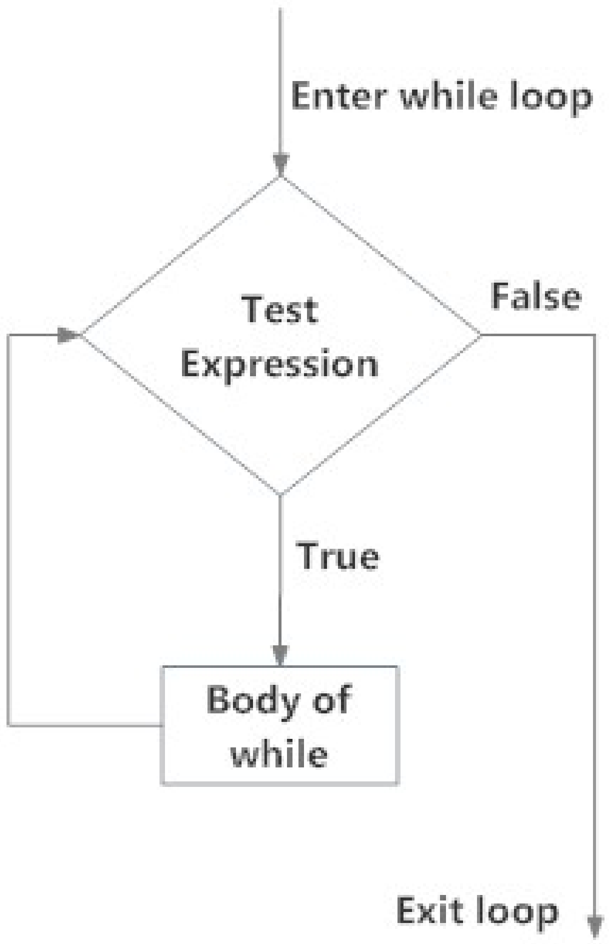


Fig: operation of while loop

More examples on *While*

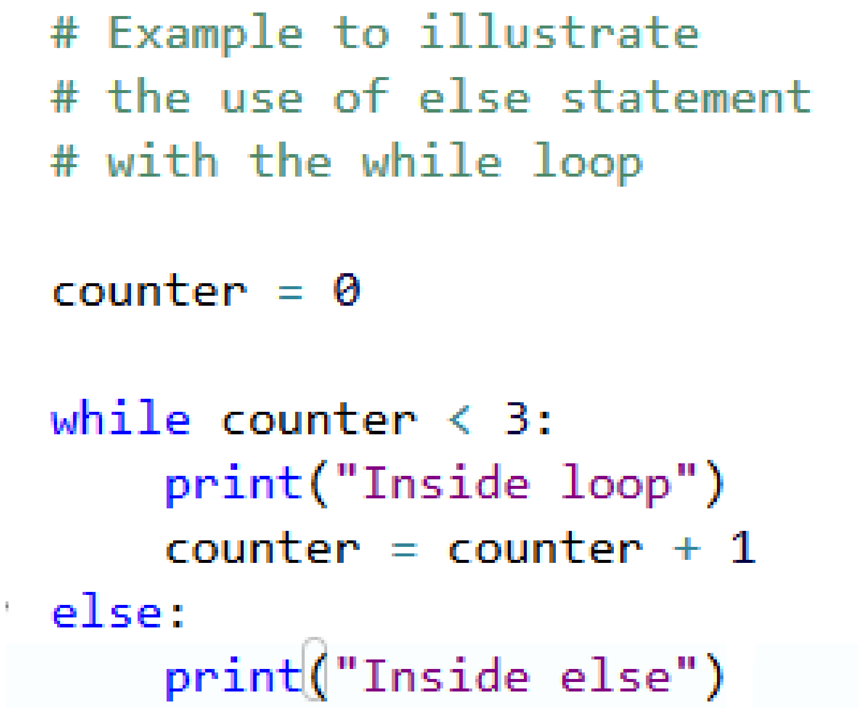
- Example #1

```python
n = input("Please enter 'hello':")
while n.strip() != 'hello':
    n = input("Please enter 'hello':")
```

- Example #2

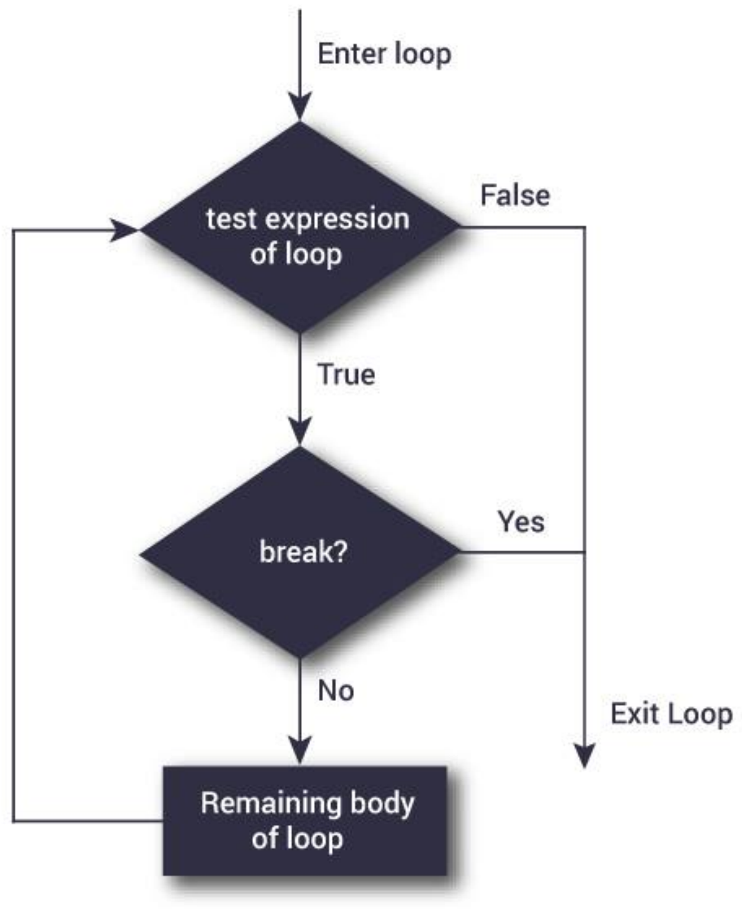
```python
while True:
    n = input("Please enter 'hello':")
    if n.strip() == 'hello':
        break
```

While and *else*

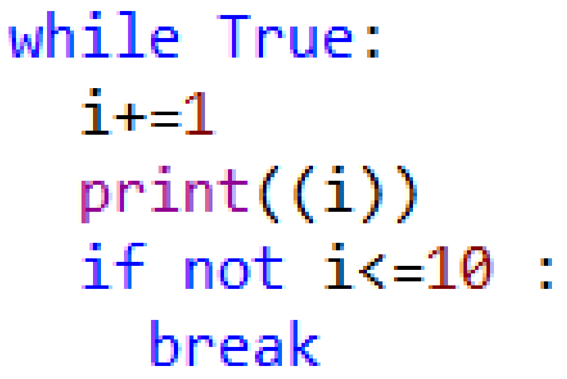


```
# Example to illustrate
# the use of else statement
# with the while loop

counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

While and *break*



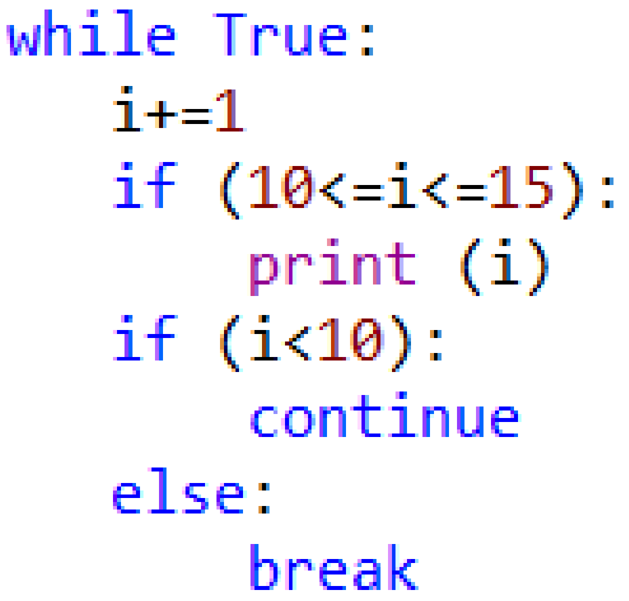
Different structure of the *while*



```python
while True:
  i+=1
  print((i))
  if not i<=10 :
    break
```

```
1
2
3
4
5
6
7
8
9
10
11
```

While with *continue*



```
while True:
    i+=1
    if (10<=i<=15):
        print (i)
    if (i<10):
        continue
    else:
        break
```

Lists and Tuples

- A list or tuple is a group of objects.

```
In [1]: myList = ['hello', -10, 2.5]
In [2]: print(type(myList))
<class 'list'>
```

- 'hello', -10 and 2.5 are said to be **elements** of myList
- Notice that each element of the list is of a different type, 'hello' is a string, -10 is an int and 2.5 is a float.
- myList contains 3 elements and has a length of 3

```
In [3]: print(len(myList))
3
```

- The first element of a list is at position 0 and the last element is at a position equal to the list's (length-1)

```
In [1]: print(myList[0])
hello
In [2]: print(myList[1])
-10
In [3]: print(myList[2])
2.5
In [4]: print(myList[3])
myList[3]
IndexError: list index out of range
```

- A reference to an element of a list is of the form:

  listname[index]

  where the index must be between 0 and len(listname)-1, otherwise an Index error occurs.

- The elements of a list are enclosed in []
- The elements of a tuple are enclosed in ()

```
In [1]: myTuple = (0, 2, 4, 6, 8)
In [2]: print(type(myTuple))
<class 'tuple'>
In [3]: len(myTuple)
5
```

- The values of the elements of a list may be changed.

```
In [4]: myList[0] = 17.5
In [5]: print(myList)
[17.5, -10, 2.5]
```

- The values of the elements of tuple cannot be changed.

```
In [6]: myTuple[2] = 99
myTuple[2] = 99
TypeError: 'tuple' object does not support item
assignment
```

```
In [1]: test = [0, 5, 10, 15, 20]
In [2]: test.append(25)
In [3]: print(test)
[0, 5, 10, 15, 20, 25]
```

- test.append(25) appends the value 25 to the end of the list

```
In [4]: test.insert(2, 99)
In [5]: print(test)
[0, 5, 99, 10, 15, 20, 25]
```

- test.insert(2, 99) inserts the value 99 into the list at position 2
- One list can be added to the end of another list:

```
In [6]: test = test + [30, 35]
In [7]: print(test)
[0, 5, 99, 10, 15, 20, 25, 30, 35]
```

```
In [1]: del test[2]
In [2]: print(test)
[0, 5, 10, 15, 20, 25, 30, 35]
```

- del test[2] deletes the element at position 2

```
In [3]: print(test.index(15))
3
In [4]: print(test.index(99))
test.index(99)
ValueError: 99 is not in list
```

- test.index(15) returns the position at which 15 occurs in the list, but test.index(99) gives an error as 99 does not occur in the list

```
In [1]: print(20 in test)
True
In [2]: print(99 in test)
False
```

- **20 in test** returns True as 20 is an element of test, but **99 in test** returns False as 99 is not an element of test.

```
In [3]: print(test)
[0, 5, 10, 15, 20, 25, 30, 35]
In [4]: print(test[-1])
35
In [5]: print(test[-2])
30
```

- test[-1] refers to the last element in a list, test[-2] refers to the second last element in test, etc.

```
In [1]: test = []
In [2]: print(test)
[]
```

- test = [] creates an empty list, a list with no elements.

- Create a list containing 10 zeros

```
In [3]: test = [0]*10
In [4]: print(test)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [5]: n = 5
In [6]: test = [1] * n
In [7]: print(test)
[1, 1, 1, 1, 1]

In [8]: n = 2.5
In [9]: test = [1] * n
test = [1] * n
TypeError: can't multiply sequence by non-int of
type 'float'
```

- Lists are often built using loops. Create a list containing the values from -40 to 40 in steps of 5.

```
In [1]: tempC = []
In [2]: valueC = -40
In [3]: maxC = 40
In [4]: incC = 5
In [5]: while valueC <= maxC:
           tempC.append(valueC)
           valueC += incC

In [6]: print(tempC)
[-40, -35, -30, -25, -20, -15, -10, -5, 0, 5,
10, 15, 20, 25, 30, 35, 40]
```

- A list of variables may refer to the elements in a list.

```
In [1]: aList = ['hello', 42, 3.14159]
In [2]: word, magic, myPi = aList
In [3]: print(word)
hello
In [4]: print(magic)
42
In [5]: print(myPi)
3.14159
```

- The number of variables before the assignment operator <u>must</u> match the number of elements in the list, otherwise an error occurs.

Tuples

- A tuple is a list of objects enclosed in ()
- Tuples are similar to lists but the elements of a tuple cannot be changed and are often thought of as constants.
- Operations that modify a tuple such as **append**, **del**, and **insert** cannot be used with tuples.
- Operations that reference an element of a tuple are permitted such as **index**.
- Often the () can be omitted when using tuples.

```
In [1]: test = 10, 2.5, 'hello'
In [2]: print(test)
(10, 2.5, 'hello')
In [3]: print(type(test))
<class 'tuple'>
```

- Tuples can be 'added' together to create a larger tuple.

```
In [1]: print(test)
(10, 2.5, 'hello')
In [2]: test = test + (-1e5, 'bye')
In [3]: print(test)
(10, 2.5, 'hello', -100000.0, 'bye')
```

- Indexing for tuples is the same as for lists.

```
In [4]: print(test[2])
hello
```

- Tuples have some desirable properties compared to lists.
  - Cannot accidently change the contents of a tuple.
  - Using tuples in programs is faster than using lists.

- The sum function returns the sum of the elements in a list or tuple.

```
In [1]: numbers = [1, 2, 3, 4, 5]
In [2]: print(sum(numbers))
15
```

- This can also be done using a loop as follows (sumLoop1.py):

```
numbers = (1, 2, 3, 4, 5, 6)
total = 0
i = 0
print("The numbers are:")
while i < len(numbers):
    number = numbers[i]
    print(number)
    total += number
    i += 1
# while
print("The total is %d" % total)
```

- The output from these statements is:

```
The numbers are:
1
2
3
4
5
6
The total is 21
```

- Compute the average of the numbers in a list (avgLoop1.py)

```
numbers = [1, 2, 3, 4, 5, 6]
total = 0
index = 0
print("The numbers are:")
while index < len(numbers):
    number = numbers[index]
    print(number)
    total += number
    index += 1
# while

print("The average is %g" % (total //
                              len(numbers)))
```

- The output from these statements is:

```
The numbers are:
1
2
3
4
5
6
The average is 3
```

- The average of 21 divided by 6 should be 3.5, what went wrong?
- Integer division was used to compute the average.
- Fix this with `total / len(numbers)`

- Compute the average of the numbers in a list (avgLoop2.py)

```
numbers = [1, 2, 3, 4, 5, 6]
total = 0
index = 0
print("The numbers are:")
while index < len(numbers):
    number = numbers[index]
    print(number)
    total += number
    index += 1
# while

print("The average is %g" % (total /
                             len(numbers)))
```

- The output from these statements is:

```
The numbers are:
1
2
3
4
5
6
The average is 3.5
```

- Write a program to display a temperature chart.
- In class demo of temperatureLists.py

```
tempC = (-40, -30, -20, -10, 0, 10, 20, 30, 40)
tempF = []
print("Celsius\tFahrenheit")
i = 0
while i < len(tempC):
    tempF.append(9./5 * tempC[i] + 32)
    print("%7d\t%10d" %(tempC[i],
                          tempF[i]))
    i += 1
# while
```

- The output from these statements is:

```
Celsius  Fahrenheit
    -40         -40
    -30         -22
    -20          -4
    -10          14
      0          32
     10          50
     20          68
     30          86
     40         104
```

- Reverse the order of the elements in a list (reverseElements.py):

```
aList = ['hi', 12, -2.5, True, 42]
idx = 0
last = len(aList) - 1
print("Original:", aList)
while idx < len(aList)//2:
    aList[idx], aList[last-idx] = \
         aList[last-idx] , aList[idx]
    idx += 1
print("Reversed:", aList)
```

- The output from these statements is:

```
Original: ['hi', 12, -2.5, True, 42]
Reversed: [42, True, -2.5, 12, 'hi']
```

- The order of the elements in a list may also be reversed using the *reverse()* function.

```
In [1]: a = [1,2,3,4,5]
In [2]: a.reverse()
In [3]: print(a)
[5, 4, 3, 2, 1]
```

- Notice the values of the elements in the list have been changed.

- Create a list containing the odd numbers between 1 and 99 (oddNumbers.py):

```
odds = []
number = 1
while number <= 99:
    odds.append(number)
    number += 2
# while
print(odds)
```

- The output from these statements is:

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77,
79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]
```

Rewrite the trajectory program to calculate the vertical position of the projectile at 0.1 increments in the horizontal direction (trajectoryLoop.py).

```
from math import pi, cos, tan
G = 9.81            # m/s**2
v0 = 5.0            # m/s
y0 = 1              # m
theta = 60          # degrees
x = 0.              # m
heights = []        # m
distances = []      # m

# display the values of the variables
print("""
gravity = %.2f m/s^2
v0      = %.1f m/s
y0      = %.1f m
theta   = %d degrees
x       = %.1f m
""" % (G, v0, y0, theta, x))

theta = theta * pi / 180  # convert theta to radians
```

```
# calculate the vertical position of the ball
# and display the result
y = y0
while y >= 0:
    heights.append(y)
    distances.append(x)
    x += .1
    y = x * tan(theta) - 1 / (2 * v0**2) * G *
        x**2 / cos(theta)**2 + y0
# while

i= 0
while i< len(heights):
    print("%.1f\t%.6f" % (distances[i],
                          heights[i]))
    i += 1
# while
```

The output from these statements is:

```
gravity = 9.81 m/s^2
v0 = 5.0 m/s
y0 = 1.0 m
theta = 60 degrees
x = 0.0 m

X        Y
0.000    1.000000
0.100    1.165357
0.200    1.315018
0.300    1.448983
0.400    1.567252
0.500    1.669825
0.600    1.756702
0.700    1.827884
0.800    1.883369
0.900    1.923158
1.000    1.947251
1.100    1.955648
1.200    1.948349
1.300    1.925354
1.400    1.886663
1.500    1.832276
1.600    1.762193
1.700    1.676414
1.800    1.574939
1.900    1.457769
2.000    1.324902
2.100    1.176339
2.200    1.012080
2.300    0.832125
2.400    0.636474
2.500    0.425127
2.600    0.198084
```

Demo trajectoryLoopQuadrartic.py

- Consider lists and tuples with only 1 element:

```
In [1]: type([1])
Out[1]: list
```

We get a list.

```
IN [2]: type((1))
Out[2]: int
```

We get an int.

```
IN [3]: type((1,))
Out[3]: tuple
```

We get a tuple.

A tuple with only one element requires a comma.

The for statement

- The **for** statement is of the form:

```
for <variable> in <list or tuple>:
    <block of statements>
```

```
In [1]: words = ['This', 'is', 'a', 'test.']
In [2]: for word in words:
            print(word, end=' ') # print on same line

This is a test.
```

- The statements that form the body of the for loop are indented.

- The first time through the for loop `word` is assigned the first element of `words` which is ‘This' and the value **`This`** is output.
- The second time through the for loop `word` is assigned the second element of `words` which is ‘is' and the value **`is`** is output.
- The third time through the for loop `word` is assigned the third element of `words` which is ‘a' and the value **`a`** is output.
- The fourth time through the for loop `word` is assigned the fourth element of `words` which is ‘test.' and the value **`test.`** is output.
- As there are no more elements in `words` the loop terminates.
- The **`end=' '`** ensures that all of the words are displayed on the same line separated by 1 space.

- The temperature conversion program can also be written using a for statement.
- In class demo of termperatureLists1.py

```
celsius = (-40, -30, -20, -10, 0, 10, 20, 30,
           40)
fahrenheit = []
print("Celsius\tFahrenheit")

for tempC in celsius:
    tempF = (9./5 * tempC + 32)
    fahrenheit.append(tempF)
    print("%7d\t%10d" %(tempC, tempF))
# for
```

- The output from this program is:

```
---------------------------------------------

Celsius Fahrenheit
    -40        -40
    -30        -22
    -20         -4
    -10         14
      0         32
     10         50
     20         68
     30         86
     40        104

Programmed by Stew Dent.
Date: Wed May 2 22:08:44 2018.
End of processing.
```

A string can be thought of as a tuple of characters.
A string can be used in a **for** statement.

```
sentence = "Programming in Python is fun!"

for letter in sentence:
    print(letter, end='')
print()
```

The output from this program is:

```
Programming in Python is fun!
```

Notice that `end=''` specifies an empty string so that the letters are printed next to each other on the same line.

This can also be done using a *while* loop.

```
sentence = "Programming in Python is fun!"
i = 0
while i < len(sentence):
    print(sentence[i], end='')
    i += 1
print()
```

The output from this program is:

```
Programming in Python is fun!
```