which is a floating-point number i.e it has a decimal point and a fractional part after the decimal.
In Python, the default behavior of the division operator '/' is this floating-point division. So if you divide 5 by 2, you will get 2.5 as the answer.

The other type of division is the integer division. This is the kind of division that you learn in middle-level mathematics class.
In integer division (also known as Euclidean division), when we divide a number (dividend) by another number(divisor), we get the result (quotient) as an integer, and we also get a remainder – another integer.
For eg., if we divide the number 5 by 2, then the quotient will be 2, and the remainder will be 1.

The modulo operator in Python `'%'` helps us find this 'remainder' value when Euclidean division is performed between the two operands.
In this tutorial, we will gain a detailed understanding of the modulo operator in Python. Before you begin, I assume that you have some Python basics.

Table of Contents

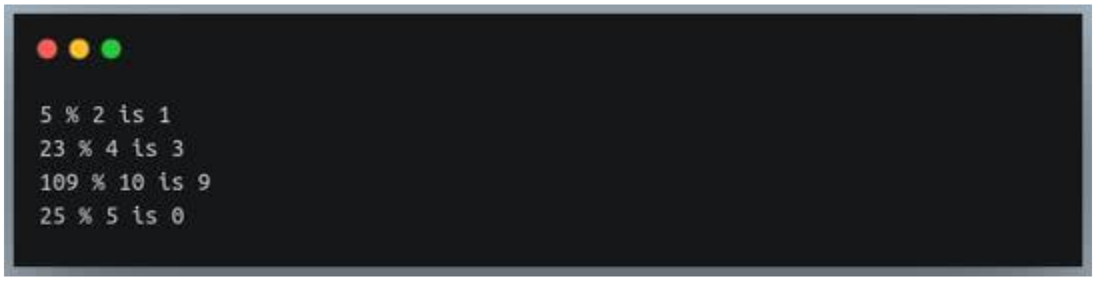
- Usage on positive integers
- Benefits of using the modulo operator
  - Testing the divisibility of a number
  - Testing if a number is even
  - Logging intermediate results in large loop operations
  - Cyclic iteration of a list
- Usage on floats
- Usage on negative numbers
- Modulo operator with strings
- Python modulo operator vs //
- Conclusion

Usage on positive integers

Let us first look at the result of the modulo operator on positive integer values.

```
a = 5 % 2

print(f"5 % 2 is {a}")

b = 23 % 4

print(f"23 % 4 is {b}")

c = 109 % 10

print(f"109 % 10 is {c}")
```

Output:



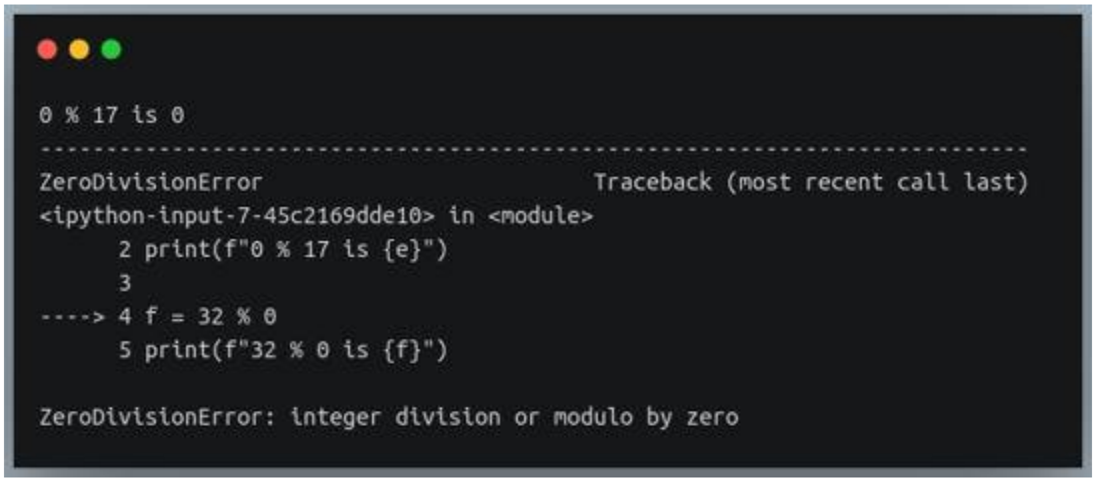
```
5 % 2 is 1
23 % 4 is 3
109 % 10 is 9
25 % 5 is 0
```

The result is in accordance with our discussion about integer division in the introduction.
If we divide 5 by 2, we get the remainder 1. If we divide 23 by 4, we get the remainder of 3. Dividing 109 by 10 gives us the remainder 9 (and quotient 10). Finally, if we divide 25 by 5, the remainder is 0 because 25 is evenly divisible by 5.

Note that if you pass 0 as the value for the second operand, you will get a `ZeroDivisionError` because we cannot divide any number by 0.
However, If the first operand is 0, then the result of modulo operation will always be zero.

```
e = 0 % 17

print(f"0 % 17 is {e}")



f = 32 % 0

print(f"32 % 0 is {f}")
```

Output:



```
0 % 17 is 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-7-45c2169dde10> in <module>
      2 print(f"0 % 17 is {e}")
      3 
----> 4 f = 32 % 0
      5 print(f"32 % 0 is {f}")

ZeroDivisionError: integer division or modulo by zero
```

Benefits of using the modulo operator

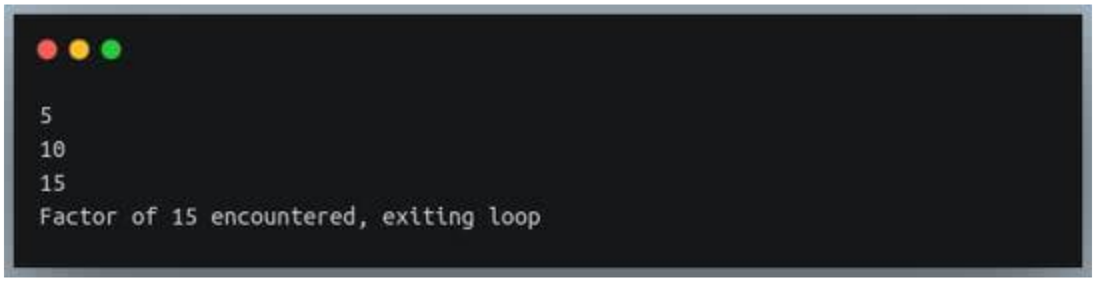
Modulo operator is used in a variety of situations in programming other than just finding the remainder after a division of two numbers. Here are a few examples of them:

Testing the divisibility of a number

In programming, we often encounter a situation where we have to test whether a given number is divisible by a certain number before moving forward in the code. For example, we may have a certain code block that can be run only if a given number is divisible by 7, or we may want to exit a loop if a number becomes divisible by 15.
We can put these divisibility tests as `if` conditions using the modulo operator. We know that if a number x is divisible by 7, the result of x % 7 will be 0.

```
for i in range(1,30):

    if i%5 == 0:

        print(i)

    if i%15 == 0:

        print("Factor of 15 encountered, exiting loop")

        break
```

Output:



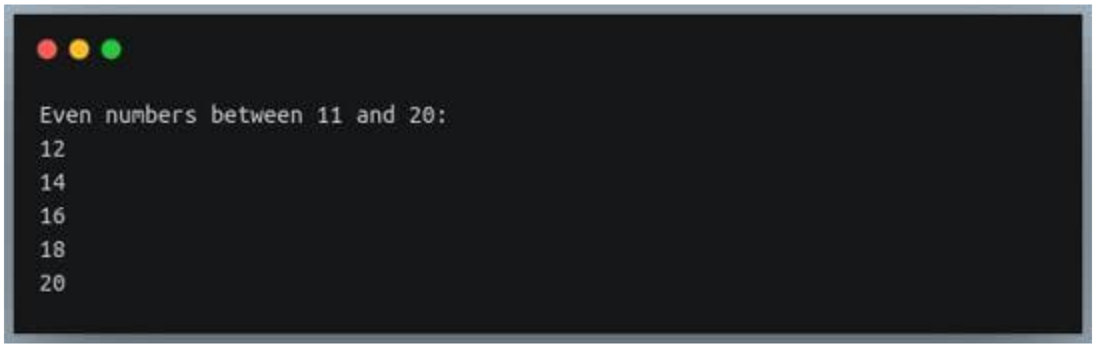
```
5
10
15
Factor of 15 encountered, exiting loop
```

Testing if a number is even

The need for testing the evenness of a number is frequently encountered in programming. We can extend the divisibility test discussed in the previous section to check the evenness of a number.
If the modulo operation between a number and 2 returns 0 as the result, then the number is even.

```
print("Even numbers between 11 and 20:")

for i in range(11,21):

    if i%2 == 0:

        print(i)
```

Output:



```
Even numbers between 11 and 20:
12
14
16
18
20
```

Logging intermediate results in large loop operations

When we execute a long, time-consuming code block involving loops with thousands of iterations, it is a good practice to log intermediate results to ensure that the code block within the loop is running fine and also to track the progress of the execution.
For eg., when training deep learning models, we run the training step for 100s or

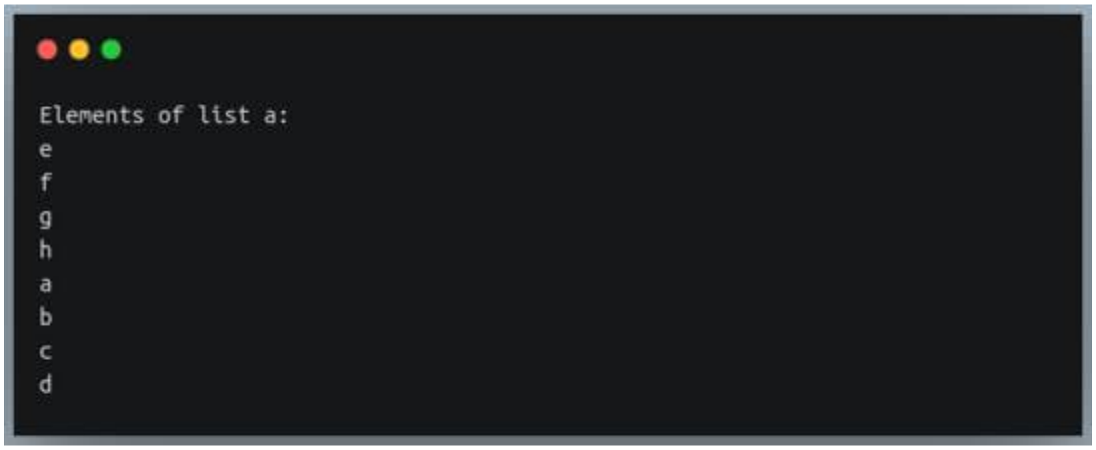
1000s of epochs.
It doesn't make sense to log results after every epoch. We can instead log results every, say 20 epochs. To do this, we can use the modulo operator as – `if epoch % 20 == 0: ...`

Cyclic iteration of a list

Modulo operation often becomes handy when we want to iterate a list or any iterable 'cyclically'. That is if we incrementally iterate over a list and if the index crosses the length of the list, it should move back to the starting position in the list. This also helps us avoid the `IndexError` when the list index goes out of range.

```
a = ["a", "b", "c", "d", "e", "f", "g", "h"]

index = 4 #start index

n = len(a)

print("Elements of list a:")

for i in range(n):

    print(a[index])

    index += 1

    index = index % n #ensuring the index remains within bounds
```

Output:



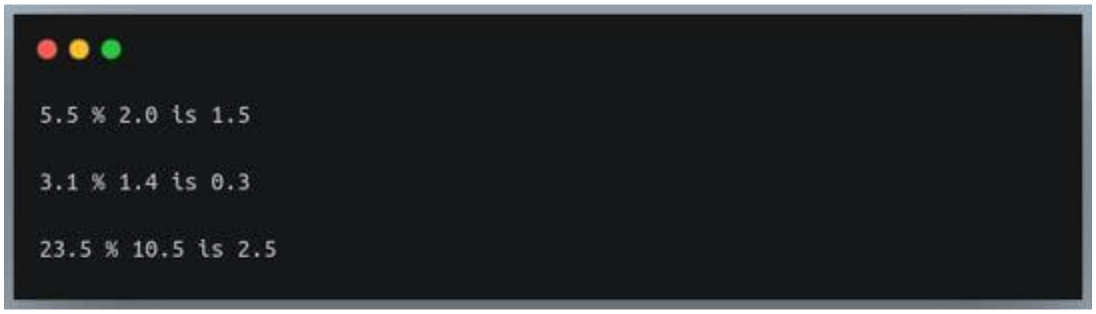
```
Elements of list a:
e
f
g
h
a
b
c
d
```

Usage on floats

We have seen the result of the modulo operator on integer operands. The result of such an operation is always an integer.
The operator also works with floating-point operands. The result, in this case, is a floating-point number.
Let us first look at some examples and then try to understand how it works on floating-point numbers.

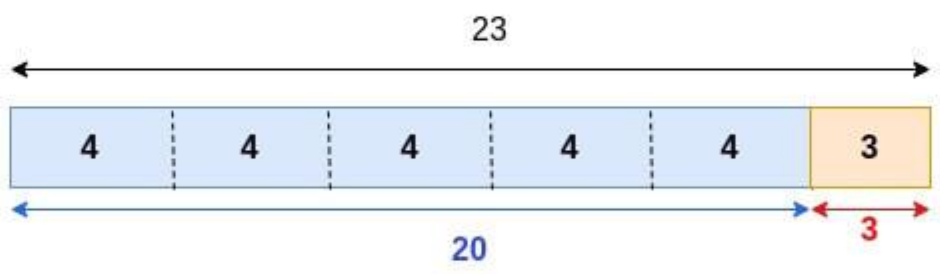
```
a = 5.5 % 2.0

print(f"5.5 % 2.0 is {round(a,2)}\n")



b = 3.1 % 1.4

print(f"3.1 % 1.4 is {round(b,2)}\n")



c = 23.5 % 10.5

print(f"23.5 % 10.5 is {round(c,2)}")
```

Output:



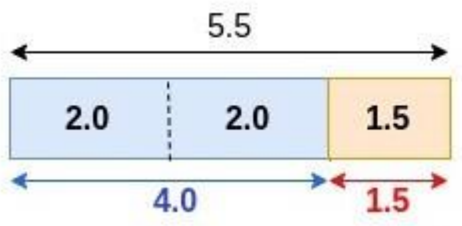
```
5.5 % 2.0 is 1.5

3.1 % 1.4 is 0.3

23.5 % 10.5 is 2.5
```

To understand how the modulo operator works on floating-point numbers, let us first redefine what the modulo operator does.
The modulo operator returns the remainder after evenly dividing the first number into as many complete parts of the second number as possible.
For eg., when we do `23 % 4`, we divide 23 into as many groups of 4 as possible (which is 5) after which, whatever remains (3), is the result of the modulo operation.



Since we can divide 23 into 5 groups of 4 (5×4 = 20), and we are left with the value 3, the result is 3.

A similar idea works for floating-point numbers.
When you do `5.5 % 2.0`, we can completely fit 2.0 in 5.5 exactly 2 times, and then we have a remainder of 1.5.



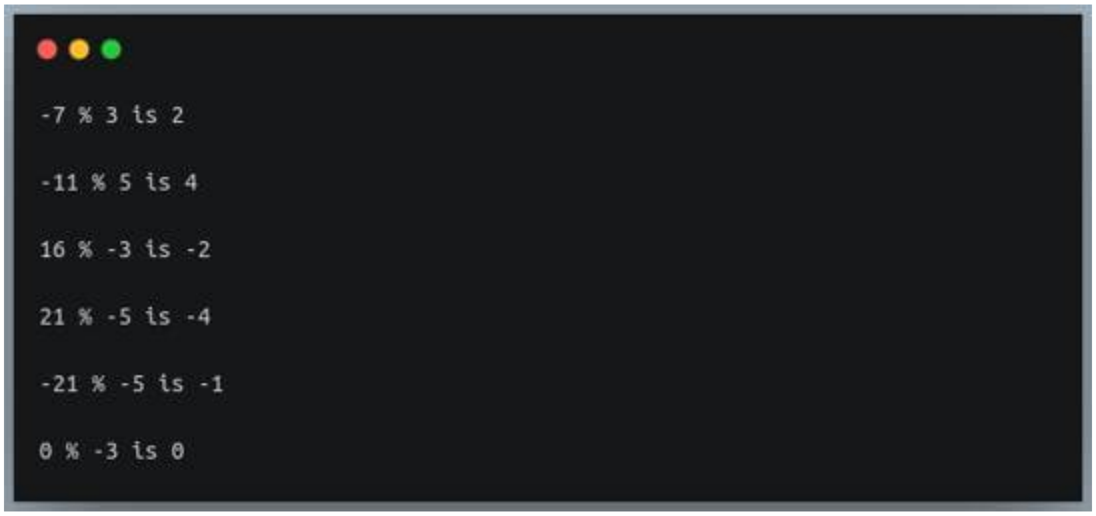
Usage on negative numbers

We have seen the result of modulo operation on positive integers and floating-point numbers. Let us now look at negative numbers.
The behavior of the modulo operator on negative numbers may initially look a bit counterintuitive, but it will make sense once you know the rules.
Let us first look at a few examples.

```
a = -7 % 3

print(f"-7 % 3 is {a}\n")



b = -11 % 5

print(f"-11 % 5 is {b}\n")
```

```
c = 16 % -3

print(f"16 % -3 is {c}\n")



d = 21 % -5

print(f"21 % -5 is {d}\n")

e = 0 % -3

print(f"0 % -3 is {e}\n")
```

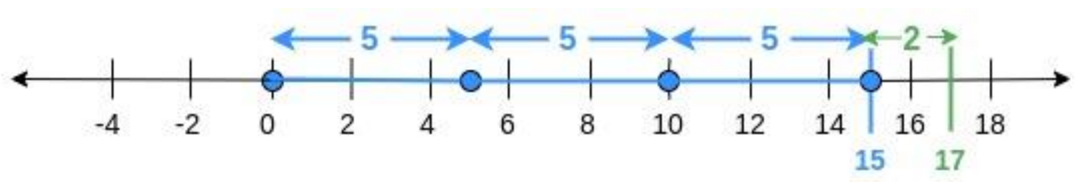
Output:



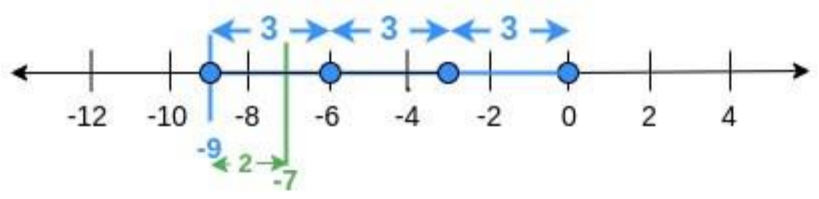
```
-7 % 3 is 2

-11 % 5 is 4

16 % -3 is -2

21 % -5 is -4

-21 % -5 is -1

0 % -3 is 0
```

The first important rule that is obvious from the output is that **the sign of the result is the same as the sign of the divisor**.
Now, to understand why we get a certain answer, let us once again discuss how the modulo operator works, this time with the context of a number line.

If you do `17 % 5` , you will get 2 as the answer.
This is because first, the floor division between 17 and 5 occurs, which gives 3 as the quotient.
Floor division (or integer division) returns the nearest integer to the left of the result of decimal division. 17 / 5 is 3.4, so floor(17 /5) is 3.
Now the product of the result of this floor division with the divisor (here 3*5 = 15) is subtracted from the dividend (here 17). The remainder (after subtraction) is the result of the modulo operation, which in this case happens to be 2.



Now, if we consider the first example with a negative operand i.e `-7 % 3` , `-7 / 3` is -2.3333, floor(-2.3333) is -3.
So we subtract 3*-3 = -9 from -7, the result is 2 and that is what we get as the result for `-7 % 3`

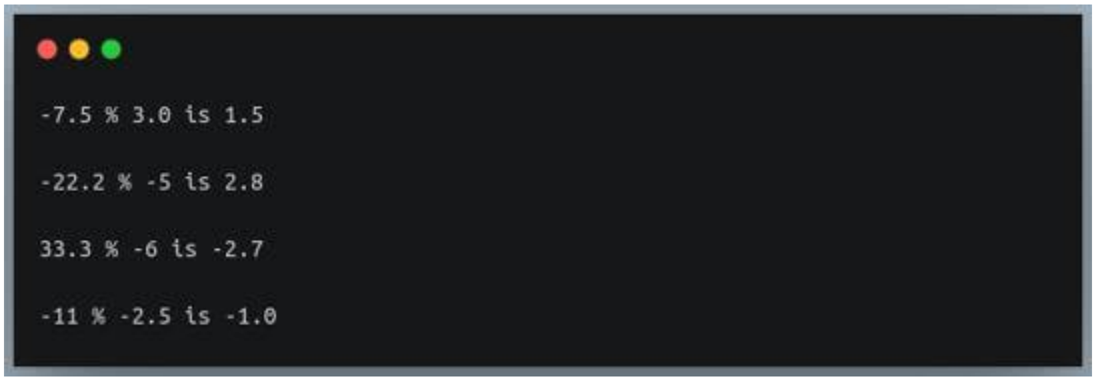


Similarly for `21 % -5` , floor(21 / -5) is -5. We subtract -5*-5 = 25 from 21 to get -4. And that is the answer for `21 % -5` .

Similar idea would work for negative floating-point operands as well.

```
a = -7.5 % 3.0

print(f"-7.5 % 3.0 is {a}\n")

b = -22.2 % 5

print(f"-22.2 % -5 is {round(b,2)}\n")

c = 33.3 % -6

print(f"33.3 % -6 is {round(c,2)}\n")

d = -11 % -2.5

print(f"-11 % -2.5 is {d}\n")
```

Output:



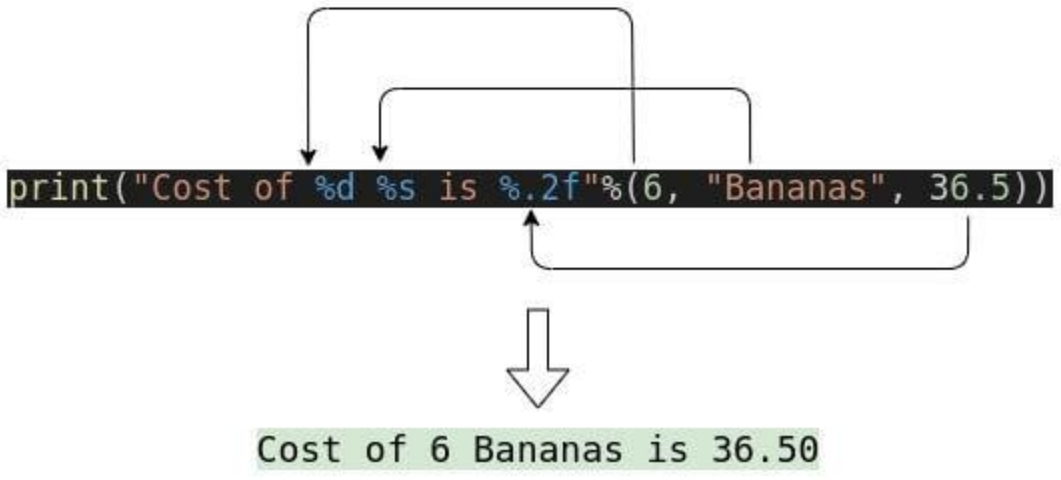
```
-7.5 % 3.0 is 1.5

-22.2 % -5 is 2.8

33.3 % -6 is -2.7

-11 % -2.5 is -1.0
```

Modulo operator with strings

Although the modulo operator is used to perform an arithmetic operation on numbers, it is used to achieve an entirely different purpose with strings.
Python modulo operator is used for formatting strings i.e for embedding values of other variables and data types in strings.

We specify the placeholders for values of different data types using the modulo operator in the string.
For example, if we want to insert an integer value at a position in a string, we will add `%d` at that position in the string. Similarly, we can specify floating-point values using `%f` .

The string is followed by a tuple containing as many arguments as the placeholders specified in the string. This tuple is also specified using the modulo operator.
This way of Python string formatting is similar to the string formatting used in the `printf` function in C language.



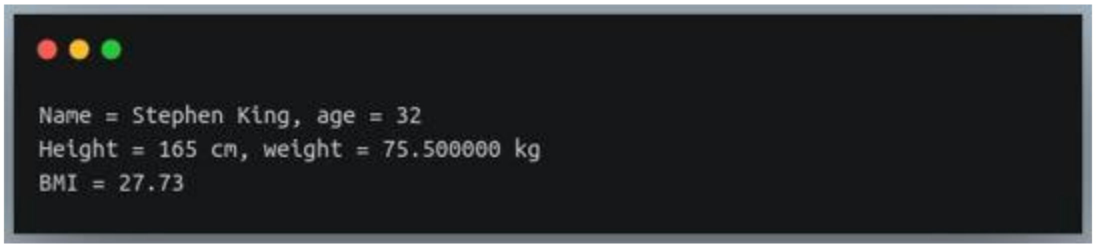
```
print("Cost of %d %s is %.2f"%(6, "Bananas", 36.5))
```

```
Cost of 6 Bananas is 36.50
```

Let us look at a few examples to understand this better.

```
name = "Stephen King"

age = 32

height = 165; weight = 75.5

print("Name = %s, age = %d" %(name, age))

print("Height = %d cm, weight = %f kg" %(height, weight))

bmi = weight/((height*0.01)**2)

print("BMI = %.2f" %bmi)
```

Output:



```
Name = Stephen King, age = 32
Height = 165 cm, weight = 75.500000 kg
BMI = 27.73
```

In the first print statement, we have inserted a string value and an integer value in the output string using the `%s` and `%d` format specifiers respectively.
In the second print statement, we have used `%d` and `%f` to insert integer and floating-point numbers in the strings.
Finally, in the third print statement, we have inserted the computed BMI value (float type) in the output string. But this time the format specifier used is `%.2f` . This is an indication to round the argument up to 2 decimal values in the string.

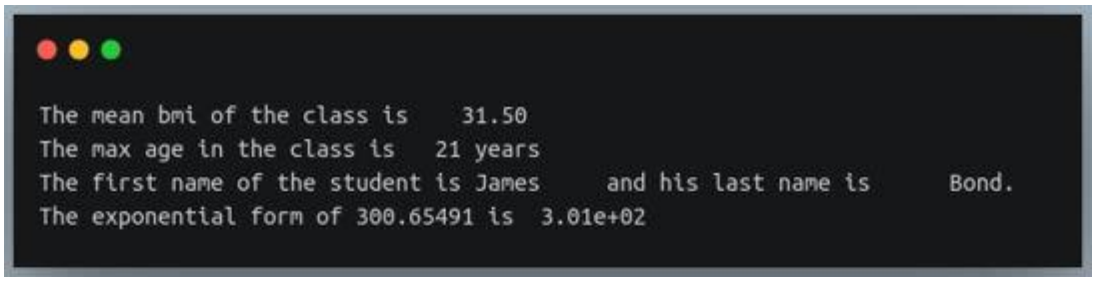
The way we specified the rounding of floating-point numbers up to 2 decimals, we can similarly modify/format the appearance of the values inside the strings in different ways using format specifiers.

For example, if we specify an integer before the datatype in the format specifier (such as `%5d` , `%10s`), it indicates the width it is supposed to occupy in the string. If the specified width is more than the actual length of the value, then it is padded with blank spaces.
If the specified width is less than the actual length, then the specified width is of no

value and the entire value will be to the string.
Let us look at a few examples.

```
print("The mean bmi of the class is %8.2f"%(31.5))

print("The max age in the class is %4d years"%(21))

print("The first name of the student is %-9s and his last name is %9s."%("James","Bond"))

print("The exponential form of 300.65491 is %9.2e"%(300.65491))
```

Output:



```
The mean bmi of the class is    31.50
The max age in the class is   21 years
The first name of the student is James     and his last name is      Bond.
The exponential form of 300.65491 is  3.01e+02
```

In the first example, we specify the total length of the floating-point value to be 8, and the precision to be 2. Hence the value 31.5 (length 4) is padded with 3 extra spaces in the beginning, and a 0 is added in the end to match the precision length. Similarly, in the second example, to display the value 21 we have used the format specifier `%4d` . This adds two extra spaces in front of 21 in the formatted string to match the length 4.

The third example shows how we can add trailing spaces instead of leading spaces. The `%-9s` indicates the minimum width of the argument “James” to be 9, and ‘-‘ indicates that any extra space has to be put at the end of the argument value in the string. Hence we see “James “(4 extra spaces) in the output string.

The last example shows how we can convert the floating-point value into its exponential form (or scientific notation) using the datatype character ‘e’.

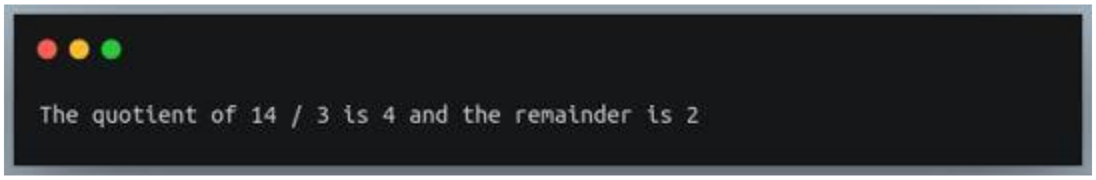
Python modulo operator vs //

The ‘ `//` ‘ operator in Python is used to perform floor division. It returns the nearest integer less than (to the left on the number line of) the result of the floating-point division between the two numbers.

In the introduction section, we learned about the integer division or 'Euclidean division', where the result of a division operation consists of two values – the quotient and the remainder.
While the modulo operator `%` returns the remainder for such a division, the floor division operator `//` returns the quotient.

```
q = 14 // 3

r = 14 % 3

print("The quotient of 14 / 3 is %d and the remainder is %d." %(q,r))
```

Output:



```
The quotient of 14 / 3 is 4 and the remainder is 2
```

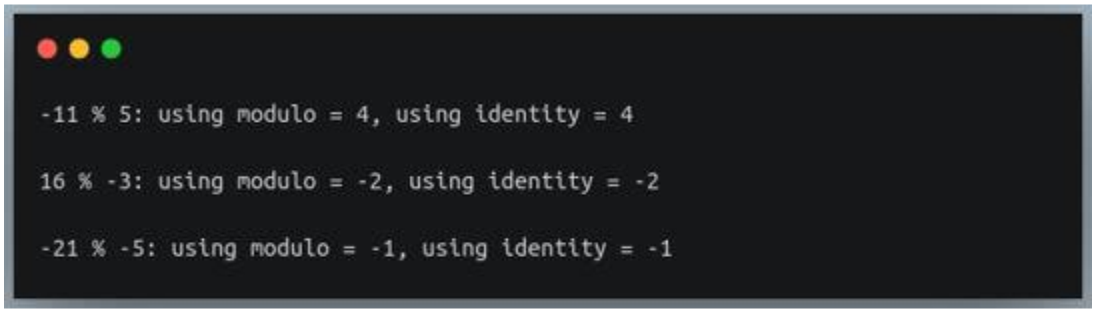
The result of the modulo operator is dependent on a floor division operation, and the two are inter-related using the following identity:
x % y = x – (x // y)*y

This identity should explain all the 'weird', counter-intuitive results we got with the negative operands to the modulo operator.
Let us revisit some of those examples and calculate the results using both the identity as well as the modulo operator.

```
a1 = -11 % 5

a2 = -11 - (-11//5)*5

print(f"-11 % 5: using modulo = {a1}, using identity = {a2}\n")



b1 = 16 % -3

b2 = 16 - (16//-3)*(-3)

print(f"16 % -3: using modulo = {b1}, using identity = {b2}\n")
```

```
c1 = -21 % -5

c2 = -21 - (-21//-5)*(-5)

print(f"-21 % -5: using modulo = {c1}, using identity = {c2}\n")
```

Output:



```
-11 % 5: using modulo = 4, using identity = 4

16 % -3: using modulo = -2, using identity = -2

-21 % -5: using modulo = -1, using identity = -1
```

All the results computed using the identity between modulo and floor division match with the result calculated using the modulo operator.

Conclusion

In this tutorial, we understood the different ways of using the modulo operator in Python.
We began by looking at the basic use of the Python modulo operator on integer values.

Then, we understood the various benefits of using the modulo operator by looking at 4 different use cases where the modulo operator is commonly used in Python. These were testing the divisibility of a number, testing the evenness of a number, logging in long loops, and cyclical iteration of lists. Through these examples, we also saw how we can use the modulo operator within `for` loops and with `if` conditions.

We then looked at the results of the modulo operator on floating-point operands. We also checked the output of modulo operation on negative numbers and dug deeper into the working of the modulo operator to understand why it returns counter-intuitive results with negative operands. Then, we deep-dived into the usage of the modulo operator for formatting strings.

Finally, we compared the modulo operator with the floor division operator and looked at how they complement each other.