Python Programming

Döngü Yapıları ve Boolean

Objectives

- Belirli ve belirsiz döngü kavramlarını Python for ve while deyimlerinde gerçekleştirildikleri şekliyle anlamak.
- Etkileşimli döngü, başlangıç ve bitiş simgesi döngü programlama modellerini ve bunların bir Python while deyimi kullanarak uygulamalarını anlamak.
- Python'da dosya sonu döngüsünü programlama modelini ve bu tür döngüleri uygulama yollarını anlamak.
- İç içe döngü yapıları da dahil olmak üzere döngü örüntülerini içeren problemlere çözüm tasarlayabilme ve uygulayabilme.
- Boole cebirinin temel fikirlerini anlamak ve Boole operatörlerini içeren Boole ifadelerini analiz edebilmek ve yazabilmek.

for

for

```
animals = ["monkey", "lion", "elephant"]
for animal in animals:
    print(animal)
```

```
for number in [1, 2, 3, 4, 5]:
    square = number * number
    print(square)
```

```
for x in "Apple":
  print(x)
```

Öğrenciler dizisinde indis öğrenci olarak tanımlanır.

Örneğin:

```
students=["Furkan", "Ahmet", "Sumeyye",
"Seyda", "Yusuf", "Adem"]
for student in students:
        print(student)
```

The range() Function in Python

- Bir sayı aralığı oluşturmak ve bunun içinde döngü yapmak istemeniz oldukça yaygındır.
- Örneğin, 1'den 1000'e kadar saymak istiyorsanız, bir liste oluşturup içine 1000 numara yerleştirmek istemezsiniz.
- Bunun yerine yerleşik range() işlevini kullanabilirsiniz.Bu işlev, yinelenebilir bir sayı aralığı oluşturmayı kolaylaştırmak için tasarlanmıştır.
- Örneğin, range() kullanarak 1'den 10'a kadar olan sayıları yazdıralım:

```
for number in range(1, 11):
        print(number)
```

Örnek:

```
for number in range(1, 21):
    if number % 2 != 0:
        print(number)
```

Not: 1,2, …, 20; n-1

Örnek:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number == 3:
        break
    print(number)
else:
    print("The loop ran from start to finish.")
```

For Loops: A Quick Review

- The `for` statement allows us to iterate through a sequence of values.

-
  ```
  for <var> in <sequence>:
     <body>
  ```
- Döngü dizin değişkeni var, dizideki ardışık her değeri alır ve döngü gövdesindeki ifadeler, her değer için bir kez yürütülür.
- Kullanıcı tarafından girilen bir dizi sayının ortalamasını hesaplayabilen bir program yazmak istediğimizi varsayalım.
- Programı genel yapmak için, herhangi bir boyuttaki sayı kümesiyle çalışmalıdır.
- Girilen her sayıyı takip etmemize gerek yok, sadece dönen toplamı ve kaç sayının eklendiğini bilmemiz gerekiyor.
- Bir dizi sayı bir tür döngü tarafından işlenebilir. n sayı varsa, döngü n kez yürütülmelidir.
- Devam eden bir yekün değere ihtiyacımız var. Bu bir akümülatör kullanacaktır.

For Loops: A Quick Review

- Input the count of the numbers, n
- Initialize sum to 0
- Loop n times
  - Input a number, x
  - Add x to sum
- Output average as sum/n

```
# average1.py
#    A program to average a set of numbers
#    Illustrates counted loop with accumulator

def main():
    n = eval(input("How many numbers do you have? "))
    sum = 0.0
    for i in range(n):
        x = eval(input("Enter a number >> "))
        sum = sum + x
    print("\nThe average of the numbers is", sum / n)
```

- Note that sum is initialized to 0.0 so that `sum/n` returns a float!

For Loops: A Quick Review

```
How many numbers do you have? 5
Enter a number >> 32
Enter a number >> 45
Enter a number >> 34
Enter a number >> 76
Enter a number >> 45

The average of the numbers is 46.4
```

range

range

- The `range` function specifies a range of integers:
  - `range(`***start*****,** ***stop***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive)
  - It can also accept a third value specifying the change between values.
    - **`range(`*****start*****, *****stop*****, *****step*****`)`** - the integers between ***start*** (inclusive) and ***stop*** (exclusive) by ***step***
  - Example:

```
for x in range(5, 0, -1):
    print x
print "Blastoff!"
```

Output:

```
5
4
3
2
1
Blastoff!
```

Range: producing lists of integer numbers

- Often you need a regularly spread list of numbers from a beginning value to an end value.
- This is done by the range function:

```
""""" range gives a list of int numbers note that end value is NOT included! """
    r1 = range(11) # 0...10
    Print(r1) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    r2 = range(5,16) # 5...15
    Print( r2) # [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
    r3 = range(4,21,2) # 4...20 step 2
    Print( r3) # [4, 6, 8, 10, 12, 14, 16, 18, 20]
    r4 = range(15, 4, -5) # 15....5 step -5
    print (r4 )# [15, 10, 5]
```

- The general syntax is, *range (<startvalue>, <endvalue>, <stepsize>)*

- Take care: A strange (and somewhat illogical) detail of the range function is that the end value is excluded from the resulting list! The range function only works for integers!

Producing lists of floating point numbers

- If you need floating point numbers, use linspace from the Numpy module, a package that is very useful for technical and scientific applications. This package must be installed first, it is available at http://www.numpy.org/
- Don't forget to import the module in your script!
- Note: Here we use a slightly different method of import that avoids confusion between names of variables and numpy funtions. There are 3 ways to import functions from a module, see appendix.

```
""" for floating point numbers use linspace and logspace from numpy!"""
import numpy as np
r5 = np.linspace(0,2,9)
print r5
[ 0. 0.25 0.5 0.75 1. 1.25 1.5 1.75 2. ]
```

- The syntax for linspace is

```
linspace ( <startvalue>, <stopvalue>, <number_of_values> )
```

- The next example gives 9 logarithmically spaced values between 100 = 102 and 1000 = 103:

```
r6 = np.logspace(2, 3, 9)
print r6
[ 100. 133.35214322 177.827941 237.13737057 316.22776602
421.69650343 562.34132519 749.89420933 1000. ]
```

for

for loop

- **`for` loop**: Repeats a set of statements over a group of values.
  - Syntax:

    `for` ***variableName*** `in` ***groupOfValues*** `:`
            statements

    - We indent the statements to be repeated with tabs or spaces.
    - ***variableName*** gives a name to each value, so you can refer to it in the ***statements***.
    - ***groupOfValues*** can be a range of integers, specified with the `range` function.

  - Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Cumulative loops

- Some loops incrementally compute a value that is initialized outside the loop. This is sometimes called a *cumulative sum*.

```
sum = 0
for i in range(1, 11):
    sum = sum + (i * i)
print "sum of first 10 squares is", sum

Output:
sum of first 10 squares is 385
```

- **Exercise:** Write a Python program that computes the factorial of an integer.

Iterating through a list: for

- If we have to do something with all the elements of a list (or another sequence like a tuple etc.) one after the other, we use a for loop.
- The example uses the names of a list of names, one after one:

```
mynames = [ "Sam", "Pit", "Misch" ]
for n in mynames:
print "HELLO ", n
HELLO Sam
HELLO Pit
HELLO Misch
```

- This can also be done with numbers:

```
from math import *
for i in range (0, 5):
print i, "\t", sqrt(i)
0 0.0
1 1.0
2 1.41421356237
3 1.73205080757
4 2.0
```

Notes:

- Python's for loop is somewhat different of the for ... next loops of other programming languages. İn principle it can iterate through anything that can be cut into slices. So it can be used on lists of numbers, lists of text, mixed lists, strings, tuples etc.
- In Python the for ... next construction is often not to be missed, if we think in a "Pythonic" way.
- Example: if we need to calculate a lot of values, it is not a good idea to use a for loop, as this is very time consuming. It is better to use the Numpy module that provides array functions that can calculate a lot of values in one bunch (see below).

Iterating through a list: for

- If we have to do something with all the elements of a list (or another sequence like a tuple etc.) one after the other, we use a for loop.
- The example uses the names of a list of names, one after one:

```
mynames = [ "Sam", "Pit", "Misch" ]
for n in mynames:
print "HELLO ", n
HELLO Sam
HELLO Pit
HELLO Misch
```

- This can also be done with numbers:

```
from math import *
for i in range (0, 5):
print i, "\t", sqrt(i)
0 0.0
1 1.0
2 1.41421356237
3 1.73205080757
4 2.0
```

Iterating through a list: for

- This can also be done with numbers:

```
from math import *
for i in range (0, 5):
print i, "\t", sqrt(i)
0 0.0
1 1.0
2 1.41421356237
3 1.73205080757
4 2.0
```

Notes:

- Python's for loop is somewhat different of the for ... next loops of other programming languages. İn principle it can iterate through anything that can be cut into slices. So it can be used on lists of numbers, lists of text, mixed lists, strings, tuples etc.
- In Python the for ... next construction is often not to be missed, if we think in a "Pythonic" way.
- Example: if we need to calculate a lot of values, it is not a good idea to use a for loop, as this is very time consuming. It is better to use the Numpy module that provides array functions that can calculate a lot of values in one bunch (see below).

Iterating with indexing

- Sometimes you want to iterate through a list and have access to the index (the numbering) of the items of the list.
- The following example uses a list of colour codes for electronic parts and prints their index and the colour. As the colours list is well ordered, the index is also the colour value.

```
""" Dispay resistor colour code values"""
colours = [ "black", "brown", "red", "orange", "yellow",
"green", "blue", "violet", "grey","white" ]
cv = list (enumerate (colours))
for c in cv:
print c[0], "\t", c[1]
```

Iterating with indexing

- The list(enumerate (....)) function gives back a list of tuples cv that contain each an index (the numbering) and the colour value as text. If we print this we see
- [(0, 'black'), (1, 'brown'), (2, 'red'), (3, 'orange'), (4, 'yellow'), (5, 'green'), (6, 'blue'), (7, 'violet'), (8, 'grey'), (9, 'white')]
- Now we iterate on this, so we get the different tuples one after the other.
- From these tuples we print c[0], the index and c[1], the colour text, separated by a tab.
- So as a result we get:

```
0 black
1 brown
2 red
...
8 grey
9 white
```

Iterating with indexing

- The list(enumerate (....)) function gives back a list of tuples cv that contain each an index (the numbering) and the colour value as text. If we print this we see
- [(0, 'black'), (1, 'brown'), (2, 'red'), (3, 'orange'), (4, 'yellow'), (5, 'green'), (6, 'blue'), (7, 'violet'), (8, 'grey'), (9, 'white')]
- Now we iterate on this, so we get the different tuples one after the other.
- From these tuples we print c[0], the index and c[1], the colour text, separated by a tab.
- So as a result we get:

```
0 black
1 brown
2 red
...
8 grey
9 white
```

Loops: break, continue, else

- `break` and `continue` like C
- `else` after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print n, 'equals', x, '*', n/x
      break
  else:
    # loop fell through without finding a factor
    print n, 'is prime'
```

Avoiding for loops: vector functions

- For loops tend to get slow if there are many iterations to do.
- They are not necessary for calculations on numbers, if the Numpy module is used. It can be found here http://www.numpy.org/ and must be installed before using it.
- In this example we get 100 values of a sine function in one line of code:

```
import numpy as np
# calculate 100 values for x and y without a for loop
x = np.linspace(0, 2* np.pi, 100)
y = np.sin(x)
print x
print y
```

```
>>> for x in xrange(1, 3): # dikkat! xrange() fonksiyonu kullanıldı.
    for y in xrange(1, 4):
        print '%d * %d = %d' % (x, y, x*y)
# Cikti-------------
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
>>>
```

Örnek 3: Döngüden erken çıkma

```
for x in xrange(3):
```

```
>>> kelime = "Merhaba"
for x in kelime:
    print x
# Cikti--------------
M
e
r
h
a
b
a
>>>
```

Liste içindeki elemanları görüntüleme

```
>>> kelimeler = ['Ali', 5, 'Oya']
for x in kelimeler:
    print x
# Cikti--------------
Ali
5
Oya
>>>
```

while

while

- `while` **loop**: Executes a group of statements as long as a condition is True.
  - good for *indefinite loops* (repeat an unknown number of times)
- Syntax:

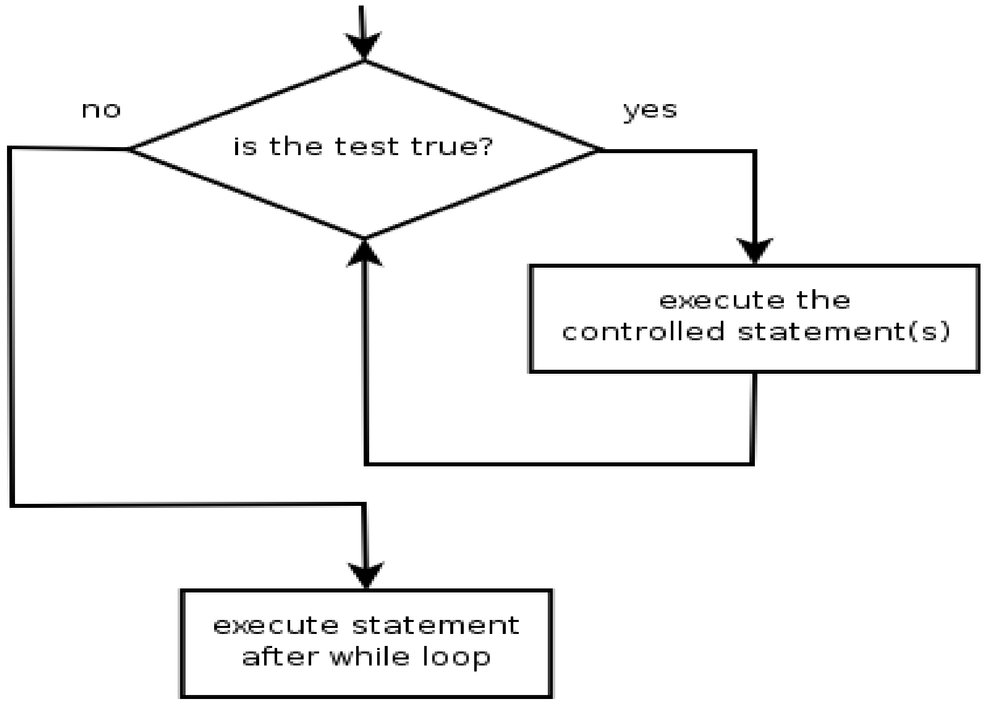
  while ***condition*:**

  statements

- Example:

```
number=1
while number < 200:
    print(number)
    number = number * 2

print(number)
d=math.sqrt(number)
print(d)
```


no
is the test true?
yes
execute the controlled statement(s)
execute statement after while loop

  - Output:

```
1 2 4 8 16 32 64 128
```

Dikkat: Yazım düzeni çok önemlidir.

Indefinite Loops

- That last program got the job done, but you need to know ahead of time how many numbers you'll be dealing with.
- What we need is a way for the computer to take care of counting how many numbers there are.
- The `for` loop is a definite loop, meaning that the number of iterations is determined when the loop starts.

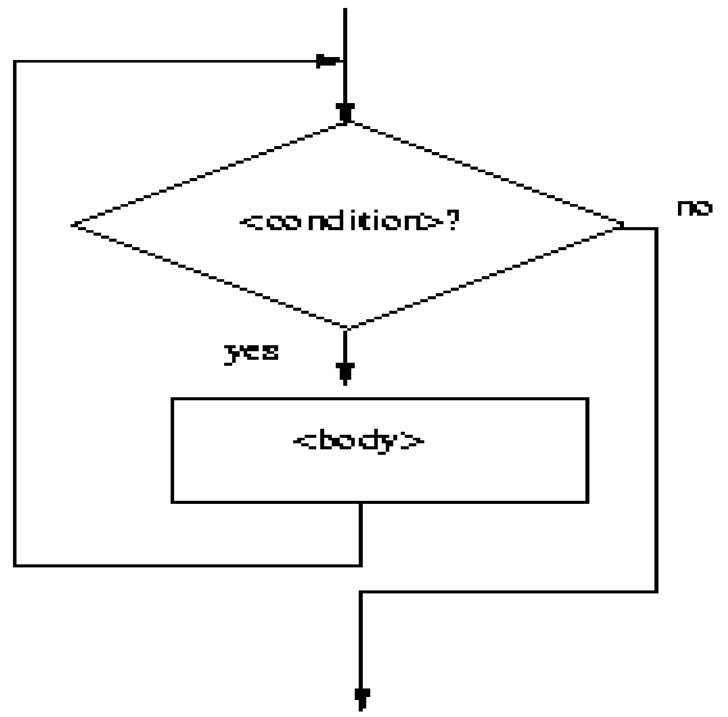
Indefinite Loops

- We can't use a definite loop unless we know the number of iterations ahead of time. We can't know how many iterations we need until all the numbers have been entered.
- We need another tool!
- The *indefinite* or *conditional* loop keeps iterating until certain conditions are met.

Indefinite Loops

-
  ```
  while <condition>:
      <body>
  ```
- `condition` is a Boolean expression, just like in `if` statements. The body is a sequence of one or more statements.
- Semantically, the body of the loop executes repeatedly as long as the condition remains true. When the condition is false, the loop terminates.

Indefinite Loops



- The condition is tested at the top of the loop. This is known as a *pre-test* loop. If the condition is initially false, the loop body will not execute at all.

Indefinite Loop

- Here’s an example of a `while` loop that counts from 0 to 10:

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

- The code has the same output as this `for` loop:

```
for i in range(11):
    print(i)
```

Indefinite Loop

- The `while` loop requires us to manage the loop variable `i` by initializing it to 0 before the loop and incrementing it at the bottom of the body.
- In the `for` loop this is handled automatically.

Indefinite Loop

- The `while` statement is simple, but yet powerful and dangerous – they are a common source of program errors.

-
  ```
  i = 0
  while i <= 10:
      print(i)
  ```
- What happens with this code?

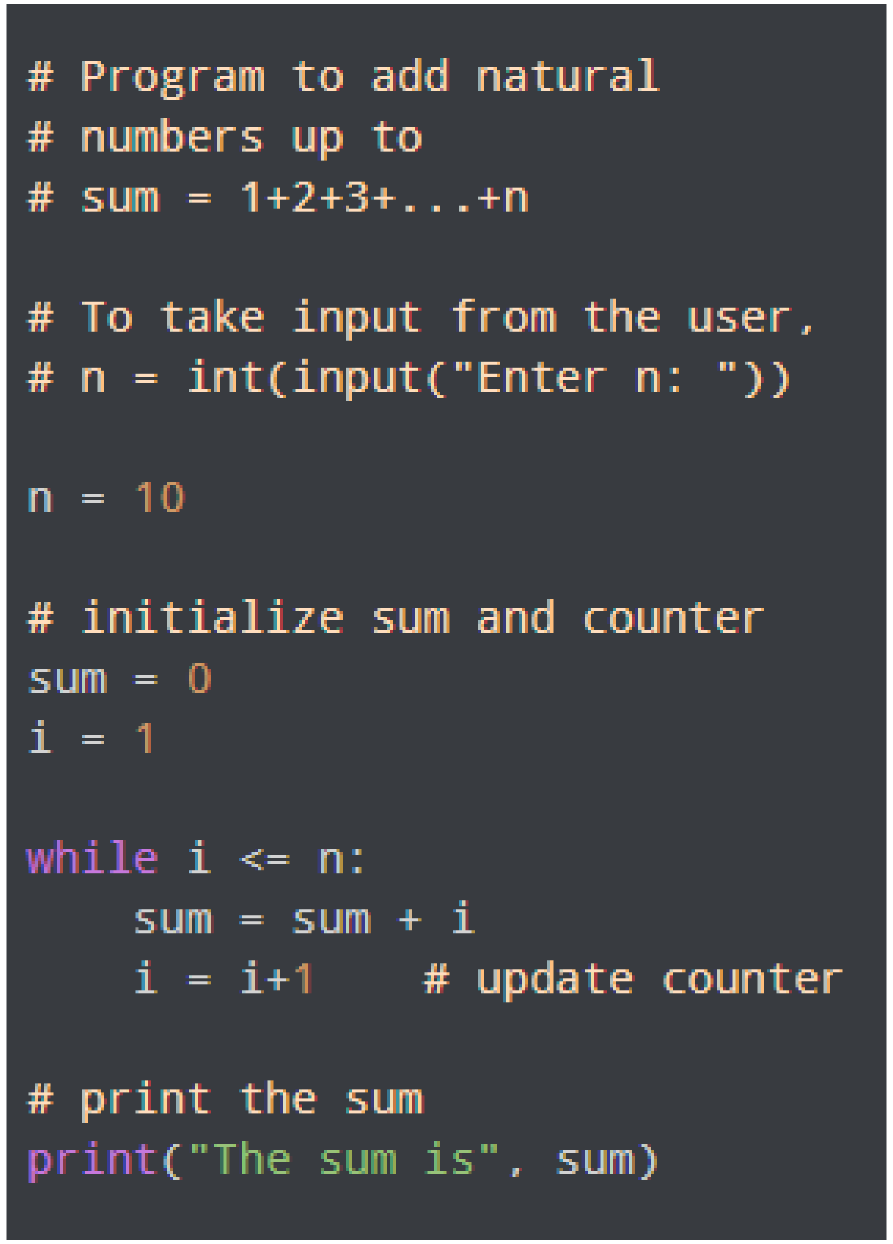
Indefinite Loop

- When Python gets to this loop, `i` is equal to 0, which is less than 10, so the body of the loop is executed, printing 0. Now control returns to the condition, and since `i` is still 0, the loop repeats, etc.
- This is an example of an *infinite loop*.

Indefinite Loop

- What should you do if you're caught in an infinite loop?
  - First, try pressing control-c
  - If that doesn't work, try control-alt-delete
  - If that doesn't work, push the reset button!

Example: While



```python
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

While

```
# Prints all letters except 'e' and 's'
i = 0
a = 'geeksforgeeks'

while i < len(a):
    if a[i] == 'e' or a[i] == 's':
        i += 1
        continue

    print('Current Letter :', a[i])
    i += 1
```

```
# Prints number of the 'e'
i = 0
J=0
a = 'geeksforgeeks'

while i < len(a):
    if a[i] == 'e':
        i += 1
        j=j+1
    else
        i += 1
print(‘Number of Letter :', j
```

While loops

- We can use the computer to do tedious (sıkıcı) tasks, like calculating the square roots of all integers between 0 and 100. In this case we use a while loop:

```
""" Calculate quare root of numbers 0 to 100"""

import math
i = 0
while i<= 100:
   print(i, "\t\t" ,math.sqrt(i)) # \t: boşluk bırakır
   i = i + 1

print("READY!")
```

```
0 0.0
1 1.0
2 1.41421356237
3 1.73205080757
.....
98 9.89949493661
99 9.94987437107
100 10.0
READY!
```

While loops

- The syntax is :

  while <condition> :

  <....

  block of statements

  ...>
- The block of statements is executed as long as <condition> is True, in our example as long as i <= 100.

- Take care:

  Don't forget the ":" at the end of the while statement

  Don't forget to indent the block that should be executed inside the while loop!

- The indentation can be any number of spaces (4 are standard), but it must be consistent for the whole block.

While loops

- Avoid endless loops!
- In the following example the loop runs infinitely, as the condition is always true:

```
i = 0
while i<= 5 :
Print i
```

- The only way to stop it is by pressing <Ctrl>-C
- Note: i = i +1 can be written in a shorter and more "Pythonic" way as
- i += 1

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""

a, b = 0, 1
while b < 10:
    print(b)
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Interactive Loops

- One good use of the indefinite loop is to write *interactive loops*. Interactive loops allow a user to repeat certain portions of a program on demand.
- Remember how we said we needed a way for the computer to keep track of how many numbers had been entered? Let's use another accumulator, called `count`.

Interactive Loops

- At each iteration of the loop, ask the user if there is more data to process. We need to preset it to “yes” to go through the loop the first time.
-
```
set moredata to “yes”
while moredata is “yes”
   get the next data item
   process the item
   ask user if there is moredata
```

Interactive Loops

- Combining the interactive loop pattern with accumulators for sum and count:
-
```
initialize sum to 0.0
initialize count to 0
set moredata to “yes”
while moredata is “yes”
   input a number, x
   add x to sum
   add 1 to count
   ask user if there is moredata
output sum/count
```

Interactive Loops

```
# average2.py
#    A program to average a set of numbers
#    Illustrates interactive loop with two accumulators

def main():
    moredata = "yes"
    sum = 0.0
    count = 0
    while moredata[0] == 'y':
        x = eval(input("Enter a number >> "))
        sum = sum + x
        count = count + 1
        moredata = input("Do you have more numbers (yes or no)? ")
    print("\nThe average of the numbers is", sum / count)
```

- Using string indexing (moredata[0]) allows us to accept “y”, “yes”, “yeah” to continue the loop

Interactive Loops

```
Enter a number >> 32
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? yes
Enter a number >> 34
Do you have more numbers (yes or no)? yup
Enter a number >> 76
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? nah

The average of the numbers is 46.4
```

Sentinel Loops

- A *sentinel loop* continues to process data until reaching a special value that signals the end.
- This special value is called the *sentinel*.
- The sentinel must be distinguishable from the data since it is not processed as part of the data.

Sentinel Loops

-
  ```
  get the first data item
  while item is not the sentinel
     process the item
     get the next data item
  ```
- The first item is retrieved before the loop starts. This is sometimes called the *priming read*, since it gets the process started.
- If the first item is the sentinel, the loop terminates and no data is processed.
- Otherwise, the item is processed and the next one is read.

Sentinel Loops

- In our averaging example, assume we are averaging test scores.
- We can assume that there will be no score below 0, so a negative number will be the sentinel.

Sentinel Loops

```
# average3.py
#    A program to average a set of numbers
#    Illustrates sentinel loop using negative input as sentinel

def main():
    sum = 0.0
    count = 0
    x = eval(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = eval(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", sum / count)
```

Sentinel Loops

```
Enter a number (negative to quit) >> 32
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> 34
Enter a number (negative to quit) >> 76
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> -1

The average of the numbers is 46.4
```

Sentinel Loops

- This version provides the ease of use of the interactive loop without the hassle of typing ‘y’ all the time.
- There’s still a shortcoming – using this method we can’t average a set of positive *and negative* numbers.
- If we do this, our sentinel can no longer be a number.

Sentinel Loops

- We could input all the information as strings.
- Valid input would be converted into numeric form. Use a character-based sentinel.
- We could use the *empty string* (“”)!

Sentinel Loops

```
initialize sum to 0.0
initialize count to 0
input data item as a string, xStr
while xStr is not empty
   convert xStr to a number, x
   add x to sum
   add 1 to count
   input next data item as a string, xStr
Output sum / count
```

Sentinel Loops

```
# average4.py
#    A program to average a set of numbers
#    Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", sum / count)
```

Sentinel Loops

```
Enter a number (<Enter> to quit) >> 34
Enter a number (<Enter> to quit) >> 23
Enter a number (<Enter> to quit) >> 0
Enter a number (<Enter> to quit) >> -25
Enter a number (<Enter> to quit) >> -34.4
Enter a number (<Enter> to quit) >> 22.7
Enter a number (<Enter> to quit) >>

The average of the numbers is 3.38333333333
```

File Loops

- The biggest disadvantage of our program at this point is that they are interactive.
- What happens if you make a typo on number 43 out of 50?
- A better solution for large data sets is to read the data from a file.

File Loops

```
# average5.py
#     Computes the average of numbers listed in a file.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in infile.readlines():
        sum = sum + eval(line)
        count = count + 1
    print("\nThe average of the numbers is", sum / count)
```

File Loops

- Many languages don't have a mechanism for looping through a file like this. Rather, they use a sentinel!
- We could use `readline` in a loop to get the next line of the file.
- At the end of the file, `readline` returns an empty string, ""

File Loops

-
  ```
  line = infile.readline()
  while line != ""
     #process line
     line = infile.readline()
  ```
- Does this code correctly handle the case where there’s a blank line in the file?
- Yes. An empty line actually ends with the newline character, and `readline` includes the newline. “\n” != “”

File Loops

```
# average6.py
#     Computes the average of numbers listed in a file.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        sum = sum + eval(line)
        count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
```

Nested Loops

- In the last chapter we saw how we could nest `if` statements. We can also nest loops.
- Suppose we change our specification to allow any number of numbers on a line in the file (separated by commas), rather than one per line.

Nested Loops

- At the top level, we will use a file-processing loop that computes a running sum and count.

```
sum = 0.0
count = 0
line = infile.readline()
while line != "":
    #update sum and count for values in line
    line = infile.readline()
print("\nThe average of the numbers is", sum/count)
```

Nested Loops

- In the next level in we need to update the `sum` and `count` in the body of the loop.
- Since each line of the file contains one or more numbers separated by commas, we can split the string into substrings, each of which represents a number.
- Then we need to loop through the substrings, convert each to a number, and add it to `sum`.
- We also need to update `count`.

Nested Loops

-
  ```
  for xStr in line.split(","):
      sum = sum + eval(xStr)
      count = count + 1
  ```
- Notice that this `for` statement uses `line`, which is also the loop control variable for the outer loop.

Nested Loops

```
# average7.py
#     Computes the average of numbers listed in a file.
#     Works with multiple numbers on a line.

import string

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        for xStr in line.split(","):
            sum = sum + eval(xStr)
            count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
```

Nested Loops

- The loop that processes the numbers in each line is indented inside of the file processing loop.
- The outer `while` loop iterates once for each line of the file.
- For each iteration of the outer loop, the inner `for` loop iterates as many times as there are numbers on the line.
- When the inner loop finishes, the next line of the file is read, and this process begins again.

Nested Loops

- Designing nested loops –
  - Design the outer loop without worrying about what goes inside
  - Design what goes inside, ignoring the outer loop.
  - Put the pieces together, preserving the nesting.

Computing with Booleans

- `if` and `while` both use Boolean expressions.
- Boolean expressions evaluate to `True` or `False`.
- So far we've used Boolean expressions to compare two values, e.g. (`while x >= 0`)

Boolean Operators

- Sometimes our simple expressions do not seem expressive enough.
- Suppose you need to determine whether two points are in the same position – their *x* coordinates are equal and their *y* coordinates are equal.

Boolean Operators

-
  ```
  if p1.getX() == p2.getX():
      if p1.getY() == p2.getY():
          # points are the same
      else:
          # points are different
  else:
      # points are different
  ```
- Clearly, this is an awkward way to evaluate multiple Boolean expressions!
- Let’s check out the three Boolean operators `and`, `or`, and `not`.

Boolean Operators

- The Boolean operators `and` and `or` are used to combine two Boolean expressions and produce a Boolean result.
- `<expr> and <expr>`
- `<expr> or <expr>`

Boolean Operators

- The `and` of two expressions is true exactly when both of the expressions are true.
- We can represent this in a *truth table*.

P	*Q*	*P and Q*
T	T	T
T	F	F
F	T	F
F	F	F

Boolean Expressions

- In the truth table, *P* and *Q* represent smaller Boolean expressions.
- Since each expression has two possible values, there are four possible combinations of values.
- The last column gives the value of `P and Q`.

Boolean Expressions

- The `or` of two expressions is true when either expression is true.

P	*Q*	*P or Q*
T	T	T
T	F	T
F	T	T
F	F	F

Boolean Expressions

- The only time `or` is false is when both expressions are false.
- Also, note that `or` is true when both expressions are true. This isn’t how we normally use “or” in language.

Boolean Operators

- The `not` operator computes the opposite of a Boolean expression.
- `not` is a *unary* operator, meaning it operates on a single expression.

P	*not P*
T	F
F	T

Boolean Operators

- We can put these operators together to make arbitrarily complex Boolean expressions.
- The interpretation of the expressions relies on the precedence rules for the operators.

Boolean Operators

- Consider `a or not b and c`
- How should this be evaluated?
- The order of precedence, from high to low, is `not`, `and`, `or`.
- This statement is equivalent to `(a or ((not b) and c))`
- Since most people don’t memorize the the Boolean precedence rules, use parentheses to prevent confusion.

Boolean Operators

- To test for the co-location of two points, we could use an `and`.
-
```
if p1.getX() == p2.getX() and p2.getY() == p1.getY():
    # points are the same
else:
    # points are different
```
- The entire condition will be true *only* when both of the simpler conditions are true.

Boolean Operators

- Say you’re writing a racquetball simulation. The game is over as soon as either player has scored 15 points.
- How can you represent that in a Boolean expression?
- `scoreA == 15 or scoreB == 15`
- When either of the conditions becomes true, the entire expression is true. If neither condition is true, the expression is false.

Boolean Operators

- We want to construct a loop that continues as long as the game is **not** over.
- You can do this by taking the negation of the game-over condition as your loop condition!

-
  ```
  while not(scoreA == 15 or scoreB == 15):
      #continue playing
  ```

Boolean Operators

- Some racquetball players also use a shutout condition to end the game, where if one player has scored 7 points and the other person hasn’t scored yet, the game is over.

-
  ```
  while not(scoreA == 15 or scoreB == 15 or \
  (scoreA == 7 and scoreB == 0) or (scoreB == 7 and scoreA == 0):
      #continue playing
  ```

Boolean Operators

- Let's look at volleyball scoring. To win, a volleyball team needs to win by at least two points.
- In volleyball, a team wins at 15 points
- If the score is 15 – 14, play continues, just as it does for 21 – 20.
- `(a >= 15 and a - b >= 2) or (b >= 15 and b - a >= 2)`
- `(a >= 15 or b >= 15) and abs(a - b) >= 2`

Boolean Algebra

- The ability to formulate, manipulate, and reason with Boolean expressions is an important skill.
- Boolean expressions obey certain algebraic laws called *Boolean logic* or *Boolean algebra.*

Boolean Algebra

Algebra	Boolean algebra
a * 0 = 0	*a* and false == false
a * 1 = *a*	*a* and true == *a*
a + 0 = *a*	*a* or false == *a*

- `and` has properties similar to multiplication
- `or` has properties similar to addition
- `0` and `1` correspond to false and true, respectively.

Boolean Algebra

- Anything `or`ed with true is true:

```
a or true == true
```

- Both `and` and `or` distribute:

```
a or (b and c) == (a or b) and (a or c)
a and (b or c) == (a and b) or (a and c)
```

- Double negatives cancel out:

```
not(not a) == a
```

- DeMorgan’s laws:

```
not(a or b) == (not a) and (not b)
not(a and b) == (not a) or (not b)
```

Boolean Algebra

- We can use these rules to simplify our Boolean expressions.

-
  ```
  while not(scoreA == 15 or scoreB == 15):
      #continue playing
  ```
- This is saying something like “While it is not the case that player A has 15 or player B has 15, continue playing.”
- Applying DeMorgan’s law:
  ```
  while (not scoreA == 15) and (not scoreB == 15):
      #continue playing
  ```

Boolean Algebra

- This becomes:

```
while scoreA != 15 and scoreB != 15
    # continue playing
```

- Isn’t this easier to understand? “While player A has not reached 15 and player B has not reached 15, continue playing.”

Boolean Algebra

- Sometimes it's easier to figure out when a loop should stop, rather than when the loop should continue.
- In this case, write the loop termination condition and put a `not` in front of it. After a couple applications of DeMorgan's law you are ready to go with a simpler but equivalent expression.

Other Common Structures

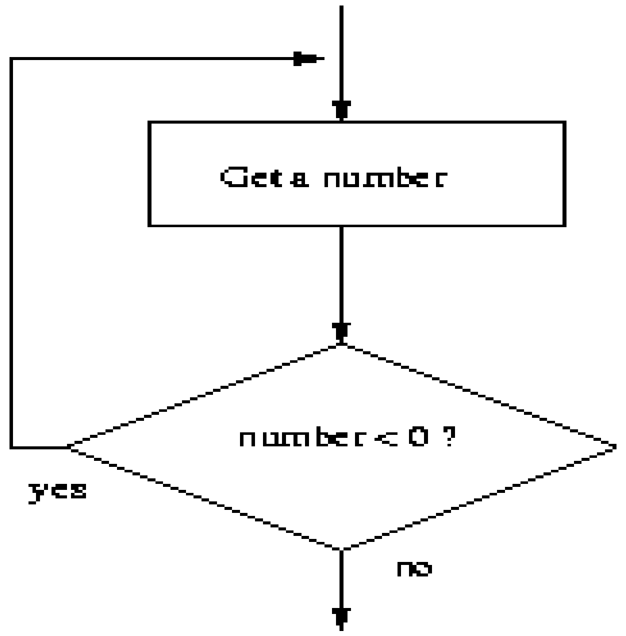
- The `if` and `while` can be used to express every conceivable algorithm.
- For certain problems, an alternative structure can be convenient.

Post-Test Loop

- Say we want to write a program that is supposed to get a nonnegative number from the user.
- If the user types an incorrect input, the program asks for another value.
- This process continues until a valid value has been entered.
- This process is *input validation*.

Post-Test Loop

-
  ```
  repeat
      get a number from the user
  until number is >= 0
  ```



Post-Test Loop

- When the condition test comes after the body of the loop it's called a *post-test loop*.
- A post-test loop always executes the body of the code at least once.
- Python doesn't have a built-in statement to do this, but we can do it with a slightly modified `while` loop.

Post-Test Loop

- We seed the loop condition so we’re guaranteed to execute the loop once.

-
  ```
  number = -1
  while number < 0:
      number = eval(input("Enter a positive number: "))
  ```
- By setting `number` to –1, we force the loop body to execute at least once.

Post-Test Loop

- Some programmers prefer to simulate a post-test loop by using the Python `break` statement.
- Executing `break` causes Python to immediately exit the enclosing loop.
- `break` is sometimes used to exit what looks like an infinite loop.

Post-Test Loop

- The same algorithm implemented with a `break`:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break # Exit loop if number is valid
```

- A while loop continues as long as the expression evaluates to true. Since `True` *always* evaluates to true, it looks like an infinite loop!

Post-Test Loop

- When the value of *x* is nonnegative, the `break` statement executes, which terminates the loop.
- If the body of an `if` is only one line long, you can place it right after the `:`!
- Wouldn’t it be nice if the program gave a warning when the input was invalid?

Post-Test Loop

- In the `while` loop version, this is awkward:

```
number = -1
while number < 0:
    number = eval(input("Enter a positive number: "))
    if number < 0:
        print("The number you entered was not positive")
```

- We're doing the validity check in two places!

Post-Test Loop

- Adding the warning to the `break` version only adds an `else` statement:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0:
        break # Exit loop if number is valid
    else:
        print("The number you entered was not positive.")
```

Loop and a Half

- Stylistically, some programmers prefer the following approach:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break # Loop exit
    print("The number you entered was not positive")
```

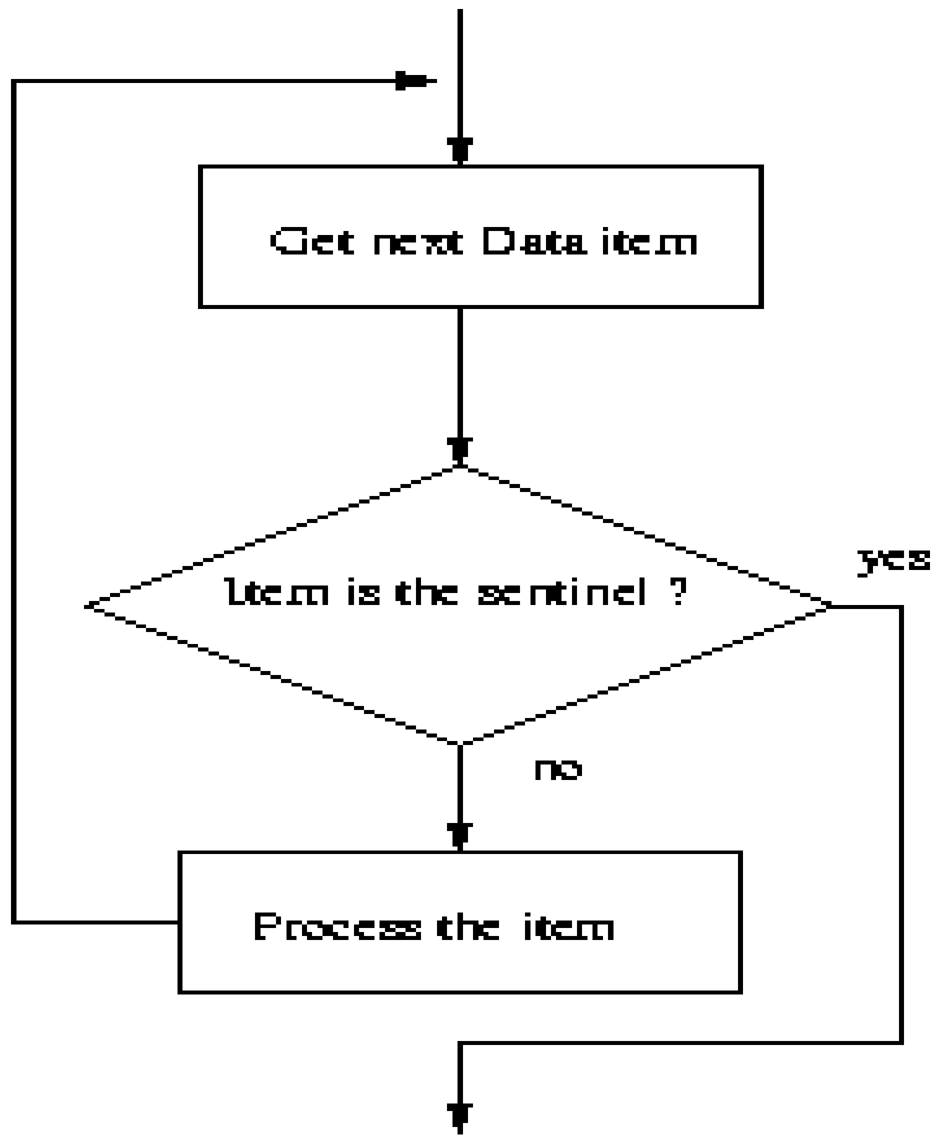
- Here the loop exit is in the middle of the loop body. This is what we mean by a *loop and a half*.

Loop and a Half

- The loop and a half is an elegant way to avoid the priming read in a sentinel loop.

-
  ```
  while True:
      get next data item
      if the item is the sentinel: break
      process the item
  ```
- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!

Loop and a Half



Loop and a Half

- To use or not use `break`. That is the question!
- The use of break is mostly a matter of style and taste.
- Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits.

Boolean Expressions as Decisions

- Boolean expressions can be used as control structures themselves.
- Suppose you’re writing a program that keeps going as long as the user enters a response that starts with ‘y’ (like our interactive loop).
- One way you could do it:

```
while response[0] == "y" or response[0] == "Y":
```

Boolean Expressions as Decisions

- Be careful! You can't take shortcuts:
  `while response[0] == "y" or "Y":`
- Why doesn't this work?
- Python has a `bool` type that internally uses 1 and 0 to represent `True` and `False`, respectively.
- The Python condition operators, like `==`, always evaluate to a value of type `bool`.

Boolean Expressions as Decisions

- However, Python will let you evaluate any built-in data type as a Boolean. For numbers (int, float, and long ints), zero is considered `False`, anything else is considered `True`.

Boolean Expressions as Decisions

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(32)
True
>>> bool("Hello")
True
>>> bool("")
False
>>> bool([1,2,3])
True
>>> bool([])
False
```

Boolean Expressions as Decisions

- An empty sequence is interpreted as `False` while any non-empty sequence is taken to mean `True`.
- The Boolean operators have operational definitions that make them useful for other purposes.

Boolean Expressions as Decisions

Operator	Operational definition
x `and` *y*	If *x* is false, return *x*. Otherwise, return *y*.
x `or` *y*	If *x* is true, return *x*. Otherwise, return *y*.
`not` *x*	If *x* is false, return `True`. Otherwise, return `False`.

Boolean Expressions as Decisions

- Consider *x* `and` *y*. In order for this to be true, both *x* and *y* must be true.
- As soon as one of them is found to be false, we know the expression as a whole is false and we don’t need to finish evaluating the expression.
- So, if *x* is false, Python should return a false result, namely *x*.

Boolean Expressions as Decisions

- If *x* is true, then whether the expression as a whole is true or false depends on *y*.
- By returning *y*, if *y* is true, then true is returned. If *y* is false, then false is returned.

Boolean Expressions as Decisions

- These definitions show that Python's Booleans are *short-circuit* operators, meaning that a true or false is returned as soon as the result is known.
- In an `and` where the first expression is false and in an `or`, where the first expression is true, Python will not evaluate the second expression.

Boolean Expressions as Decisions

- `response[0] == "y" or "Y“`
- The Boolean operator is combining two operations.
- Here’s an equivalent expression:
  `(response[0] == "y") or ("Y")`
- By the operational description of `or`, this expression returns either `True`, if response[0] equals “y”, or “Y”, both of which are interpreted by Python as true.

Boolean Expressions as Decisions

- Sometimes we write programs that prompt for information but offer a default value obtained by simply pressing `<Enter>`
- Since the string used by `ans` can be treated as a Boolean, the code can be further simplified.

Boolean Expressions as Decisions

-
  ```
  ans = input("What flavor fo you want [vanilla]: ")
  if ans:
      flavor = ans
  else:
      flavor = "vanilla"
  ```
- If the user just hits `<Enter>`, `ans` will be an empty string, which Python interprets as false.

Boolean Expressions as Decisions

- We can code this even more succinctly!

```
ans = input("What flavor fo you want [vanilla]: ")
flavor = ans or "vanilla‟
```

- Remember, any non-empty answer is interpreted as `True`.
- This exercise could be boiled down into one line!

```
flavor = input("What flavor do you want
            [vanilla]:” ) or "vanilla"
```

Boolean Expressions as Decisions

- Again, if you understand this method, feel free to utilize it. Just make sure that if your code is tricky, that it’s well documented!

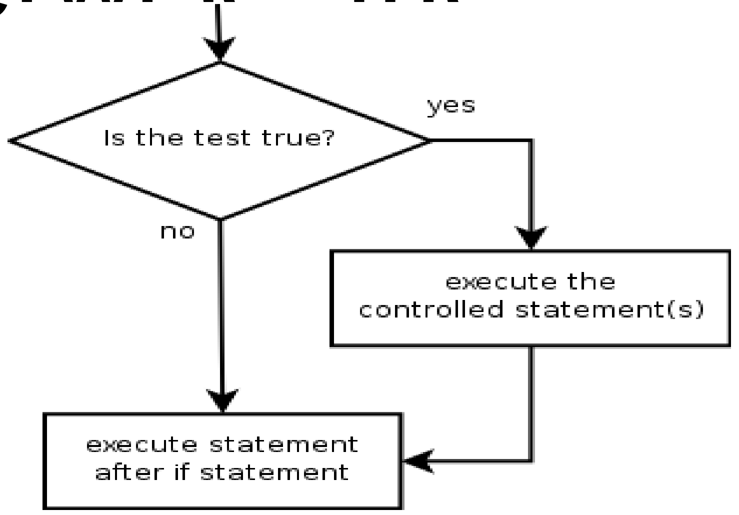
if

if

- `if` **statement**: Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.
  - Syntax:

    `if` ***condition*** `:`

    statements


Is the test true?
yes
no
execute the controlled statement(s)
execute statement after if statement

- Example:

```
gpa = 3.4
if gpa > 2.0:
    print "Your application is accepted."
```

if/else

- **`if/else` statement**: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.
  - Syntax:

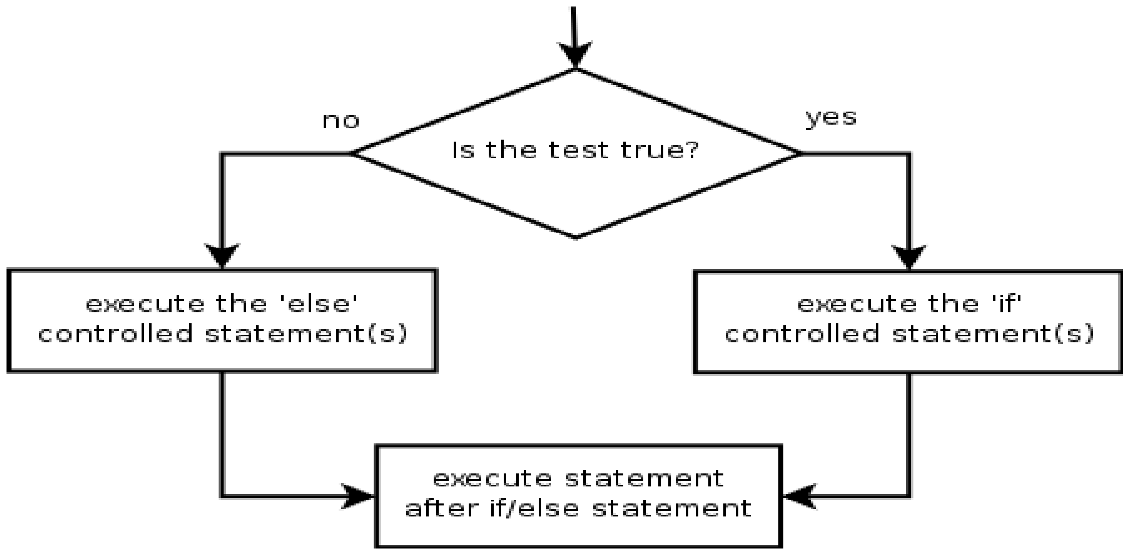
```
if condition:
    statements
else:
    statements
```

- Example:

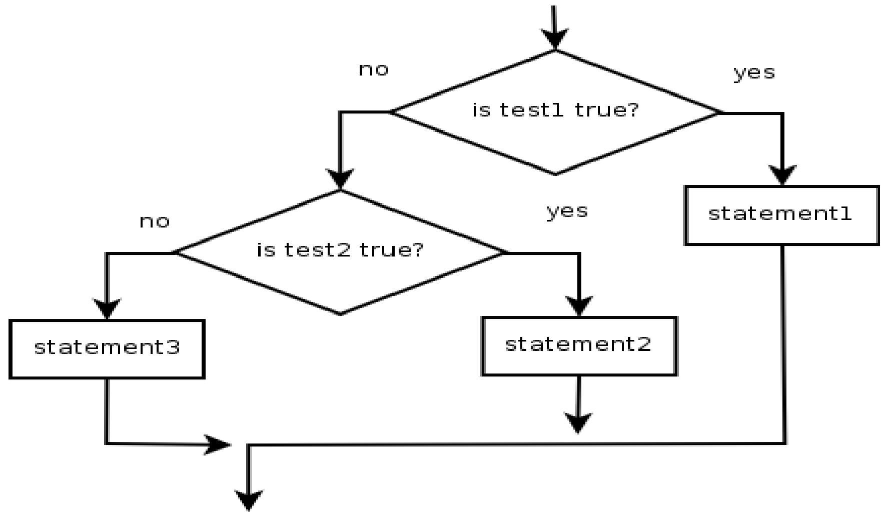
```
gpa = 1.4
if gpa > 2.0:
    print "Welcome to Mars University!"
else:
    print "Your application is denied."
```

- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```



Is the test true?
no
yes
execute the 'else' controlled statement(s)
execute the 'if' controlled statement(s)
execute statement after if/else statement



Testing conditions: if, elif, else

- Sometimes it is necessary to test a condition and to do different things, depending on the condition.
- Examples: avoiding division by zero, branching in a menu structure etc.
- The following program greets the user with "Hello Tom", if the name he inputs is Tom:

```
s = raw_input ("Input your name: ")
if s == "Tom":
print "HELLO ", s
```

- Note the indentation and the ":" behind the if statement!
- The above program can be extended to do something if the testing condition is not true:

```
s=input ("Input your name: ")
if s == "Tom":
   print( "Hello ", s)
else:
   print( "Hello unknown")
```

Testing conditions: if, elif, else

- It is possible to test more than one condition using the elif statement:

```
s = input ("Input your name: ")
if s == "Tom":
    print( "Hello ", s)
elif s == "Carmen":
    print("I'm so glad to see you ", s)
elif s == "Sonia":
   print("I didn't expect you",s)
else:
   print("Hello unknown")
```

Note the indentation and the ":" behind the if, elif and else statements!

Conditions

- can check for sequence membership with `is` and `is not`:

```
>>> if (4 in vec):
...   print '4 is'
```

- chained comparisons: a less than b AND b equals c:

```
a < b == c
```

- and and or are short-circuit operators:
  - evaluated from left to right
  - stop evaluation as soon as outcome clear
- Can assign comparison to variable:

```
>>> s1,s2,s3='', 'foo', 'bar'
>>> non_null = s1 or s2 or s3
>>> non_null
foo
```

- Unlike C, no assignment within expression

```
>>> puan=78
if (puan>=60 and puan<=100):   # Eğer puan 60 dan büyük ve 100 den küçükse 'Geçti' yaz.
   print("Gecti")

elif (puan>=0 and puan<60):   # Yok eğer puan 0 dan büyük ve 60 dan küçükse 'Kaldı' yaz.
   print("Kaldi")

else :   # Yukarıdaki iki şart da false ise (gerçekleşmese) 'Yanlış değer' yaz.
   print ("Yanlis değer")

Gecti :   # sonuç
>>>
```