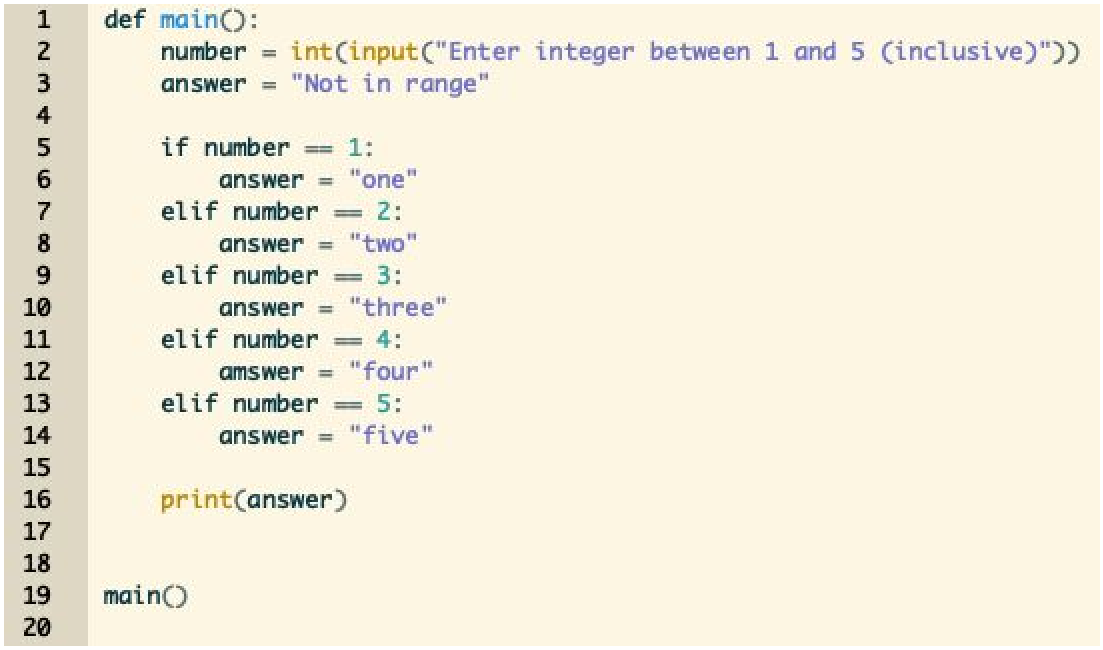
So we've learned a lot in this module so far. But before we wrap the module up, I want to take some time to point out some common programming errors that occur in your programs. As our programs become more complex, it becomes even more challenging to find some of these errors.

So we want to reflect on some of them now. One of the most common programming errors in Python comes from typos in our code. Take a look at this code.



```
1   def main():
2       number = int(input("Enter integer between 1 and 5 (inclusive)"))
3       answer = "Not in range"
4
5       if number == 1:
6           answer = "one"
7       elif number == 2:
8           answer = "two"
9       elif number == 3:
10          answer = "three"
11      elif number == 4:
12          amswer = "four"
13      elif number == 5:
14          answer = "five"
15
16      print(answer)
17
18
19  main()
20
```

Notice that this code is quite similar to the problem that we've been looking at in this module. It is simply a different solution.

This solution, however, has a problem. If you were to enter the value 4 into the keyboard, the program would print "not in range". Can you see why? Notice that on line 12 of this code the word answer was accidentally typed as amswer. That is a common typographical error, since N and M are right next to each other on the keyboard.

This type of error is not strictly the result of using conditionals, but rather a result of the way Python manages its variables. Python simply creates a new variable anytime assignment is used, so this is not an error, at least not to Python. This type of error is called a **logical error**.

As we get more familiar with programming, we get fewer and fewer syntactical errors and start to become more and more adept at handling logical errors. And this is just one type. Other logical errors arise from writing Boolean expressions. In fact, Python makes it pretty easy to write fairly complex Boolean expressions.

To help find logical errors in our program, it is often useful to carefully consider an expression to make sure that is the one that you intended. Consider the following:

```
number > 0 or number < 10.
```

What values of number makes this expression true and what values of number makes the expression false?

You might have noticed that this expression is always true regardless of the value of number. A Boolean expression that is always true is called a tautology and often occurs in code when logical `and` and logical `or` are confused. It is possible to write an expression that's always false too.

Consider this one:

```
number < 0 and number > 0
```

This is known as a contradiction. And again is often the result of confusing logical `and` and logical `or`. When your program has errors it is always worth carefully considering the Boolean expressions in your conditionals.

Okay, suppose that you wanted to print the word ok if the value of the variable x is one, two or three. You might write this.

```
if x == 1 or 2 or 3:
    print("ok")
```

When you run this code in Python, it runs. But unfortunately, it always prints ok regardless of the value of x. The reason for this is comparison has a higher precedence than the `or`, so Python interprets this code as though you had written this:

```
if (x == 1) or 2 or 3:
    print("ok")
```

The expression x equal one is either true or false, but the integer two is always interpreted to be true, and integer three as well because they're not zero. And anything `or` true is always true.

The correct way to write this statement, is like we did in the program in the previous video on logical operators. That would have been to write this one:

```
if x==1 or x==2 or x==3:
    print("ok")
```

Obviously the added complexity here is needed in order for the statement to be correct.

It is however often the case, that we write statements that are overly complex. Take a look at these two expressions:

1. `not (a ==b and not (c ==d))`
2. `a !=b or c==d`

Clearly the second statement is not as complex as the first one, but they are logically equivalent to each other. And we can convince ourselves of this using a truth table to evaluate both expressions.

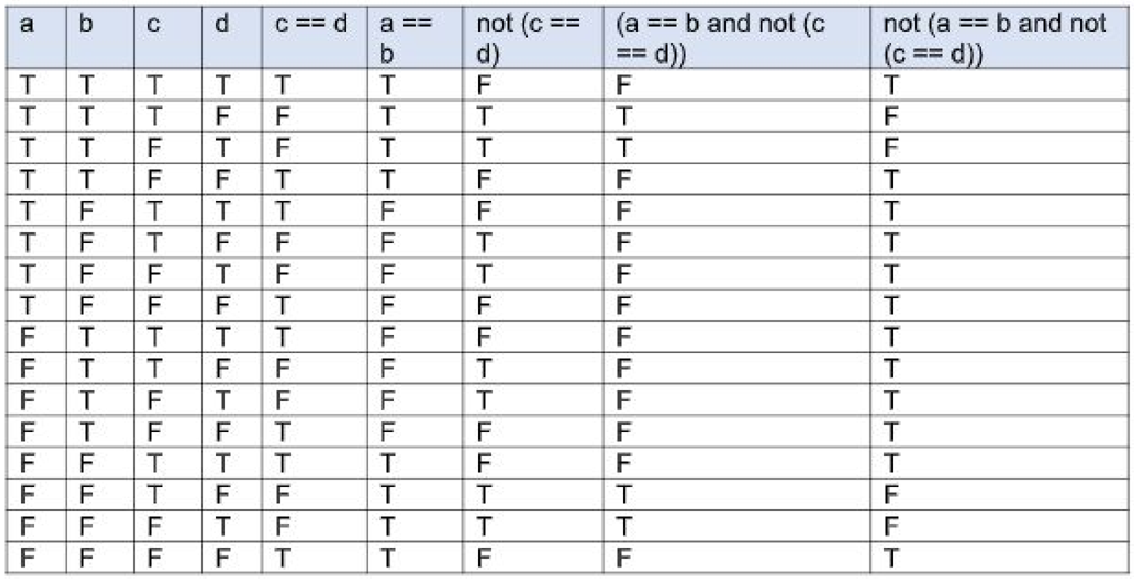
Okay, let's do that. We start the truth table by adding a, b, c, and d as the first four columns in the table. Then we need to write out all of the possible combinations of true and false for these variables. Since there are four of them, that means there are sixteen rows.

We then build each piece of the Boolean expression one step at a time following the rules of precedence and associativity. Since the parentheses are evaluated first, we evaluate the innermost parentheses first, c is equal to d. This is a combination of columns three and four being combined by the `is equal` operator.

This evaluates to `true` when c and d have the same value. Next, comparison is evaluated for a is equal to b which combines the variables in columns one and two. The next operator to be evaluated is the `not` inside the outer parentheses `not c = d,` which is the negation of column five.

Then the `and` is evaluated which will combine the values of columns six and seven. Finally, we negate column eight.

First Statement Resulting Truth Table



a	b	c	d	c == d	a == b	not (c == d)	(a == b and not (c == d))	not (a == b and not (c == d))
T	T	T	T	T	T	F	F	T
T	T	T	F	F	T	T	T	F
T	T	F	T	F	T	T	T	F
T	T	F	F	T	T	F	F	T
T	F	T	T	T	F	F	F	T
T	F	T	F	F	F	T	F	T
T	F	F	T	F	F	T	F	T
T	F	F	F	T	F	F	F	T
F	T	T	T	T	F	F	F	T
F	T	T	F	F	F	T	F	T
F	T	F	T	F	F	T	F	T
F	T	F	F	T	F	F	F	T
F	F	T	T	T	T	F	F	T
F	F	T	F	F	T	T	T	F
F	F	F	T	F	T	T	T	F
F	F	F	F	T	T	F	F	T

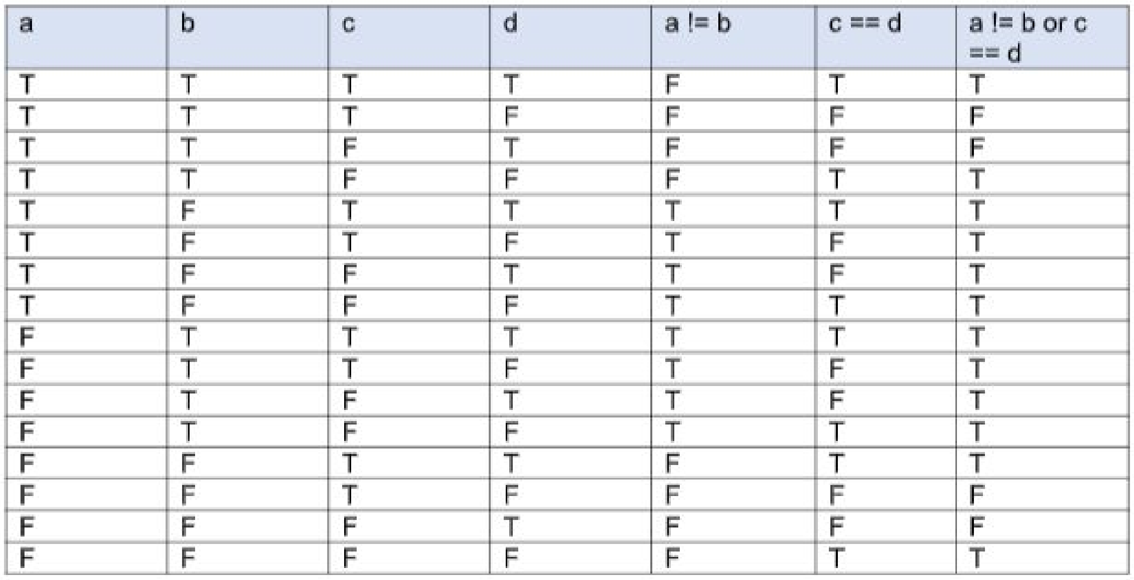
By breaking down each statement into pieces, we can determine the truth table values of the complex expression without making mistakes.

Now that we have built it for the first expression, we can build it for the second.

Starting from the same place with all of the combinations of values a, b, c and d we will then evaluate it following the rules of associativity and precedence. We start by evaluating a not equal to b combining the values of the columns one and two. We then evaluate c is equal to d by combining the values in three and four.

Finally, we apply the or operator to combine the values of comms five and six to get our final result

Second Statement Resulting Truth Table

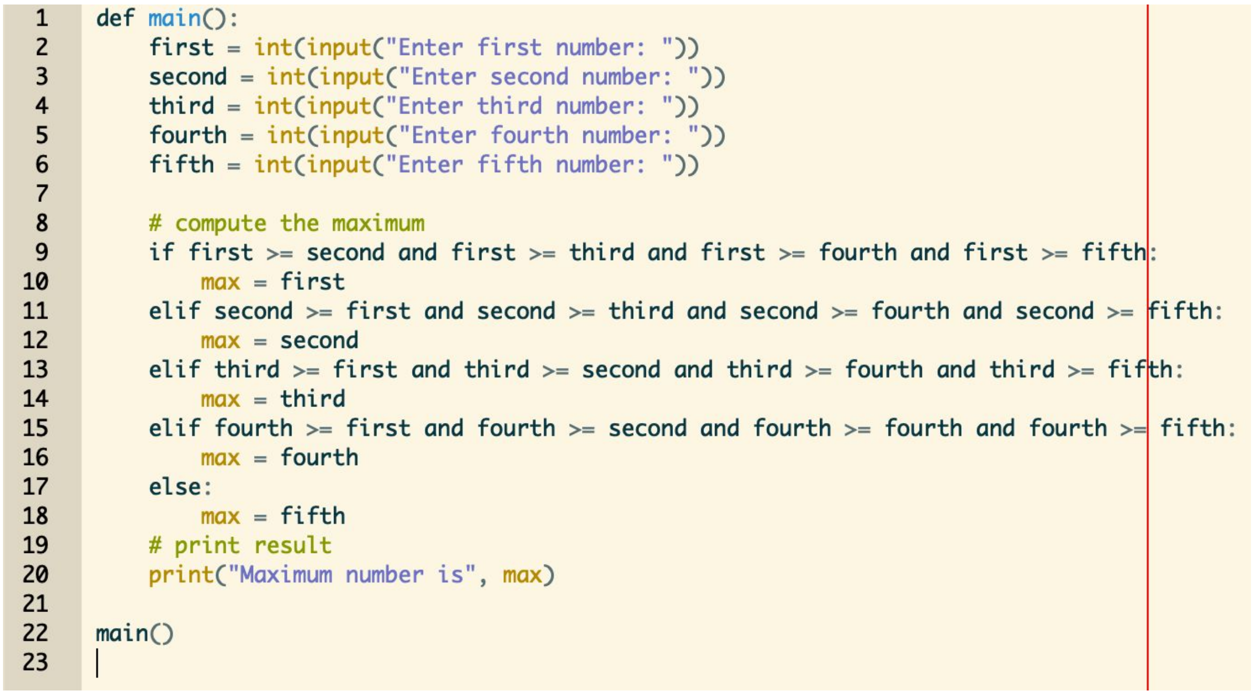


a	b	c	d	a != b	c == d	a != b or c == d
T	T	T	T	F	T	T
T	T	T	F	F	F	F
T	T	F	T	F	F	F
T	T	F	F	F	T	T
T	F	T	T	T	T	T
T	F	T	F	T	F	T
T	F	F	T	T	F	T
T	F	F	F	T	T	T
F	T	T	T	T	T	T
F	T	T	F	T	F	T
F	T	F	T	T	F	T
F	T	F	F	T	T	T
F	F	T	T	F	T	T
F	F	T	F	F	F	F
F	F	F	T	F	F	F
F	F	F	F	F	T	T

Since the values of the element in the first two columns are the same by design, and the values in the last column are as well, we know that these two expressions are logically equivalent.

The second one is not only easier to understand, it was way fewer steps to evaluate. In general, there's some guidelines to use when writing Boolean expressions. Simpler logic is easier to understand. Simpler logic is easier to write and get working. The longer and more complicated expressions increase your risk of typographical errors that manifest as logical errors and logical errors are really hard to find.

Simpler logic can be more efficient. Every relational comparison and Boolean operation takes time to execute. So the fewer they are, the more efficient your program is. Simpler logic is easier to modify and extend. Let's take a look at one more example. Consider the problem of determining the maximum value of five integers entered by the user.
One possible solution would be the following:



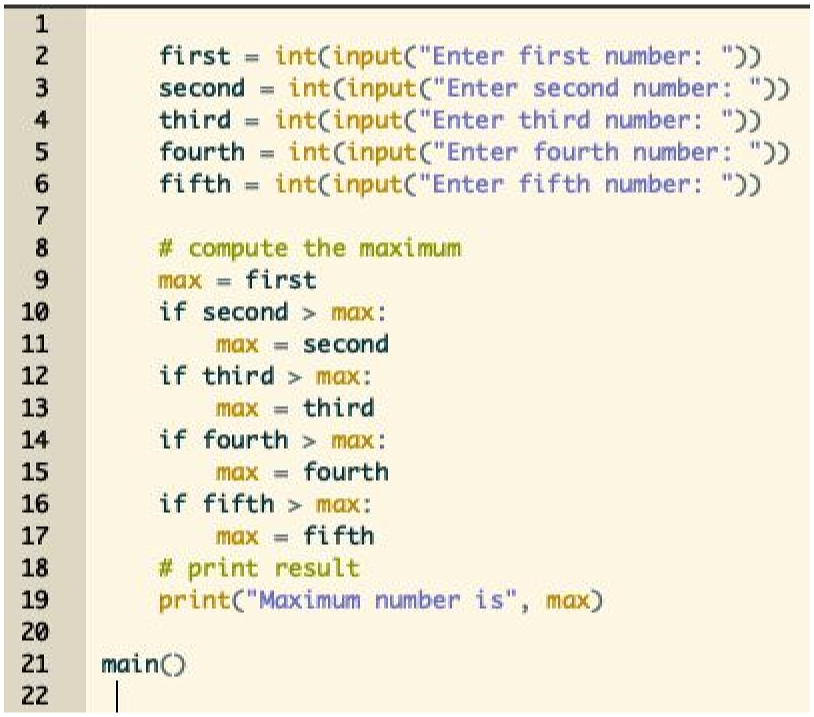
```python
def main():
    first = int(input("Enter first number: "))
    second = int(input("Enter second number: "))
    third = int(input("Enter third number: "))
    fourth = int(input("Enter fourth number: "))
    fifth = int(input("Enter fifth number: "))

    # compute the maximum
    if first >= second and first >= third and first >= fourth and first >= fifth:
        max = first
    elif second >= first and second >= third and second >= fourth and second >= fifth:
        max = second
    elif third >= first and third >= second and third >= fourth and third >= fifth:
        max = third
    elif fourth >= first and fourth >= second and fourth >= fourth and fourth >= fifth:
        max = fourth
    else:
        max = fifth
    # print result
    print("Maximum number is", max)

main()
```

Now, when we run this, we notice it gives the correct result most of the time, but not always. This is not good, and it's even worse because it's possible that we would run through a couple of scenarios without testing all of them.

And this code would have been put into production with a bug. You might look at this and say, "there's the bug!", but chances are you won't because of the complexity of the logic in the conditional statements. This can make it hard to determine where the error actually is.

What if we were to take a different approach to this problem that would greatly simplify the logic. Suppose that we could guess that the maximum value was the first number entered and then check to see if any of the other values could beat the maximum so far by being larger than the max.

Let's look at code that does that:



```
    first = int(input("Enter first number: "))
    second = int(input("Enter second number: "))
    third = int(input("Enter third number: "))
    fourth = int(input("Enter fourth number: "))
    fifth = int(input("Enter fifth number: "))

    # compute the maximum
    max = first
    if second > max:
        max = second
    if third > max:
        max = third
    if fourth > max:
        max = fourth
    if fifth > max:
        max = fifth
    # print result
    print("Maximum number is", max)

main()

```

Here the maximum variable is storing the maximum so far. We start by guessing that maximum value is the first number entered and then sequentially check it against the rest of them modifying the maximum so far if any of the values are greater than what we have seen so far.

This is why this is often called the **guess and check pattern**. We will be seeing this again in future modules.

Okay, these are just a few of the logical errors that come as the code that we write becomes more complex. There are more, but we'll let you find those.

Okay, that's all for now. Thanks for watching a line online.