

CSC 241 Notes
Yosef Mendelsohn

# The 'in' operator

We have used this operator many times such as when iterating through a list:

for n in lstNumbers:

Or when using the range() function:

 for i in range(10):

**However, this very convenient operator allows you to check if a value is present in a collection.**

* For example, you might want to see if the number 3 is present in a list.
* Or you might want to see if the letter 'q' is present in a string (which is a collection of characters).

>>> lstNums = [1, 2, 3, 5, 7, 11, 13] #a list with 7 integers

>>> lstPets = ['rabbit','goldfish','emu']

>>> s = "Doh!"

The ‘**in**’ command returns true if the first argument is present inside the second argument:

>>> 2 in lstNums

True

>>> 4 in lstNums

False

>>> 'emu' in lstPets

True

>>> 'Emu' in lstPets

False

>>> '!' in s

True

>> 'oh' in s

True

# Loop patterns: Accumulator loop

A common pattern in loops is to accumulate some value in every iteration of the loop. This value can be a numeric value such as a count of something, or it might refer to a list that we keep appending to, etc.

For example, given a list of numbers we might want to determine the **sum** of those numbers.

To do this we need to introduce a variable that will contain the sum. This variable will need to be initialized to 0, and then a **for** loop is used to add every number in the list to the variable.

>>> lst = [3, 2, 7, -1, 9]

>>> sum = 0

>>> for number in lst:

 sum = sum + number

>>> print(sum)

20

In this example, **sum** is the **accumulator**. The assignment sum = sum + n increments the value of sum by n.

This type of operation, i.e. one in which we add a value to an existing variable is so common that there is a shortcut for it:

sum += n

Note: This shortcut also works for various other common mathematical operations such as subtraction, multiplication, division. For example, let's say you had a variable called total. If each time through the loop you wanted to double the value of this variable you could use:

total = total \* 2

OR

total \*= 2

Many – perhaps most – programmers tend to use this version.

Recall from the textbook that there is a built-in function sum that can be used to add up the values in a list.

>>> lst = [3, 2, 7, -1, 9]

>>> sum(lst)

20

So in *this* case a solution using a loop wasn’t necessary.

However, most of the time, a convenient built-in function that just so happens to do exactly what you want is not going to be available!

**Example**: Suppose we want to **multiply all the numbers** in the list.

A similar approach as the one we used for the sum would work:

>>> lst = [3, 2, 7, -1, 9]

>>> product = 0

>>> for i in lst:

 product = product \* i #Or: product\*=i

>>> print(product)

0

Clearly something went wrong. What is the problem?

Note that we initialized our variable product to 0. Since anything times 0 equals 0, that's what our calculation will end up being.

Instead we need to initialize product to 1, which is neutral for multiplication.

>>> lst = [3, 2, 7, -1, 9]

>>> product = 1

>>> for i in lst:

 product \*= i

>>> product

-378

In the previous two examples the accumulators were of a numerical type.

There can be other types of accumulators. In the following examples, the accumulators are strings and lists.

**Practice problem 1**: Write a function called upAbbrev that takes a string as an argument and returns a string consisting of all upper-case characters in the string.

It would be used as follows:

>>> upAbbrev(“Cubs Win the Pennant!”)

‘CWP’

**Practice problem 2**: Write a function divisors that takes a positive integer n as an argument and returns the list of all positive divisors of n.

**Question**: Do you remember what we mean by positive divisor? It's a question you must be able to answer in the future!

**Answer**: It's the situation where when you divide one number by another, the remainder is 0. In programming we do this using the modulus operator, %. For example, positive divisors of 10 are 1,2,5,10.

It would be used as follows:

>>> divisors(6)

[1, 2, 3, 6]

>>> divisors(11)

[1, 11]

>>> divisors(12)

[1, 2, 3, 4, 6, 12]

**Practice problem 3**: An acronym is a word formed by taking the first letters of the words in a phrase and making a word from them. For example, RAM is an acronym for “random access memory”.

Write a function acronym that takes a phrase as a parameter and returns the acronym for that phrase. Note: The acronym should be all uppercase, even if the words in the phrase are not capitalized.

It would be used as follows:

>>> makeAcronym('random access memory')

'RAM'

>>> makeAcronym('internal revenue service')

'IRS'

**Hint**: You'll need to break your phrase up into separate words. There is a function you definitely want to become familiar with that does this.

**Answer**: The split() function.

# Loop patterns: Nested loops

Some problems can only be solved using multiple loops together.

To see how to use multiple loops, consider the following (somewhat artificial) problem: Suppose we want to write a function nested() that takes a positive integer n and prints to the screen the following n lines:

0 1 2 3 … n-1

0 1 2 3 … n-1

…

0 1 2 3 … n-1

For example:

>>> nested(5)

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

We’ve already seen that in order to print one line we can write the following:

>>> for i in range(n):

 print(i, end=" ")

0 1 2 3 4

In order to get n such lines (5 lines in this case), we need to repeat the above loop n times (5 times in this case). We can do that with an additional outer for loop which will repeatedly execute the above inner for loop:

for j in range(n):

 for i in range(n):

 print(i, end = " ")

0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4

Great – we're close!!

Still, this doesn’t produce exactly we wanted. Setting end to a space forces all of the numbers onto a single line, which is not what we want.

We would like to have a new line printed between each line printed by the inner for loop. We can do that as follows:

>>> for j in range(n):

 for i in range(n):

 print(i, end = " ")

 print()

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

0 1 2 3 4

**Practice problem**: Write a function multmult that takes two lists of integers as parameters and returns a list containing the products of integers from the first list with the integers from the second list.

For example, it would be used as follows:

>>> multmult([3, 4, 1], [2, 0])

[6, 0, 8, 0, 2, 0]

**Important**: This is a CLASSIC example of a situation where you should not, not, not start by typing out code! Rather, you should spend a few minutes with a pen and paper and *think* (yuck) through the problem and come up with a strategy for solving it.

As always, solutions are in this week's examples file.