IIT Madras
BSc Degree

Home Lesson-3.4

Lesson-3.3

Nested loops

Consider the following problem:

> Find the number of ordered pairs of positive integers whose product is 100. Note that order matters: (2, 50) and (50, 2) are two different pairs.

Solution

```
1 count = 0
2 for a in range(1, 101):
3     for b in range(1, 101):
4         if a * b == 100:
5             count = count + 1
6 print(count)
```

The code given above is an example of a nested loop. Lines 2-5 form the outer loop while lines 3-5 form the inner-loop. There are multiple levels of indentation here. Line-3 is the beginning of a new `for` loop, so line-4 is indented with respect to line-3. As line-4 is an if statement, line-5 is indented with respect to line-4.

This problem could have been solved without using a nested loop. The nested loop is not an efficient solution. It is left as an exercise to the reader to come up with a more efficient solution to this problem. Let us look at one more problem:

> Find the number of prime numbers less than n, where n is some positive integer.

Solution

```
n = int(input())
count = 0
for i in range(2, n + 1):
    flag = True
    for j in range(2, i):
        if i % j == 0:
            flag = False
            break
    if flag:
        count = count + 1
print(count)
```

The basic idea behind the solution is as follows:

- The outer for loop goes through each element in the sequence `2, 3, ..., n`. `i` is the loop variable for this sequence.
- We begin with the guess that `i` is prime. In code, we do this by setting `flag` to be `True`.
- Now, we go through all potential divisors of `i`. This is represented by the sequence `2, 3, ..., i - 1`. Variable `j` is the loop variable for this sequence. Notice how the sequence for the inner loop is dependent on `i`, the loop variable for the outer loop.
- If `j` divides `i`, then `i` cannot be a prime. We correct our initial assumption by updating `flag` to `False` whenever this happens. As we know that `i` is not prime, there is no use of continuing with the inner-loop, so we break out of it.
- If `j` doesn't divide `i` for any `j` in this sequence, then `i` is a prime. In such a situation, our initial assumption is right, and `flag` stays `True`.
- Once we are outside the inner-loop, we check if `flag` is `True`. if that is the case, then we increment count as we have hit upon a prime number.

Some important points regarding nested loops:

- Nesting is not restricted to `for` loops. Any one of the following combinations is possible:
  - `for` inside `for`
  - `for` inside `while`
  - `while` inside `while`
  - `while` inside `for`
- Multiple levels of nesting is possible.

`while` versus `for`

`for` loops are typically used in situations where the number of iterations can be quantified, whereas `while` loops are used in situations where the number of iterations cannot be quantified exactly. This doesn't mean that the number of iterations in a `for` loop is always constant. For example:

```
n = int(input())
for i in range(n):
    print(i ** 2)
```

In the code given above, the number of iterations will keep varying every time the code is run with a different input. But given the knowledge of the input, the number of iterations is fixed. On the other hand, consider the following example:

```
x = int(input())
while x > 0:
    x = int(input())
```

The number of iterations in the above code can be determined only after it terminates. There is no way of quantifying the number of iterations as an explicit function of user input.

print: `end`, `sep`

`end`

Consider the following problem:

> Accept a positive integer `n` as input and print all the numbers from 1 to n in a single line separated by commas.

For a given value of `n`, say `n = 9`, we want the output to be:

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

The following solution won't work:

```
n = int(input())
for i in range(1, n + 1):
    print(i, ',')
```

For `n = 9`, this will give the following output:

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

Thankfully, the print function provides a way to solve this problem:

```
n = int(input())
for i in range(1, n):
    print(i, end = ',')
print(n)
```

For `n = 9`, this will give the required output:

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

Whenever we use the `print` function, it prints the expression passed to it and immediately follows it up by printing a newline. This is the default behaviour of `print`. It can be altered by using a special argument called `end`. The default value of `end` is set to the newline character. So, whenever the end argument is not explicitly specified in the print function, a newline is appended to the input expression by default. In the code given above, by setting `end` to be a comma, we are forcing the `print` function to insert a comma instead of a newline at the end of the expression passed to it. It is called `end` because it is added at the end. To get a better picture, consider the following code:

```
print()
print(end = ',')
print(1)
print(1, end = ',')
print(2, end = ',')
print(3, end = ',')
```

This output is:

```

,1
1,2,3,
```

Even though nothing is being passed to the print function in the first line of code, the first line in the output is a newline because the default value of `end` is a newline character (`'\n'`). No expression is passed as input to print in the second line of code as well, but `end` is set to `,`. So, only a comma is printed. Notice that line-3 of the code is printed in line-2 of the output. This is because `end` was set to `,` instead of the newline character in line-2 of the code.

`sep`

If multiple expressions are passed to the `print` function, it prints all of them in the same line, by adding a space between adjacent expressions. For example:

```
print('this', 'is', 'cool')
```

The output is:

```
this is cool
```

What if we do not want the space or if want some other separator? This can be done using `sep`:

```
print('this', 'is', 'cool', sep = ',')
```

The output is:

```
this,is,cool
```

We could also have an empty string as the separator:

```
print('this', 'is', 'cool', sep = '')
```

The output will then be:

```
thisiscool
```

`end` and `sep`

Let us look at one final example that makes use of both `end` and `sep`:

> Accept a positive integer `n`, which is also a multiple of 3, as input and print the following pattern:

```
|1,2,3|4,5,6|7,8,9|...|n - 2,n - 1,n|
```

For `n = 9`, we would like to print:

```
|1,2,3|4,5,6|7,8,9|
```

Solution

```
n = int(input())
print('|', end = '')
for i in range(1, n + 1, 3):
    print(i, i + 1, i + 2, sep = ',', end = '|')
print()
```

Notice that the `for` loop iterates in steps of 3 starting from 1. To print the comma separated triplet `i,i + 1,i + 2`, `sep` is set to `,`. After printing each triplet, the symbol `|` needs to be printed. This is achieved by setting `end` to be equal to `|`. Line-2 makes sure that the symbol `|` is present at the beginning of the pattern. The last `print` statement outside the loop is there so that the prompt can move to the next line on the console once the pattern has been printed. You can try removing the last line and see how that changes the output on the console.