ADVANCED PYTHON PROGRAMMING

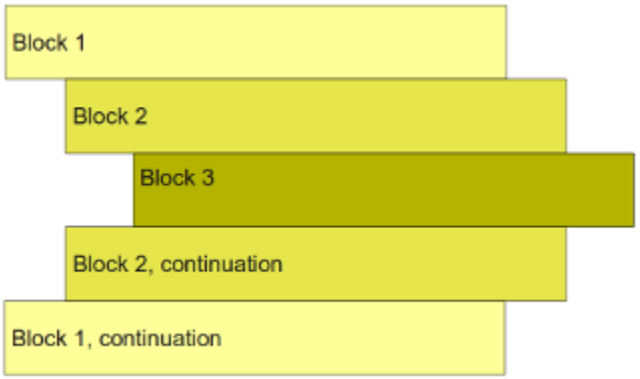
Control Flow

Peter Lo

Program Structure

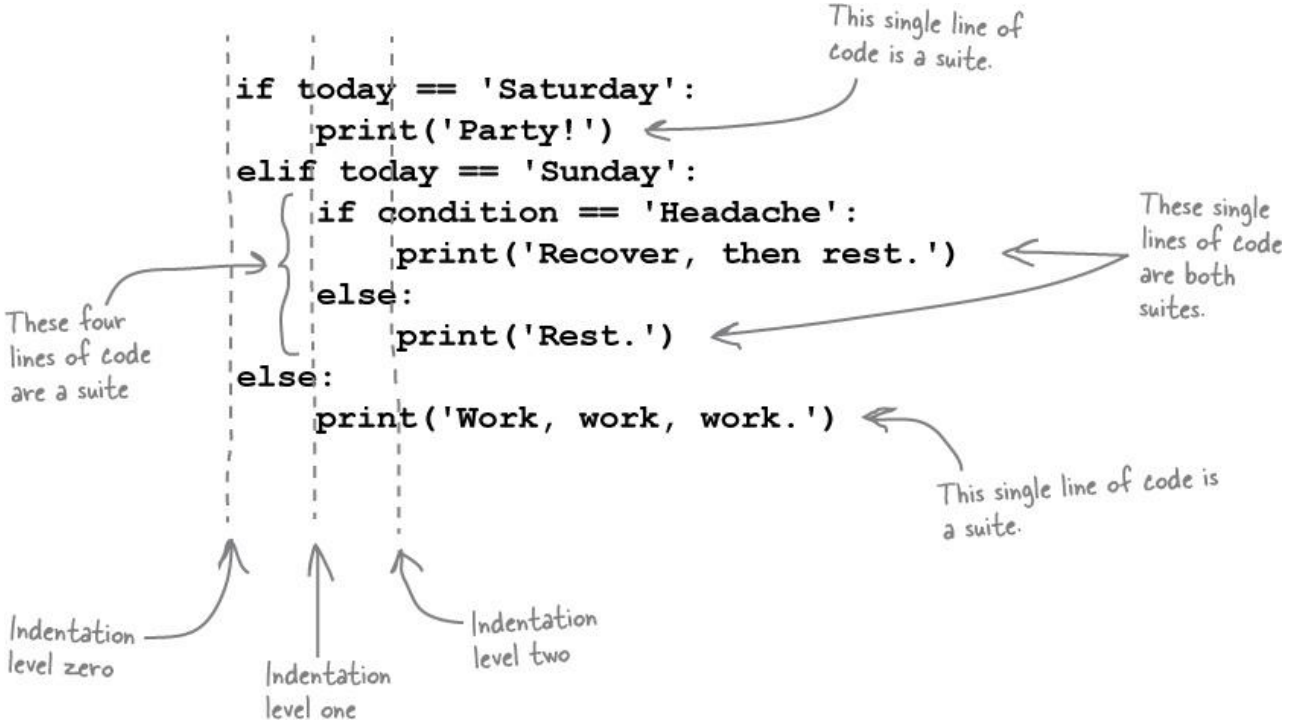
Blocks

- A block is a group of statements in a program or script.
- Usually consists of at least one statement and declarations for the block, depending on the programming or scripting language.
- A language which allows grouping with blocks, is called a Block Structured Language.
- Blocks can contain blocks as well, so we get a nested block structure.
- A block in a script or program functions as a means to group statements to be treated as if they were one statement.
- In many cases, it also serves as a way to limit the lexical scope of variables and functions.



Structuring with Indentation

- Python programs get structured through indentation,
  - i.e. code blocks are defined by their indentation



Relational and Logical Operators

Relational Operators

- Relational operators are used to compare values. It either returns True or False according to the condition.

Operator	Meaning	Example
>	Greater that - True if left operand is greater than the right	x > y
<	Less that - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

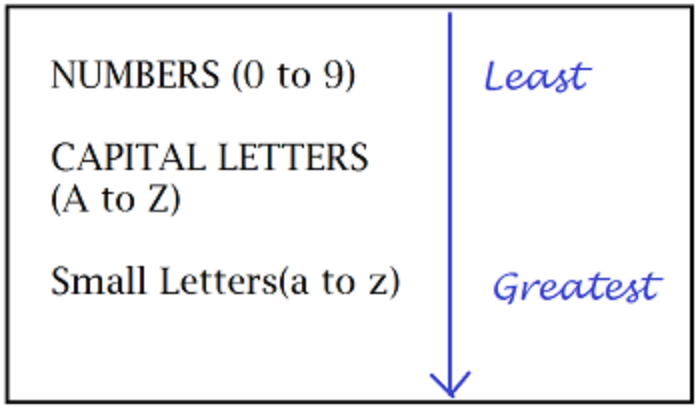
Example

Python

```
>>> 2 < 5
True
>>> 4 > 10
False
>>> 3 >= 3
True
>>>
>>> 5 == 6
False
>>> 6 != 9
True
```

Ordering in Python

- It's important to note that, in Python, strings are compared based on the alphabetical order and ASCII code values of their characters.
- The ascending order for these characters based on their respective ASCII codes is as follows:



Ascending order of characters based on ASCII codes

Logical Operators

- Logical operators are used to combine conditional statements.

Operator	Example	Meaning

`not`	`not x`	`True` if `x` is `False`
		`False` if `x` is `True`
		(Logically reverses the sense of `x`)

`or`	`x or y`	`True` if either `x` or `y` is `True` `False` otherwise
`and`	`x and y`	`True` if both `x` and `y` are `True` `False` otherwise

AND

- Returns True if both statements are true

P	Q	P and Q
True	True	True
True	False	False
False	True	False
False	False	False

```
# returns True because 5 is greater than 3 AND 5 is less than 10
x = 5
x > 3 and x < 10
```

```
True
```

OR

- Returns True if one of the statements is true

P	Q	P or Q
True	True	True
True	False	True
False	True	True
False	False	False

```
# returns True because one of the conditions are true (5 is greater than 3, but 5 is not less than 4)
x = 5
x > 3 or x < 4
```

```
True
```

NOT

- Reverse the result, returns False if the result is true

P	not P
True	False
False	True

```
# returns False because not is used to reverse the result
x = 5
not(x > 3 and x < 10)
```

```
False
```

Example

- To test if n falls between 2 and 5:
  - (2 < n) and (n < 5)
- A complete relational expression must be on either side of the logical operators and / or.
- The following is not a valid way to test if n falls between 2 and 5:
  - (2 < n < 5)

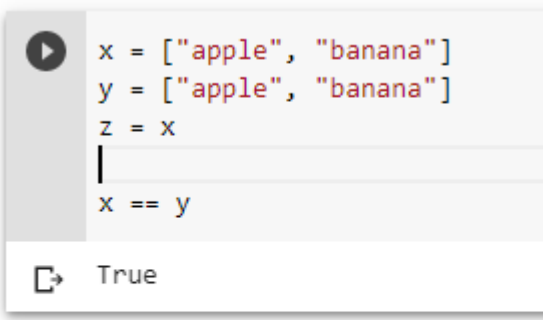
Identity Operators

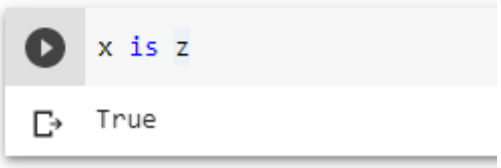
- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

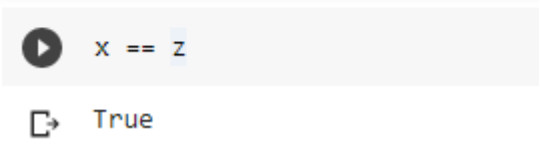
Operator	Description
is	Returns True if both variables are the same object
is not	Returns True if both variables are not the same object

Identity Operators vs Relational Operator

- The **==** operator compares the value or equality of two objects, whereas the Python **is** operator checks whether two variables point to the same object in memory.







Membership Operators

- Membership operators are used to test if a sequence is presented in an object:

Operator	Description
in	Returns True if a sequence with the specified value is present in the object
not in	Returns True if a sequence with the specified value is not present in the object

```
x = ["apple", "banana"]

"apple" in x
```

```
True
```

Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

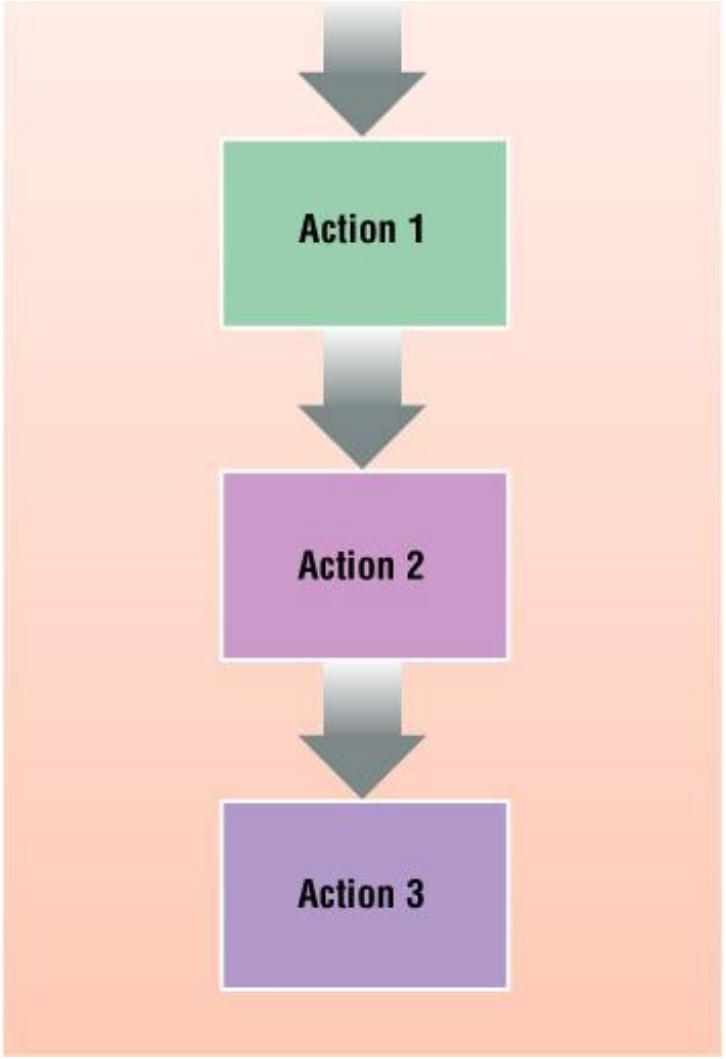
Control Structure

Control Structure

- Three types of control structure, derived from structured programming:
  - Sequences of instructions
  - Selection of alternative instructions (or groups of instructions)
  - Iteration (repetition) of instructions (or groups of instructions)

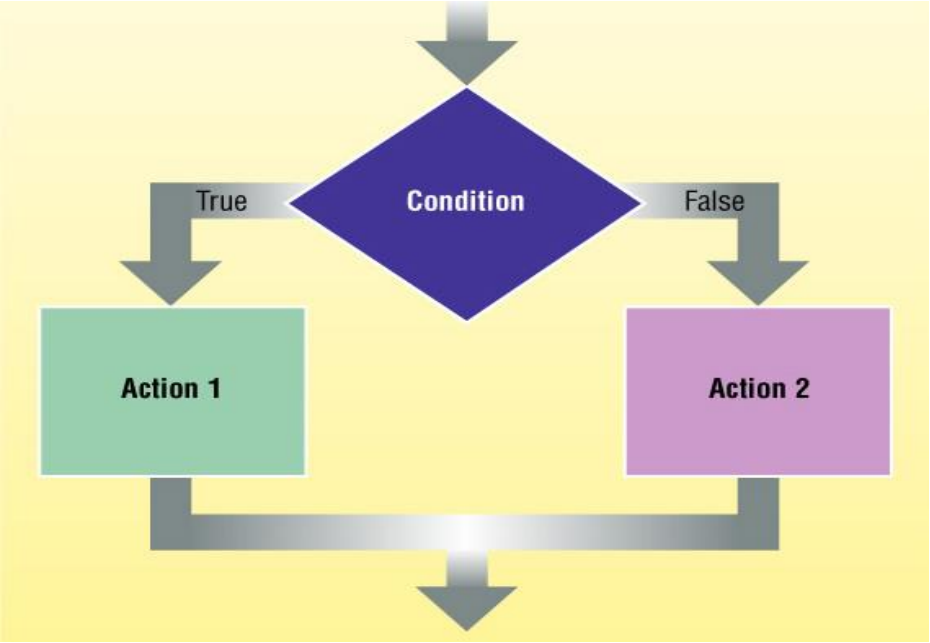
The Sequence Structure

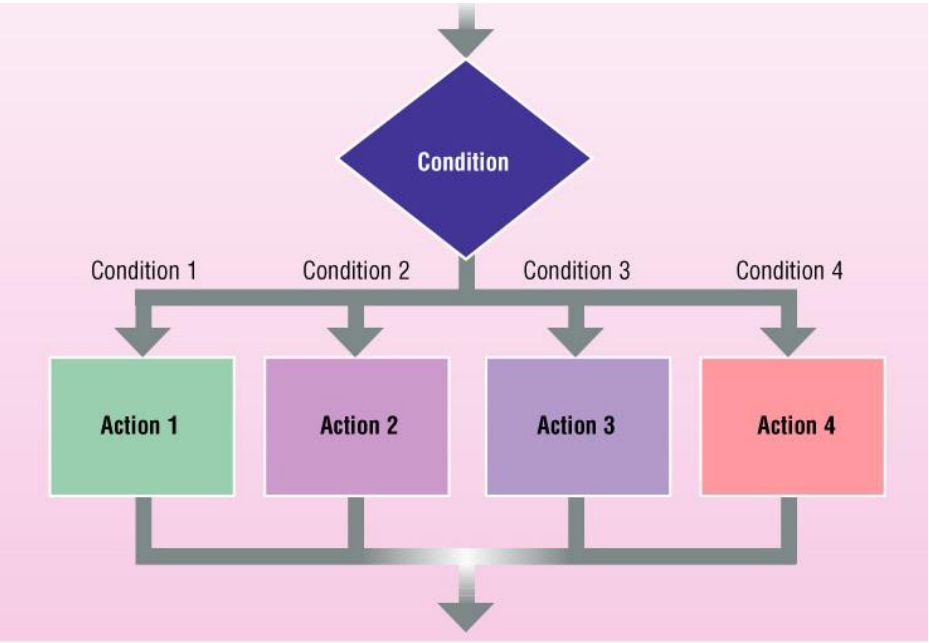
- Directs computer to process program instructions in a particular order
- Set of step-by-step instructions that accomplish a task



The Selection Structure

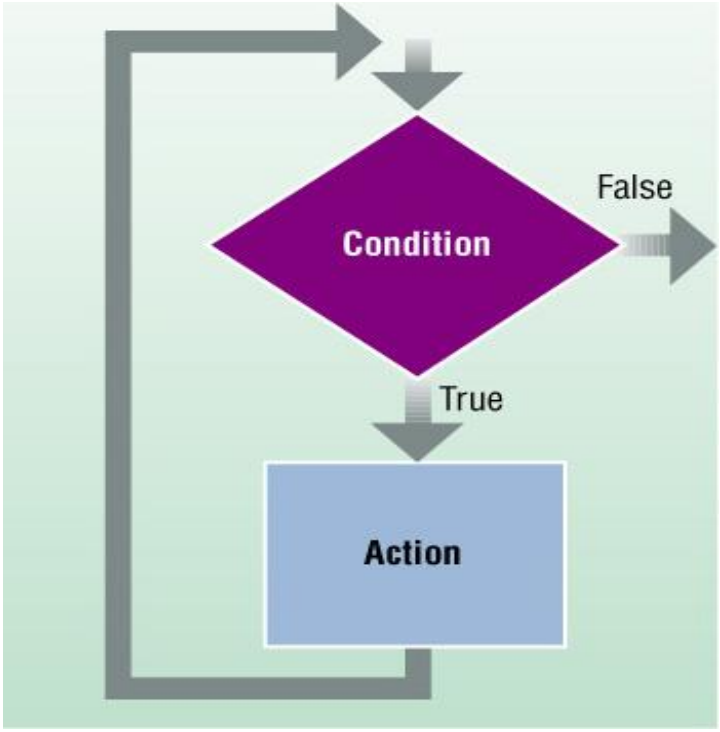
- Also called the **Decision Structure**, makes a decision and then takes appropriate action based on that decision
- Used every time you drive your car and approach an intersection

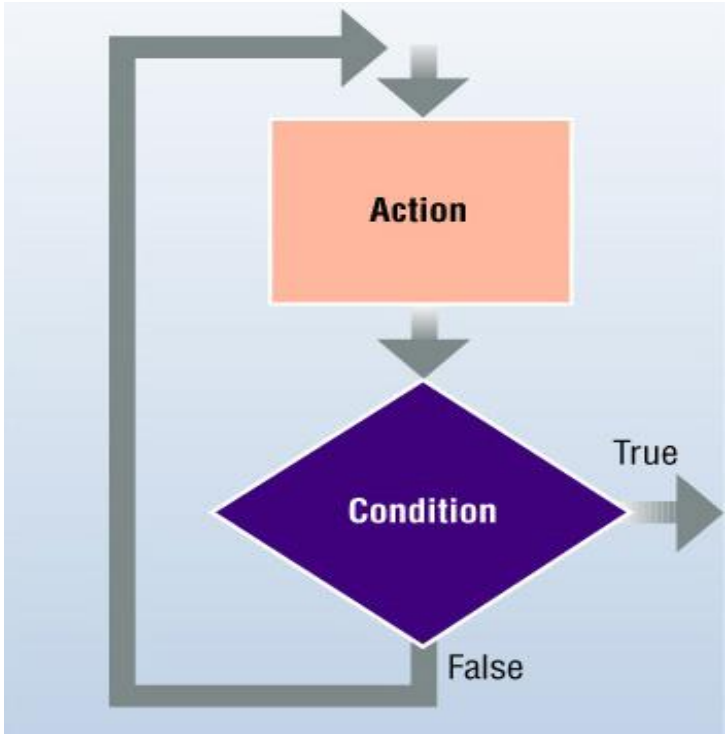




The Looping Structure

- Directs computer to repeat one or more instructions until some condition is met
- Also referred to as a **Loop**, **Repeating** or **Iteration**





The Selection Structure

Decision Making by if statement

- Conditional statements in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false.
- Decision making are handled by **if** statements in Python.
  - **if**: one test condition
  - **if…else**: two test condition
  - **If…elif…else**: multiple test condition

Simple if Statements

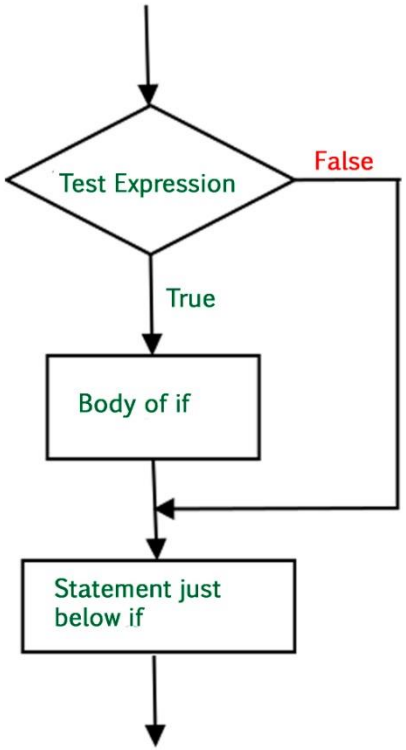
- If test expression is evaluated to true (nonzero) , statements inside the body of if is executed. If test expression is evaluated to false (0) , statements inside the body of if is skipped

example

```
x=20
y=10
if x > y :
  print(" X is bigger ")
```

output

```
X is bigger
```



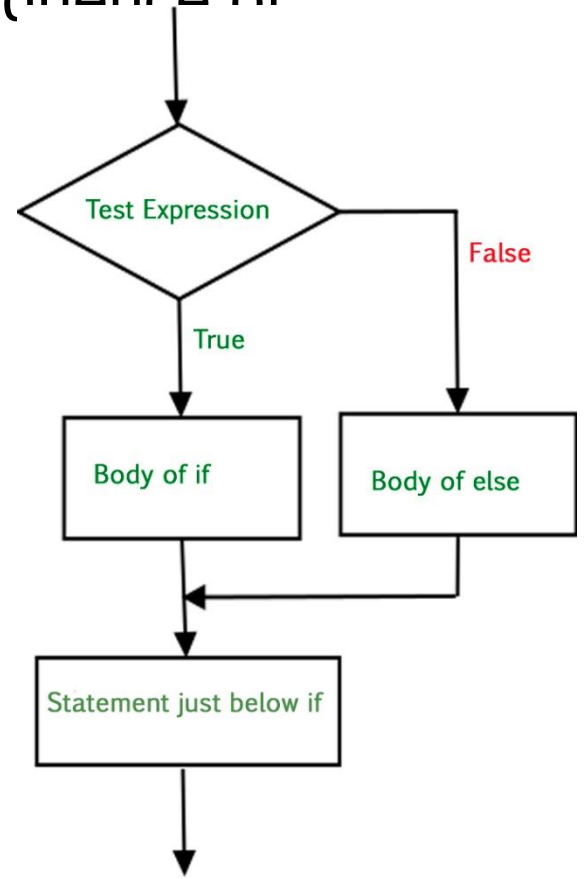
Two Branch if-else Statements

- The **else** statement is to specify a block of code to be executed, if the condition in the if statement is false. Thus, the else clause ensures that a sequence of statements is executed

```
x=10
y=20
if x > y :
  print(" X is bigger ")
else :
  print(" Y is bigger ")
```

output

```
Y is bigger
```



Multiple Tests if-elif Statements

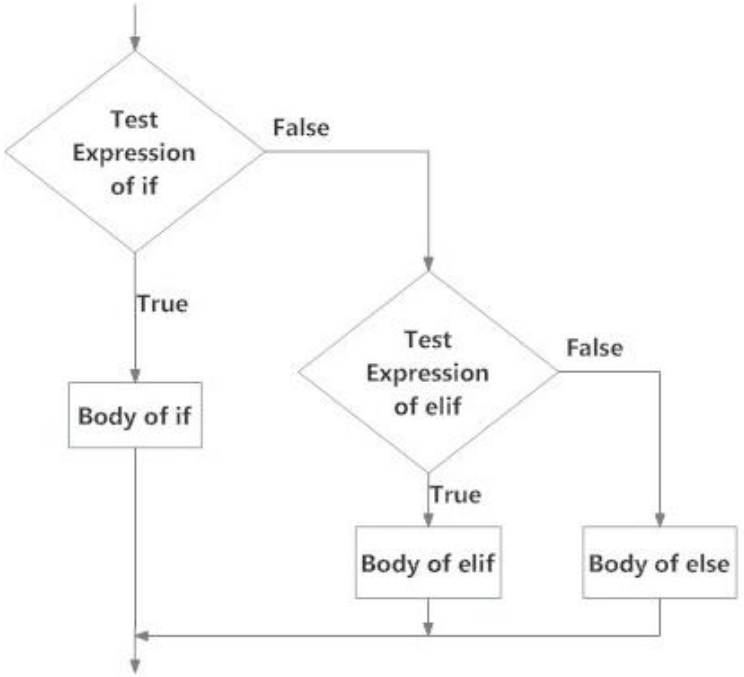
- The **elif** is short for else if and is useful to avoid excessive indentation.

example

```
x=500
if x > 500 :
  print(" X is greater than 500 ")
elif x < 500 :
  print(" X is less than 500 ")
elif x == 500 :
  print(" X is 500 ")
else :
  print(" X is not a number ")
```

output

```
X is 500
```



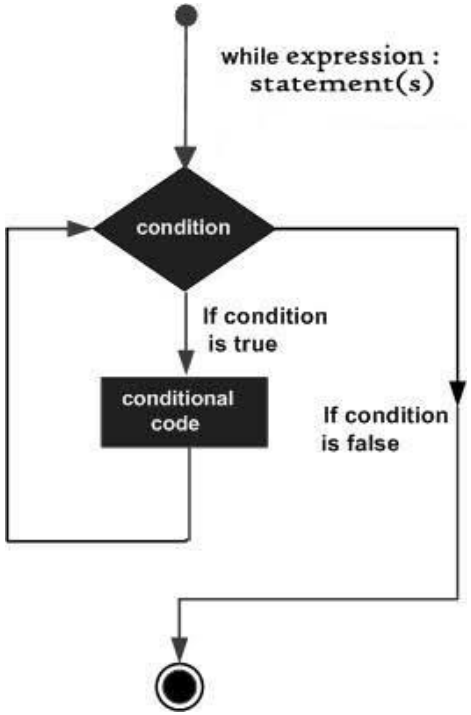
The Looping Structure

Looping

- A loop statement allows us to execute a statement or group of statements multiple times. There are two types of loops in Python:
  - **while** loop
    - Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
  - **for** loop
    - Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

The while Loop

- The while loop tells the computer to do something as long as the condition is met.
- Its construct consists of a block of code and a condition.
- The condition is evaluated, and if the condition is true, the code within the block is executed.
- This repeats until the condition becomes false.



The while loop

- While condition is true, do something
  - Two parts:
    - Condition tested (just like in If/else)
    - Set of Statements repeated as long as condition is true
  - General format:

    while *condition***:**

        statements
  - Notice
    - Indentation like If statement.
    - while statement ends when indentation ends
    - ‘:’ after condition

Example

- The block consisting of the print and increment statements, is executed repeatedly until count is no longer less than 5. With each iteration, the current value of the index count is displayed and then increased by 1

```
count = 0
while (count < 5):
    print ("The count is:", count)
    count += 1

print ("Outside Loop, count = ", count)
```

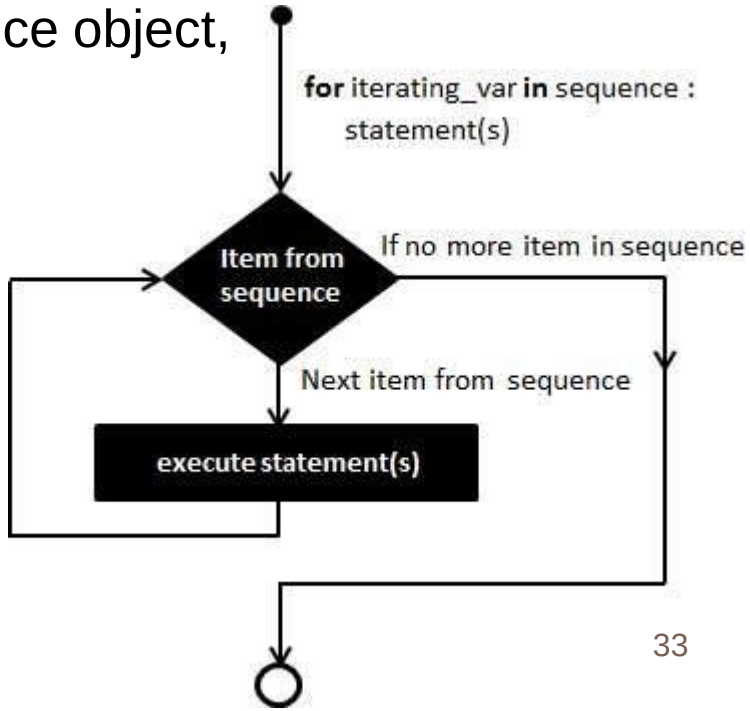
- When the above code is executed, it produces the following result

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Outside Loop, count =  5
```

The for Loop

- The for loop is an iterator based for loop.
- It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables.
- The for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through.
- The general syntax looks like this

```
for <variable> in <sequence>:
        <statements>
else:
        <statements>
```



The for Loop

- Count-Controlled loop: performs loop a specific number of times
  - Use a **for** statement to write count-controlled loop
    - Designed to work with sequence of data items
      - Iterates once for each item in the sequence
    - General format:

      for *variable* **in** *[val1, val2, etc]***:**

      statements
    - Target variable: the variable which is the target of the assignment at the beginning of each iteration

Simple For Loop

- A simple for loop to print 1 to 5

```
for num in [1, 2, 3, 4, 5]:
    print (num)
```

```
1
2
3
4
5
```

1st iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

2nd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

3rd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

4th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

5th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

For Loop in Numeric/String List

- For loop can be applied to numeric or string list:

```
Sample_List = [ 1, 3, 5, 7, 9]

for i in Sample_List:
    print (i)
```

```
1
3
5
7
9
```

```
Sample_List = ["a", "b", "c", "d", "e"]

for i in Sample_List:
    print (i)
```

```
a
b
c
d
e
```

For Loop in List

- For loop can be applied to a list.

```
Sample_List = [ "Freddie", 9, True, 1.1, 2001, ["Bone", "Little Ball"]]
for i in Sample_List:
    print (i)
```

```
Freddie
9
True
1.1
2001
['Bone', 'Little Ball']
```

Using range Function in for Loop

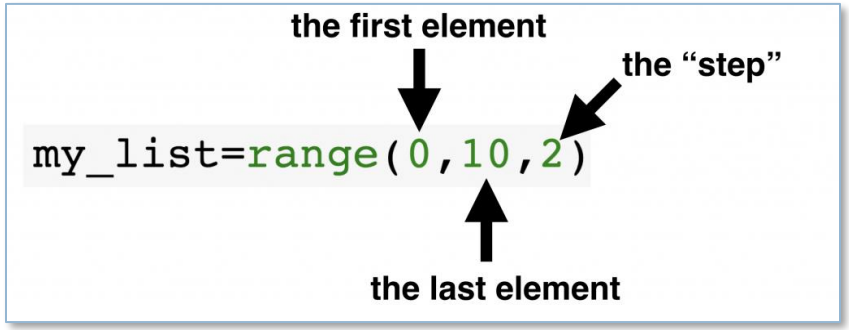
- Simplifies the process of writing a for loop
  - range returns an iterable object
    - Iterable: contains a sequence of values that can be iterated over
  - General Format:

    for *num* **in range(** *arg1, [arg2, [arg3]]* **):**

        statements
    - range characteristics:
      - One argument: used as ending limit
      - Two arguments: starting value and ending limit
      - Three arguments: third argument is step value

For Loop with Range

- A for-loop is to iterate some integer variable in increasing or decreasing order.
- Such a sequence of integer can be created using the function range(min_value, max_value):



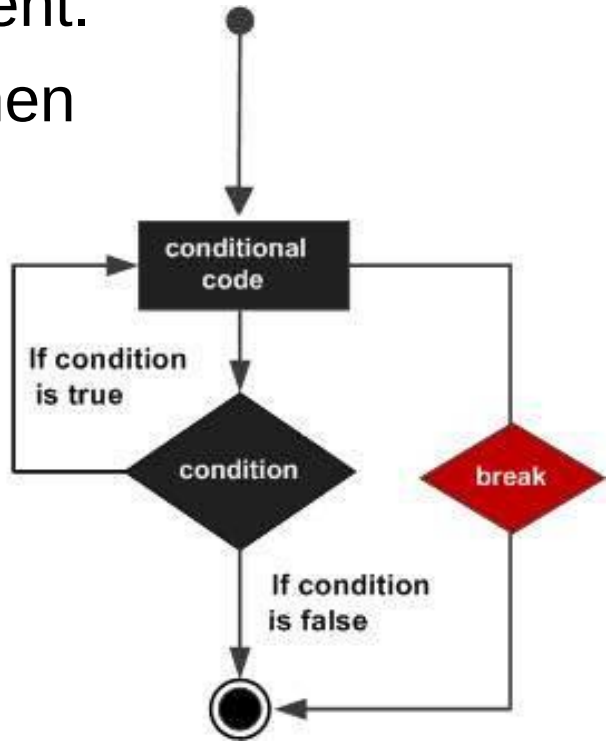

```
>>> # One parameter
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> # Two parameters
>>> for i in range(3, 6):
...     print(i)
...
3
4
5
>>> # Three parameters
>>> for i in range(4, 10, 2):
...     print(i)
...
4
6
8
>>> # Going backwards
>>> for i in range(0, -10, -2):
...     print(i)
...
0
-2
-4
-6
-8
```

Loop Control Statement

- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
  - **break** statement
    - Terminates the loop statement and transfers execution to the statement immediately following the loop.
  - **continue** statement
    - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
  - **pass** statement
    - The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
  - **else** statement
    - The else block just after for/while is executed only when the loop is NOT terminated by a break statement

The break statement

- It terminates the current loop and resumes execution at the next statement.
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop.



Example

- Let's look at an example that uses the break statement in a for loop

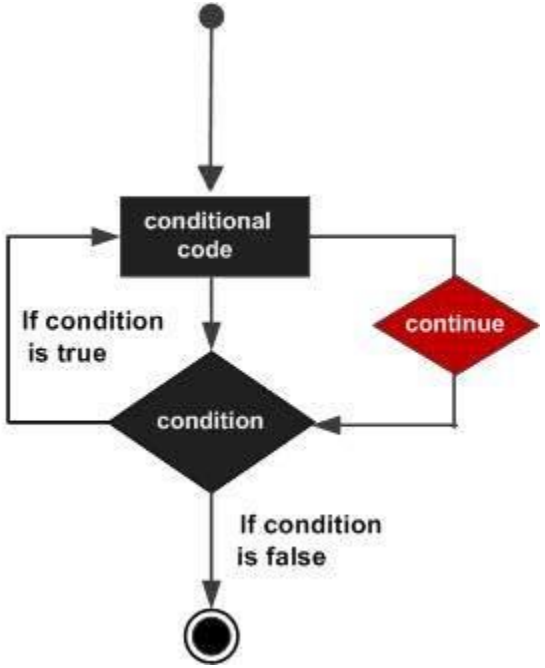
```
for i in range(5):
    if i == 3:
        break
    print (i)

print("Outside Loop")
```

- When we run this code, our output will be the following

```
0
1
2
Outside Loop
```

The continue statement

- It returns the control to the beginning of the while loop.
- The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.



Example

- The difference in using the continue statement rather than a break statement is that our code will continue despite the disruption when the variable number is evaluated as equivalent to 3.

```
for i in range(5):
    if i == 3:
        continue
    print (i)

print("Outside Loop")
```

- Let's take a look the result

```
0
1
2
4
Outside Loop
```

The pass statement

- When an external condition is triggered, the pass statement allows you to handle the condition without the loop being impacted in any way; all of the code will continue to be read unless a break or other statement occurs.
- As with the other statements, the pass statement will be within the block of code under the loop statement, typically after a conditional if statement.

Example

- Sample code:

```
for i in range(5):
    if i == 3:
        pass
    print (i)

print("Outside Loop")
```

- When the above code is executed, it produces following result

```
0
1
2
3
4
Outside Loop
```

The else Statement

- Python supports to have an else statement associated with a loop statement.
  - If the **else** statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
  - If the **else** statement is used with a while loop, the else statement is executed when the condition becomes false.

Example: while loop

- The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed

```
count = 0
while count < 5:
    print ("The count is:", count)
    count += 1
else:
    print ("Outside Loop, count = ", count)
```

- When the above code is executed, it produces the following result

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Outside Loop, count =  5
```

Example: for loop

- The following example, the else statement will only be executed if no element of the array is even, i.e. if statement has not been executed for any iteration. Therefore, for the array [1, 9, 8] the if is executed in third iteration of the loop and hence the else present after the for loop is ignored. In case of array [1, 3, 5] the if is not executed for any iteration and hence the else after the loop is executed.

```
InputList = [1, 9, 8]

for n in InputList:
    if n % 2 == 0:
        print ("even number found")
        break

# This else executes only if break is NEVER
# reached and loop terminated after all iterations.
else:
    print ("No even number")
```

```
even number found
```

```
InputList = [1, 3, 5]

for n in InputList:
    if n % 2 == 0:
        print ("even number found")
        break

# This else executes only if break is NEVER
# reached and loop terminated after all iterations.
else:
    print ("No even number")
```

```
No even number
```