THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 12
Loops

Objectives

- To understand the concepts of definite (for) and indefinite (while) loops.
- To understand interactive loop and sentinel loop and their implementations using a while statement.
- To be able to design and implement solutions to problems involving loop patterns including nested loop structures.

for Loop: Revision

```
for i in range(10):
   # do something
#-------------------------------------------------
myList = [2,3,4,9,10]
for x in myList:
   # iterates through the list elements
   # do something that involves the list elements
#-------------------------------------------------
myString = “hello there, hello world!”
for ch in myString:
   # iterates through the string characters
#-------------------------------------------------
infile = open(someFile, "r")
for line in infile:
   # iterate through the lines of the file
infile.close()
```

Indefinite Loops

- Definite loops can be used only if we know the number of iterations ahead of time, i.e. before the loop starts.
- Sometimes, we don't know how many iterations we need until all the data has been entered.
- The indefinite or conditional loop keeps iterating until certain conditions are met.

Indefinite Loops

- while <condition>:
  <body>

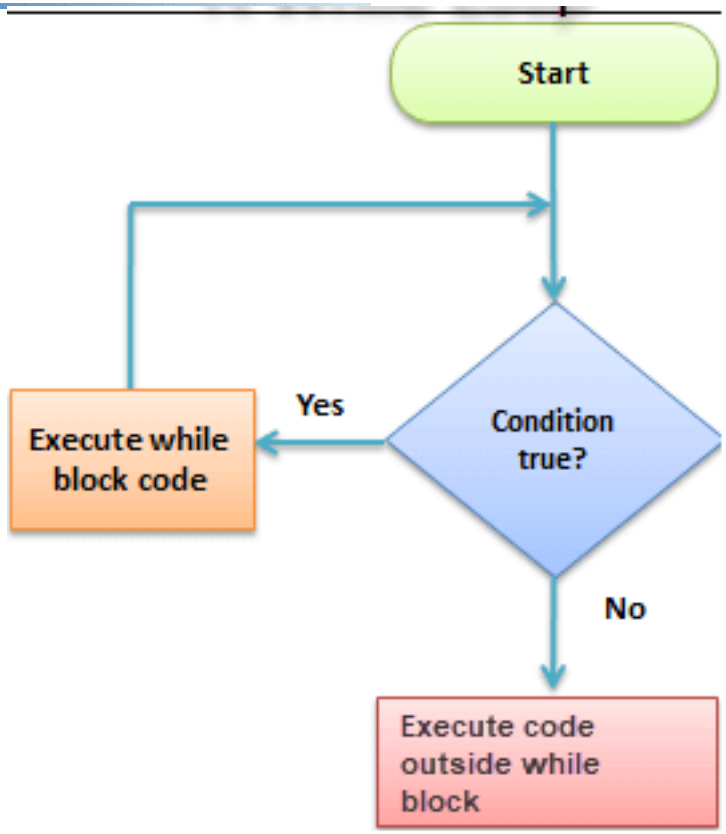
- <condition> is a Boolean expression, just like in `if` statements. <body> is a sequence of one or more statements.

- 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 Loops

- Example of a `while` loop that counts from 0 to 9:

```
i = 0
while i < 10: # valid but poor use of while
    print(i)
    i += 1
```

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

```
for i in range(10) :# this is the right way
    print(i)
```

- 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 Loops

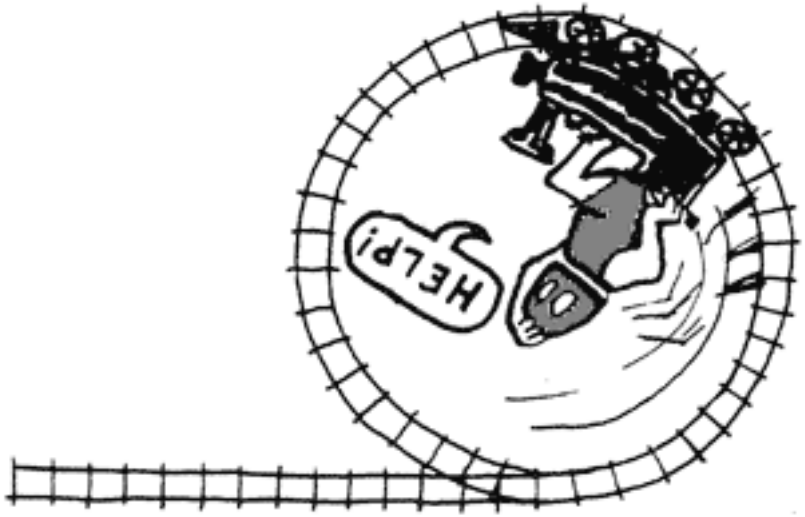
- The `while` statement is simple, but yet powerful and dangerous – they are a common source of program errors.

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

- What happens with this code?
- The value of `i` never changes inside the loop body.
- This is an example of an infinite *loop*.

Getting out of an Infinite Loop

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



www.forth.com

Interactive Loops

- A good use of the indefinite loop is to write interactive loops that allow a user to repeat certain portions of a program on demand.
- Remember that we need to keep track of how many numbers had been entered? Let's use another accumulator, called `count`.
- 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.

Interactive Loops

```
#    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].lower() == 'y':
        x = float(input("Enter a number >> "))
        sum += x
        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`", "`Y`" 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.
- 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

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

def main():
    sum = 0.0
    count = 0
    x = float(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum += x
        count += 1
        x = float(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

- Now we can use of the interactive loop without the hassle of typing `'y'` all the time.
- BUT we can't average a set of positive **and negative** numbers.
- If we do this, our sentinel can no longer be a number.
- 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

```
#  A program to average a set of numbers
#  Using empty string as loop sentinel

def main():
  sum = 0.0
  count = 0
  xStr = input("Enter a number (<Enter> to quit) >> ")
  while xStr != "":
    sum += float(xStr)
    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
```

Nested Loops

- In the same way that you have an if statement within an if statement, you can have loops within loops
- For example, rather than having 1 number per input line, have multiple, comma–separated numbers per line

Nested Loops

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

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in infile:
        # update sum and count for values in line
        for xStr in line.split(","):
            sum += float(xStr)
            count += 1
    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.*

Other Loop Structures – Post-Test Loop

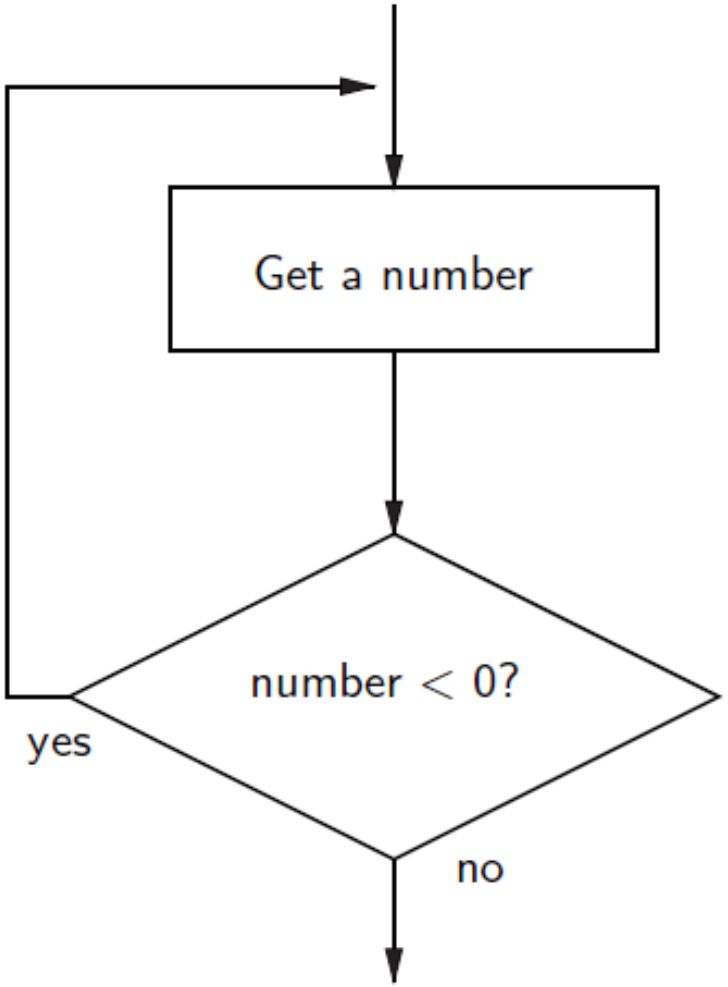
- Say we want to write a program that is supposed to get a nonnegative number from the user.
- If the user types an incorrect input, the program asks for another value.
- This process continues until a valid value has been entered.
- This process is *input validation*.

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

```
# A program to average a set of numbers
# Using Post-Test loop which will be execute at least once

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

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:
    xStr = input("Enter a number (<Enter> to quit) >> ")
    if xStr == "":
        break # Exit loop
```

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

Loop and a Half

- Stylistically, some programmers prefer the following approach:

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

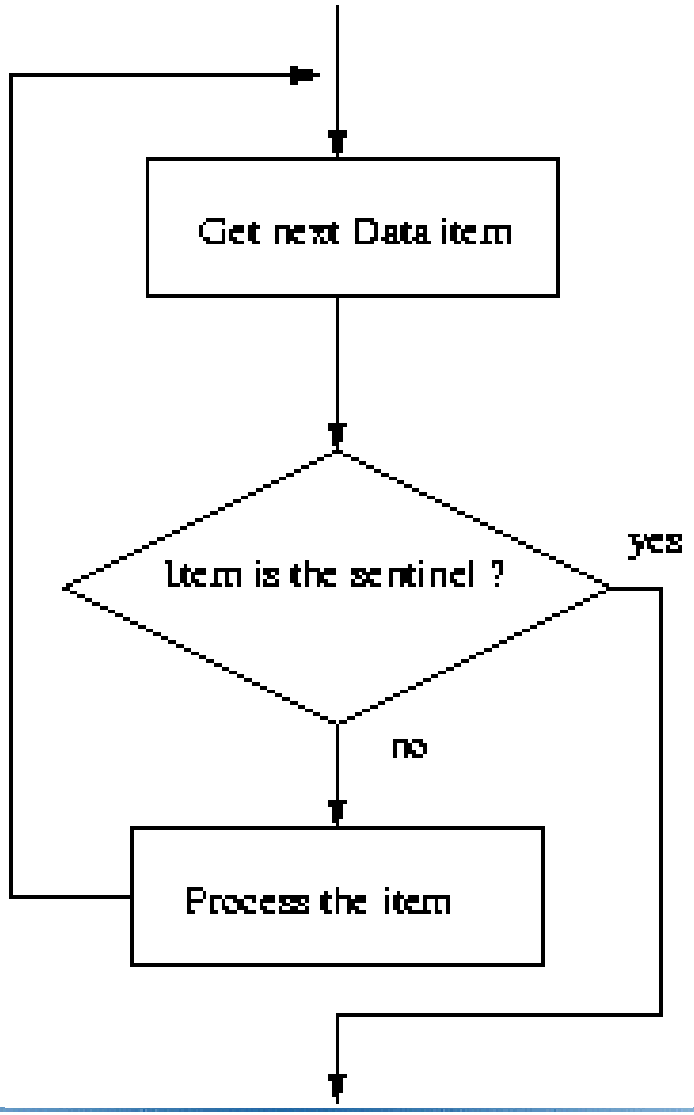
Loop and a Half

- 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.

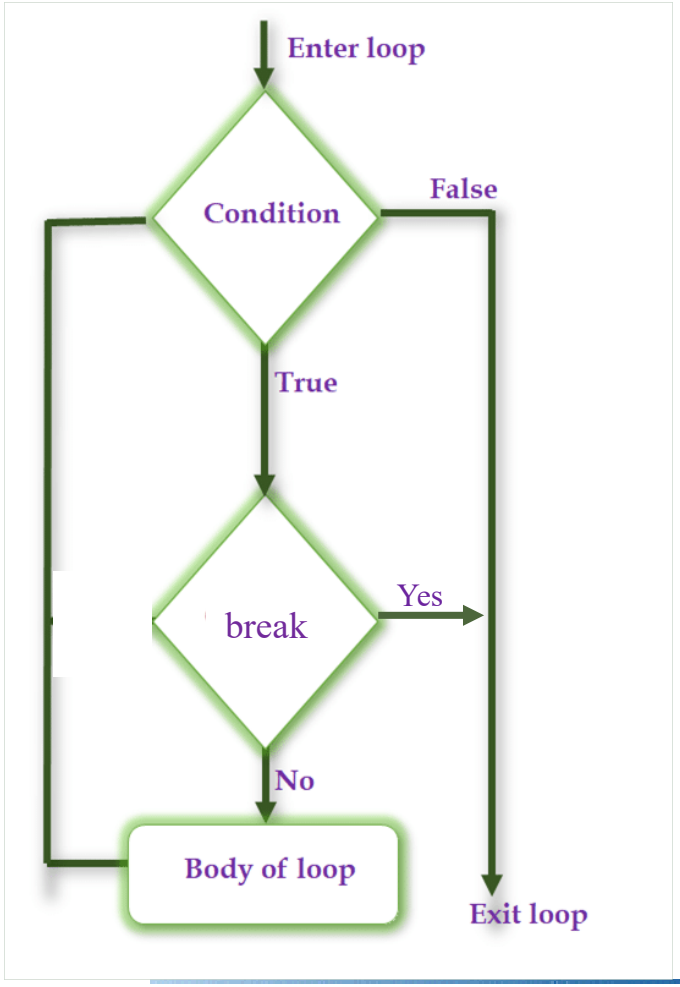
continue statement

- Continue statement returns the control to the beginning of the loop

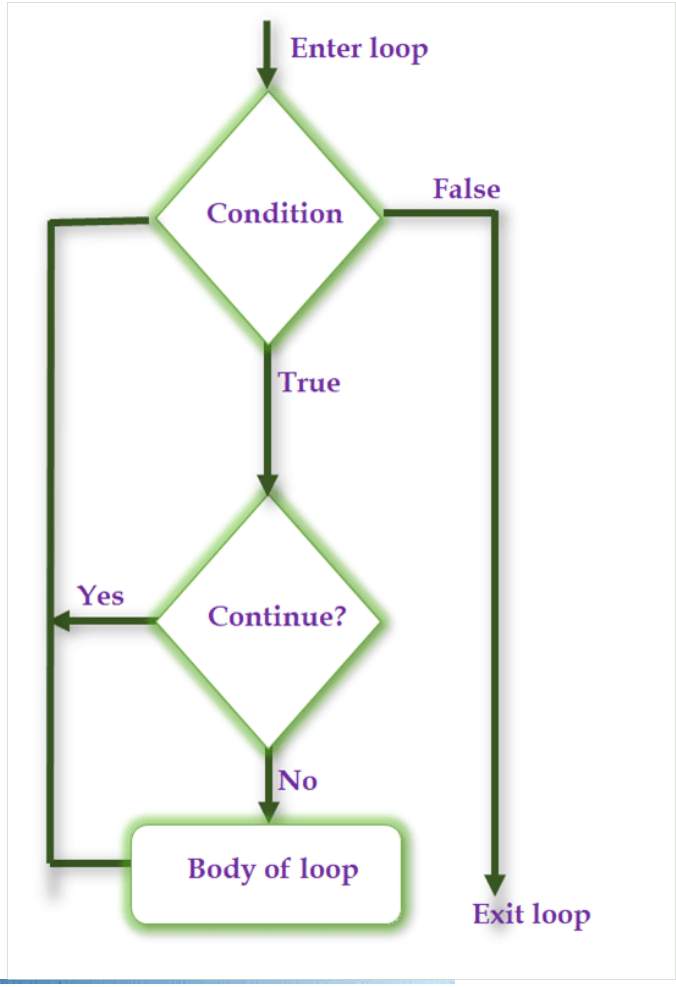
```
# print only even numbers up to 10
for i in range(11):
    if i % 2 == 1:  # % is "modulus" operator
        continue
    print(i)
```

break and continue comparison

break statement



continue statement



Summary

- Indefinite loops
- Interactive loops
- Sentinel loops
- Post-Test loops
- Break statement
- Loop and a Half
- Continue statement