IST 1025

Introduction to Programming
The `while` Loop and Indefinite Loops

Another Example Problem

$$y = \sqrt{x}$$

Sir Isaac Newton developed a method to approximate the value of the square root of a given number. Suppose **x** is the number and `guess` is an approximation of the square root of **x**. Then a closer approximation to the actual square root of **x** is equal to

`(guess + x / guess) / 2`

The initial value of `guess` is `x / 2`

Try Some Guesses in the Shell

```
>>> from math import sqrt
>>> x = 2
>>> sqrt(x)
1.4142135623730951
>>> guess = x / 2
>>> guess
1.0
>>> guess = (guess + x / guess) / 2
>>> guess
1.5
>>> guess = (guess + x / guess) / 2
>>> guess
1.4166666666666665
>>>
```

Use a Loop to Repeatedly Guess

The approximation algorithm repeatedly replaces a guess with a better approximation. Write a program that allows the user to specify the value of **x** and the number of iterations used in this approximation. The program computes and displays the resulting estimate of the square root of **x**, as well as Python's own estimate using Python's `math.sqrt` function.

Use a **for** Loop

```python
import math

x = float(input("Enter a number: "))
count = int(input("Enter the number of iterations: "))

guess = x / 2
for i in range(count):
    guess = (guess + x / guess) / 2
print("Guess:  %0.15f" % guess)
print("Actual: %0.15f" % math.sqrt(x))
```

We perform a definite number of approximations

Or Use a `while` Loop
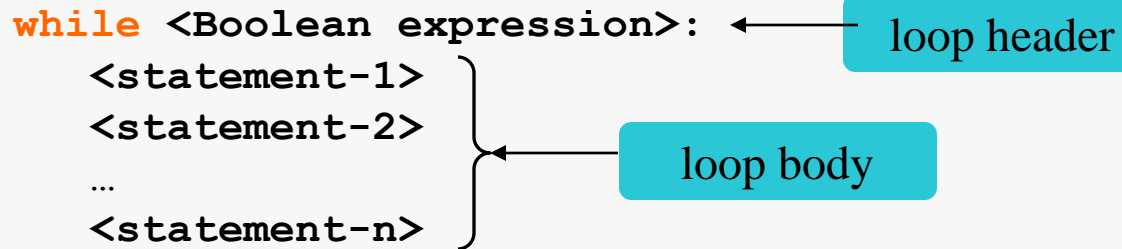
```python
import math

x = float(input("Enter a number: "))
count = int(input("Enter the number of iterations: "))

guess = x / 2
i = 0
while i < count:
    guess = (guess + x / guess) / 2
    i += 1
print("Guess:  %0.15f" % guess)
print("Actual: %0.15f" % math.sqrt(x))
```

This loop is also count-controlled, and is equivalent in meaning to the `for` loop

Syntax of the `while` Loop

```
while <Boolean expression>:          ← loop header
    <statement-1>
    <statement-2>                    ← loop body
    …
    <statement-n>
```
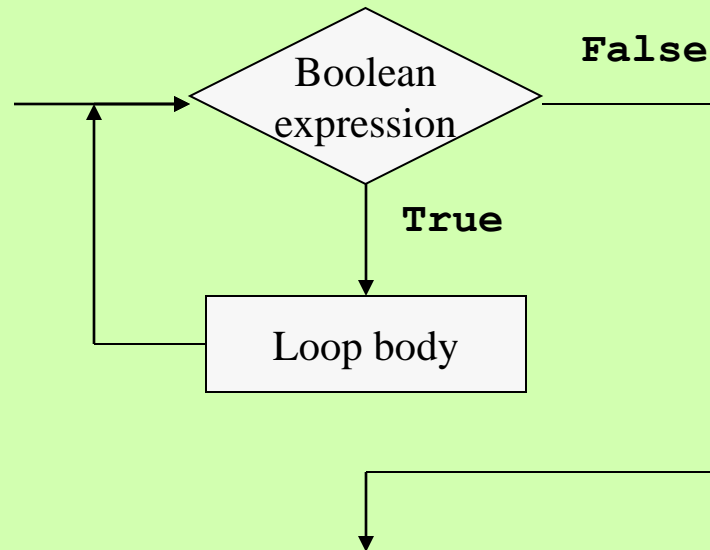
A *Boolean expression* can be, among other things, a comparison of two numbers or strings, using ==, !=, <, >, <=, >=

These operations return the Boolean values `True` or `False`

Note that == means *equal to*, whereas = means *assignment*

Behavior of the `while` Loop


```
while <Boolean expression>:          # The condition
    <sequence of statements>         # The loop body
```
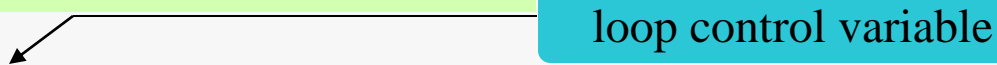
Logic of the **`while`** Loop

```
i = 0
while i < count:
    guess = (guess + x / guess) / 2
    i += 1
```

Before the loop header, the *loop control variable* must be set to the appropriate initial value

Somewhere in the loop body, a statement must reset the loop control variable so that the Boolean expression eventually becomes **False**, to terminate the loop

These steps are *automatic* in the **for** loop

Which Loop Should I Use?

- A **`for`** loop works well when the number of iterations is predictable (usually count-controlled)

- A **`while`** loop works well when the number of iterations is not predictable

- How could the # of iterations be unpredictable?

Another Way to Look at Newton's Method

We repeatedly improve the **guess** until the difference between the square of the **guess** and **x** reaches a given tolerance value.

```
tolerance = 0.00001
while |guess² - x| > tolerance
    guess = (guess + x / guess) / 2
```

We cannot predict how many times this process will be repeated

This is called an *indefinite loop* or a *conditional iteration*

The `while` Loop

```python
import math

x = float(input("Enter a number: "))

tolerance = 0.00001
guess = x / 2
while abs(guess ** 2 - x) > tolerance:
    guess = (guess + x / guess) / 2
print("Guess:  %0.15f" % guess)
print("Actual: %0.15f" % math.sqrt(x))
```

Add a tolerance value and quit the loop when the condition becomes false

Sentinel-Controlled Loops

- An indefinite loop can test for the presence of a special value called a *sentinel*

- When the sentinel is reached, the loop should stop

An Example Problem

Input test scores from the user and print the average score.

The user signals the end of input by simply pressing the enter or return key.

This value shows up as an empty string, which is the sentinel value.

Sentinel-Controlled Loops

```python
total = 0
count = 0
data = input("Enter a score or just return to quit: ")

while data != "":
    total += int(data)
    count += 1
    data = input("Enter a score or just return to quit: ")

if count == 0:
    print("No scores were entered.")
else:
    print("The average is", total / count)
```

The input statement must be written twice

Simplify with a **break** Statement

```python
total = 0
count = 0


while True:
    data = input("Enter a score or just return to quit: ")
    if data == "":
        break
    total += int(data)
    count += 1

if count == 0:
    print("No scores were entered.")
else:
    print("The average is", total / count)
```

break causes an immediate exit from the enclosing loop

When Should I Use a **break**?

- When the loop body executes at least once but the number of iterations is unpredictable

- The condition of the enclosing **while** loop is just **True**

- The condition for loop exit is placed in an enclosing **if** statement

- A **break** may also be helpful within a **for** loop, but more on that at a later date . . .

Input Checking *redux*

```python
rate = int(input('Enter the interest rate[0-100]: '))
if rate < 0 or rate > 100:
    print('ERROR: Rate must be between 0 and 100!')
else:
    interest = principal * rate / 100
    print('Your interest is', interest)
```

We need to continue this process until the user enters a legitimate number

Input Checking *redux*

```python
while True:
    rate = int(input('Enter the interest rate[0-100]: '))
    if rate < 0 or rate > 100:
        print('ERROR: Rate must be between 0 and 100!')
    else:
        break
interest = principal * rate / 100
print('Your interest is', interest)
```

The loop controls the input process

The computation process occurs only after the loop has finished