BUILDING UP STRINGS

Recall that a string (a `str`) is a sequence of characters. You have been using strings a lot, whenever you have put quotes around something! We will discuss `strings` a lot more soon, as they are quite different to the other variable types we have been using up until now.

Concatenation:

When using the '+' operator with `strings`, it does not mean addition, it means "stick together", or "***concatenate***". You can only concatenate `strings` with `strings`!

Example:

```
x = "55"
y = "15"
print(x + y)  # prints 5515, not 70
```

Often we want to build up a lot of stuff to output at once. For example, you may want to build up an entire grocery receipt to print all at once instead of printing one line at a time. In order to build-up a larger piece of output you can use a `str`, which can act like a container that you repeatedly add stuff to (or stick on to the end of → *concatenation*!).

Example:

```
def number_chain(max_num: int) -> str:
    '''
    Return a string with the numbers from 1 to max_num all in a row, separated
    by '-' characters (ensuring each number takes up 3 characters).
    Ex: if max_num is 11, returns "1--2--3--4--5--6--7--8--9--10-11-"
    '''
    message = ""  #this is the empty string... NOT a blank space!
    for num in range (1, max_num+1):
        message += f"{num:-<3}"  #note that we can use f-strings!
    return message
```

We could call this function in a print statement: `print(f"=>{number_chain(16)}<=")`
which would output: `"=>1--2--3--4--5--6--7--8--9--10-11-12-13-14-15-16-<="`

Compare the following 2 examples:

```
for num in range(100000):
      print("ICS3U1!")
```

```
out = ""
for num in range(100000):
    out += ("ICS3U1!\n")
print(out)
```

Each of these essentially does the exact same thing. The one on the left executes a `print` statement 100000 times. The one on the right executes a `print` statement just ***once*** but builds up the string a little more on each iteration by replacing it with a new one.

Escape Sequences:

When building up `str`s there are special characters that you may want to use such as a tab `("\t")`, a new-line `("\n")`, a quote `("\"")`, and a backslash `("\\")`. You can use these just like any other characters. They must be *inside* the quotes in order to appear properly:

Example:

```
print("/\\\tHello\nMy name is: \"Adam\"")
```

…would print:

```
/\    Hello
My name is: "Adam"
```

Exercises:

*For each exercise, remember to think about if you should be using a **while** loop, or a **for**-loop!*

1. Repeatedly ask the user for a number, until the user enters 0. Output the set of ***positive*** input numbers *only*, separated by a comma, in a ***single*** output statement.

Example: If the user input 8, then 5, then -6, then -7, then 4, then 2, then 0, the output would be:

```
8,5,4,2,
```

Extra Add-on: Try to do this so that there is no extra comma at the end of the line.
Hint: think of how you can use an if-else.

2. Write a function called **`text_block()`** that takes a single string as an argument. It will then first determine the number of characters in the string (*see an earlier for-loop exercise example*), then will build and ***return*** a string where each letter in the word is on its own 'line', and each line has that letter repeated based on the string length.

Example: calling **`text_block("PYTHON")`** would return the string:

```
"PPPPPP\nYYYYYY\nTTTTTT\nHHHHHH\nOOOOOO\nNNNNNN\n"
```

When this string is ***printed*** it would appear as shown here →
e.g. **`print(text_block("PYTHON"))`**

```
PPPPPP
YYYYYY
TTTTTT
HHHHHH
OOOOOO
NNNNNN
```

3. Write a function called **`number_trapezoids()`** that takes an `int` argument (n) and returns a string in the format $/1n\backslash/2n\backslash/3n\backslash..../10n\backslash$, but where the values of $1n$, $2n$ etc are calculated.

Example: calling `number_trapezoids(5)` would return the string:

```
"/5\/10\/15\/20\/25\/30\/35\/40\/45\/50\"
```

4. DNA strands are made up of 4 different 'bases': guanine, adenine, thymine, and cytosine. These are denoted by upper-case 'G', 'A', 'T', and 'C'. A sample DNA strand might look like "GTAACGTA". When DNA strands pair up to form their double-helix shape, they pair with their *complementary* strand using *base pairs*. The base pairings are guanine-cytosine, and adenine-thymine. Write a function called **`get_complement()`** that takes a DNA strand as an argument, and returns the complement of that DNA strand (as a string).

Example: calling **`get_complement("ATGCGGAT")`** would return **`"TACGCCTA"`**

5. Write code that repeatedly asks the user for an id#, then a first name, then a last name, until the user enters *nothing* for an id number (i.e. just hits the Enter key). Once the user has done this, output a ***table*** of all the ID's and the names, in the exact format as below, in a singly executed `print` statement:

```
1. 123456789      Foster, Adam
2. 987654321      Smith, John
3. 111222333      Johnson, Jane
```
...*etc*

6. Write a function called **`factor_chain_2_3()`** that takes an `int` as an argument, and returns a string as defined below. The function will build up a string showing the product of 2's and 3's that the number has as its prime factors. If the number has ONLY 2's and 3's as prime factors, return this chain string. Otherwise, return `"N/A"`.

Example: If the argument was 24, the returned string would be: `"2 x 2 x 2 x 3"`
Example: If the argument was 60, the returned string would be: `"N/A"`

Extra *Building Up Strings* Exercises:

1. Write a function called **`even_odd()`** that takes an integer as an argument and then generates a string of that many ***random*** digits (between 0 and 9). The string should be built so that all the even digits are 'first' (on the left) then the odd digits are 'last' (on the right).
For example: **`even_odd(9)`** might return the string **`'020649177'`**, or **`'863519773'`**, etc.

2. Write a function called **`boxify()`** that takes two arguments: a string of characters and an integer 'width'. The function will take all the characters in the given string and will build & return a 2-dimensional version based on the given width.
For example: **`boxify("1234567890", 3)`** would return the string **`'123\n456\n789\n0'`**. Notice the placement of the **`\n`** characters. When that string is printed it would look like:

```
     123
     456
     789
     0
```

Another example:

```
    text = 'We acknowledge we are hosted on the lands of the Mississaugas of the
            Anishinaabe, the Haudenosaunee Confederacy and the Wendat.'
    print(boxify(text, 12))
```

would output:

```
    We acknowled
    ge we are ho
    sted on the 
    lands of the
     Mississauga
    s of the Ani
    shinaabe, th
    e Haudenosau
    nee Confeder
    acy and the 
    Wendat.
```