Introduction

It's time to start learning how to collect information from a user. So far our programs have been one sided, we have been outputting information and results to the user. The input of our programs has been hard wired in. In this lesson and those to come we are going to learn how to receive input from a user, for example, name, age, location. In Python input from a user is received using the *input()* function.

Python's input() function

When Python's *input()* function is called within a program, the program is paused and waits for a user to enter text. Once the input has been received, Python assigns that input to a variable which makes it easy for us to work with. Let's take a look:

In [1]:

```
message = input("Tell me your name and I will output it to the screen: ")
print(message)
```

```
Tell me your name and I will output it to the screen: Tony
Tony
```

In this code example, the *input()* function takes one argument which is the prompt that the user sees on screen. When the prompt has been displayed, the program waits for the user to respond, which in this case means the user entering their name. The program will continue only after the user presses the enter key. The response is assigned to the variable *message*. We then use a *print* statement to display the input back to the user.

> Not all code editors run programs that can prompt the user for input. If your program involves asking the user for input you may need to run it from the terminal.

Let's take a look at another example:

In [2]:

```
name = input("Please enter your name to begin: ")
print(f"\nHello, {name}")
```

```
Please enter your name to begin: Tony

Hello, Tony
```

There will be times, such as explaining something to the user, when you need to use a long prompt. In these cases you can assign your prompt to a variable and pass that variable to the *input()* function. Let's take a look:

In [3]:

```
prompt = "Welcome to the setup of your new software."
prompt += "\nEvery time you run this software we can greet you by your name."
prompt += "\nWhat is your name?"

name = input(prompt)
print(f"\nHello, {name}")
```

```
Welcome to the setup of your new software.
Every time you run this software we can greet you by your name.
What is your name?Tony

Hello, Tony
```

Accepting numerical input

When Python's *input()* function receives input from a user, it interprets everything it receives as a string. Let's take a look:

In [4]:

```
age = input("What age are you? ")
```

```
What age are you? 29
```

In [5]:

```
type(age)
```

Out[5]:

```
str
```

In this example we have entered the number 40 as user input and assigned it to the variable *age*. We then asked Python to tell us what that data type is, using the *type()* function. Python tells us that the variable *age* is of type 'str' or string. If you are running these examples in a terminal you will see the number outputted wrapped in single quotes, '40'. This is the terminal telling you the output is a string. If all you are doing is displaying the input back to user then you won't have any

problems. However, if you try and use the input as a number you will run into difficulty. Let's take a look:

In [7]:

```
score = input("What is your high score? ")
score >= 100
```

```
What is your high score? 100

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [7], in <cell line: 2>()
      1 score = input("What is your high score? ")
----> 2 score >= 100

TypeError: '>=' not supported between instances of 'str' and 'int'
```

Here we are returned with a *TypeError*. Why? Python is performing a numerical comparison. It is comparing the user input of '100' with the number 100. Python can't compare a string with a number because they are two different data types.

We can fix this error by using the *int()* function. This function tells Python to treat the input as a number. Let's see how it works:

In [8]:

```
score = input("What is your high score? ")
score = int(score)
score >= 100
```

```
What is your high score? 100
```

Out[8]:

```
True
```

In this code example, we ask the user what is their high score. Python will firstly interpret the user input as a string. At line 2 we convert the string into a number using the *int()* function. Python then runs the numerical comparison test to see if the score that that user inputted is greater than or equal to the number 100. Let's build on this example:

In [10]:

```
score = input("What is your high score? ")
score = int(score)

if score >= 50:
    print("Congratulations, you've made it onto the leaderboard!")
else:
    print("Sorry your score is not high enough. Please try again.")
```

```
What is your high score? 75
Congratulations, you've made it onto the leaderboard!
```

In [11]:

```
type(score)
```

Out[11]:

```
int
```

Here our program is able to perform the numerical comparison without any issues because at line 2, *score = int(score)* converts the user input into a number.

The Modulo Operator

Our next Python topic always inspires a lot of questions from my students and rightly so. The modulo operator (%), divides one number by another and returns the remainder:

In [12]:

```
5 % 3
```

Out[12]:

```
2
```

In [13]:

```
8 % 7
```

Out[13]:

```
1
```

In [14]:

```
4 % 2
```

Out[14]:

```
0
```

The modulo operator simply tells you what the remainder of one number divided by another is. Simply as it may seem the modulo operator can be used in very powerful ways. When one number is divided by another and the remainder is 0, the modulo operator always returns 0. Let's take a look:

In [16]:

```
number = input("Input a number to check if it is an even or odd number: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is uneven.")
```

```
Input a number to check if it is an even or odd number: 7

The number 7 is uneven.
```

This piece of code works because even numbers are always divisible by two. If the modulo of a number is zero, the number is even. You will see this little piece of code in a lot of programs as you learn more Python.

Exercises

- Write a program that asks a user what type of book they would like to read. Print a message out for the book type, for example, "I think we do have some books on history. Let me see if I can find you one."
- You are booking dinner for your company party. Write a program that asks the user how many people will be attending the dinner. If the answer is more than six, print a message that informs there user that there are no tables available. If the answer is six or less inform that user that their booking is confirmed.
- Create a program that asks a user for a number and then let the user know if the number is odd or even.

While loops

We've learned about *for* loops which take a collection of items and runs a block of code once for each item in the collection. *While* loops run as long as, or *while* a condition is true.

We can use a *while* loop to create a simple counter:

In [17]:

```
number = 1

while number <=10:
    print(number)
    number += 1
```

```
1
2
3
4
5
6
7
8
9
10
```

At line 1 we create a new variable called *number* and assign it the value 1. This will be the number that we'll start counting at.

At line 3 we define a *while* loop. This loop will keep counting while the value of *number* is less than or equal to 10. As you can see at the end of the first line of the *while* loop we have a double colon (:).

At line 4 we have an indented *print* statement, which means it is inside the loop. This *print* statement prints to the screen the value of the variable *number*. At line 5 we then add 1 to that value. How are we adding 1 to the value of the variable *number*? The += operator performs the action of *number = number + 1*.

Python repeats the loop while the condition is less than or equal to 10, <= 10. Python prints 1 and then adds 1, which makes the value of the variable *number* 2. 2 is less then 10, so Python prints 2 and adds *1* again, which makes the value of the variable *number* now 3. This loop is repeated until the value of the variable *number* is 10.

while loops are used frequently in Python. For example how does a program know to keep running if a user has not pressed the *quit* button?

Choosing when to quit a program

For our next code example let's define a quit value.

In [18]:

```
message = "\nTell me your name and I will output it to the screen:"
message += "\nEnter 'quit' to end the program. "

name = ""
while name != 'quit':
    name = input(message)
    print(name)
```

```
Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. Tony
Tony

Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. Frank
Frank

Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. quit
quit
```

In this code example we define a prompt which gives a user some instructions on what to do when inside the program. We then create a new empty variable called *name* at line 4. This variable will store whatever value the user enters. Why is the variable *name* empty? Python needs something to check the first time it reaches line 5, the *while* line. The first time the program runs and Python reaches the *while* statement it needs to compare the value of *name* to 'quit', but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give *name* an initial value. Even though this initial value is just an empty string, it allows our program to perform the comparison that makes the *while* loop work. The *while* loop at line 5 will run so long as the value inside the variable *name* is not 'quit'.

This program works well, except that it prints the word 'quit' as if it were an actual message to display to the screen.

In [19]:

```
message = "\nTell me your name and I will output it to the screen:"
message += "\nEnter 'quit' to end the program. "

name = ""
while name != 'quit':
    name = input(message)

    if name != 'quit':
        print(name)
```

```
Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. Tony
Tony

Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. Frank
Frank

Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. quit
```

Now, our program performs a quick check before displaying the message and only prints the *name* variable if it does not match the quit value.

Using a flag

Previously, our programs have performed certain tasks while a given condition was true. This works well for small, simple programs like the ones we have been creating but what happens when our programs start to get bigger? We could have programs in which several different events cause it to stop running.

If we have a word processing program we could quit the program by selecting the close icon, selecting File >> Quit, or maybe a combination of keys.

For a program that should run only as long as many conditions are true, we can define one variable that determines whether or not the entire program is active. In Python this variable is called a *flag* and it acts a signal to our programs. We can write programs that continue to run while the flag is set to *True* and stop running when any of several events sets the value of the flag to *False*. As a result, our overall *while* statement needs to check only one condition.

Let's take a look at an example:

In [20]:

```
message = "\nTell me your name and I will output it to the screen:"
message += "\nEnter 'quit' to end the program. "

active = True
while active:
    name = input(message)

    if name == 'quit':
        active = False
    else:
        print(name)
```

```
Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. frank
frank

Tell me your name and I will output it to the screen:
Enter 'quit' to end the program. quit
```

At line 4 we have a new variable *active* which is set to *True*. This means that our program is starting in an active state. So long as *active* is *True* the *while* loop will run, line 5.

Inside the *if* statement at line 8, we check the value of *name*. If our program detects that a user has entered 'quit' we set *active* to *False*. This will stop the *while* loop. Anything other than 'quit' and the program will continue to run.

This program makes it easier for us to add other tests that might cause the *active* state to become *False*. As just mentioned, the addition of this code is useful in complicated programs that require several ways to stop them running.

Using break to exit a loop

There will be times when you need to exit a *while* loop without running any of the remaining code, no matter the results of any conditional tests. When you need this functionality, use the *break* statement. The *break* statement directs the flow of your programs. You can use it to control which lines of code are executed and which are not. This gives you control over what code is executed and when and when it is not.

Let's take a look at an example:

In [21]:

```
message = "\nWhat day of the week is it?"
message += "\n(Enter 'quit' to exit the program.)"

while True:
    day = input(message)

    if day == 'quit':
        break
    else:
        print(f"I didn't know it was {day.title()}!")
```

```
What day of the week is it?
(Enter 'quit' to exit the program.)Monday
I didn't know it was Monday!

What day of the week is it?
(Enter 'quit' to exit the program.)tuesday
I didn't know it was Tuesday!

What day of the week is it?
(Enter 'quit' to exit the program.)quit
```

In this code example we have a *while* loop starting at line 4. A *while* loop that starts with *while True* will run forever unless it reaches a *break* statement. In this example the loop will continue running and asking the user what day of the week it is until 'quit' is entered. When the program detects 'quit', the *break* statement runs, which forces Python to exit the loop.

A *break* statement can in used in any of Python's loops.

using continue in a loop

The *continue* statement will return Python to the beginning of a loop rather than breaking out of a it completely as with the *break* statement. Let's look at an example:

In [22]:

```
number = 0
while number < 10:
    number += 1
    if number % 2 == 0:
        continue

    print(number)
```

```
1
3
5
7
9
```

In this example we first set the variable *number* to 0 at line 1. 0 is obviously less than 10 so Python enters the loop. Inside the loop we add 1 to the *number* variable. Now *number* has the value of 1. Next, at line 4 the *if* statement checks the modulo of *number* and *2*. If the result is 0 then the *continue* statement tells Python to ignore the rest of the loop and return to the beginning. If the *number* variable is not divisible by 2, the rest of the loop is executed and Python prints the number.

Avoiding infinite loops

If you don't stop a *while* loop it will continue to run forever. Let's take a look:

In [23]:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
1
2
3
4
5
```

OK, so that works. But if you leave out line 4 you might be waiting a while for your program to finish.

In [ ]:

```
x = 1
while x <= 5:
    print(x)
```

In this code example the value of *x* is 1 and will never change. Because *x* will never change the conditional test *x <= 5* is always true and so the *while* loop will run forever.

If you write a program, and it gets stuck in an infinite loop, we've all done it, you can exit it by:

- pressing the stop icon in Jupyter
- pressing **CTRL-C** in terminal
- just close the terminal window
- close or quit the code editor

Exercises

- Write a loop that asks a user what extras they would like with their car. As they enter each extra print a message letting the user that their extra has been ordered. Provide the user with a way to quit the loop
- You have been contracted to build the pricing system for an airline company. They have 3 destinations, America for $100, Europe for $200, and Australia $300. Write a loop which asks the user their destination and then tell them how much it will cost.

While loops with lists and dictionaries

Our programs so far have used only one piece of information, the user's input. This is not scalable when we start to work with large numbers of users and information. To help us we can use lists and dictionaries with *while* loops.

To modify a list as you work through it, use a while loop. A list shouldn't be modified inside a *for* loop as Python will have problems keeping track of the items inside the list. Using *while* loops with lists and dictionaries will allow us to collects, store and organize lots of input to perform additional tasks on later.

Moving items from one to list to another

In the example below we have users who signed up to a newsletter via a website using their email addresses. In order for the users to start receiving the newsletter they need to confirm their email addresses. This scenario requires two lists. The first to hold unverified email addresses and a second list to store users who have verified their email addresses. Let's take a look at how we could do this:

In [25]:

```
# Start with email addresses that need to be verified
# and a list to hold confirmed email addresses
unconfirmed_email_addresses = ['tony@example.com', 'jane@test.com', 'frank@test.com']
confirmed_email_addresses = []

# Confirm each email address until there are no more left
# Move each confirmed email address to the list of confirmed email address
while unconfirmed_email_addresses:
    current_email_address = unconfirmed_email_addresses.pop()

    print(f"Confirming email address: {current_email_address}")
    confirmed_email_addresses.append(current_email_address)

# Display all confirmed email addresses
print("\nThe following email addresses have been confirmed:")
for confirmed_email_address in confirmed_email_addresses:
    print(confirmed_email_address)
```

```
Confirming email address: frank@test.com
Confirming email address: jane@test.com
Confirming email address: tony@example.com

The following email addresses have been confirmed:
frank@test.com
jane@test.com
tony@example.com
```

Nice! 👍

At line 1 we have a list of unconfirmed email addresses. We have an empty list created a line 2 to hold confirmed email addresses. The *while* loop at line 8 runs as long as the list *unconfirmed_email_addresses* is not empty. The *pop()* method at line 9 removes unconfirmed email addresses one at a time from the end of the list *unconfirmed_email_addresses*.

Finally we confirm each email address by printing a confirmation message and then adding it to the list of confirmed email addresses. When the list of unconfirmed email addresses is empty the loops stops and our *print* statements are displayed on screen.

Removing all instances of specific values from a list

In a previous lesson, we've used *remove()* to remove specific values from a list. This method works well when the item that you want to remove appears only once in a list. But what if it appears several times and you want to remove all of them?

Let's take a look:

In [26]:

```
employees = ['tony', 'frank', 'dina', 'beth', ' pete', 'karen', 'frank', 'frank']

print(employees)

while 'frank' in employees:
    employees.remove('frank')

print(employees)
```

```
['tony', 'frank', 'dina', 'beth', ' pete', 'karen', 'frank', 'frank']
['tony', 'dina', 'beth', ' pete', 'karen']
```

Our first list of employees at line 1 contains several instances of the name 'frank'. We print out the list at line 3 to show them. At line 5 we create a *while* loop which Python enters because it finds at least one instance of 'frank' in the employees list. Once inside the loop Python removes the first instance of 'frank' and then returns to line 5, the *while* line, it then reenters the loop when it finds another instance of 'frank' in the employees list. Python proceeds to remove each instance of 'frank' until this name is no longer in the list. When this happens Python exits the loop and prints out the new list of employees.

Adding user input to a dictionary

Let's create a program that asks new employees for their name and department. We'll store the data gathered from users in a dictionary.

In [ ]:

```
responses = {}

# Set a flag to indictate that the program is active
program_active = True

while program_active:
    # Ask for the employees name
    employee_name = input("\nWhat is your name?")
    department = input("What department have you been assigned to? ")

    # Store the user's response in the dictionary
    responses[employee_name] = department

    # Find out if they have been assigned a mentor
    repeat = input("Have you been assigned a new joiner mentor? (yes / no)")
    if repeat == 'yes':
        program_active = False

    # Close the program and output results
    print("\n---- New Employee Details ----")
    for employee_name, department in responses.items():
        print(f"{employee_name} has started in {department} and has been assigned a mentor.")
```

```
What is your name?Frank
What department have you been assigned to? Legal
Have you been assigned a new joiner mentor? (yes / no)no

---- New Employee Details ----
Frank has started in Legal and has been assigned a mentor.
```

In this code example we start by creating an empty dictionary at line 1 called *responses*. We then set a flag at line 4, *program_active* to indicate that the program is running. As long as the program_active flag is *True*, Python will run the code in the *while* loop.

Within the *while* loop, at lines 8 and 9, an employee is asked to enter their name and the department that they will be working in. The responses back from the employee are stored in the *responses* dictionary. The employee is then asked if they have been assigned a mentor at line 15. If they answer *yes*, the *program_active* flag is set to *False* and the *while* loop will stop running and the final code block at line 20 will execute. If the employee answers *no* the program enters the *while* loop again.

Exercises

- Take the last code example and modify it so that if a new employee has not been assigned a mentor they are asked to contact HR. Update the program so that it ends cleanly whether or not an employee responds with 'yes' or 'no'.
- Make a list called *training_available* and fill it with some training course names. Next create an empty list called *assigned_training*. Loop through the list of training available and print a message informing an employee that they have been assigned to a training course and give the course name. As each training course is assigned move it to the list of assigned training. After all training has been assigned print a message saying that there are no more training slots available.