Introduction to Python Language

REPETITIVE STRUCTURES (LOOPS)

OBJECTIVES

- To understand the concepts of definite and indefinite loops as they are realized in the Python for and while statements.
- To understand the programming patterns interactive loop and sentinel loop and their implementations using a Python while statement.
- To be able to design and implement solutions to problems involving loop patterns including nested loop structures.
- To understand the basic ideas of Boolean algebra and be able to analyze and write Boolean expressions involving Boolean operators.

For Loops

- For loop allows us to iterate through a sequence of values.

```
for <var> in <sequence>:
    <body>
```

The loop index variable `var` takes on each successive value in the `sequence`, and the statements in the `body` of the loop are executed once for each value.

- **Example:** Suppose we want to write a program that can compute the average of a series of numbers entered by the user.

We can generate a design for this problem:

```
input the count of the numbers, n
initialize total to 0
loop n times
    input a number, x
    add x to total
output average as total / n
```

o We can almost totaly implement this algorithm into Python.

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

```
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( ) function: returns a sequence of numbers, in a given range.

Syntax:

range(start, stop, step)

Example:

```
for i in range(5):
    print(i, end=" ")
print()
```

```
0 1 2 3 4
```

```
for i in range(5, 20):
    print(i, end=" ")
```

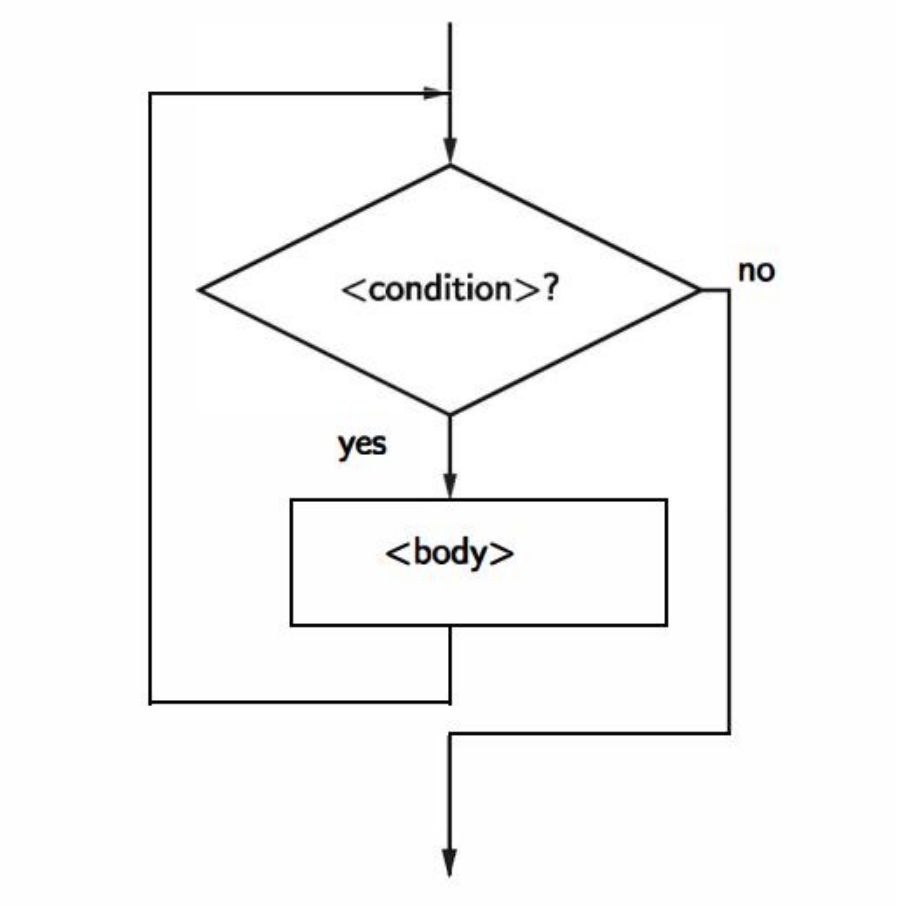
```
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Indefinite Loops (while loop)

- Our averaging program is certainly functional, but it doesn't have the best user interface.
- It starts by asking to user how many numbers there are. It is OK for handful of numbers, but what if I have a whole page of numbers to average? It might be a significant burden to go through and count them up.
- It would be much nicer if the computer could take care of counting the numbers for us.
- Unfortunately, as you no doubt recall, the for loop (in its usual form) is a definite loop, and that means the number of iterations is determined when the loop starts.
- An **indefinite loop** keeps iterating until certain conditions are met. There is no guarantee ahead of time regarding how many times the loop will go around.

o In Python, an indefinite loop is implemented using a while statement. Syntactically, the `while` is very simple:

```
while <condition>:
    <body>
```



Flowchart of a while loop

Here `condition` is a Boolean expression, just like in if statements. The `body` is, as usual, a sequence of one or more statements.

- The `body` of the loop executes repeatedly as long as the condition remains true.
- When the `condition` is false, the loop terminates.
- The condition is always tested at the top of the loop, before the loop body is executed. This kind of structure is called a **pre-test loop**.
- If the loop condition is initially false, the loop body will not execute at all.

o Here a simple example that prints the numbers from 0 to 10.

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

o This code will have the same output as if we had written a for loop like this:

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

Notice that the while version requires us to take care of initializing i before the loop and incrementing i at the bottom of the loop body. In the for loop, the loop variable is handled automatically.

- The simplicity of the while statement makes it both powerful and dangerous.
- Suppose we forget to increment `i` at the bottom of the loop body in the counting example:

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

- What will the output from this program be? When Python gets to the loop, i will be 0, which is less than 10, so the loop body executes, printing a 0. Now control returns to the condition; i is still 0, so the loop body executes again, printing a 0. Now control returns to the condition; i is still 0, so the loop body executes again, printing a 0 . . . .
- This is an example of an **infinite loop**.

- Usually, you can break out of a loop by pressing <Ctrl>-c (holding down the <Ctrl> key and pressing c) . If your loop is really tight, this might not work, and you'll have to resort to more drastic means (such as <Ctrl><Alt><Delete> on a PC) . If all else fails, there is always the trusty reset button on your computer.

Common Loop Patterns

Interactive Loops

- One good use of the indefinite loop is to write **interactive loops**.
- The idea behind an interactive loop is that it allows the user to repeat certain portions of a program on demand.

- Let's take a look at this loop pattern in the context of our number-averaging problem.
- We want to modify the program so that it keeps track of how many numbers there are. We can do this with another accumulator-call it `count`-that starts at 0 and increases by 1 each time through the loop.
- To allow the user to stop at any time, each iteration of the loop will ask whether there is more data to process. The general pattern for an interactive loop looks like this:

```
set moredata to "yes"
while moredata is "yes"
    get the next data item
    process the item
    ask user if there is moredata
```

- Combining the interactive loop pattern with accumulators for the `total` and `count` yields this algorithm for the averaging program:

```
initialize total to 0.0
initialize count to 0
set moredata to "yes"
while moredata is "yes"
    input a number, x
    add x to total
    add 1 to count
    ask user if there is moredata
output total / count
```

- Here is the corresponding Python program:

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

Notice this program uses string indexing (**moredata [0]**) to look just at the first letter of the user's input. This allows for varied responses such as "yes," "y," "yeah," etc. All that matters is that the first letter is a "y."

- Here is the output of our program:

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

The average of the numbers is 46.4
```

Sentinel Loops

- A better solution to the number-averaging problem is to employ a pattern commonly known as a **sentinel loop**.
- A sentinel loop continues to process data until reaching a special value that signals the end. The special value is called the **sentinel**.
- Any value may be chosen for the sentinel. The only restriction is that it be distinguishable from actual data values.
- The sentinel is not processed as part of the data.

- Here is a general pattern for designing sentinel loops:

```
get the first data item
while item is not the sentinel
    process the item
    get the next data item
```

-Notice how this pattern avoids processing the sentinel item.
-If the first item is the sentinel, the loop immediately terminates and no data is processed.

- We can apply the sentinel pattern to our number-averaging problem. The first step is to pick a sentinel. We can safely assume that no score will be below 0. The user can enter a negative number to signal the end of the data.

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

- Now we have a useful form of the program:

```
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
```

This sentinel loop solution is quite good, but there is still a limitation. **The program can't be used to average a set of numbers containing negative as well as positive values**.

In order to have a truly unique sentinel, we need to broaden the possible inputs. Suppose that we get the input from the user as a string.

- One simple solution is to have the sentinel value be an **empty string**. If the user types a blank line in response to an input Gust hits <Enter>), Python returns an empty string. We can use this as a simple way to terminate input. The design looks like this:

```
initialize total 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 total
    add 1 to count
    input next data item as a string, xStr
output total / count
```

- Translating it into Python yields this program:

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

- Here is an example run, showing that it is now possible to average arbitrary sets of numbers:

```
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
```

Computing with Booleans

- In the next sections, we will develop a simulation for the game of racquetball. Part of the simulation will need to determine when a game has ended.
- Suppose that scoreA and scoreB represent the scores of two racquetball players. The game is over as soon as either of the players has reached 15 points.
- Here is a Boolean expression that is true when the game is over:

```
scoreA == 15 or scoreB == 15
```

- When either score reaches 15, one of the two simple conditions becomes true, and, by definition of `or`, the entire Boolean expression is true. As long as both conditions remain false (neither player has reached 15) the entire expression is false.
- Our simulation will need a loop that continues as long as the game is not over.

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

- We can also construct more complex Boolean expressions that reflect different possible stopping conditions.
- A game also ends when one of the players reaches 7 and the other has not yet scored a point.
- For brevity, we'll use **`a`** for **scoreA** and **`b`** for **scoreB**. Here is an expression for game-over when shutouts are included:

```
a == 15 or b == 15 or (a == 7 and b == 0) or (b == 7 and a == 0)
```

Boolean Algebra

- Boolean expressions obey certain algebraic laws similar to those that apply to numeric operations. These laws are called **Boolean logic** or **Boolean algebra**.
- The following table shows some rules of algebra with their correlates in 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**

- From these examples, you can see that and has similarities to multiplication, or has similarities to addition, and 0 and 1 correspond to false and true.

- Here are some other interesting properties of Boolean operations. Anything **`or`** 'ed with true is just true.

```
( a or True ) == True
```

- Both **`and`** and **`or`** distribute over each other.

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

- A double negative cancels out.

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

- The next two identities are known as DeMorgan's laws.

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

- The following table demonstrates DeMorgan's first law:

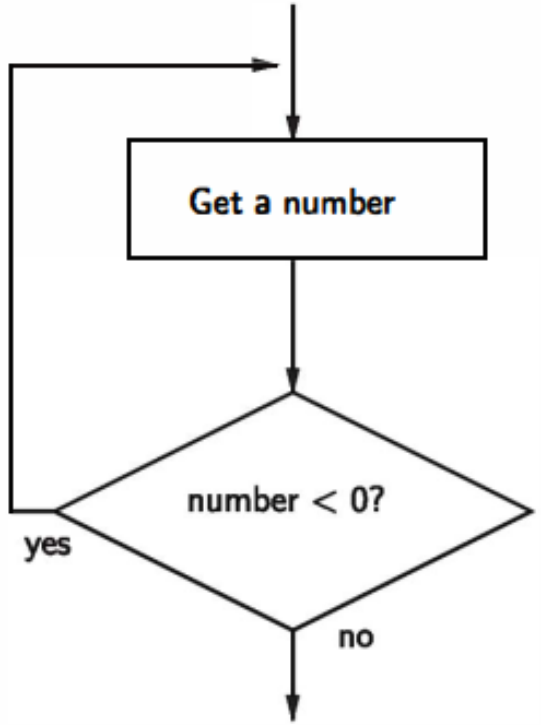
a	b	a or b	not (a or b)	not a	not b	(not a) and (not b)
T	T	T	**F**	F	F	**F**
T	F	T	**F**	F	T	**F**
F	T	T	**F**	T	F	**F**
F	F	F	**T**	T	T	**T**

Other Common Structures: Post-test Loop

- Suppose you are writing an input algorithm that is supposed to get a nonnegative number from the user. If the user types an incorrect input, the program asks for another value. It continues to re-prompt until the user enters a valid value.
- This process is called **input validation**. Well-engineered programs validate inputs whenever possible. Here is a simple algorithm:

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

The idea here is that the loop keeps getting inputs until the value is acceptable.



Flowchart of post-test loop

- Unlike some other languages, Python does not have a statement that directly implements a post-test loop.
- However, this algorithm can be implemented with a while by "seeding" the loop condition for the first iteration:

```
number = -1  # Start with an illegal value to get into the loop.
while number < 0:
    number = float(input("Enter a positive number: "))
```

This forces the loop body to execute at least once and is equivalent to the post test algorithm.

- Some programmers prefer to simulate a post-test loop more directly by using a Python **`break`** statement.
- Executing `break` causes Python to immediately exit the enclosing loop.
- Often a `break` statement is used to leave what looks syntactically like an infinite loop.
- Here is the same algorithm implemented with a break:

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

- It would be nice if the program issued a warning explaining why the input was invalid. Adding a warning to the version using break only requires adding an else to the existing if.

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

Other Common Structures: Loop and a Half

- Some programmers would solve the warning problem from the previous section using a slightly different style:

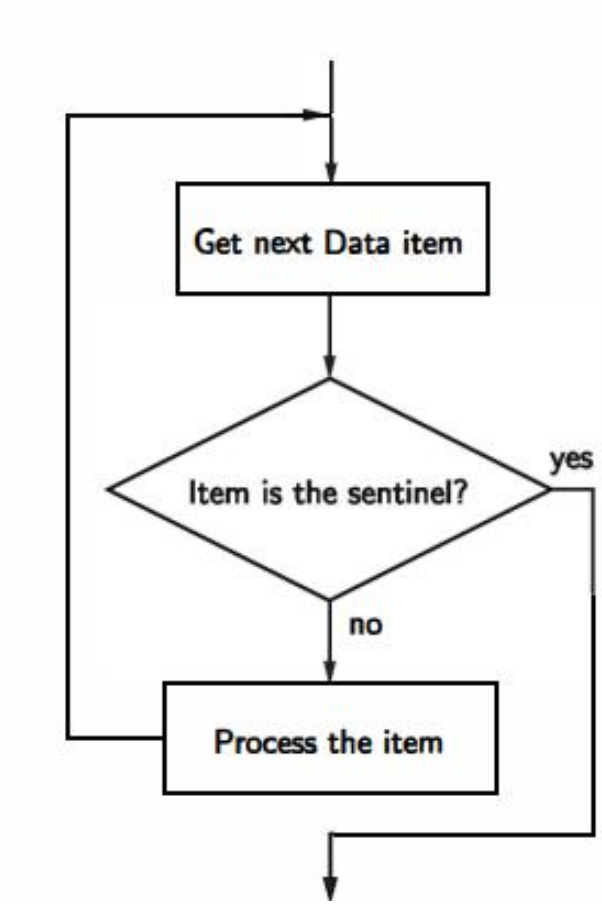
```
while True:
    number = float(input("Enter a positive number: "))
if number >= 0: break   # Loop exit
print("The number you entered was not positive")
```

- Here the loop exit is actually in the middle of the loop body.
- This is called *a loop and a half*.

- Here is the general pattern of a sentinel loop implemented as a loop and a half:

```
while True:
    get next data item
    if the item is the sentinel: break
    process the item
```

```
while True:
    get next data item
    if the item is the sentinel: break
    process the item
```



Loop-and-a-half implementation of sentinel loop pattern

Bool ean Expressions as Decisions

- Sometimes Boolean expressions themselves can act as control structures. Consider writing an interactive loop that keeps going as long as the user response starts with a "y." To allow the user to type either an upper- or lowercase response, you could use a loop like this:

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

- You must be careful not to abbreviate this condition as you might think of it in English: "While the first letter is Y or 'Y' ". The following form does not work:

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

- Frequently, programs prompt users for information but offer a default value for the response. The default value, sometimes listed in square brackets, is used if the user simply hits the <Enter> key. Here is an example code fragment:

```
ans = input("What flavor do you want [vanilla]: ")
if ans != "":
    flavor = ans
else:
    flavor = "vanilla"
```

- Exploiting the fact that the string in `ans` can be treated as a Boolean, the condition in this code can be simplified as follows:

```
ans = input("What flavor do you want [vanilla]: ")
if ans:
    flavor = ans
else:
    flavor = "vanilla"
```

- In fact, this task can easily be accomplished in a single line of code:

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

Next Lecture

Defining Functions