Session 4: While loops and Try

While loops

Let's look at another way to print the square:

```
print("**********")
i = 0
while i<8:
    print("*        *")
    i+=1
print("**********")
```

The idea with the while loop is very simple: the code in the body of the loop will be executed repeatedly, while the condition remains true - in other words, until the condition becomes false. In each iteration, i gets incremented by 1, until it reaches 8 and we exit the loop.

As a result, the loop is run 8 times: for `i=0`, `i=1`, etc. up until `i=7`. On the 8th run, at the start of which `i=7`, `i` will be incremented to 8 in the body of the loop, which makes the condition false.

We can use while loops to do more things. Imagine we wanted to create a simple security program that asks our user for a password until they enter the correct one. One way to do this is:

```
SECRET_PASSWORD = "pass123"

user_input = input("Please enter a password: ")

while user_input != SECRET_PASSWORD:
  print("Access denied: wrong password.")
  print() # Print a newline. This is just here to make things look nicer.
  user_input = input("Please enter the password: ")

print("Access granted.")
```

In this case, while the user keeps entering the wrong password, the program will keep prompting them for a new one.

Now, if the user enters the correct password on their first try, the body of the loop will not be executed. So we can't use a for loop here, because we don't have any list to iterate down, nor do we know exactly how many times the body of the loop has to run.

In conclusion, while loops are used when the body of the loop has to loop an unknown number of times.

For loops and while loops

This for loop that calculates 1 + 2 + ... + n

```
sum = 0
for i in range(1, n + 1):
  sum += i
```

can be turned into a while loop:

```
sum = 0
i = 1
while i < n + 1:
  sum += i
  i += 1
```

In the for loop, the loop variable `i` would be automatically increased with each pass through the loop. But with a while loop, we have to do that ourselves, writing at the start `i = 1` and in the middle `i += 1`. So the while loop looks a bit more cumbersome.

On the other hand, there are many things that for loops are just not built for (pun intended). Recall this code that prints the items in a list backwards:

```
items = ["first", "second", "third"]

n = len(items)
for i in range(n):
    print(items[n - 1 - i])
```

Using a while loop, this becomes quite a lot simpler, because instead of fiddling around with subtracting stuff in the list index, we can just iterate backwards.

```
items = ["first", "second", "third"]

n = len(items)
i = n - 1
while i >= 0:
    print(items[i])
    i -= 1
```

Here are some more examples of for loops and while loops:

For loops	While loops
Adding 3 teaspoons	Adding to taste
Doing 10 reps	Training to failure
Watering plants	Picking a basketful of apples

For loops	While loops
Looking at every item in a list	Looking for a specific item in a list

How not to brick your device

What goes wrong if you try to calculate the product of the first n natural numbers with this code?

```
n = int(input("Please enter a number: "))

product = 1
i = 1
while i <= n:
  product *= i

print("The product of the first " + str(n) + " natural numbers is " +
str(sum))
```

We forgot to increment our `i`. We don't have an `i += 1` in our loop. This means that the loop will run with `i=1`, `i=1`, `i=1`, ... and not stop until we interrupt the program.

Everybody makes this mistake sometimes, from the most casual of hobbyists to the highest paid of professionals. **Keep this in mind if your program seems to "hang".**

To interrupt a program that's currently hanging:

- On Google Colab, just press the "stop" sign in the code's cell.
- In Visual Studio Code or any other editor with a command line, try pressing Ctrl-C.

Worked Exercise 1. Digit Count

Let's write a program that, when given an integer, tells us how many digits long that integer is.

```
n = int(input("Enter a number: "))

length = 0
while ________:
    ________

print("Your number is " + str(length) + " digits long")
```

The insight here is that the number of digits in a number is equal to the number of times it needs to be divided by 10 until it drops below 1. For example, 87 / 10 = 8.7 but 87 / 100 = 0.87 < 1; since we needed to divide 87 by 10 twice to put it below 1, we know that 87 has two digits.

```
n = int(input("Enter a number: "))
```

```
length = 0
while n >= 1:
    n /= 10
    length += 1

print("Your number is " + str(length) + " digits long")
```

Are we happy with this solution?

Well... we're told that the user will give us an *integer*, and integers can be negative. This solution doesn't handle negative numbers well, because no matter how many times you divide a negative number by 10, it stays negative, which means it stays less than 1. Thankfully we spotted that potential infinite loop, or our code would have made someone's computer hang.

Here's an adjustment that handles negative numbers by looping so long as n is at least 1 OR at most -1:

```
n = int(input("Enter a number: "))

length = 0
while n >= 1 or n <= -1:
    n /= 10
    length += 1

print("Your number is " + str(length) + " digits long")
```

Exercise 1

For each piece of code here, identify the difference between it and the previous code segment. Predict its result, and then run it. Is the output what you expected?

```
# a)
i = 8
while(i>0):
  print(i)
  i-=1
```

```
# c)
i = 8
while(i>0):
  i-=1
  print(i)
```

```
# d)
i = 8
```

```
while(i>=0):
  i-=1
  print(i)
```

```
# e)
i = 8
while(i>0):
  i-=3
  print(i)
```

```
# e)
i = 8
while(i>0):
  i = 2 * i - 9
  print(i)
```

Exercise 2. High-Pass

Create a program, which repeatedly asks the user to input a number until their number is greater than 9. A run of your program might look like:

```
Please enter a number: 5
That number is too small!
Please enter a number: 9
That number is too small!
Please enter a number: 10
That's a good number!
```

Exercise 3. Positive-Pass

Earlier in the notes we saw this loop that calculates the product of the first n positive numbers:

```
n = int(input("Enter a number: "))

product = 1
i = 1
while i <= n:
  product *= i
```

Modify the code to give an error message if user inputs a negative number, and asks the user to input again.

P.S. Don't brick your device!

```
Sample:
Please enter a non-negative integer: -99
Invalid number! Please enter again.
Please enter a non-negative integer: -3
Invalid number! Please enter again.
Please enter a non-negative integer: 4
24
```

Exercise 4. Checkout

This program represents a checkout machine.

Write a program that lets the user input as many numbers as they like. Then, when they enter an empty string ("") into the program, print back the sum of the numbers they've entered.

```
Add: 2
Add: 10
Add: 18
Add: 70
Add: 5000
Add:
Your total is: 5100
```

(In an actual checkout machine, instead of the user entering prices into the till, the barcode scanner does that for them, of course.)

Exercise 5. Zeroing

This is a program that finds use when calibrating some sensors, and also when companies comply with legal requests to delete sensitive data.

Write a program that sets every entry in a list to 0. In other words, starting from this code:

```
numbers = [1.5,-2.2,-0.3,0.6,8,-0.2] # actual numbers may vary

# your code here

print(numbers)
```

your program should output

```
[0, 0, 0, 0, 0, 0]
```

(b) (Free Response) You can also do this exercise with a for loop. Which one do you think fits better here? Explain.

Exercise 6. Cutoff (Hard)

Given a list filled with words, write a program that slices off everything after the word "not". For instance, given the list of words:

```
words = ['slightly', 'greatly', 'passionately', 'madly', 'not', 'cookies',
'cream']
```

your program should print something along the lines of

```
['slightly', 'greatly', 'passionately', 'madly']
```

Here's some code that might be helpful:

```
slice_index = 0
while ____________:
  slice_index += 1

words = words[0:slice_index]
```

For top marks, make sure your code works when given a list of words that does NOT contain the word "not":

```
words = ['doe', 'ray', 'me', 'far', 'sew']
```

should print

```
['doe', 'ray', 'me', 'far', 'sew']
```

Exercise 7. Unlimited Echo (Hard)

Write a program that lets the user shout as many times as they want. Then, when they stop shouting (by entering an empty string "" into the program), the program should shout everything they said back at them in reverse order:

```
Shout: it's fun
Shout: to stay
Shout: at the
Shout: Y
Shout: M
Shout: C
Shout: A
```

```
Shout:
A
C
M
Y
at the
to stay
it's fun
```

This exercise is quite similar to Exercise 4.

(Hint: You'll want to collect the shouts into a list.)

Exercise 8. Average (Hard)

Write a program that calculates the average of numbers. The program should terminate when the empty string ("") is inputted.

```
Please enter a number: 1
Please enter a number: 3
Please enter a number: 5
Please enter a number: -1
Please enter a number:
The average was: 2.0
```

This exercise is quite similar to Exercise 4.

You can find all the solutions here.

Extension A. Completing the shopping list

Return to your code from Session 3 Exercise 4 (the shopping list). Using a while loop, modify your program so that the user doens't have to enter the size of the shopping list in advance; instead, once they enter the empty string (""), the program should exit the loop (and proceed to print out the items in the list as before).

Worked Exercise 2. Adding 1

The problem: Given a number represented by its digits, write a program that adds 1 to that number.

```
number = [9,5,8,9,9] # representing the number 95899
# add 1 to number here...
print(number) # you should get [9,5,9,0,0]
```

Nine times out of ten, when we add 1 to a number represented like this, the only thing that changes is the smallest digit. The remaining one time out of ten is when the last digit is a 9. In this case, we then have to add 1 to the tens digit - which can propagate up through the number.

The program we're looking for will be something like

```
add_index = len(number) - 1 # the index of the last item in number
while number[add_index] == 9: # if the current digit is a 9:
  number[add_index] = 0 # add 1 to it, bringing the 9 to a 10
  add_index -= 1 # carry the 1 to the higher place value

number[add_index] += 1
```

Are we happy with this solution?

Well... what if the number is all nines?

```
number = [9,9,9,9,9]
```

In this case, our `add_index` will decrease from 4 to 3, 2, 1, 0; then, since number[0] is still 9, `add_index` will decrease to -1. Then, the next time through the loop, when Python tries to access number[-1], well... you'll have to read the answer to Session 3 Exercise 2 to find out 😃

All that to say, we need to check if `add_index` has gone below 0, and if it has, we add a 1 to the front of the number. In code, this looks like this:

```
if add_index < 0:
  number.insert(0, 1) # add a 1 to the front of the number
else:
  number[add_index] += 1
```

We also need to stop the loop if add_index goes into the negatives. This means we need to make the condition in our while loop a bit more complicated:

```
# stop the loop if number[add_index] != 9 OR if add_index < 0
# in other words: run the loop while number[add_index] == 9 AND add_index
>= 0
while add_index >= 0 and number[add_index] == 9:
  # etc...
```

```
add_index = len(number) - 1
while add_index >= 0 and number[add_index] == 9:
  number[add_index] = 0
  add_index -= 1

if add_index < 0: # if the number is all nines:
  number.insert(0, 1) # add a 1 to the front of the number
else:
  number[add_index] += 1
```

```
print(number)
```

Try

In the password checking loop at the top of this section, we saw how a program could protect itself from incorrect input, to some extent. But the program still breaks if the user does something wildly incorrect such as

```
favorite_number = int(input("Please enter your favorite number."))
# User enters the word "Puppy"
# Program crashes!
# ValueError: invalid literal for int() with base 10: 'Puppy'
```

The try keyword is a stronger way of protecting a program against incorrect input. Programmers tend to worry a lot about making sure nonsensical user input can't mess up their precious programs (but for all that worrying, they get it wrong anyway)

For example, here's some code that makes sure that checks if the user inputted an integer.

```
user_input = input("Please input an integer.")

try:
    user_input = int(user_input)
    print("That was an integer.")
except:
    print("That was not an integer.")
```

Inside the `try` block, Python *tries* to convert the user's input into an integer. If it suceeds, the rest of the code in the `try` block can run, which means Python prints "That was an integer." But if the conversion fails, Python panics.

Usually, when Python panics, you get an error message on your screen. This is called **throwing an exception**. But because the error happened inside a `try` block, Python understands that this error was accounted for, and calmly moves on to the `except` block, where it prints "That was not an integer."

Worked Exercise 3: Enforce Integer Input

Here's how we can use `try` to wait until the user inputs an integer.

We start with the code from before.

```
user_input = input("Please input an integer.")

try:
  user_input = int(user_input)
  print("That was an integer.")
except:
  print("That was not an integer.")
```

If the try block fails, we want to ask the user for another input, and enter the try block again. Sounds like a job for a while loop.

```
user_input = input("Please input an integer.")

while some_condition:
  try:
    user_input = int(user_input)
    print("That was an integer.")
  except:
    print("That was not an integer.")
```

What's the condition we want to set here? It's kind of hard to tell. We could use a variable to tell us whether we can exit the loop or not:

```
user_input = input("Please input an integer.")

successful_conversion = False
while not successful_conversion:
  try:
    user_input = int(user_input)
    print("That was an integer.")
    successful_conversion = True
  except:
    user_input = print("That was not an integer. Please try again: ")
```

But that's kind of clunky, isn't it? Okay, how about this? Let's introducte a new function: `type()`.

The `type` function behaves exactly as you would expect:

```
type(5) == int
type(5.0) == float
type("5.0") == str
type([5, 5.0]) == list
```

Using `type()`, we can complete our integer converter very cleanly.

```
user_input = input("Please input an integer.")

while type(user_input) == str:
  try:
    user_input = int(user_input)
  except:
    user_input = input("That was not an integer. Please try again: ")
```

Exercise 9. Bad at instructions (Free Response)

Add a counter to the "enforce integer input" code to show the user how many times their input has been incorrect.

Whether you show the user their incorrect count every loop, or only once (just at the end), is up to you, and the format of the statement is also up to you. 😃

Exercise 10. Finally (Free Response)

Read the following code and predict its result.

```
x = input("Enter a number: ")
try:
  print(1 + int(x))
except:
  print("That was not a number.")
finally:
  print("Goodbye!")
```

Now run the code. Try inputting numbers and not-numbers. What happens? Is this what you expected?

Try to think of a situation where the `finally` keyword might be useful. (If there are any.)

Exercise 11. Your numbers are numbered

Create a program that accepts user inputs until the user has inputted two integers.

```
Input: 4
Input: score
Input: and
Input: 7
You have inputted the maximum allowed number of integers. Goodbye!
```

Exercise 12. Pick from eight

This code represents offering a number of options to the user. These might represent menu items, insurance plans, or anything in between.

Create a block of code that only accepts an input that is a number from 1 to 8.

```
```

```
Please choose an option: 9
That is not an option. Please try again: wow
That is not an option. Please try again: 2
Option 2 selected!
```

Exercise 13. One of each

Create a program that accepts two user inputs and makes sure that exactly one of them is an integer.

```
The first input: 4
The second input: 6
That is too many integers. Please try again.
The first input: spoon
The second input: fork
That is too few integers. Please try again.
The first input: 42
The second input: life
Exactly one of these is an integer!
```

Exercise 14. You number your numbers (Hard)

(a) Edit your program from Exercise 11 so that it can accept user inputs until the user has inputted a number of integers that they input.

```
Number of integers to be inputted: spoon
That is not an integer. Please try again: 3
Input: 1
Input: 2
Input: buckle my shoe
Input: 3
You have inputted the maximum allowed number of integers. Goodbye!
```

(b) What will happen if the user inputs a negative number? What about zero? What would some reasonable behaviours for your program be? Should it ask them for another number, or immediately say goodbye? Explain your choice, and implement that change in your code.