UCL

INST0004 Programming 2

Lecture 03: Control Flow Structures

Module Leader:
Dr Daniel Onah

2023-24


UCL

Copyright Note

Important information to adhere to

Recap of Previous Lecture 2

Recap of previous lecture

Re: Cap☺

Let's remind ourselves of last week's lecture.

- We looked at variables and declaration
- We discussed constant variables
- We discussed arithmetic operations and expressions
- We looked at function call and arguments
- We discussed Python libraries and modules
- We looked at strings in Python

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce the concept of control structure, decisions and functions in Python. At the end of the lecture, you should be able to:

- To implement decisions using if statements
- To compare integers, floating-point numbers, and strings
- To implement while loops, sentinel control loop
- To write a program using for loop
- To write statements using Boolean expressions
- To develop strategies for testing your programs
- To validate user input

Introduction to Control Structures

What are control functions

Normally, statements in a program are executed in the order in which they are written. This is called **sequential execution**. Various Python statements enable the programmer to specify the next statement to be executed and this might be other than the **next one in the sequence**. This is called **transfer of control**. Transfer of control is achieved with **Python control structures**. This section discusses the background of control structure development and the specific tools Python uses to transfer control in a program.

Introduction to Control Structures

What are control functions

Before writing a program to solve a particular problem, *it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem*. When writing a program, it is equally essential to understand the **types of building blocks** that are *available and to use within a proven program-construction principles*.

One of the most **essential features** of computer programs is **their ability to make decisions**. Similar to a train that changes its tracks. This is largely depending on how the switches within the train are set. A computer program can take ***different actions*** *depending on inputs and the constructs created within the program*.

Program Flow Structures

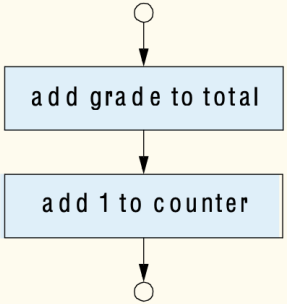
What are program structure

In Python programming, all programs could be written in terms of three **control structures** namely, ***the sequence structure***, ***the selection structure*** and ***the repetition structure***. The sequence structure is built into Python. Unless directed otherwise, the computer executes Python statements sequentially.

Flowchart Sequence

Importance of flowchart structure

A flowchart segment is a typical sequence structure in which program flow structure could be illustrated graphically. We could also perform sequence of two or more calculations sequentially. A flowchart is a tool that provides graphical representation of an algorithm or to depict portion of an algorithm.




```
total = total + grade
counter = counter + 1
```

Decision: If Statement

How to construct decision statement?

The **if statement** is used to implement and construct a program decision logic. When the condition within the *if statement* **block** is fulfilled, a set of decision statement is then executed within the body of the conditional statement. Otherwise, **another set of statements would be executed**.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for creating an if statement
"""
if condition :
    statements
```

Decision: If Statement

How to construct decision statement?

Let's look at an example on how we could create **if else** conditional statements.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for creating an if else statement
"""
if condition :
    block statements_1 # execute statement 1
else :
    block statements_2 # otherwise, execute statement 2
```

Decision: If Statement

Decision making procedure

An if statement condition that is **true** or **false**. Often uses relational operators: ==, !=, <, ≤, >, ≥. The **colon** in the if statement indicates a compound block statement. In creating the if conditional statement, the **if** and **else** clauses must be aligned.

- *If the condition is **true**, the statement(s) in branch of the **if statement** are executed in sequence; however, if the condition is **false**, they are skipped.*
- *If the condition is **false**, the statement(s) in the branch of the **if statement** are executed in sequence; however, if the condition is **true**, they are skipped.*

Decision: If Statement

Decision making problem

Suppose we want to write a program that would request user input about a floor number. In the building, floor number **13** is not available on the lift button. The program should be able to determine the actual floor number based on the specific set condition. If the floor input entered by the user is greater than 13, the program should subtract **1** from the input to get to the actual floor. If the user type number **14** this should take the user to the **13** floor and so on.

"It is a challenge to allow them to survive on their own, for this would make them to evolve"
- **Star Trek: Yoyager**

Decision: If Statement

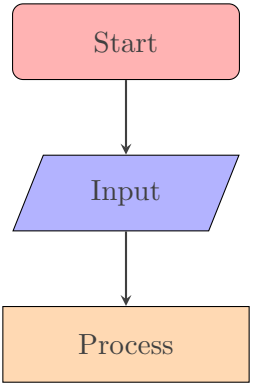
Decision making solution

Let's look at a simple example.

```
#!usr/bin/env python
"""
A program to determine the floor number of a building
"""
floor = int(input("Enter floor number: "))
buildingFloor = 0
if floor > 13 :
    buildingFloor = floor - 1
else :
    buildingFloor = floor
print(buildingFloor)
```

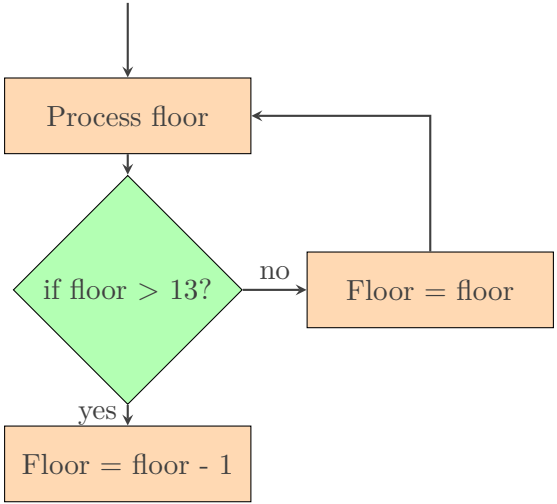
Decision: If Statement

How to construct decision statement using flowchart



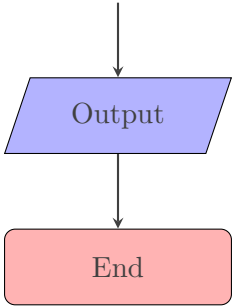
Decision: If Statement

How to construct decision statement using flowchart



Decision: If Statement

How to construct decision statement using flowchart



Decision: If Statement

Declaring If constructs

Some of the If constructs in Python are represented as a **compound statement**. This statement span across multiple lines which consist of a ***header*** and a **statement block** for processing the result from the condition. In Python, the if statement is a good example of a compound statement. A compound statement requires a colon **(:)** at the end of the header. The statement blocks in Python are made up of a group of one or more statements, all of which are **indented** within the **if constructs** all to the **same indentation level**. Ideally, a **statement block** *begins after the line of the header and ends at the first indented statement.* While, Python enforces proper indentation, you can still use any number of spaces to indent statements within a block. However, all statements within an **if-statement construct** should have the same indentation level throughout.Remember that **comments** are not statements and thus can be indented to any level and this would be ignored in **Python**.

Decision: If Statement

Declaring If constructs

Let's look at an example.

```
#!/usr/bin/env python
"""
A program to demonstrate the total sale
"""
# request for total sale
totalSales = input("Enter total sale ($): ")
totalSales = int(totalSales)
# conditional statement
if totalSales > 1000.00: # The header of if statement block ends with a colon
    discount = totalSales * 0.05 # Lines in the block are indented to the same level
    totalSales = totalSales - discount
    print("Discount of ${} applied".format(discount))
```

Nested Statement

Nested If else constructs

In Python programming, if & else and other statement blocks can be **nested** inside another statement blocks. This indicates that the statements are part of a given **compound statement**. As you know, in the case of the if construct, the statement blocks specifies the instruction(s) that would be executed if and only if the condition is TRUE or otherwise, ignored (or skipped) if the condition is FALSE.

Nested Statement

Nested If else constructs

```
#!/usr/bin/env python
"""
A program to demonstrate nested if else statement
"""
totalSales = float(input("Enter total sale: "))
if totalSales > 1000.0:
    discount = totalSales * 0.05
    print("You've got a discount of   %.2f"%discount)
else:
    difference = 1000 - totalSales
    if difference < 10.0:
        print("To receive our 5% discount, make a purchase of above
     1000 ")
    else:
        print("You would need to spend   %.2f more to receive a discount
     of 5 percent."%difference)
```

Python Syntax

Syntax in Python

Python requires block structured code as part of its syntax. The **alignment of statements** within a Python program specifies which statements are part of a **given statement block**.

- How do you move the cursor from the leftmost column to the appropriate indentation level?.

A perfectly reasonable strategy is to hit or tap the space bar a sufficient number of times. With **most editors**, you can use the Tab key instead. A **tab** moves the cursor to the next indentation level. Some editors even have an option to fill in the **tabs automatically**.

Program duplicate

What to avoid here.

```
#!/usr/bin/env python
"""
A program to avoid duplication
"""
Floor = int(input("Enter floor number: "))
if Floor > 13:
    buildingFloor = Floor - 1
    print("The building floor is: ", buildingFloor)# repetition/
    duplication to avoid
else:
    buildingFloor = Floor
    print("The building floor is: ",buildingFloor) # avoid repetition/
    duplication
```

Better program

Improved from previous example

```
#!/usr/bin/env python
"""
A program to avoid duplication better program
"""
Floor = int(input("Enter floor number: "))
if Floor > 13:
    buildingFloor = Floor - 1
else:
    buildingFloor = Floor
print("The building floor is: ",buildingFloor) # One print statement
```

Removing duplication from our program code is particularly important when we try to maintain our programs for a long period of time. When we have two or more sets of statements with the same outcome effect, it can easily happen that we may modifier one and forget to update the others.

Conditional Expressions

What is conditional expression?

In Python, a conditional expression is *a single statement written in a single line or continued to the next line without a break* in most cases. The conditional expression is **not a compound statement** therefore it does not require a **colon(:)** to begin a statement block. you can definitely use a conditional expression wherever you expect a value to be. The value of the expression is either ***value_1* if the condition is TRUE** or ***value_2* if it is FALSE**.

Syntax

```
#!/usr/bin/env python
"""
A conditional expression syntax
"""
value_1 if condition else value_2
```

Conditional Expressions

What is conditional expression?

Conditional expression is a powerful construct that you would see in some Python programming tasks. For example, we can compute a conditional expression as follows:

```
#!/usr/bin/env python
"""
A program to demonstrate conditional expression
"""
Floor = int(input("Enter floor number: "))
# conditional expression
buildingFloor = Floor - 1 if Floor > 13 else Floor
print(buildingFloor)
```

Conditional Expressions

What is conditional expression?

The previous conditional expression is equivalent to the **if else** construct below.

```
#!/usr/bin/env python
"""
A program to demonstrate conditional expression equivalent
"""
# conditioan expression equivalent to:
Floor = int(input("Enter floor number: "))
if Floor > 13:
    buildingFloor = Floor - 1
else:
    buildingFloor = Floor
print(buildingFloor)
```

Exercises

Try them yourself!

1. In some Asian countries, the number **14** is considered unlucky. Some building owners play it safe and skip both the **thirteenth** and the **fourteenth** floor. How would you modify the sample program in the lecture to handle such a building?
2. Consider the following if statement to compute a discounted price:

```
if actualPrice > 300:
    discountedPrice = actualPrice - 25
else:
    discountedPrice = actualPrice - 15
```

What is the discounted price if the actual price is 115? 500? 605?

Exercises

Try them yourself!

3. Compare this **if statement** with the one in exercise 2:

```
if actualPrice < 300:
    discountedPrice = actualPrice - 15
else:
    discountedPrice = actualPrice - 25
```

   Do the two statements always compute the same value? If not, when do the values differ?

4. The variables **fuelAmount** and **fuelCapacity** hold the actual amount of fuel in the vehicle and the size of the fuel tank of a vehicle. If less than 5% is remaining in the fuel tank, a status light should show a **red** color; otherwise it shows a **green** color. Simulate this process by printing out either **"red"** or **"green"**.

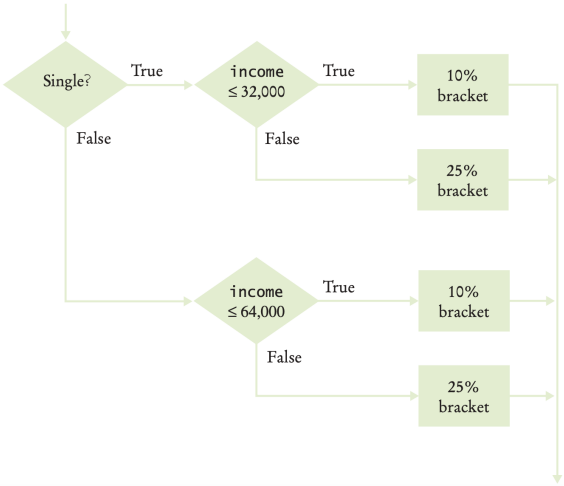
Nested Branches

Nested branch construct

Nested branch is a construct that includes a **conditional statement** within another **if statement**. The process of including an if statement in another **if statement** is known as nested set of statements.

Nested Branches

Nested branch construct



Nested Branches

Example

Let's look at an example.

```
#!/usr/bin/rnv python
"""
A program to compute income taxes, using a simplified tax schedule.
"""
# initialize constant variables for the tax rates and rate limits
RATE_1 = 0.10
RATE_2 = 0.25
RATE_1_SINGLE_LIMIT = 32000.0
RATE_2_MARRIED_LIMIT = 64000.0

# request income and marital status
income = float(input("Please enter your income:"))
maritalStatus = input("Please enter (s) for single or (m) for married:")
```

Nested Branches

Example

```
14 # Compute taxes
15 tax_1 = 0.0
16 tax_2 = 0.0
17 # nested branchesconditional statement
18 if maritalStatus == "s":
19     if income <= RATE_1_SINGLE_LIMIT:
20         tax_1 = RATE_1 * income
21     else:
22         tax_1 = RATE_1 * RATE_1_SINGLE_LIMIT
23         tax_2 = RATE_1 * (income * RATE_1_SINGLE_LIMIT)
```

Nested Branches

Example

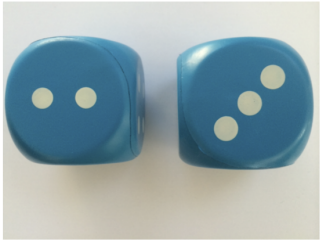
Because marital Status is not "**s**", we move to the else branch of the ***outer if statement*** in (line 24). Because income is not <= 64000, we move to the else branch of the ***inner if statement*** in (line 28). The values of **tax1** and **tax2** are then updated in (line 29 & 30).

```
24 else:
25     if income <= RATE_2_MARRIED_LIMIT:
26
27         tax_1 = RATE_1 * income
28     else:
29         tax_1 = RATE_1 * RATE_2_MARRIED_LIMIT
30         tax_2 = RATE_2 * (income - RATE_2_MARRIED_LIMIT)
31 totalTax = tax_1 + tax_2
32 # print the outcome
33 print("The tax is %0.2f"% totalTax)
```

Multiple If Else If Statement

Multiple if else if statements in Python

In the previous section we looked at a two-way branch with an **if statement**. In many situations, we can have more than two cases in our Python program. In this section, you will see how to implement a decision with multiple if else_if statements. Multiple if statements can be combined to evaluate complex problem decisions.



Multiple If Else If Statements

Example

```
#!/usr/bin/env python
"""
A program to demonstrate multiple if else statements
"""
grade = int(input("Enter your grade: "))
if grade >= 40 and grade <= 49:
    print("You have passed with D grade!")
else :
    if grade >= 50 and grade <=59:
        print("You have passed with C grade!")
    else :
        if grade >= 60 and grade <= 69 :
            print("You have passed with B grade!")
```

Multiple If Else If Statements

Example

```
14          else :
15              if grade >= 70 and grade <=100:
16                  print("You have passed with A grade!")
17              else :
18                  if grade > 100:
19                      print("Grade out of 100 percent mark range!")
20                  else:
21                      print("You have failed!")
```

Multiple Elif Statement - Simple

Multiple if else if statements in Python

But this becomes difficult to read and, as the number of branches increases, the code begins to shift further and further to the right due to the required Python indentation. Python provides other special construct known as *elif* for creating **if statements containing multiple branches** as illustrated in the previous section. Using this ***elif*** statement construct, we can rewrite the code segment in a more simpler and concise way with less code constructs. As soon as one of the five conditional tests succeeds, the **corresponding code block construct is display**, and no further tests will be ***required or attempted***. Otherwise, if none of the five cases applies, the final **else clause will be triggered**, and the *default message in the block will be displayed.* 135

Multiple Elif Statement - Simple

Multiple if else if statements in Python

```
#!/usr/bin/env python
"""
A program to demonstrate multiple elif statement
"""
grade = int(input("Please enter your grade: "))
if grade >= 40 and grade <= 49:
    print("You have passed with D grade!")
elif grade >= 50 and grade <= 59:
    print("You have passed with C grade!")
elif grade >= 60 and grade <= 69:
    print("You have passed with B grade!")
elif grade >= 70 and grade <= 100:
    print("You have passed with A grade!")
elif grade > 100:
    print("Grade out of 100 percent mark range!")
else:
    print("You have failed!")
```

Loops structure

Process of loops in Python

When we write a loop we tend to repeat specific part of our program operation within the loop construct several times as long as the condition is TRUE. This means some part of our program is repeated over and over, until our target goal is achieved. In Python loop constructs are very significant for computations that required repeated processing steps involving large data.

Loops structure

Process of loops in Python

In this section, you would learn about *loop statement constructs* in details, as well as the techniques for writing programs that process input and simulate real-world activities. *We will explore loop statements that repeatedly address a given task until the goal of the task is reached*.

While Loop structure

Process of a while loops in Python

In **Python programming**, the *while* statement is used to implement program repetition.

Syntax:

```
while condition:
    statements
```

As long as the condition remains *TRUE*, the statements inside the while loop block will continue its execution. This statement **block** is called the **body** of the **while** statement.

While Loop

Process of a while loops in Python

Let's look at the bank account problem from the lecture 2 tutorial. You have an account balance of £10,000 that earns a yearly interest of 5%. How many years does it take for the account balance to double the original balance. Using the while loop construct, we will want to repeat the following steps in our program.

Pseudocode:

- Repeat the following steps while the balance is less than £20,000
  - *increment the year by **1***
  - *compute the interest as balance X **0.05** (i.e., 5% interest)*
  - *add the interest to the balance*
- display the final year value

While Loop

Process of a while loops in Python

To solve this bank account double task, we will want to remember to ***increment the year counter and add the interest rate to the balance only while the balance is less than the target balance of £20,000***. Lets see an example while construct:

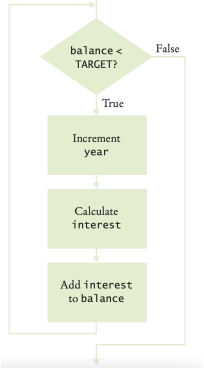
```
1 while balance < TARGET:
2     year = year + 1
3     interest = balance * RATE/100
4     balance = balance + interest
```

Note that because the interest earned also earns interest, therefor, a bank account balance would also grow exponentially.

While Loop

Process of a while loops in Python

The while statement is a very popular loop construct. *The flow of execution loops from one point within the program to another where the condition is tested or executed to be* ***TRUE***. Let's see how this is illustrated in the flowchart below.



While Loop

Process of a while loops in Python

In the bank account double program, for this to work successfully, we will need to initialise **the balance**, **the target**, **the rate** and **the year**. The bank account balance grows exponentially.

While Loop

Key Facts

Important facts about while loop

- the variables should be initialised outside the while loop construct
- put a colon at the end of the while condition (at the header of the while loop).
- the statement in the while loop block are executed while the condition is **TRUE**
- the statement in the body of the compound statement must be indented to the same column position.
- if the condition never becomes **FALSE**, then an infinite loop would occur

While Loop

Process of a while loops in Python

We often wish to repeat the execution a sequence of statements in a given number of times. Here, you can simply use a *while* loop controlled by a **counter**. For example:

```
1 count = 1 # intiatlise the counter
2 while count <= 15:  # check the counter condition is TRUE
3     print(count)
4     count = count + 1 # update the loop variable
```

This process is sometimes refer to as ***count-controlled*** or sometimes known as ***definite***. But, in while loop this is called ***event-controlled*** loop because it executes until an event occurs, i.e., when the program reaches its target. We know that our loop for the bank account double program would execute a definite number of times. In our example, this would be 10 times. However, we do not know how many iterations it would take to accumulate the target balance, in this case we can consider this notion as ***indefinite***.

While Loop

Process of a while loops in Python

Let's see the solution to the account balance double task.

```
#!/usr/bin/env python
"""
A program to demonstrate bank account double
"""
def main():
    # initialise the variables
    balance = 10000
    TARGET = balance * 2
    RATE = 5
    year = 0
```

While Loop

Process of a while loops in Python

```
11      # use a while construct
12      while balance < TARGET:
13          year = year + 1
14          interest = balance * RATE/100
15          balance = balance + interest
16      print("Balance when account doubled is: {}".format(round(balance,2))
        )
17      print("It would take {} years".format(year))
18
19  # start program
20  main()
```

While Loop: *Infinite Loop*

Infinite Loop Error

One of must common errors in while loop is the **infinite loop**. Infinite loop is a loop that runs forever and can be *stopped only by terminating the program or restarting the computer*. **If there are output statements in the loop, then many lines of output flash by on the screen**. Otherwise, the program just sits there and hangs, seeming to do nothing. On some systems, you can stop or kill a hanging program by hitting ***Ctrl + C***. Another option is to ***close the window in which the program runs***. A common reason for infinite loops is forgetting to update the variable that controls the loop:

While Loop: *Infinite Loop*

Infinite Loop Error

Let's look at an example of code construct that would lead to infinite loop.

```
1  #!/usr/bin/env python
2  def main():
3      balance = 10000
4      RATE = 5
5      year = 1
6      while year  <= 20:
7          interest = balance * RATE/100
8          balance = balance + interest
9          print(balance)
10
11 # start program
12 main()
```

The program did not increment the year (**year = year + 1**) in the loop. Therefor, the year variable always stays at 1, and the loop never comes to an end.

While Loop: *Better Program*

Better program

```
#!/usr/bin/env python
def main():
    balance = 10000
    RATE = 5
    year = 1
    while year  <= 20:
        interest = balance * RATE/100
        balance = balance + interest
        year += 1
    print(round(balance, 2))

# start program
main()
```

While Loop: *Infinite Loop*

Infinite Loop Error

Let's look at another common example of a infinite loop. When we accidentally increment a counter value that should be decremented (or **vice versa**).

```
#!/usr/bin/env python
def main():
    balance = 10000
    RATE = 5
    year = 20
    while year  > 0:
        interest = balance * RATE/100
        balance = balance + interest
        year += 1
# start program
main()
```

The year variable should have been decremented, not incremented. This is a **common error** because incrementing counters is so much common than decrementing.

While Loop: *Better Program*

Better program

```
#!/usr/bin/env python
def main():
    balance = 10000
    RATE = 5
    year = 20
    while year  > 0:
        interest = balance * RATE/100
        balance = balance + interest
        year -= 1
    print(round(balance,2))
# start program
main()
```

While Loop: *Sentinel Control Loop*

What is sentinel value?

In this section, you will be exposed to new ways loop processing sequence of input values. We will explore method of indicating the end of the sequence while we are reading sequence of input. In programming ***sentinel value*** *denotes the end of an input sequence or the border between input sequences*. Sometimes we want to prompt the user to keep entering input values within our program, we can then decide to set the value **0** or **999** as a *sentinel control value* to finish the sequence. Once the user enters the sentinel value set, this should end the program input sequence. Also, we can use negative number such as **-1** to terminate the sequence if any of the values *0* or *999* is allowed as input in the sequence. Such a value which is not part of the required input, but serves as a signal for the program termination is called ***sentinel***.

While Loop: *Sentinel Control Loop*

Example of Sentinel Control Loop

You can use any sentinel value such as a negative number (**e.g., -1**) to stop the loop. Let's see the syntax for sentinel control loop.

```
while . . . :
    salary = float(input("Enter a salary or -1 to finish: "))
    if salary >= 0.0 :
        total = total + salary
        count = count + 1
```

When the loop is entered for the initial time, no data value has been read. We must make sure to initialize the salary variable with a value that will satisfy the while loop condition so that the loop will be executed at least once. Let's look at an example of how to create sentinel control within a while loop construct.

While Loop: *Sentinel Control Loop*

Example of Sentinel Control Loop

```
#!/usr/bin/env python
"""
A program to demonstrate sentinel control loop
"""
def main():
    total = 0.0 # initialise the total
    count = 0 # initialise the count
    salary = 0.0 # initialise the salary
    while salary != -1 : # sentinel control loop
        salary = float(input("Enter a salary or -1 to finish: "))
        if salary >= 0.0 :
            total = total + salary
            count = count + 1
            average = total/count
    print("The total salary is: {}".format(total))
    print("The average salary is: %.2f"%(average))
main()
```

While Loop: *Sentinel Control Loop*

Another Example of Sentinel Control Loop

Let's write the same program in the previous session in a different way.

```
\#!/usr/bin/env python
"""
A program to compute average salary using sentinel control loop
"""
# Initialize variables to maintain the running total and count.
total = 0.0
count = 0
# Initialize salary to any non-sentinel value.
salary = 0.0
# Process data until the sentinel is entered.
while salary >= 0.0 :
    salary = float(input("Enter a salary or -1 to finish: "))
    if salary >= 0.0 :
        total = total + salary
        count = count + 1
```

While Loop: *Sentinel Control Loop*

Another Example of Sentinel Control Loop

```
16 # Compute and print the average salary.
17 if count > 0 :
18     average = total / count
19     print("Average salary is", average)
20 else :
21     print("No data was entered.")
```

While Loop: *Sentinel Control Loop*

Another Example of Sentinel Control Loop

Sometimes you would want to only initialise the input variable with a value other than a sentinel. ***Although this also solves the problem, it requires the implementation of decision control of an if statement in the body of the loop in order to test for the sentinel value.*** Another popular approach is to declare two input statements, one before the loop to obtain the first value and another at the bottom of the loop after the counter iteration to read additional values.

While Loop: *Sentinel Control Loop*

Another Example of Sentinel Control Loop

```
#!/usr/bin/env python
"""
A program to compute average salary using sentinel control loop
"""
# Initialize variables to maintain the running total and count.
total = 0.0
count = 0
# Initialize salary to any non-sentinel value.
salary = 0.0
# Process data until the sentinel is entered.
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0.0 :
        total = total + salary
        count = count + 1
        salary = float(input("Enter a salary or -1 to finish: "))
```

While Loop: *Sentinel Control Loop*

Another Example of Sentinel Control Loop

```
16 # Compute and print the average salary.
17 if count > 0 :
18     average = total / count
19     print("Average salary is", average)
20 else :
21     print("No data was entered.")
```

If the first value entered by the user is the sentinel, then the body of the loop is never executed. Otherwise, the value is processed just as it was in the earlier version of the loop. The input operation before the loop is known as the ***priming read***, because it prepares or initializes the loop variable. **The input operation at the bottom of the loop is used to obtain the next input**. It is known as the **modification read**, because it modifies the loop variable inside the loop. **Note that this is the last statement to be executed before the next iteration of the loop**. If the user enters the sentinel value, then the loop terminates. **Otherwise, the loop continues processing the input**.

Break Statement

What is a **break statement?**

The break statement breaks out of the enclosing loop, independent of the loop condition. A break statement, when executed in a while, for, do ...while or cases and so on, **causes immediate exit from the statement** (i.e., it causes the program control to proceed with the first statement or condition that is **TRUE** and exit). ***When the break statement is encountered, the loop is terminated, and the statement following the loop is executed***. The break statement allows you to control the flow of a while loop and not end up with an **infinite loop**. In the loop-and-a-half case, break statements can be beneficial. However, it is difficult to lay down clear rules as to when they are safe and when they should be avoided.

Break Statement *Example*

Example of break statement

Let's look at an example of a break statement in Python.

```
#!/usr/bin/env python
"""
A program to demonstrate a break statement
"""
def main():
    count = 0
    for i in range(1, 11): # iterate over 10 numbers from 1 - 10
        count += 1
        if (count == 5): # if number is 5, terminate or exit the loop
            print("\nUsed break to exit the program at %d\n"%(count))
            break # exit the remaining code in loop if count is 5
        print("%d"%(count))
# start program
main()
```

Continue Statement

What is a **continue statement?**

The **continue statement**, unlike the **break statement** *does not terminate the program at the position of call or use, this just skip that condition and move on or continue to the next statement within the loop construct*. continue statement, when executed in a while, for or do ... while, ***skips the remaining statements in the loop body and proceeds with the next iteration of the loop***.

- *In while and do ...while statements, the program evaluates the loop-continuation test immediately after the continue statement executes.*
- *In a for loop statement, the increment expression executes, then the program evaluates the loop-continuation test.*

Continue Statement *Example*

Example of continue statement

Let's look at an example of a continue statement in Python.

```
#!/usr/bin/env python
"""
A program to demonstrate a continue statement
"""
def main():
    count = 0
    for i in range(1, 11):# iterate over 10 numbers from 1 - 10
        count += 1
         # if number is 5, skip and continue to the remaining numbers
        if (count == 5):
            print("\nUsed continue to skip printing %d\n"%(count))
            continue # skip remaining code in loop if count is 5
        print("%d"%(count))
# start program
main()
```

For Loop

For loop implementation

For loop is an orderly sequence of iterative steps in a program.It is very common to visit each character in a string stored in a variable. The **for loop** control sequence makes this process easy to program.

Syntax:

```
for variable in container:
    statements
```

Syntax:

```
for variable in range(...):
    statements
```

For Loop

For loop implementation

For example, **suppose we want to print the character of a string one after the other per line**. We cannot simply print the string using the print function. *Instead, we need to iterate over the characters in the string and print each character individually*. Here is how you use the for loop to accomplish this task:

```
#!/usr/bin/env python
"""
A program to print a string using for loop
"""
def main():
    name = "Danny"
    for char in name:
        print(char)
main()
```

For Loop

For loop implementation

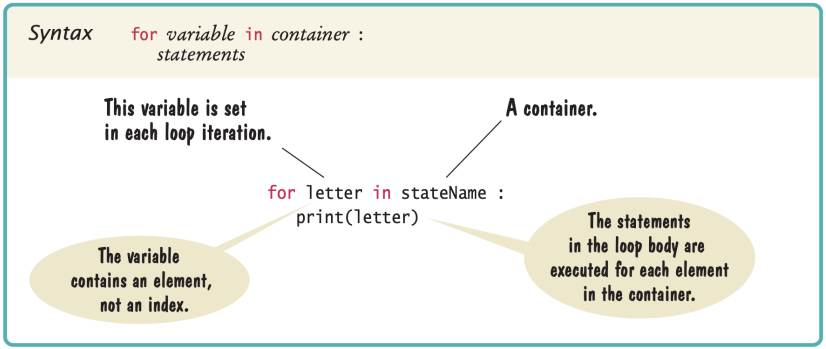
Writing the same program in while loop would be:

```
#!/usr/bin/env python
"""
A program to demonstrate using whilel oop to work as for loop
"""
# equivalent while loop program from previous for loop
def main():
    i= 0
    name = "Danny"
    while i < len(name) :
        char = name[i]
        print(char)
        i=i+ 1
main()
```

For Loop

For loop implementation

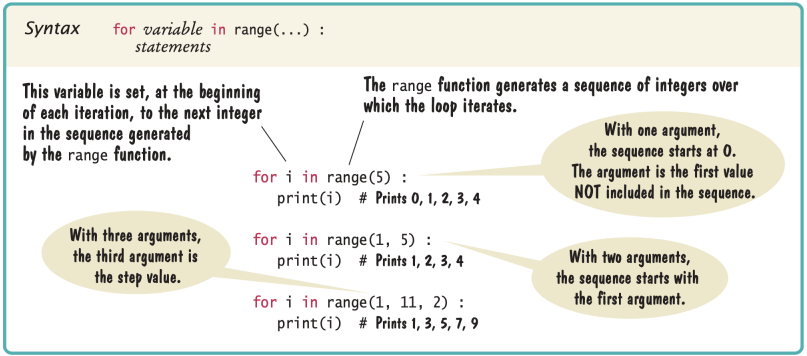
Note an important difference between the *for loop* and the while loop. In the for loop, the element variable **char** is assigned **name[0]**, **name[1]**, and so on. In the *while loop*, the index variable **i** is assigned *0, 1*, and so on.



For Loop

For loop implementation

As you have noticed in prior lectures, loops that iterate over a range of integer values are very common. To simplify the creation of such loops, Python provides the range function for generating a sequence of integers that can be used with the for loop.



For Loop

For loop implementation

By default, the range function creates the sequence in steps of 1. This can be changed by including a step value as the third argument to the function:

```
#!/usr/bin/env python
"""
A program to demonstrate using loop to print numbers in ascending order
"""
for i in range(1, 10, 2) : # i = 1, 3, 5, ..., 9              print(i)
```

Now, only the odd values from **1 to 9** are displayed. We can also have the for loop count down in descending order instead of up.

For Loop

For loop implementation

In the program listing below, the numbers are displayed in descending order in a sequence from **10 to 1**.

```
#!/usr/bin/env python
"""
A program to demonstrate using loop to print numbers in descending order
"""
for i in range(10, 0, -1) : # i = 10, 9, 8, ..., 1          print(i)
```

Finally, you can use the range function with a single argument. When you do, the range of values starts at zero.

```
#!/usr/bin/env python
"""
A program to demonstrate using loop to print a sting ten times
"""
for i in range(10) : # i = 0, 1, 2, ..., 9
    print("Hello") # Prints Hello ten times
```

Nested Loop

Nested loop implementation

We discussed nested if statements and nested branch statement previously. Similarly, in Python programming, complex iterations sometimes require a nested loop to make it work easily. Nested loop is a loop inside another loop statement. When processing tables, nested loops occur naturally. An outer loop iterates over all rows of the table. An inner loop deals with the columns in the current row. **This nested loop must be placed inside the preceding loop. This means that the inner loop is nested inside the outer loop**

The hour and minute displays in a digital clock are an example of nested loops. The hours loop 12 times, and for each hour, the minutes loop 60 times.

Nested Loop

Nested loop implementation

Let's look at an example of a nested loop.

```
#!/usr/bin/env python
"""
This program prints a table of powers of x using a nested loop construct
"""
def main():

    # Initialize constant variables for the max ranges.
    NMAX = 4
    XMAX = 10
    # Print table header. nested loop
    for n in range(1, NMAX + 1):
        print("%10d" % n, end="")
    print()
```

Nested Loop

Nested loop implementation

```
14      for n in range(1, NMAX + 1) :
15          print("%10s" % "x ", end="")
16      print("\n", " ", "-" * 35)
17      # Print table body.
18      for x in range(1, XMAX + 1) :
19      # Print the x row in the table.
20          for n in range(1, NMAX + 1) :
21              print("%10.0f" % x ** n, end="")
22          print()
23  main()
```


Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following:

- We discuss control flow structure - if - elif and else
- We look at the concept of multiple condition statements
- We looked at various nested statement and conditional expressions
- We discussed nested branch statement
- We discussed about loops, while loop, for loop and nested loop
- We looked at sentinel control loop

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 3 Decisions, Chapter 4 Loops*
- Learning Python (5th Edition): **By Mark Lutz** - *Chapter 12 If Tests and Syntax Rules, Chapter 13 While and For Loops*
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 4 For Loops, Strings and Tuple: The World Jumble Game*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 5 If Statements, Chapter 5 User Input and While Loops*

Next Lecture 4

In week 4

Lecture 4: **Principles of Object-Oriented Programming**