Other Man: Well I'm very sorry but you didn't pay!
Man: Ah hah! Well if I didn't pay, why are you arguing?
Ah HAAAAAAHHH! Gotcha!
Other Man: No you haven't!
Man: Yes I have! If you're arguing, I must have paid.
*Other Man: Not necessarily. I *could* be arguing in my spare time.*
Man: I've had enough of this!
Other Man: No you haven't.
Man: Oh shut up!

Introduction to Software Design and Implementation

Asking a Question

There comes a time when input from the user is required to be able to collect data to process. This can be done using the input function. This function takes a string prompt as an argument. When the code is run the user sees the prompt and types in the data. input takes this data in as a **string**, i.e. a piece of text. An example of using input follows.

```
name = input("What is your name? ")
print("Hello", name + "! Have a nice day.")
```

Saving as `input.py` and running the code, the output is similar to

```
>>>
What is your name? Arthur, King of the Britons
Hello Arthur, King of the Britons! Have a nice day.
```

Notice that the `input` prompt has a space at the end after the question mark. This space is included to separate the question from the user's response in the interaction. Without it, the interaction would look less appealing to the user:

```
What is your name?Arthur, King of the Britons
```

As the example shows, strings can be joined together to form one string for use in a print function. This is very useful in situations such as this where we want to print a combination of messages and values (as in our example). The examples show both methods of joining strings together for printing. The **,** is the first one used, it joins any items together with spaces automatically placed in between. The second is the **+** symbol, it joins the items together by adding one string to the next to form a single string.

`input` Syntax
variable_name = input(*string_prompt*)

`input` Semantics
variable_name is given the string value of what the user types in after being prompted.

`print` Syntax
print(*item1, item2, ..., itemn*)

`print` Semantics
Each *`item`* is displayed in the interpreter, with a space separating each item. `print` also takes two **optional** arguments. The `sep` argument changes what the separator between the items. If not given, the items are separated with a space otherwise, the items will be separated by the given string. For example:

```
print(item1, item2, ..., itemn, sep="separator")
```

Results in the items being separated by the "separator" string instead of spaces. The `end` argument changes what is printed after all the items. Multiple `print` function calls will display output on separate lines, unless the `end` argument is changed as the default is to end prints with a newline character. For example:

```
print(item1, item2, ..., itemn, end="ending")
```

Will end the print with the "ending" string after all the items are printed.

True or False

In programing there is always a time when a test is required. This can be used, for example, to see if a number has a relationship with another or if two objects are the same.

There are several character combinations that allow for testing.

```
== is equal to
!=  not equal to
<   Less than
>   Greater than
<=  Less than or equal to
>=  Greater than or equal to
```

```
>>> 1 == 1
True
>>> 2 != 1
True
>>> 2 < 1
False
>>> "Tim" < "Tom"
True
>>> "Apple" > "Banana"
False
>>> "A" < "a"
True
>>> type(True)
<class 'bool'>
```

As can be seen these statements result in either **True** or **False**. True or False are the two possible values of the type `bool` (short for boolean). Also note that upper and lower case letters in strings are not equal and that an upper case letter is less than the corresponding lower case letter. The reason for this is that computers can only understand numbers and not any characters. Therefore, there is a convention set up to map every character to a number. This convention has become the ASCII scheme. ASCII makes the upper case letters be the numbers 65 through to 90 and the lower case letters the numbers 97 to 122.

Making Decisions

The ability to do a test has no use if it can not be used in a program. It is possible to test and execute a body of code if the test evaluates to True.

This is done using the **if** statement.

Let's start with a simple example. The following code is an example of an if statement that will display a hello message if the name input by the user is "Tim".

```
name = input("What is your name? ")

if name == "Tim" :
    print("Greetings, Tim the Enchanter")
```

Saving this code as `if.py` and running, the output from this code looks like:

```
>>>
What is your name? Tim
Greetings, Tim the Enchanter

>>>
What is your name? Arthur, King of the Britons
```

As can be seen if name does not equal "Tim" then nothing is output from the code.

If Statement Syntax

```
if test:
   body
```

If statements start with an `if` followed by a *test* and then a colon. The *body* of code to be executed starts on a new line and indented following the Python indentation rules.

Semantics

If *test* evaluates to `True` then *body* is executed. Otherwise, *body* is skipped.

What if we want to run a different block of code if the test is False?

This requires an **if, else** statement

Our example can be modified to print a different message if name is not "Tim".

```
name = input("What is your name? ")

if name == "Tim" :
    print("Greetings, Tim the Enchanter")
else :
    print("Hello", name)
```

Saving this code as `if_else.py` , the output from this code looks like:

```
>>>
What is your name? Tim
Greetings, Tim the Enchanter

>>>
What is your name? Arthur, King of the Britons
Hello Arthur, King of the Britons
```

If-Else Syntax

```
 if test:
    body1
 else:
    body2
```

The if segment of the If-Else statement is the same is for if. After *body1* of the if on a new line and de-dented is an else followed by a colon. Then *body2* starts on a new line and indented again following the Python indentation rules.

Semantics

If *test* evaluates to `True` then *body1* is executed. Otherwise *body2* is executed.

It is also possible to carry out multiple tests within the same if statement and execute different blocks of code depending on which test evaluates to True. We can do this simply by using an **if, elif, else** statement.

Our example can be modified further to look like the following

```
name = input("What is your name? ")

if name == "Tim" :
    print("Greetings, Tim the Enchanter")
elif name == "Brian" :
    print("Bad luck, Brian")
else :
    print("Hello", name)
```

Saving this code as `if_elif_else.py` , our example now has the following output:

```
>>>
What is your name? Tim
Greetings, Tim the Enchanter

>>>
What is your name? Brian
Bad luck, Brian

>>>
What is your name? Arthur, King of the Britons
Hello Arthur, King of the Britons
```

If-Elif-Else Syntax

```
if test1:
    body1
elif test2:
    body2
.
.
.
elif testn:
    bodyn
else:
    bodyn1
```

The if segment of the If-Elif-Else statement is the same as the if statement. This is the followed by an elif on a new line and de-dented from the body of the if. This is followed by the next test and a colon. The body of this test starts on a new, indented line. This is repeated for all the elif statements required. Then (if required) an else statement is last as described in the If-Else syntax section.

Semantics

If *test1* evaluates to `True` then *body1* is executed. Otherwise, if *test2* evaluates to `True` *body2* is executed. If all the tests are tested and if none evaluate to `True` then *bodyn1* is executed. In other words, the first (and only the first) `True` test in the if-elif-elif-... chain executes its body. If there is no `else` statement and none of the *tests* are `True`, then nothing is executed.

The *`test`* of an if/elif statement is known as the **condition**, because it specifies when the body will execute. if, elif, and else statements are also known as **conditional statements**.

Being Repetitive

We are off to a good start, but the interaction is not very long. We are not doing much before we abruptly end the conversation. For our next addition to the program, we would like to be able to talk to the user for as long as we can. Let's accomplish this by asking the user for a topic, talking about that topic, then asking for another topic. Here is an example of what we might want:

```
What is your name? Tim
Greetings, Tim the Enchanter
What do you want to talk about? Python
Do you like Python? yes
Why do you think that? it's easy to use
I also think that it's easy to use
What do you want to talk about? coconuts
Do you like coconuts? no
Why do you think that? they cannot migrate
I also think that they cannot migrate
What do you want to talk about? CSSE1001
Do you like CSSE1001? very much
Why do you think that? the course notes are very useful
I also think that the course notes are very useful
What do you want to talk about? nothing
Okay. Goodbye, Tim!
```

To do this, we will need to have a way to repeat the discussion until the conversation is over. We can use a construct called a **while** loop. We need to consider what code should be repeated (in this case, the discussion of a topic) and when it should keep going (in this case, when the topic is not "nothing"). Let's update our code to include the repetition. This code can be downloaded as `interaction_while.py`

```
name = input("What is your name? ")
if name == "Tim" :
    print("Greetings, Tim the Enchanter")
elif name == "Brian" :
    print("Bad luck, Brian")
else :
    print("Hello", name)

topic = input("What do you want to talk about? ")
while topic != "nothing" :
    like = input("Do you like " + topic + "? ")
    response = input("Why do you think that? ")
    print("I also think that", response)
    topic = input("What do you want to talk about? ")

print("Okay. Goodbye, " + name + "!")
```

The `topic = input(...)` line above the loop asks for the first topic. The `while topic != "nothing"` line checks if the given condition is `True`, and if it is, then the loop body repeatedly performs the actions until the condition is `False`. Notice that the last line of the body asks for a new topic, which is used as the topic for the next repetition of the loop. If that new topic is `nothing`, then the loop test `topic != "nothing"` becomes `False`, so the while loop will stop running and the code will continue after the loop body (where the indentation stops).

Notice that we are asking if the user likes the topic, but we are not using the response that the user gives. Ignoring the input from the user is very unusual behaviour, but we have done it here to simplify the example.

Run this code to experiment with it. What happens when the first topic is "nothing"? Why does that happen?

One thing that might seem a bit odd about our code is that the `topic = input(...)` line occurs in two places, once before the while loop, and once at the end of the loop body. This has to happen this way because the topic needs to be entered in before the `topic != "nothing"` test happens. We can avoid this by exiting the loop from inside the body. This is done using the **break** keyword. When `break` is executed inside a loop, the program will immediately exit the loop and continue after the body. We can use this with an if statement to specify how the loop should finish. This code is available at `interaction_break.py`

```
name = input("What is your name? ")
if name == "Tim" :
    print("Greetings, Tim the Enchanter")
elif name == "Brian" :
    print("Bad luck, Brian")
else :
    print("Hello", name)
```

```
while True :
    topic = input("What do you want to talk about? ")
    if topic == "nothing" :
        break
    like = input("Do you like " + topic + "? ")
    response = input("Why do you think that? ")
    print("I also think that", response)

print("Okay. Goodbye, " + name + "!")
```

The first change that we have made is the condition test of the while loop. Since the loop keeps going as long as the condition is `True`, that means that "`while True`" will keep going until the `break` is reached (or it will go on forever if there is no `break`; loops that go forever are called *infinite loops*). Using `break` in this way means that the `topic = input(...)` line only has to appear once in our code.

In many situations, it is considered bad programming practice to have logic that exits a loop from within the middle of the loop body. This is because it complicates understanding the loop's behaviour. The reader needs to understand the loop's logical condition plus find and understand the break logic in the middle of the loop body. In the example above, the loop body is short and simple enough that finding the break logic is not difficult. However, once the logic becomes more complex and the code longer it would be harder to read. Try to avoid writing code that exits a loop from the middle of the loop's body. But, be aware that it is possible and you may need to identify this type of logic in someone else's code.

While Loop Syntax

```
while test:
    body
```

The first line contains the word `while` followed by a boolean *`test`* and a `:`. Following is the body, an indented block of code.

Semantics

If *`test`* evaluates to `True` then the *`body`* is executed. Then the test is evaluated again, if it is still `True`, then the body is executed again. This process repeats until the test fails (becomes `False`). Each repetition is called an **iteration** through the loop, and the act of performing a repeated task is called **iterating**. When the test becomes `False` (or if it is `False` to start with), we "exit the loop" and execute the next statement after the indented block.

If a `break` statement is executed inside the loop, the loop will exit immediately.