## COUNTED (FOR) LOOPS

```
for item in <collection>:
    statements
```

A different type of loop for repetition is the counted/fixed (or *for*-) loop. Although anything can be done with a while-loop, fixed/counted loops are much easier to structure in several cases, where a specific set/range of values needs to be iterated over.

Example: Write a for-loop that prints out the numbers from 1 to 10.

```
for num in range(1, 11):
    print(num)
```

Note that this loop does the exact same thing as the example of the while-loop on page 34.

Rather than loop based on a condition, the loop *iterates* a set number of times. In the given example, it loops an ***iterator*** variable over the numbers in the range from 1 to 10 (note that you use 1-more than the upper-end of the range). The *iterator* in this example is the variable '`num`'. An iterator is just like any variable, except its value will change on every ***iteration*** of the loop. On the first iteration, its value is 1. On the second, its value is 2, etc. These values are based on how the ***range*** is defined. You can also define a range starting from 0 using a single value in the call to the range function. For example, using `range(10)` in the above example would iterate the variable `num` over the values from 0 to 9. Note that the iterator does not need to actually be used in the *body* of your loop. You could simply use it to count the number of times that the loop will execute. Using the version of `range()` that takes a single value is often used in this case. Finally, a 3rd version of `range()` can be used that defines the "***step value***", which is how much the iterator increments on each iteration.

*Example*: `range(1, 18, 2)` , the iterator values would be 1, 3, 5, 7, 9, 11, 13, 15, 17.
`range(18, 2, -2)` , the iterator values would be 18, 16, 14, 12, 10, 8, 6, 4.

### Iterables

The `range()` function is a special built-in function in Python that produces a special `range` *type* of variable, which is a "virtual sequence of numbers". A `range` sequence is one example of an ***iterable***, which is any collection of values that can be looped (or *iterated*) over. Another iterable is a `str` (string). When looping over a string, the iterator variable will take the value of each individual character one at a time. For example:

```
some_str = input("Enter something: ")
for char in some_str:  ← Note that you do not need to use range()
    print(char)        ← This example will print one character per line
```

### Should I Use A For-Loop Or A While Loop?

The for-loop structure is quite tidy for many loops that you will write. In general, you will use a for-loop when you know ***exactly*** how many times a loop will execute. Examples of this would be finding the average of 10 marks, looping over every character in a string, or adding up the numbers from 10 to some number input by the user (even though you don't know this exact value, a variable will store this value and you can use that variable in your range).

The while-loop structure will generally be used when you ***do not*** know exactly how many times a loop will execute. An example of this you have seen is asking the user to input numbers, until they enter "-1". Another example of this would be a bank machine that allows a user to perform multiple transactions until the user says they are done. A cash register is another example, where you do not know in advance how many items there are to scan. The termination of these `while` loop examples is always based on a *condition* becoming `False`.

## Exercises

**1.** Write a function called `range_sum()` that takes 2 integers as arguments. Returns the sum of all the integers between (and *including*) those two integers. Note that the user could pass the higher/lower integers in either order. For example, if they pass 2, and -8, the program would sum the following: $-8 + -7 + -6 + -5 + -4 + -3 + -2 + -1 + 0 + 1 + 2$ which is $-33$.

**2.** Write a function called `frac_sum()` that takes an integer argument '$n$', and returns the sum of $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$.

*Sample call*: `print(frac_sum(50))` prints `4.499205338329423`

**3.** Write a function called `frac_power_sum()` that takes an integer argument '$n$', and returns the sum of $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{2^n}$

*Sample call*: `print(frac_power_sum (50))` prints `0.9999999999999991`

*Note that the denominators are all powers with base-two ($2^1$, $2^2$, $2^3$, $2^4$, etc). Can you see what the iterator should be here?*

**4.** Write a function called `pi_ish()` that takes an integer argument '$n$', and returns the sum of $\frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + \frac{4}{k}$., where $k$ is the largest odd number less than or equal to $n$. Note the alternating signs ($+$, $-$) between each term.

*Sample call*: `print(pi_ish(1000000))` prints `3.141590653589692`

**5.** Write a function called `num_spaces()` that takes a string argument and returns the number of *space* characters in that string. *Note: **don't** use string methods (e.g. count), if you know those.*

**6.** Modify your program from the `while`-loops exercises 1 – 3 which asked for a user's marks. Instead of asking marks until the user enters 'quit', first ask the user how many marks they are going to enter, and then ask them for that many marks (one at a time). Produce the same output.

**Nested Loops:**

Earlier we saw how we could have if-statements inside of if-statements. We can also have if-statements inside of loops, and loops inside of if-statements, and loops inside of loops!

*Example*: Output a grid with 4 rows and 7 columns of "#" signs.

```
# for each row in the grid...
for row in range(4):
    # for each column in the row...
    for col in range(7):
        print("#", end = "")  # add one more '#', keep cursor on same line!
    print("")# move output cursor to next line
```

**7.** Write a `times_table()` function that takes an `int` 'n' argument, and prints an n-times table to n x n.

Example: if argument was 4:

```
1  2  3  4
2  4  6  8
3  6  9  12
4  8  12  16
```

…though *nicer* output would be:

```
1   2   3   4
2   4   6   8
3   6   9   12
4   8   12  16
```

**8.** Write a `dice_totals()` function that takes two `int` arguments each representing the number of sides on a dice. The function will add the total sum of all possible pairings of values on the two dice. For example, if it was a **6** and a **4** sided dice you could roll 1+1=**2**, 1+2=**3**, 1+3=**4**, 1+4=**5**, 2+1=**3**, 2+2=**4**, … 6+3=**9**, 6+4=**10**. The total sum returned in this case would be **144**.

**Additional for-loop Exercises – looping over strings**

The following exercises all involve looping over characters in a string. An example of this is shown in the middle of Sheet 39.

Here is an example that determines how many digits in a numeric string of digits are above 5

```
def count_above_5(number_string: str) -> int:
    '''
    Returns the number of digits in number_string that are above 5
    '''
    counter = 0
    for digit in number_string:
        if int(digit) > 5:
            counter += 1
    return counter
```

This function could be tested using commands such as the following (note the arguments are **strings**):

```
print( count_above_5("12345678987654321") )      → outputs 7
print( count_above_5("11223300") )               → outputs 0
```

We will see in the near future how there are more advanced techniques for working with strings. For now I do not want you exploring those. Solutions to these exercises should all be based on what we have seen.

**Exercises:**

**9.** Complete the function **count_letter()** that takes a string of text and a single character string as arguments. It will return how many times that letter appeared in the given text. You do not need to handle upper & lowercase versions of the same letter.

***Function header***: `def count_letter(word, letter):`

***Sample call***: `print( count_letter("richview collegiate", "e") )` → prints **3**

**10.** Complete the function **count_letters()** that takes a string of text and a string of characters to look for as arguments. It will return how many times any of those letters appeared in the given text. You do not need to handle upper & lowercase versions of letters.

***Function header***: `def count_letters(word, letters):`

***Sample call***: `print( count_letters("richview collegiate", "lie") )` → prints **8**

**11.** Complete the function **sum_all_digits()** that takes a string of numeric characters and returns the sum of all the digits. You do not need to handle invalid digit characters.

***Function header***: `def sum_all_digits(number_str):`

***Sample call***: `print( sum_all_digits("123987") )` → prints **30**

**12.** Complete the function **sum_alternate_digits()** that takes a string of numeric characters and returns the sum of every other digit (starting from the first). You do not need to handle invalid digit characters.

***Function header***: `def sum_alternate_digits(number_str):`

***Sample call***: `print( sum_alternate_digits("123987") )` → prints **12** (1+3+8 = 12)

***Extra Challenge*:** Can you make exercise 3 handle 'negatives' in the string? For example, `sum_all_digits("1-239-87")` would return 10, since 1–2+3+9–8+7 = 10