Rossitza S. Marinova

Programming and Algorithms I

Repetitions

Objectives

Learning about the

- Concept of a loop
- Event-controlled and counter-controlled loops
- Loops in **`Python`**
  - The `for` loops
  - The `while` loops
- Looping applications

What is the Purpose of a Loop?

- Control flow statements such as `if`, `for`, and `while` are used to change the sequential order of statements execution.
- The `if` statement is used to check a condition and *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.
- A **loop** is a statement that is used to: execute one or more statements repeatedly until a goal is reached. Sometimes these one-or-more statements will not be executed at all—if that's the way to reach the goal.

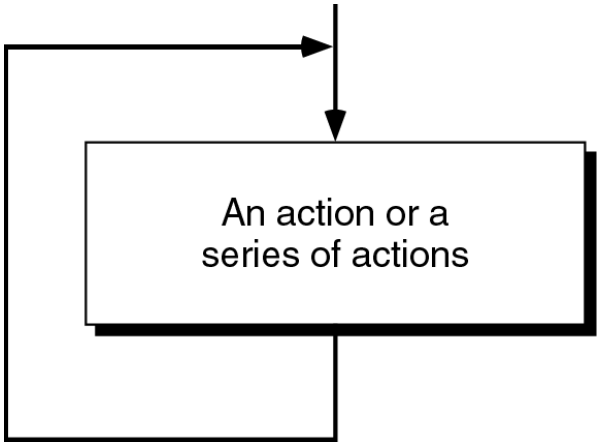
Concept of a Loop

- **Loop** is a group of instructions computer executes repeatedly while some condition remains true
- To make sure that a loop ends, we must have a **condition** that controls it.

  In other words, we must design the loop so that before or after each **iteration**, it checks to see if it is done.

  - If it is not done, it repeats one more time;
  - If it is done, it exits the loop.

  This test is known as a **loop control expression**.



Event-Controlled and Counter Controlled Loops

All the possible expressions that can be used in a loop limit test can be summarized into two general categories: event-controlled loops and counter-controlled loops.

- In an **event-controlled loop**, an event changes the loop control expression from true to false.
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data"
  - Explicit (controlled by the loop) or implicit (controlled by some external condition) updating process
- We use a **counter-controlled loop** when we know the number of times an action is to be repeated.
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions

Loops in Python

- Python has two loop statements: the `while` and the `for`.
- Both of them can be used for event-controlled and counter-controlled loops.
- The `while` are most commonly used for event-controlled loops.
- The `for` is usually used for counter-controlled loop.
- The loop constructs continue when the limit control test is true and terminate when it is false. This consistency of design makes it easy to write the limit test.
- The `while` and the `for` statements have an optional `else` clause.

Using a Loop to Solve the Investment Problem

The algorithm for an investment problem:

1. Start with a year value of 0 and a balance of $10,000.
2. **Repeat** the following steps
while the balance is less than $20,000:
   - Add 1 to the year value.
   - Multiply the balance value by 1.05 (a 5 percent increase).
3. Report the final year value as the answer.

"Repeat .. while" in the problem indicates a loop is needed.

To reach the goal of being able to report the final year value, adding and multiplying must be repeated some unknown number of times.

Using a Loop to Solve the Investment Problem

The statements to be controlled are:

- Incrementing the **`year`** variable
- Updating the **`balance`** variable using a variable for the **`rate`**

```
year = year + 1
balance = balance * (1 + rate / 100)
```

Using a Loop to Solve the Investment Problem

The condition, which indicates when to ***stop*** executing the statements, is this test:

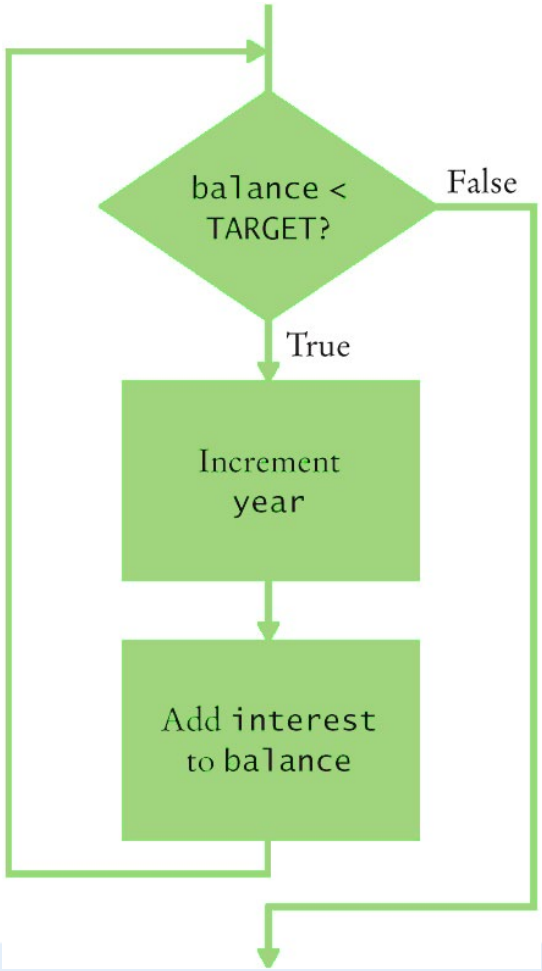
```
balance < target
```

Using a Loop to Solve the Investment Problem

Here is the complete `while` statement:

```
while balance < target:
    year = year + 1
    balance = balance * (1 + rate / 100)
```

Flowchart of the Investment Calculation's `while` Loop



The Complete Investment Script

```
rate = 5
initialBalance = 10000
target = 2 * initialBalance
year = 0
balance = initialBalance
while balance < target:
    year = year + 1
    balance = balance * (1 + rate / 100)
print('The investment doubled after', year, 'years.')
```

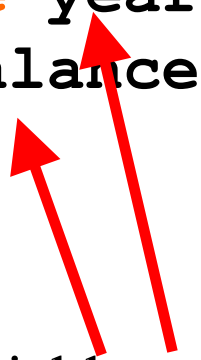
This script will output:

```
The investment doubled after 15 years.
```

Common Error – Infinite Loops

- Forgetting to update the variable used in the condition is common.
- In the investment program, it might look like this.

```
year = 1
while year <= 20:
    balance = balance * (1 + rate / 100)
```



- The variable **year** is not updated in the body

A Not Really Infinite Loop

- Due to what is called “wrap around”, the previous loop ***will*** end.
- At some point the value stored in the **`int`** variable gets to the largest representable positive integer. When it is incremented, the value stored “wraps around” to be a negative number.

That definitely stops the loop!

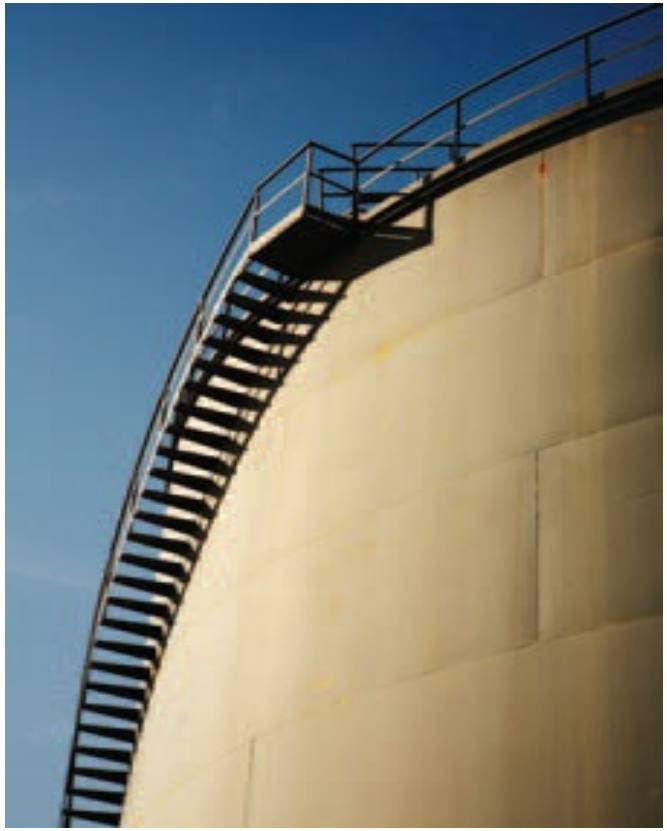
Common Error with `while` Loops

- The programmer wants to count down from 10. What is wrong and how can it be fixed?

```
i = 10
while i == 0:
    print (i)
    i -= 1
```

Note that `i -= 1` is equivalent to `i = i - 1`.

The `for` Loop



To execute statements a certain number of times

You ***"simply"*** take 4,522 steps!!!

The `for` Loop

```
for counter in [v1,v2,…,vn]
    statements
else:
    statements
```

It is usually used to cause the *statements* to happen a certain number of times.

Example:

```
for i in range(1, 10, 2):
    print(i)
else:
    print('The for loop is over')
```

The `range()` function in `for` loops

If you need to iterate over a sequence of numbers, the built-in function `range()` comes in handy:

Syntax: `range([start],stop,[step])`

It generates lists containing arithmetic progressions, for example:

`range(1,10,2)` generates `[1, 3, 5, 7, 9]`

Try using `for` statement with:

```
range(3,8)
range(1,11,3)
range(-4,-10,-2)
```

Solving a Problem with a `for` Statement

- Earlier we determined the number of years it would take to (at least) double our balance.
- Now let's see the interest in action:
  - We want to print the balance of our savings account over a five-year period.
  - The "…over a five-year period" indicates that a `for` loop should be used. Because we know how many times the statements must be executed we choose a `for` loop.

Solving a Problem with a `for` Statement

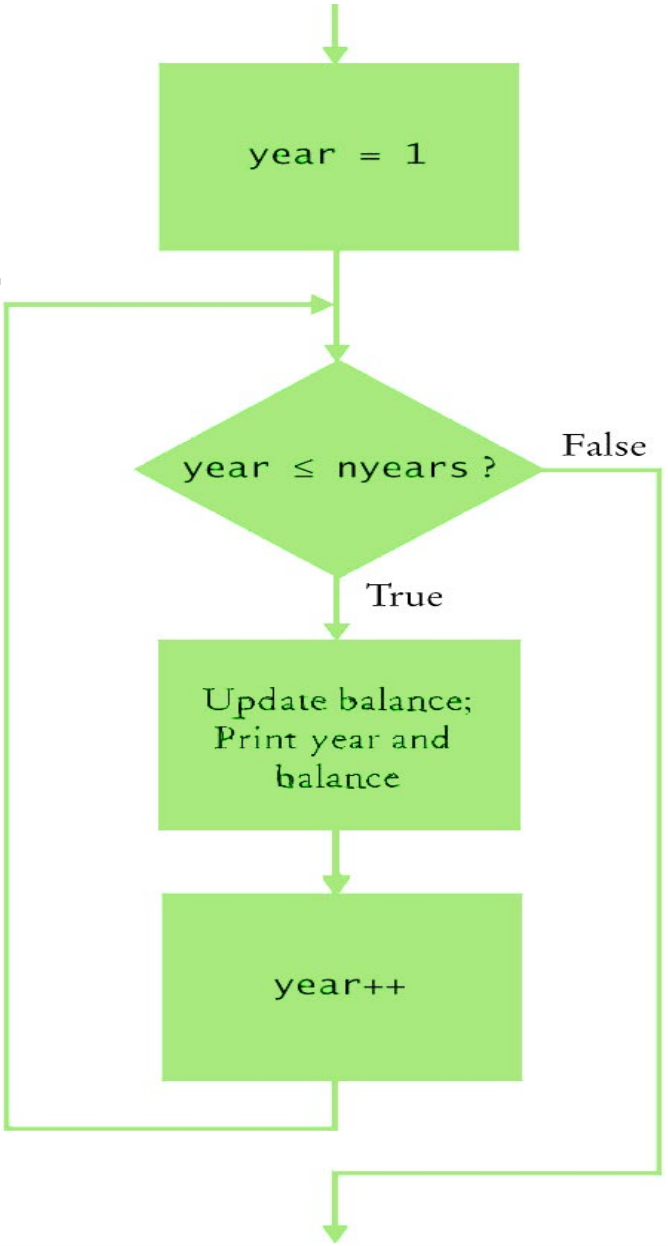
The output should look something like this:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The `for` Loop

Flowchart of
the investment calculation's
`while` loop

Easily written using a `for` loop



Solving a Problem with a `for` Statement

Two statements should happen five times.

So use a `for` statement.

They are:

update balance

print year and balance

```
for year in range(1, nyears)
    # update balance
    # print year and balance
```

The Modified Investment Program Using a `for` Loop

```
rate = 5
initialBalance = 10000
balance = initialBalance
nyears = int(input('Enter number of years: '))
for year in range(1, nyears):
    balance = balance * (1 + rate / 100)
    print('Year: ', year, '\t Balance: ', '%.2f' % balance)
```

The Modified Investment Program Using a `for` Loop

A run of the program:

```
Enter number of years: 10
Year:   1       Balance:   10500.00
Year:   2       Balance:   11025.00
Year:   3       Balance:   11576.25
Year:   4       Balance:   12155.06
Year:   5       Balance:   12762.82
Year:   6       Balance:   13400.96
Year:   7       Balance:   14071.00
Year:   8       Balance:   14774.55
Year:   9       Balance:   15513.28
```

Nested Loops

For each hour, 60 minutes are processed – a nested loop.

Nested Loops

- Nested loops are used mostly for data in tables as rows and columns.
- The processing across the columns is a loop, as you have seen before, “nested” inside a loop for going down the rows.
- Each row is processed similarly so design begins at that level. After writing a loop to process a generalized row, that loop, called the “inner loop,” is placed inside an “outer loop.”

Nested Loops

Write a program that uses nested for loops to print a multiplication table.

The output should be something like this:

```
      Multiplication Table
--------------------------------
  |   1  2  3  4  5  6  7  8  9
--------------------------------
1 |   1  2  3  4  5  6  7  8  9
2 |   2  4  6  8 10 12 14 16 18
3 |   3  6  9 12 15 18 21 24 27
4 |   4  8 12 16 20 24 28 32 36
5 |   5 10 15 20 25 30 35 40 45
6 |   6 12 18 24 30 36 42 48 54
7 |   7 14 21 28 35 42 49 56 63
8 |   8 16 24 32 40 48 56 64 72
9 |   9 18 27 36 45 54 63 72 81
```

Nested Loops

- The first step is to solve the "nested" loop.
- There are ten columns and in each column we display the product. Using **i** to be the number of the row we are processing, we have:

```
for i in range(1,10):
    print(i,'|',end='')
    for j in range(1,10):
        print('%3d' % (i*j), end='')
    print('\n',end='')
```

- You would test that this works in your code before continuing. If you can't correctly print one row, why try printing lots of them?

Nested Loops

```
print('     Multiplication Table')
print('-----------------------------')
print('   ', end='')
for j in range(1,10):
    print('%3d' % j, end='')
print('\n-----------------------------')
```

```
for i in range(1,10):
    print(i,'|',end='')
    for j in range(1,10):
        print('%3d' % (i*j), end='')
    print('\n',end='')
```

all rows

prints row **i**

(don't forget to indent, nestedly)

Other Statements Related to Looping

- Three other statements are related to loops:
  - **`break`**, **`continue`** and **`pass`**.
- The **`break`** statement is used to **`break`** out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has been completely iterated over.
- An important note is that if you **`break`** out of a for or while loop, any corresponding loop else block is not executed.
- The **`continue`** statement is used to tell Python to skip the rest of the statements in the current loop block and to **`continue`** to the next iteration of the loop.
- The **`pass`** statement is used in Python to indicate an empty block of statements.

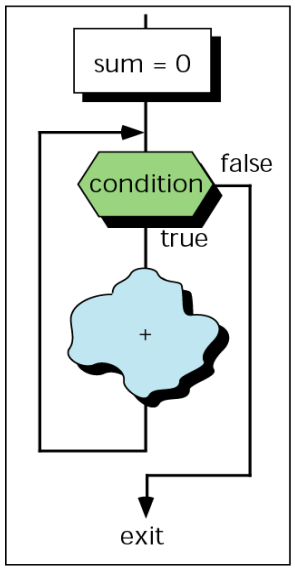
Looping Applications

We examine four common applications for loops: summation, product, smallest or largest, and inquires. With few exceptions, each loop contains initialization code, looping code, and disposition code. Disposition code handles the result of the loop, often by printing it, other times by simply returning it to the calling function.

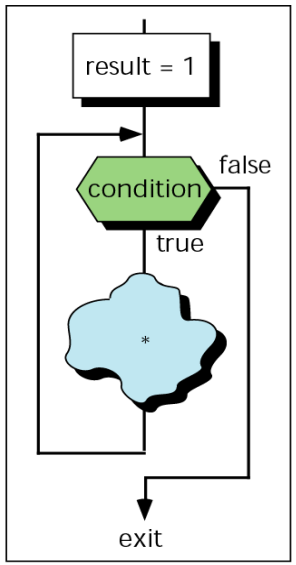
- Summation and product

  Three logical parts:

  - *Initialization* of any necessary working variables, such as the sum accumulator
  - The *loop*, which include the summation code and any data validation code
  - The *disposition code* to print or return the result



Summation

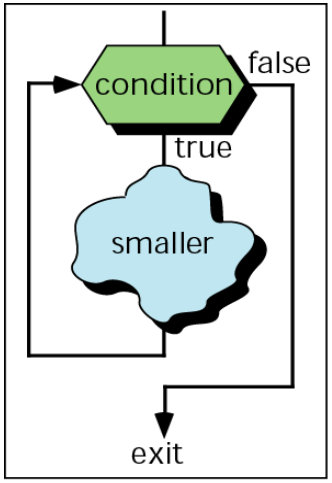


Product

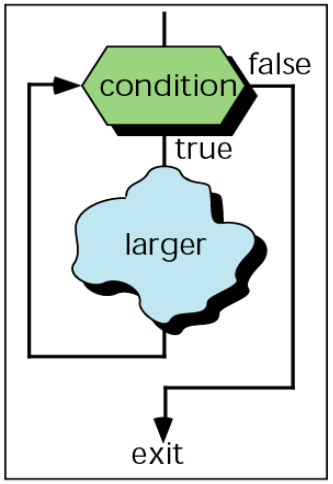
Looping Applications

- Smallest and largest

  You will often encounter situations in which you must determine the smallest or largest among a series of data. This is also a natural looping structure.



smallest



largest

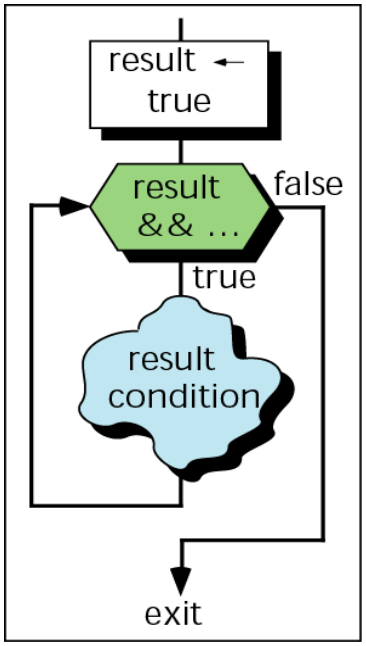
Looping Applications

- Inquires

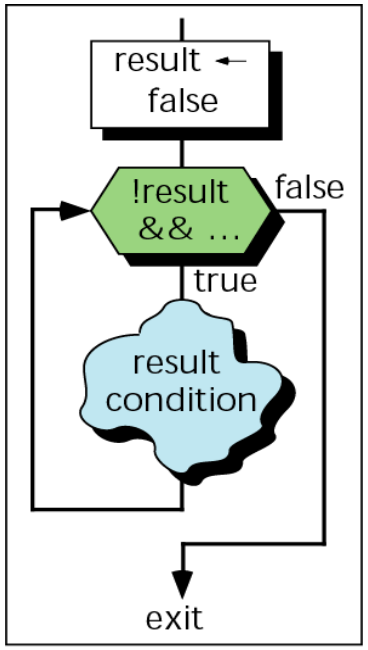
An inquiry is simply a question, asked of the computer program.

In programming we often encounter one or two basic inquiry types:

- **any** – if one or more data meet the criteria
- **all** – when we have a list of data and we want to make sure that all of them meet some specified criteria.



all



any

Questions

1. There are three things wrong with this program. Find and correct them.

```
print("This program takes 3 numbers and returns the sum.")
total = 0
for i in range (3) :
    x = input(" Enter a number : ")
    total = total + i
print(" The total is:",x)
```

2. Write a Python program that asks the user for seven numbers. Then print the total, the number of positive entries, the number entries equal to zero, and the number of negative entries.