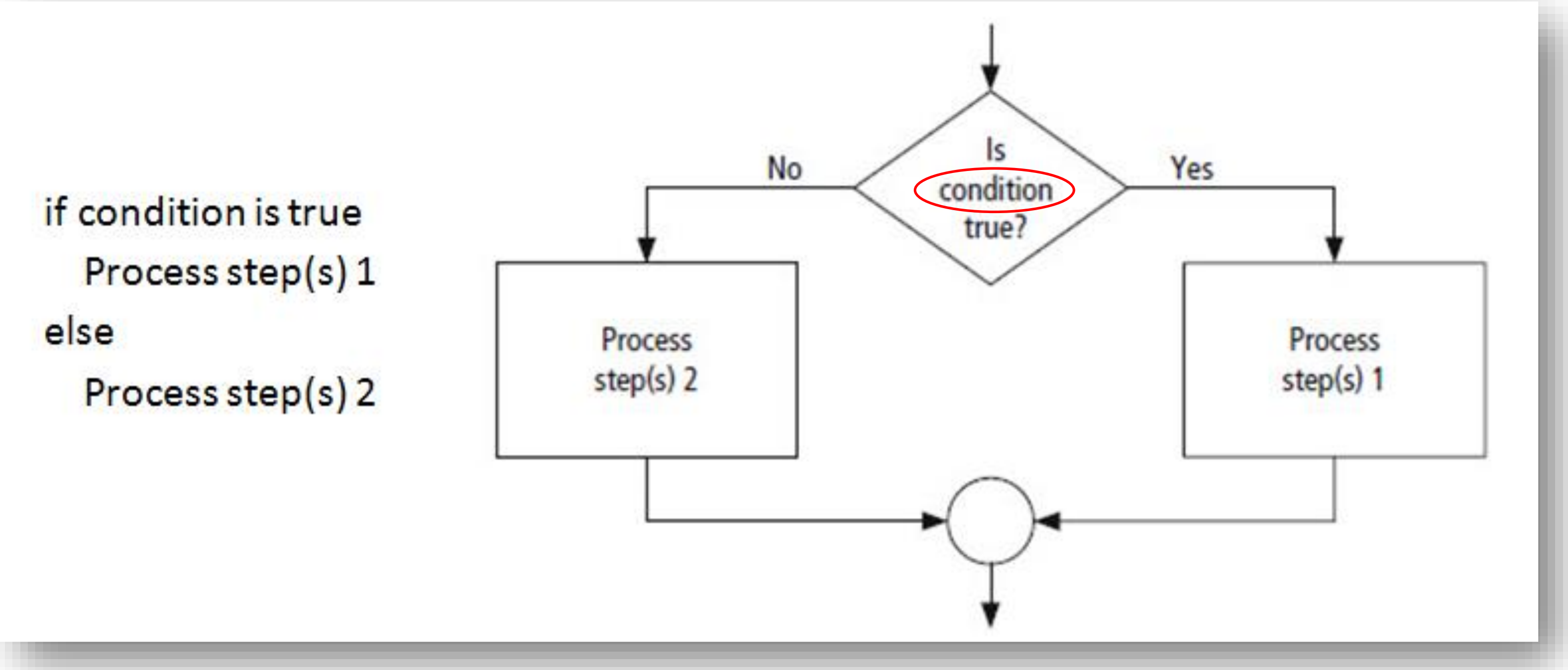
Relational and Logical Operators

Section 1

Chapter 3

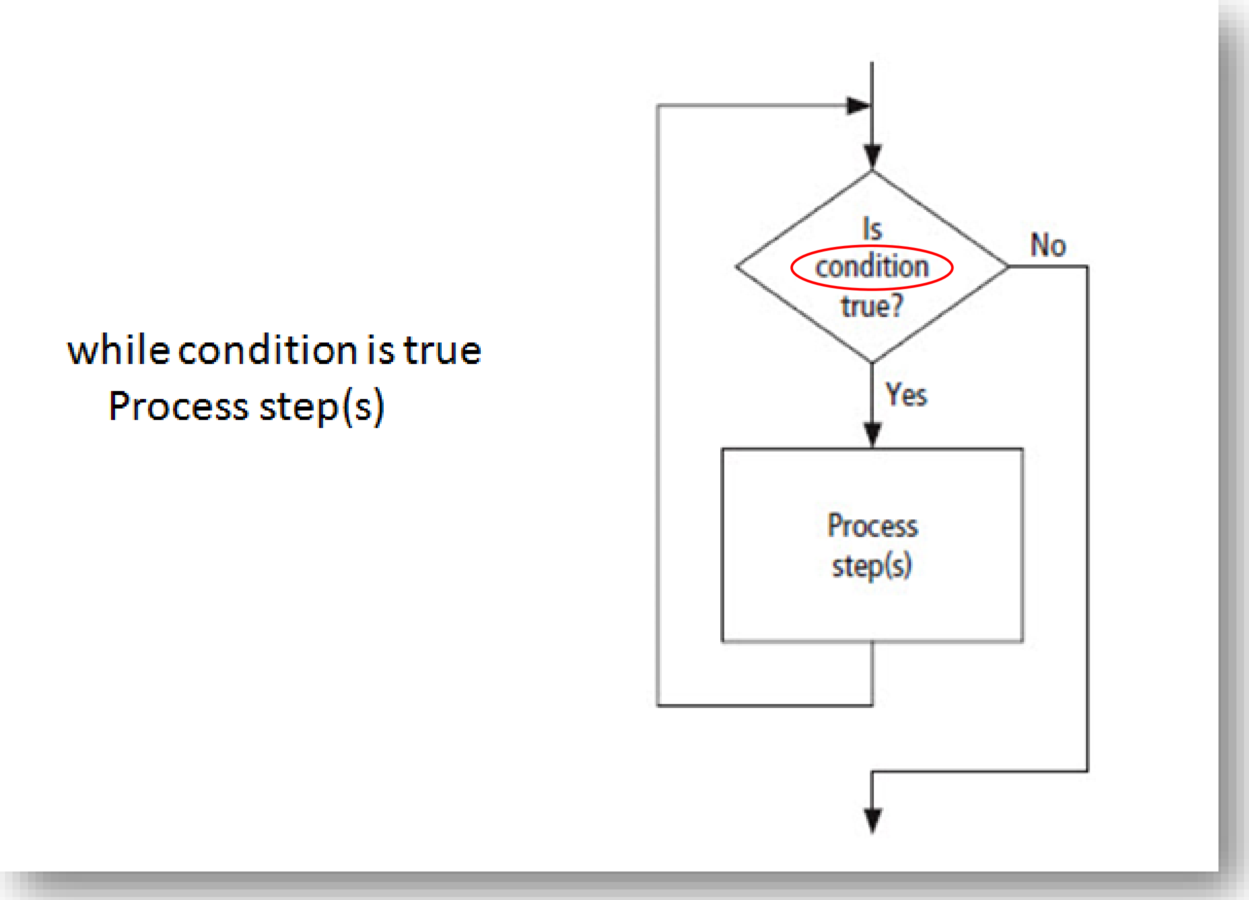
Quiz 5

Recall from Week 1: Decision Structure



```
if condition is true
    Process step(s) 1
else
    Process step(s) 2
```

Recall from week 1: Repetition Structure



```
while condition is true
   Process step(s)
```

Conditions

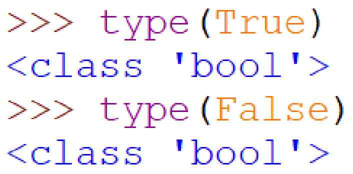
- A *condition* is an expression that evaluates to either `True` or `False`.

- Conditions are used to make decisions
  - Choose between options
  - Control loops

- Conditions typically involve
  - Relational operators (e.g., <, >=)
  - Logical operators (e.g., and, or, not)

The **bool** Data Type

- Objects **True** and **False** are said to have Boolean data type - **bool**.



```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Relational Operators

Relational Operators

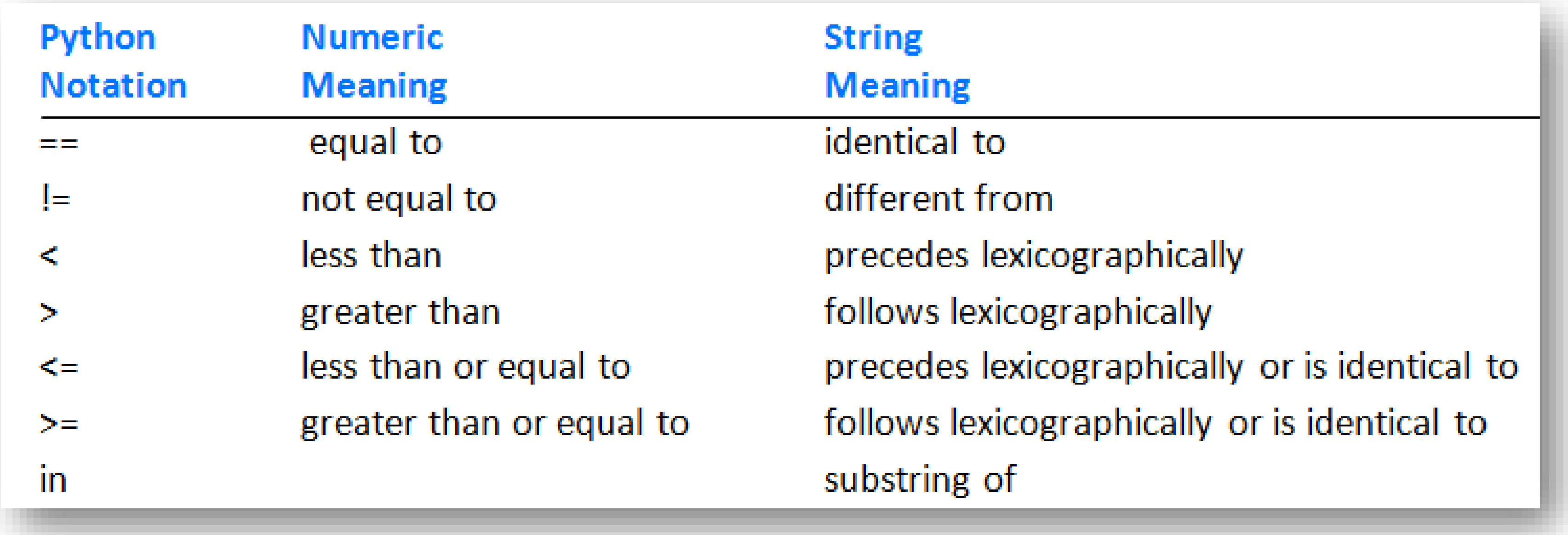
- Relational operators (e.g. ==,<, >)can be applied to
    - Numbers
    - Strings
    - Lists
    - Tuples

Relational operators for numbers

```
>>> 2==2 # Equal to. Notice there are two =
True
>>> 2!=2 # Not equal to
False
>>> 2!=4
True
```

```
>>> 3>1
True
>>> 3>=1 #Greater than or Equal to
True
>>> 3<1
False
>>> 3<=1 # 3 is neither Smaller than nor Equal to 1.
False
```

Relational Operators



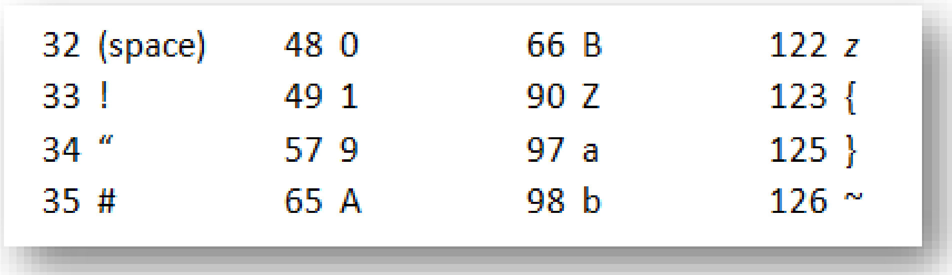
Python Notation	Numeric Meaning	String Meaning
==	equal to	identical to
!=	not equal to	different from
<	less than	precedes lexicographically
>	greater than	follows lexicographically
<=	less than or equal to	precedes lexicographically or is identical to
>=	greater than or equal to	follows lexicographically or is identical to
in		substring of

ASCII Values

- American Standard Code for Information Interchange (**ASCII**) values determines the order used to compare **strings** with relational operators.

- Associated with keyboard letters, characters, numerals
  - ASCII values are numbers ranging from 32 to 126.

- A few ASCII values:

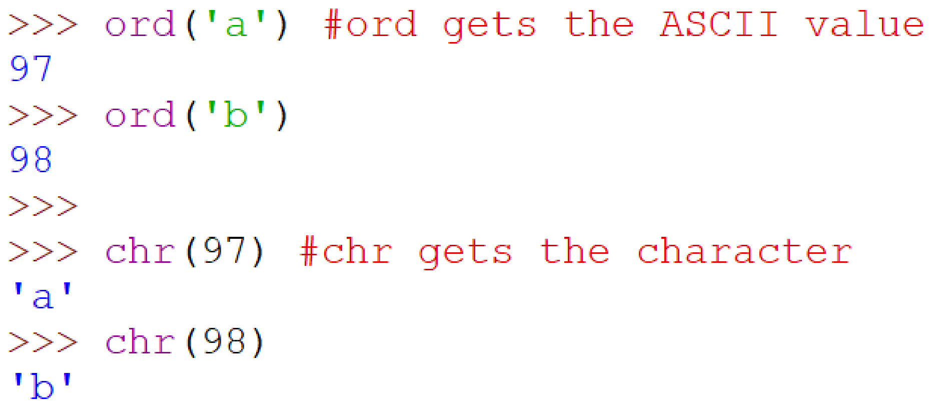


32	(space)	48	0	66	B	122	z
33	!	49	1	90	Z	123	{
34	“	57	9	97	a	125	}
35	#	65	A	98	b	126	~

ASCII Values

- Function **`ord(str)`** returns ASCII values for single character strings.

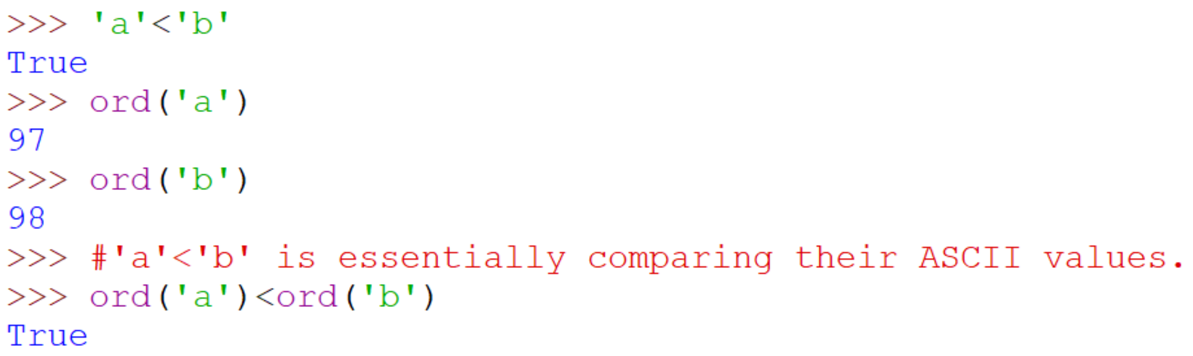
- Function **`chr(n)`** returns the string associated with the ASCII value n.



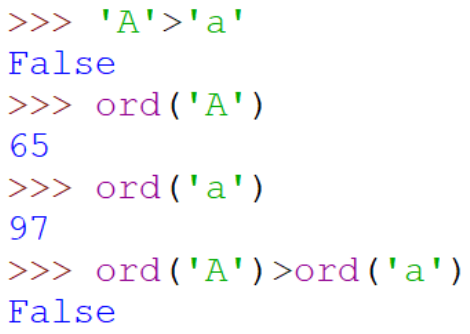
```
>>> ord('a') #ord gets the ASCII value
97
>>> ord('b')
98
>>>
>>> chr(97) #chr gets the character
'a'
>>> chr(98)
'b'
```

Relational operators for strings: comparisons

- String comparisons is based on comparisons of their ASCII values.



```
>>> 'a'<'b'
True
>>> ord('a')
97
>>> ord('b')
98
>>> #'a'<'b' is essentially comparing their ASCII values.
>>> ord('a')<ord('b')
True
```



```
>>> 'A'>'a'
False
>>> ord('A')
65
>>> ord('a')
97
>>> ord('A')>ord('a')
False
```

Relational operators for iterables: comparisons

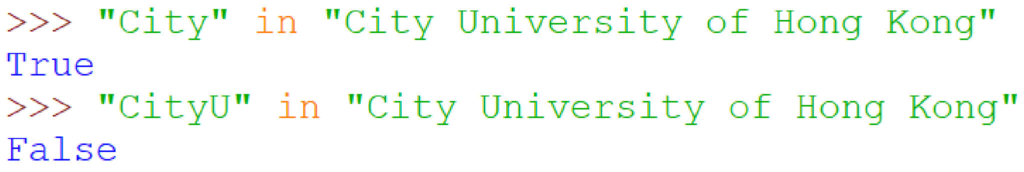
- Recall: strings, lists, and tuples are iterable objects - their characters/items can be returned one at a time.

- Comparisons of iterables are done character- or item-wise from left to right.

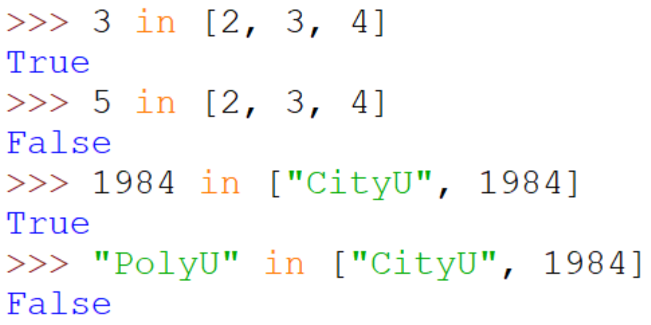
```
>>> "Amy"<"amY" #True because 'A'<'a'.
True
>>> "aMy"<"amY" #'a'=='a' evaluates to be True. But 'M'<'m'.
True

>>> [1, 10000, 100000]<[2, 0.001, 0.000001] #True because 1<2
True
>>> [4, 7, 3] > [4, 8, 1] #False because 7>8 is False, even though 3>1 is true.
False
>>> [4, "amY", 3]>[4, "Amy", 9] #True becuase "amY">"Amy" is True.
True
```

Relational operators for iterables: `in`

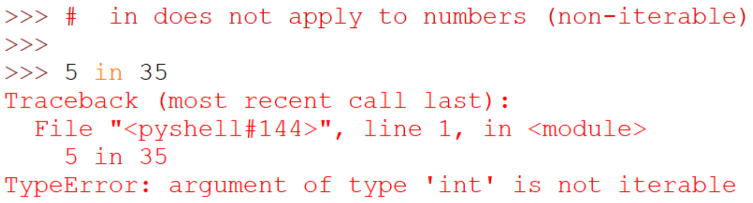
- str1 `in` str2 is evaluated to be True if str1 is a substring of str2.
- An object `in` a list/tuple is evaluated to be True if the object is an item in the list/tuple.



```
>>> "City" in "City University of Hong Kong"
True
>>> "CityU" in "City University of Hong Kong"
False
```



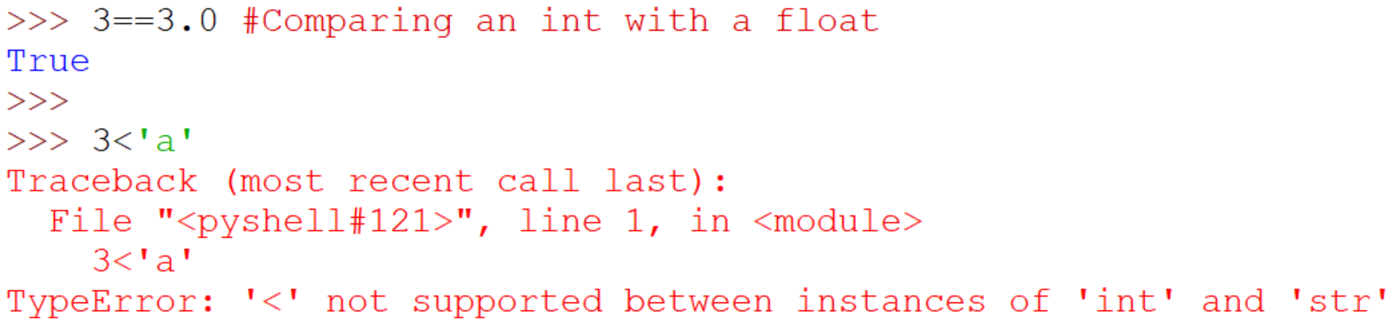
```
>>> 3 in [2, 3, 4]
True
>>> 5 in [2, 3, 4]
False
>>> 1984 in ["CityU", 1984]
True
>>> "PolyU" in ["CityU", 1984]
False
```



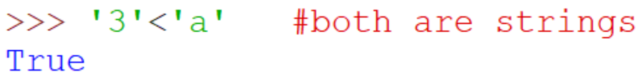
```
>>> #  in does not apply to numbers (non-iterable)
>>>
>>> 5 in 35
Traceback (most recent call last):
  File "<pyshell#144>", line 1, in <module>
    5 in 35
TypeError: argument of type 'int' is not iterable
```

Relational Operators

- Some rules
  - An int *can* be compared to a float.
  - Otherwise, objects of different types *cannot* be compared



```
>>> 3==3.0 #Comparing an int with a float
True
>>>
>>> 3<'a'
Traceback (most recent call last):
  File "<pyshell#121>", line 1, in <module>
    3<'a'
TypeError: '<' not supported between instances of 'int' and 'str'
```



```
>>> '3'<'a'   #both are strings
True
```

Decision Structures

Section 2

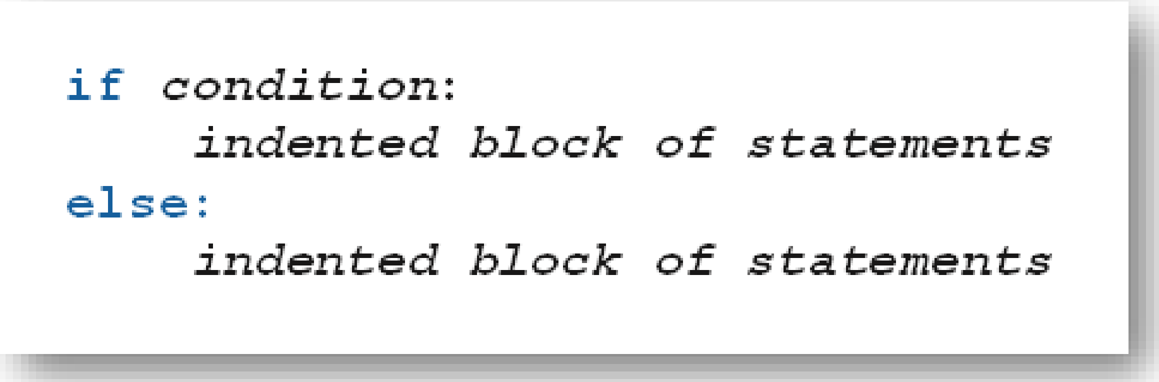
Chapter 3

Decision structures

- Decision structures are also known as **branching structures**

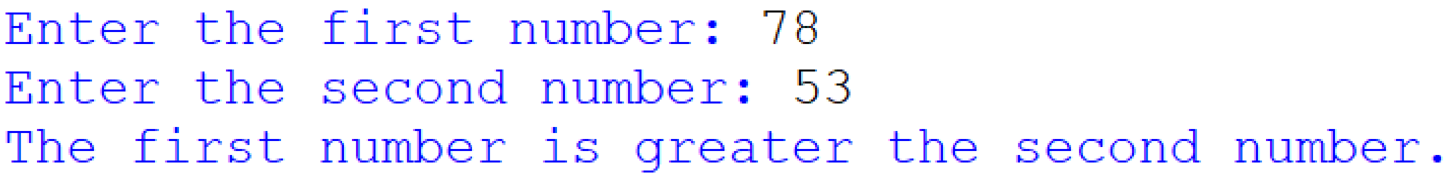
- Allow program to decide on the course of action
  - Based on whether a certain condition is true or false.

- Form:

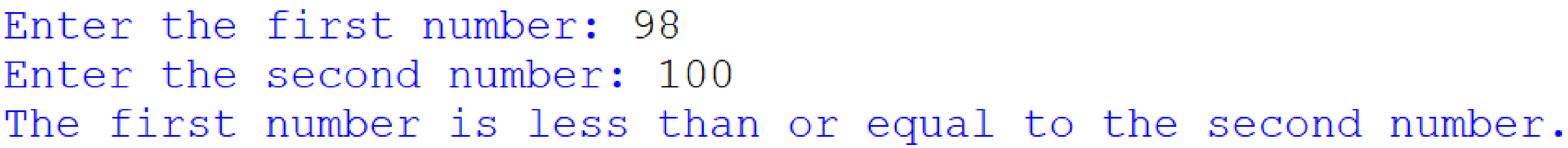


```
if condition:
    indented block of statements
else:
    indented block of statements
```

Example 1: Comparing two numbers

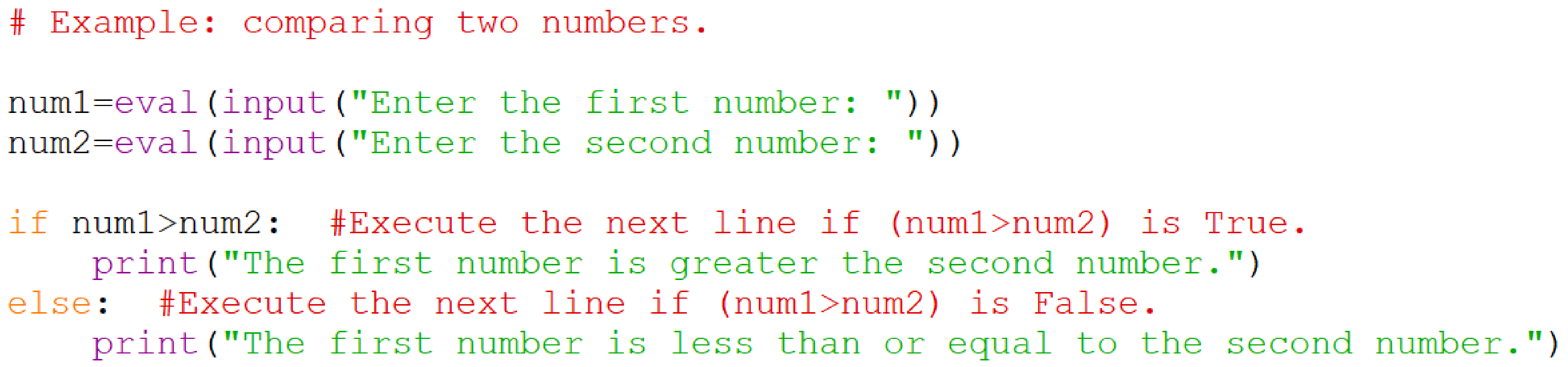


```
Enter the first number: 78
Enter the second number: 53
The first number is greater the second number.
```



```
Enter the first number: 98
Enter the second number: 100
The first number is less than or equal to the second number.
```

Example 1: Comparing two numbers

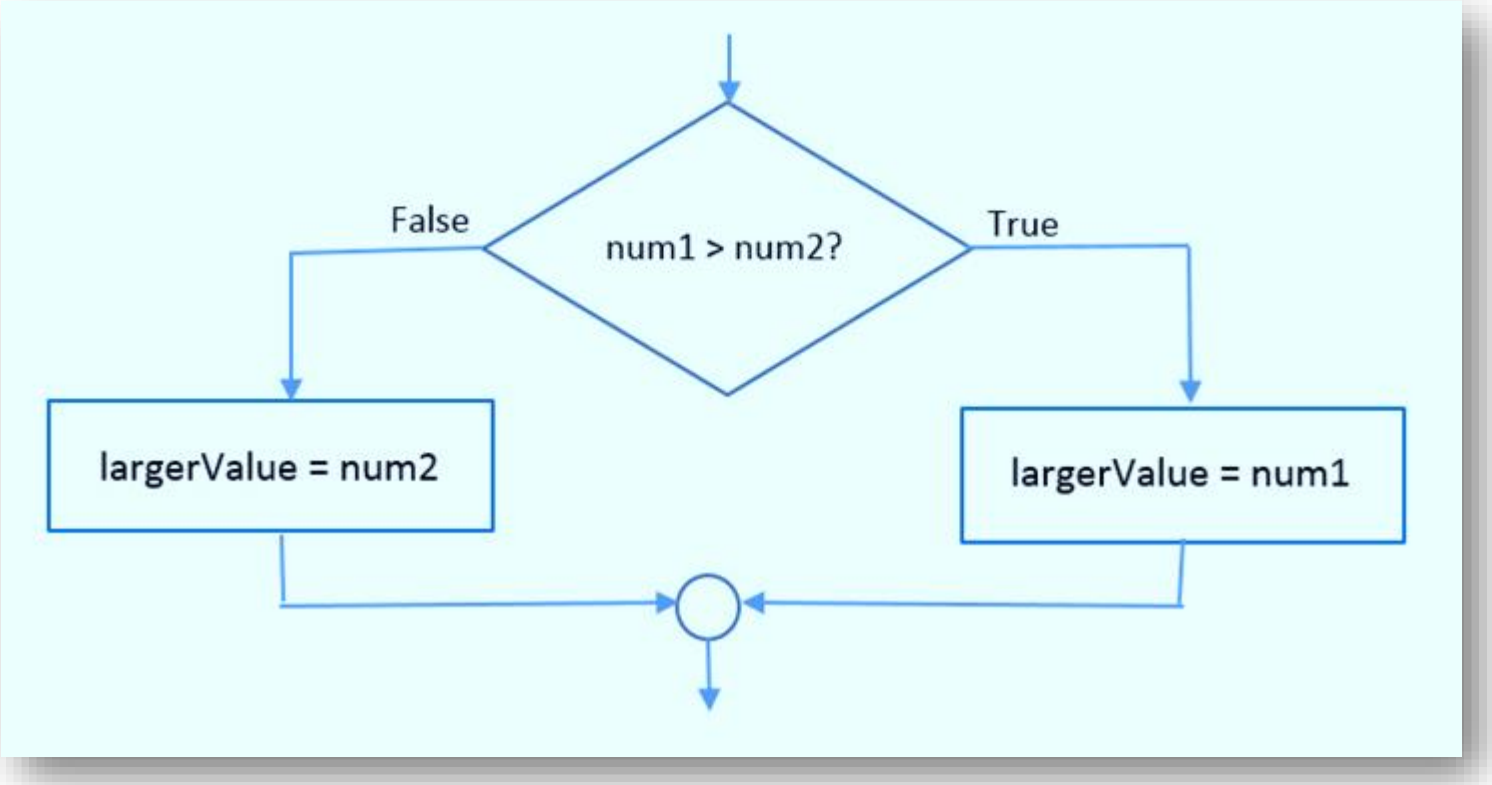


```
# Example: comparing two numbers.

num1=eval(input("Enter the first number: "))
num2=eval(input("Enter the second number: "))

if num1>num2:  #Execute the next line if (num1>num2) is True.
    print("The first number is greater the second number.")
else:  #Execute the next line if (num1>num2) is False.
    print("The first number is less than or equal to the second number.")
```

Some minor but essential points.

- Don’t forget the **colon** after if-else.
- The **indent** should be achieved with the return key after the colon.
    - Do not try to use spaces for the indented block.

Example 1: Comparing two numbers



if Statements

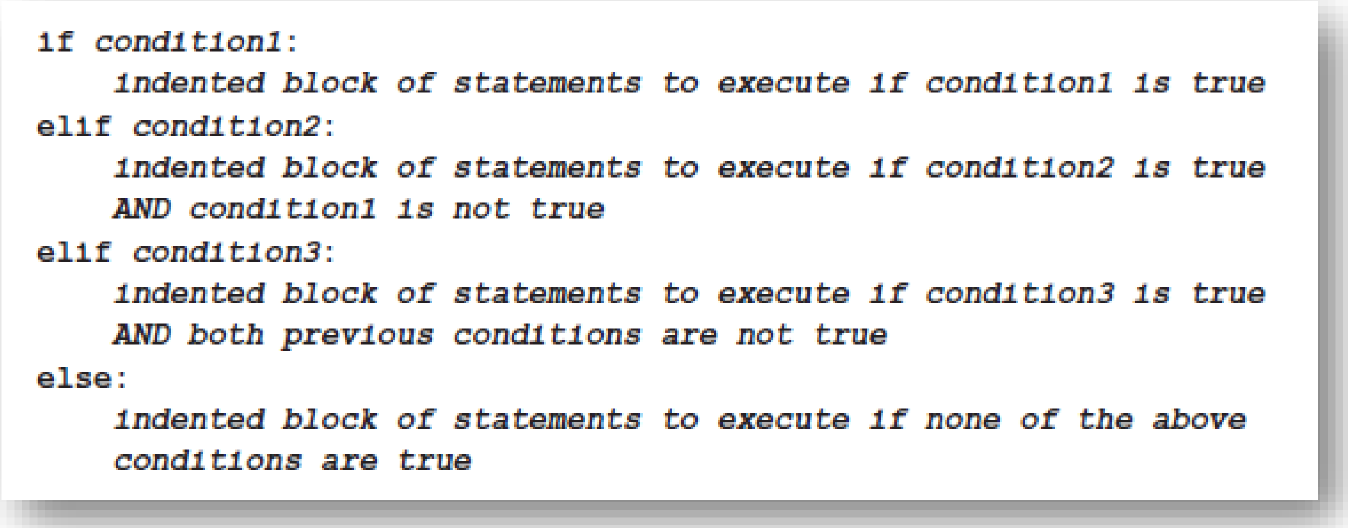
- The *else* part of an *if-else* statement can be omitted.

- When the condition is false
  - The execution continues with the line after the if statement block

The *elif* Clause

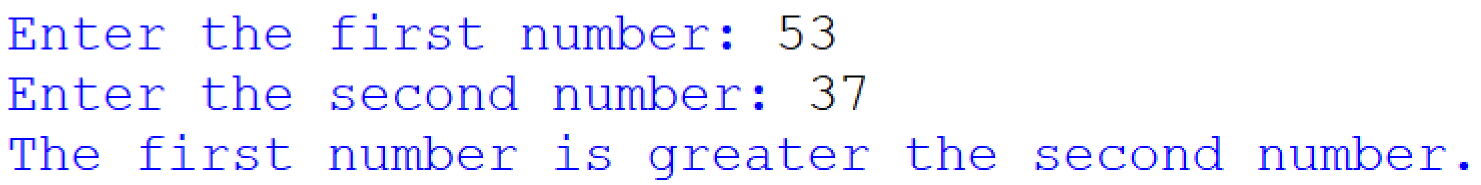
- *elif* allows for more than two possible alternatives.

- You can add as many *elif* as needed.

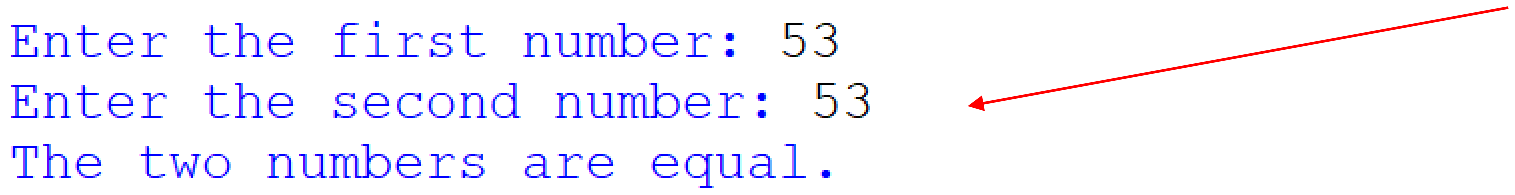


```
if condition1:
    indented block of statements to execute if condition1 is true
elif condition2:
    indented block of statements to execute if condition2 is true
    AND condition1 is not true
elif condition3:
    indented block of statements to execute if condition3 is true
    AND both previous conditions are not true
else:
    indented block of statements to execute if none of the above
    conditions are true
```

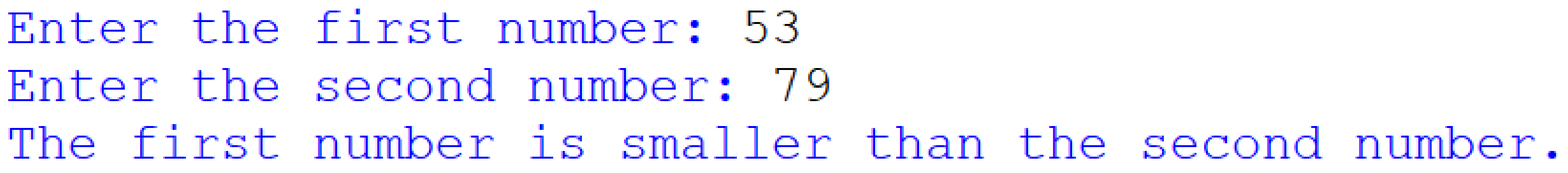
Example 2: Comparing two numbers



```
Enter the first number: 53
Enter the second number: 37
The first number is greater the second number.
```



```
Enter the first number: 53
Enter the second number: 53
The two numbers are equal.
```



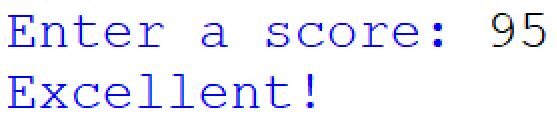
```
Enter the first number: 53
Enter the second number: 79
The first number is smaller than the second number.
```

Example 2: Comparing two numbers

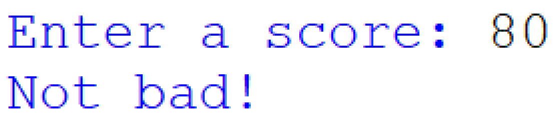
```python
# Example: comparing two numbers.

num1=eval(input("Enter the first number: "))
num2=eval(input("Enter the second number: "))

if num1>num2: #condition 1
    print("The first number is greater the second number.")
elif num1==num2: #### added condition 2 via elif.
    print("The two numbers are equal.")
else: #else takes care of everything not covered by conditions 1, 2.
    print("The first number is smaller than the second number.")
```

Example 3

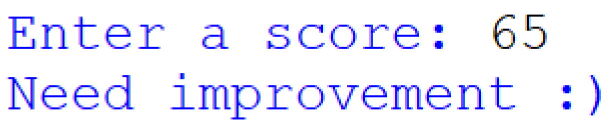
Score	Evaluation
Above 90	Excellent!
70-90	Not bad!
Below 70	Need improvement :)



```
Enter a score: 95
Excellent!
```

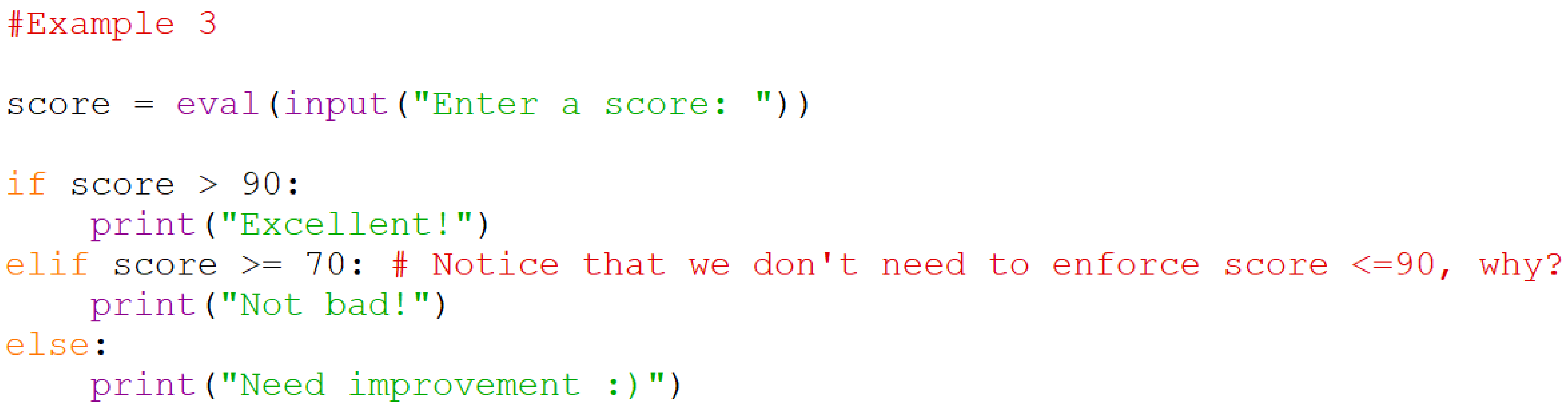


```
Enter a score: 80
Not bad!
```



```
Enter a score: 65
Need improvement :)
```

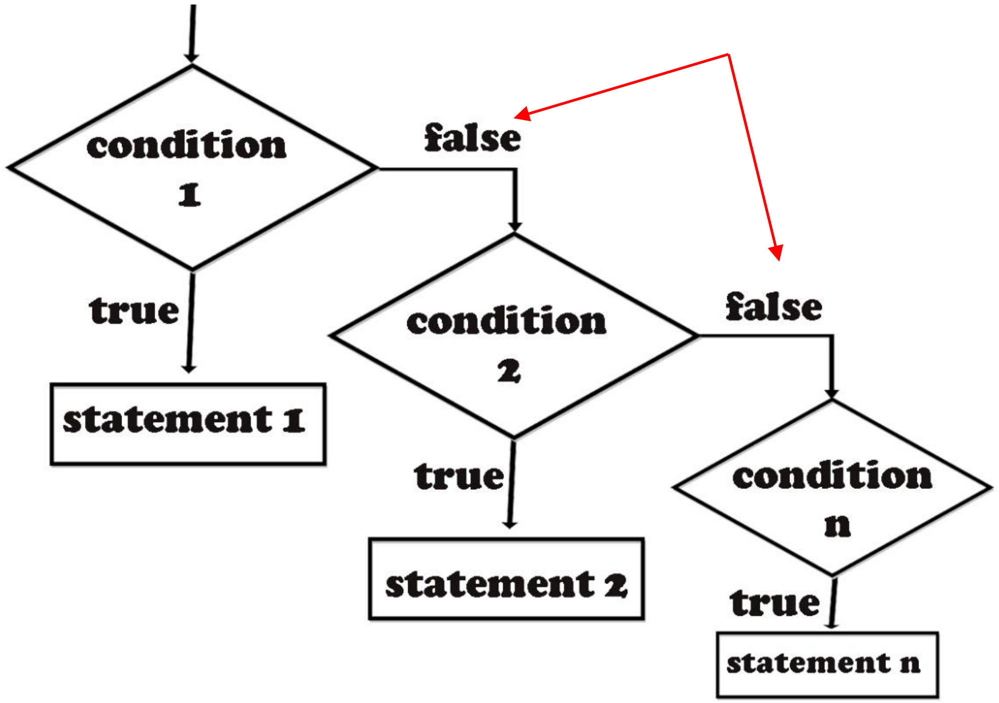
Example 3



```
#Example 3

score = eval(input("Enter a score: "))

if score > 90:
    print("Excellent!")
elif score >= 70: # Notice that we don't need to enforce score <=90, why?
    print("Not bad!")
else:
    print("Need improvement :)")
```

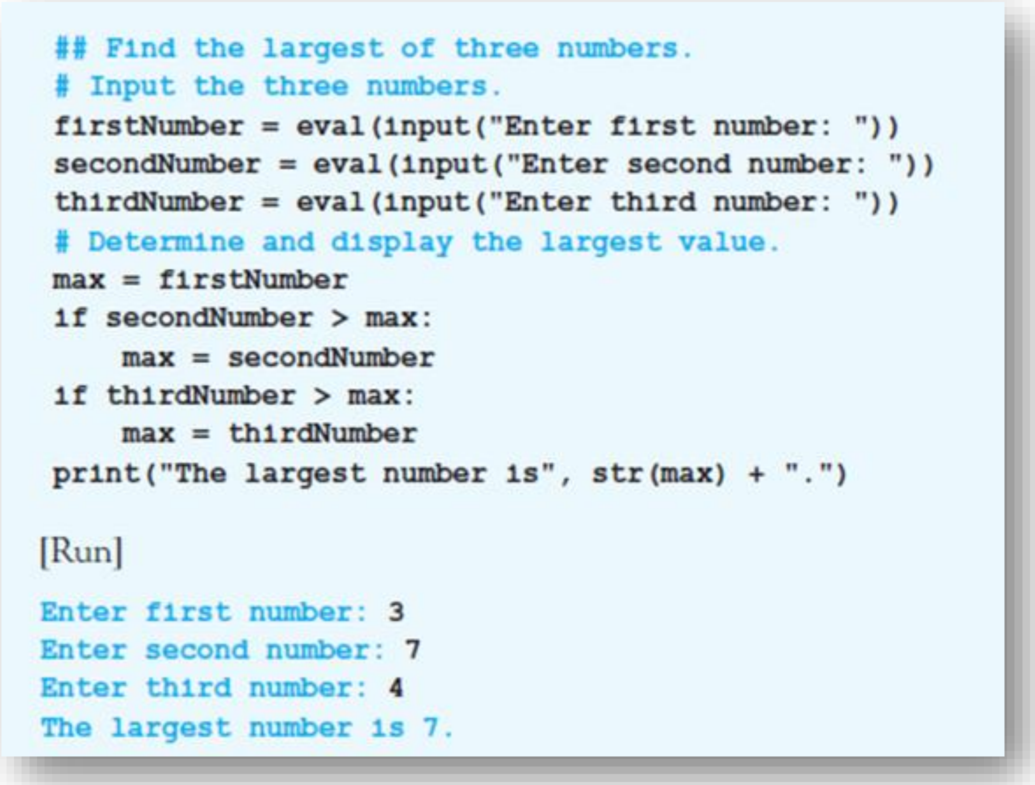
The *elif* Clause

- For condition 2, we already know that condition 1 is False.
- For condition 3, we already know that condition 1 and 2 are False.
- For condition n, …



if Statements

- Example: A program can contain two *if* statements

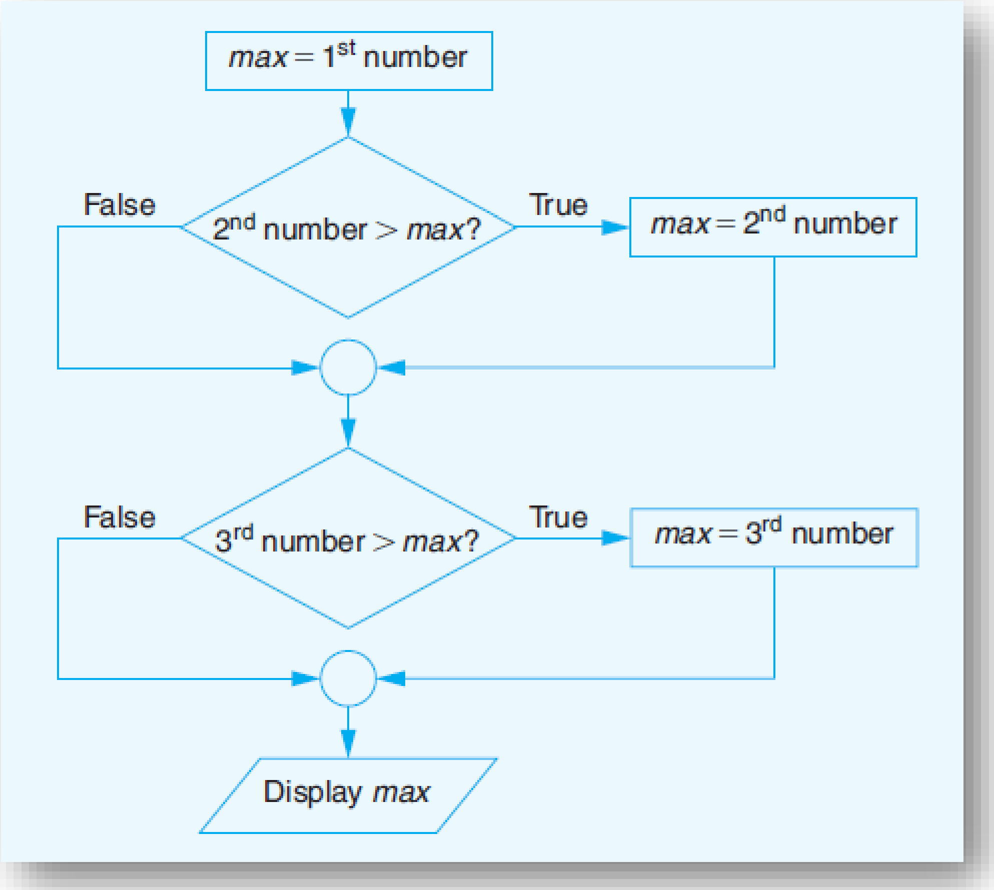


```
## Find the largest of three numbers.
# Input the three numbers.
firstNumber = eval(input("Enter first number: "))
secondNumber = eval(input("Enter second number: "))
thirdNumber = eval(input("Enter third number: "))
# Determine and display the largest value.
max = firstNumber
if secondNumber > max:
    max = secondNumber
if thirdNumber > max:
    max = thirdNumber
print("The largest number is", str(max) + ".")
```

[Run]

```
Enter first number: 3
Enter second number: 7
Enter third number: 4
The largest number is 7.
```

Two *if* Statements



Logical Operators

Logical Operators

- Logical operators allows combining multiple conditions into one condition.
- Logical operators are the reserved words **`and`**, **`or`**, and **`not`**
- Conditions that use these operators are called compound conditions

Logical Operators

- Given: *cond1* and *cond2* are conditions
  - `cond1 and cond2` is true if both conditions are true
  - `cond1 or cond2` is true if either or both conditions are true
  - `not cond1` true if cond1 is false

Logical Operators

```
>>> 3<5 and 3>1   # True because both conditions are True
True
>>> 3<5 and 3==1  # False because at least one of them is False
False


>>> 3<5 or 3==1   # True because at least one of them is True
True
>>> 3>5 or 3==1  # False because both conditions are False
False


>>> not 3==1  # True because 3==1 is False
True
>>> not 1==1  # False because 1==1 is True
False
```

Short-Circuit Evaluation

- Consider the condition **`cond1 and cond2`**
  - If Python evaluates **`cond1`** as false, it does not bother to check **`cond2`**

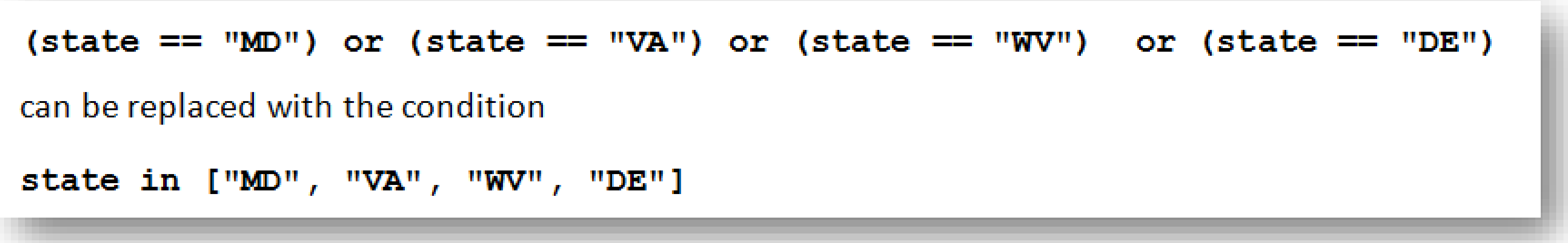
- Similar for **`cond1 or cond2`**
  - If Python finds **`cond1`** true, it does not bother to check **`cond2`**

- Think why this feature helps for

```
(number != 0) and (m == (n / number))
```

Simplifying Conditions

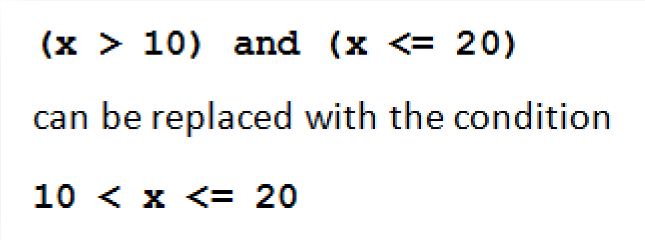
- Lists or tuples can sometimes be used to simplify long compound conditions



```
(state == "MD") or (state == "VA") or (state == "WV")  or (state == "DE")
```

can be replaced with the condition

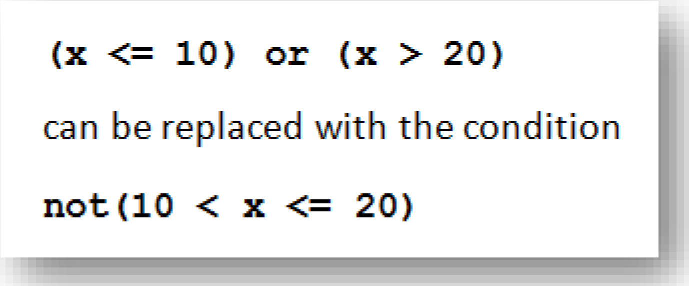
```
state in ["MD", "VA", "WV", "DE"]
```



```
(x > 10) and (x <= 20)
```

can be replaced with the condition

```
10 < x <= 20
```



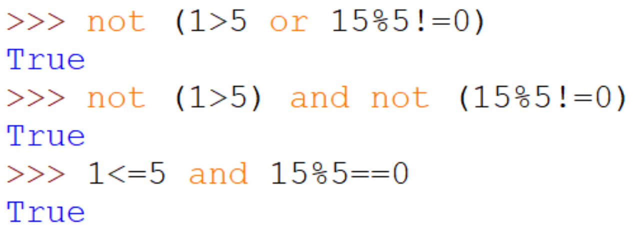
```
(x <= 10) or (x > 20)
```

can be replaced with the condition

```
not(10 < x <= 20)
```

Simplifying Conditions

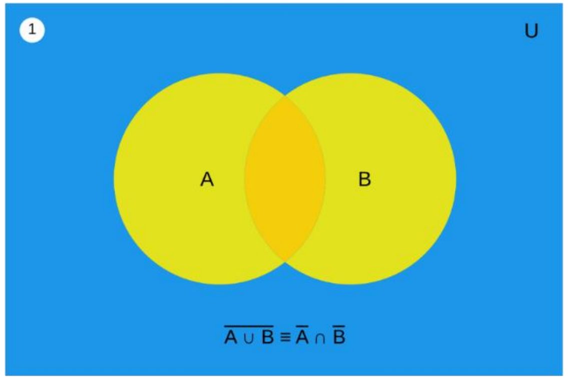
By De Morgan's Laws,

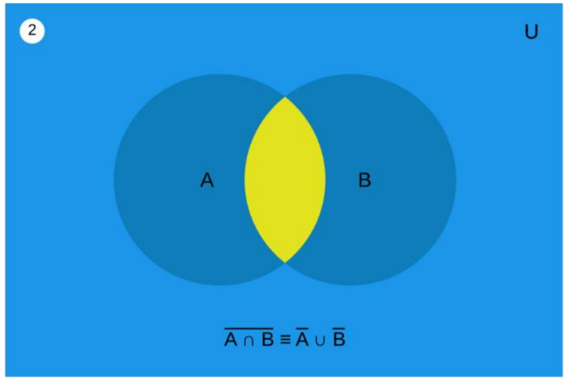


```
>>> not (1>5 or 15%5!=0)
True
>>> not (1>5) and not (15%5!=0)
True
>>> 1<=5 and 15%5==0
True
```

- **`not(cond1 or cond2)`**
  is equivalent to
  `not(cond1) and not(cond2)`

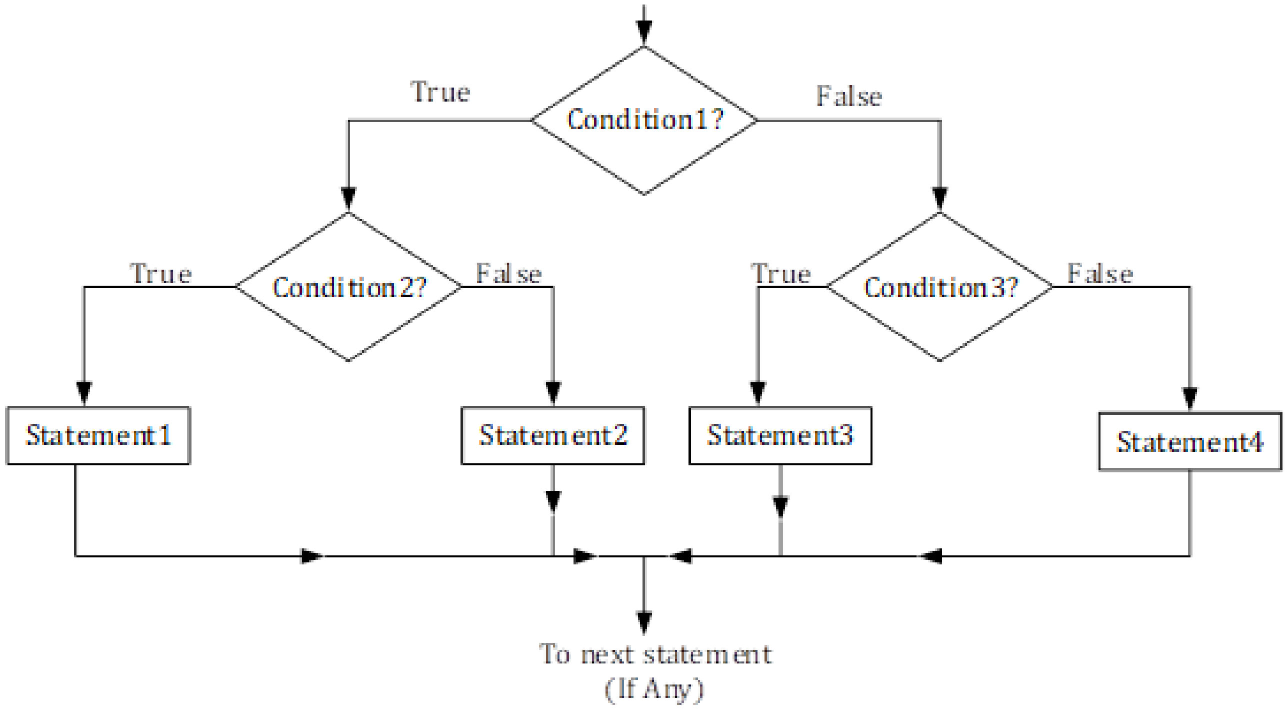
- **`not(cond1 and cond2)`**
  is equivalent to
  `not(cond1) or not(cond2)`





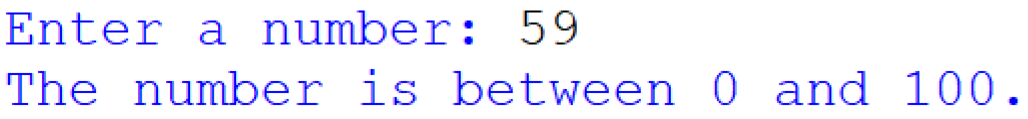
Nested *if-else* Statements

- Indented blocks of *if-else* and *if* statements can contain other *if-else* and *if* statements.
  - The *if-else* statements are said to be **nested**

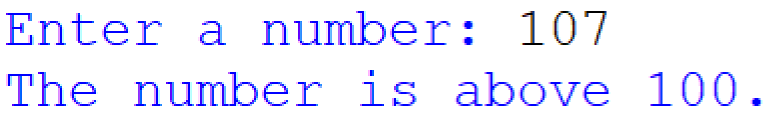


Example 4

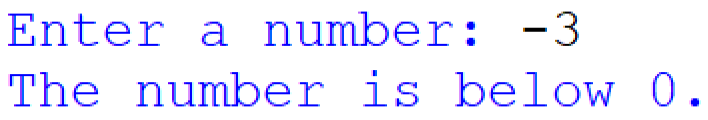
- Decide whether a number is between 0 and 100.



```
Enter a number: 59
The number is between 0 and 100.
```

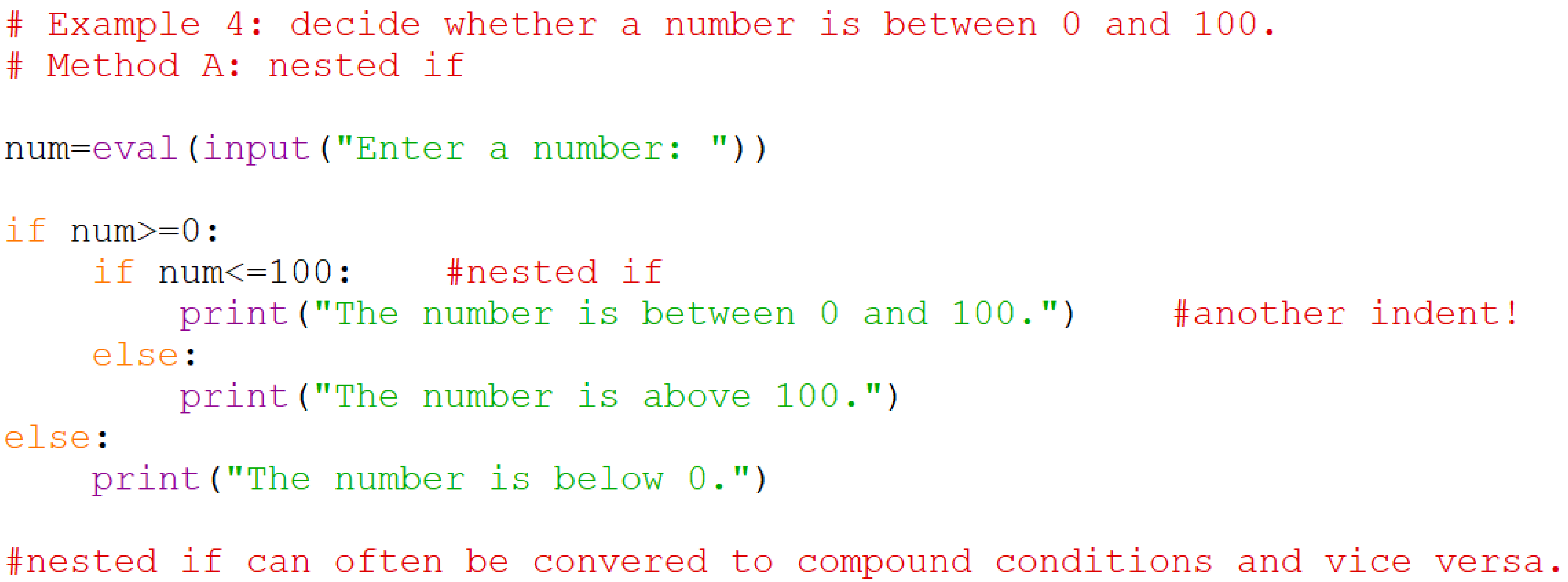


```
Enter a number: 107
The number is above 100.
```



```
Enter a number: -3
The number is below 0.
```

Example 4: Nested if



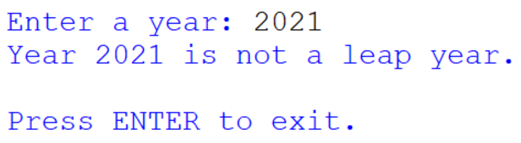
```
# Example 4: decide whether a number is between 0 and 100.
# Method A: nested if

num=eval(input("Enter a number: "))

if num>=0:
    if num<=100:    #nested if
        print("The number is between 0 and 100.")    #another indent!
    else:
        print("The number is above 100.")
else:
    print("The number is below 0.")

#nested if can often be convered to compound conditions and vice versa.
```

Example 4: compound condition

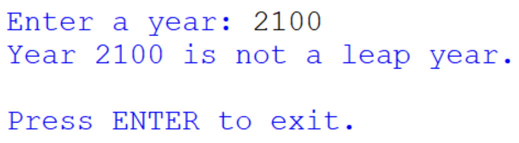
```
# Example 4: decide whether a number is between 0 and 100.
    # Method B: compound conditions.

num=eval(input("Enter a number: "))

if num>=0 and num<=100:  # alternatively, just write 0 <= num <= 100
    print("The number is between 0 and 100.")
elif num>100:
    print("The number is above 100.")
else:
    print("The number is below 0.")


#nested if can often be converted to using compound conditions and vice versa.
```

Classwork 5. Leap Years

- In the Gregorian calendar, each leap year has 366 days instead of 365, by adding one additional day in February. Every year divisible by four is a leap year, except for years divisible by 100 but not by 400. For instance, 1600, 2000, 2020 are leap years, but 1700, 1800, 1900 are not.
- Write a Python program that determines whether a given year entered by the user is a leap year. The output should resemble the following. Include the exit line. Upload the .py file and output screenshot on Canvas.



```
Enter a year: 2021
Year 2021 is not a leap year.

Press ENTER to exit.
```



```
Enter a year: 2100
Year 2100 is not a leap year.

Press ENTER to exit.
```