# INST0004 Programming 2

## Lecture 06: Recursion in Python

Module Leader:
Dr Daniel Onah

2023-24


UCL

# Copyright Note

Important information to adhere to

# Summary of Previous Lecture

## Recap of previous lecture

Re: Cap☺

Last week we looked at ...

- how to develop a class structure and components
- understand class and objects creation
- we looked at concepts of classes and types of constructors
- we looked at how to developed a simple pseudocode programs
- we looked at how to create algorithm programs
- various Python methods for the designing suitable algorithms and class structures

# Learning Outcomes

## The learning outcomes for the lecture

The objective of this week's lecture is to introduce the concept recursion in Python programming. At the end of the lecture, you should be able to:

- understand the **basic concepts of recursion in Python**
- understand how to write a recursive **factorial function**
- understand how to write a recursive function for **raised to power** problem
- understand how to define a recursive function for a **binary search** problem
- understand how to define a recursive function to solve **Tower of Hanoi** problem

# Recursion

## Recursion the idea ...

A recursion procedure refers to itself. **Recursive function is a function that calls itself**. *You can refer to a procedure inside the procedure's own definition or declaration*. You have seen instances of functions calling other functions. Function **A** can call function **B**, which can the call function **C**. It's also possible for a method to call itself. A function that *calls itself **directly** or **indirectly** in an **iterative** order* is called a **recursive function**. The number of times that a function calls itself is know as the **depth of recursion**.

# Ingredients of a Recursive Function

## Recursion the process ...

All recursive functions mush have the following:

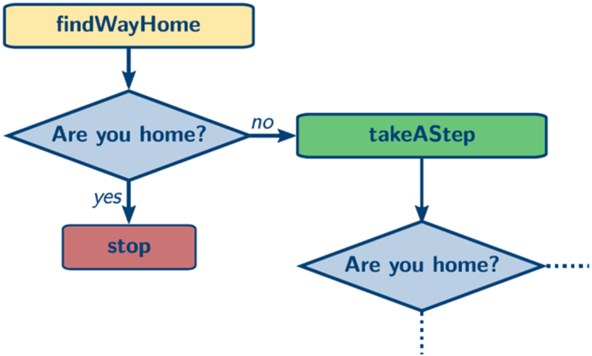
- **Base Case**: knowing when to stop
- **Work toward Base Case:** make the problem simpler(**standard case**)
- **Recursive Call:** the function calls itself on a simpler problem
- ***You can have as many base and standard cases as you need in a recursive function.***

(Compare the Market, 2009–☺)

# A Simple Recursive Procedure

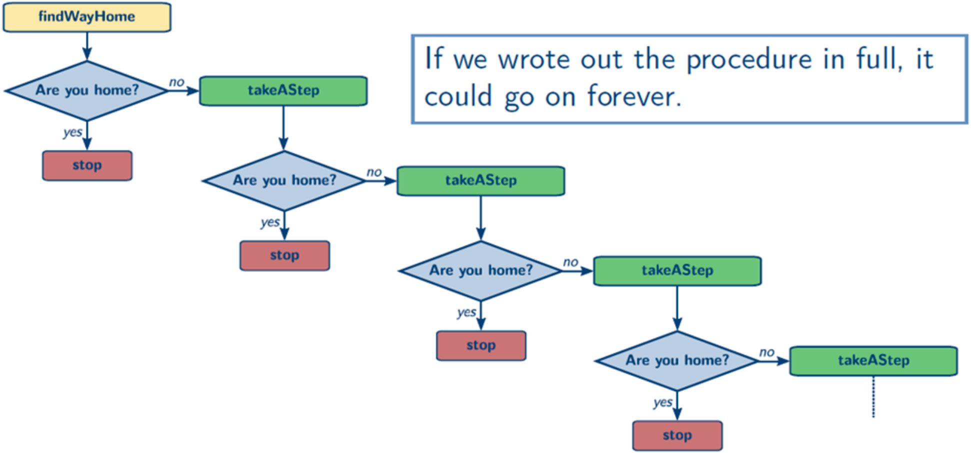
## Finding your way home procedure...

Let's look at a simple procedure for walking home.



# A Simple Recursive Procedure

Finding your way home procedure...




If we wrote out the procedure in full, it could go on forever.

# A Simple Recursive Procedure

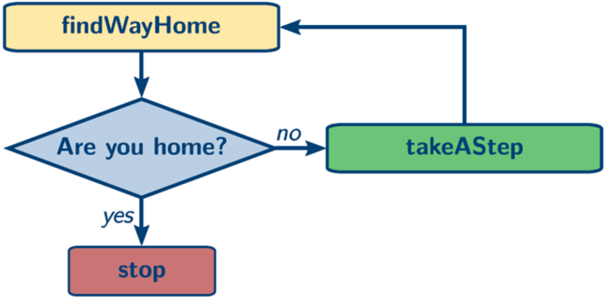
## Finding your way home procedure...

A more better procedure should have be to use a loop to iterate over the conditions.

- Instead, we can simply take a step then say **'follow the procedure again'!**
- Slightly **different from** saying simply 'take steps until you are home', e.g., an iteration or a loop

# A Simple Recursive Procedure

Finding your way home procedure...




- This provides a **'procedural conditional loop'!**
- Only when the condition is satisfied, then the looping would stop and the program ends!

# Recursive Pseudocode

## A Recipe to finding your way home...

Let's look at a simple recipe for finding your way home. We would represent this with a simple pseudocode structure:

To find your way home!

- If you are at home, stop moving
- If not:
  - Then take one step toward home.
  - Now find your way home.

This we can say is an example of a recursive procedure! **why?**

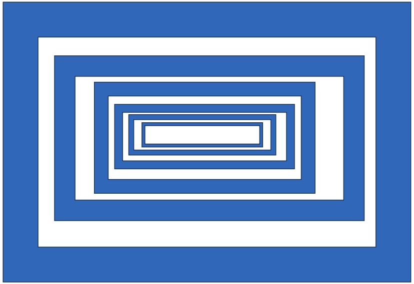
***Because***:

```
A Recursive procedure certainly and always refers to itself in
   programming!
```

# Recursive Function

## Recursive function call...

A function definition that **includes a call to itself** is said to be **recursive**. Python allows function to be recursive in a program.



# Recursive Function Error

Recursive function call error...

A **RecursionError** in Python *is caused by a function calling itself recursively **without a proper base case***. **Python has a limit on the number of times a function can call itself recursively**. This is to ensure that the function does not execute indefinitely. If this limit is exceeded by a recursive function, a **RecursionError** is raised. In Python *RecursionError* is an exception that occurs when the maximum recursion depth is exceeded. ***This typically occurs when a function calls itself recursively, and the recursion doesn't have a proper stopping condition (base case)***. In this section, we will look at an example of a typical recursive error to help us understand this concept better.

# Recursive Function Error

Recursive function call error...

```python
#!/usr/bin/env python
"""
A program demonstrating recursive error
"""
def recursiveFunction():
    recursiveFunction()
    # no base case ... this would lead to a recursive error

recursiveFunction()
```

RecursionError

```
line 6, in recursiveFunction
    recursiveFunction()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

# Recursive Function Error

Recursive function call error...

In the previous example, since the recursive function recursiveFunction() ***does not have a terminating condition, calling it creates an infinite loop as the function keeps calling itself over and over again until the*** **RecursionError:** maximum recursion depth exceeded error occurs.

# Fixing Recursion Error

## Fixing recursive error in Python

Let's look at a few ways we could approach fixing a recursion error in Python:

1. **Adding a base case**: The most common cause of a recursion error is that the function does not have a **base case** to stop the recursion. In such cases, a base case should be added to the function to stop the recursion when a condition is met.
2. **Increasing the recursion limit**: Python has a default maximum recursion depth of **1000**. If a function exceeds this limit, it can be increased using the **sys.setrecursionlimit(n)** function. You should be very careful when increasing the limit as this can cause a crash to your program if the recursion is not properly controlled.
3. **Using an iterative approach**: If a recursive approach is causing a recursion error, it may be possible to use an iterative approach instead e.g. a **for** or while loop. This can reduce the risk of hitting the maximum recursion depth, and in some cases can also lead to more efficient and easier to understand code.

# More on common recursive error

## Resolving recursive issues ...

Example of a common recursive error : RecursionError: maximum recursion depth exceeded in comparison.

**Optimize Your Recursion:** First, try to optimize your recursive function. Make sure that your recursive calls have a **proper base case**, and you're making progress towards that base case with each recursive call. If the recursion depth is **too deep**, you may need to change your algorithm to use iteration or a different approach.

**Increase Recursion Limit:** If you believe your recursive approach is valid, but it's hitting the recursion limit due to a particularly deep problem, you can increase the recursion limit using the **sys** module. However, this should be done with caution, as it can lead to **stack overflow errors or consume excessive memory**.

# A Simple Recursive Function

Example: using iterative approach

Let's look at a program that counts the words in a sentence before the **full-stop** ( . )

```
#!/usr/bin/env python
"""
A program to count the words users type in before the full-stop.
"""
def main():
    # user type in a sentence that ends with a dot or full-stop '.'
    words = input("Type in some text, ending with a '.': ")
    # creating a list from the words in the sentence using split()
    words = words.split()
    # use an iter() function to iterate over the list items
    words = iter(words)
    count = int(countWords(words))
    print("You have typed in " , count , " words. ")
```

# A Simple Recursive Function

## Example: using iterative approach

```
def countWords(words):
    # iterate over the list and get next word
    for word in words:
        for text in word:
    # base case
            if(text.endswith(".")):
                return 1
    # standard case or recursive case
    # returns 1 + number of words in the list
        return 1 + countWords(words) # recursive function calls itself
main()
```

# Reflection

## Reflection on the word count example

Stop and think about what are the possible errors you could correct in the previous word count example program.

**What happens if there is space before the full stop?**

- Ideally, since the program intention was to count the number of words in a sentence ... then this should be the aim.
  - However, if you type in a sentence with a space at the last word before the full-stop, the space would be counted as part of the word count.

# A Simple Recursive Function

Example: using iterative approach

Let's look at a program that *removes the space* in a sentence between the **full-stop** ( . )

```python
#!/usr/bin/env python
"""
A program to count the words users type in before the full-stop.
Here we want to avoid counting the space before a full-stop.
"""
def main():
    import re # import regular expression(regex)
    # user type in a sentence that ends with a dot '.'
    words = input("Type in some text, ending with a '.': ")
    # remove the space after the last word before the full-stop.
    removeSpace = re.sub(r'\s(?=[\.,:;])', "", words)
    words = removeSpace.split()
    words = iter(words)
    count = int(countWords(words))
    print("You have typed in " , count , " words. ")
```

# A Simple Recursive Function

## Example: using iterative approach

```python
def countWords(words):
    # iterate over the list and get next word
    for word in words:
        for text in word:
    # base case
            if(text.endswith(".")):
                return 1
    # standard case or recursive case
    # returns 1 + number of words in the list
        return 1 + countWords(words) # recursive function calls itself
main()
```

# Using Normal Function

## Example: using simple iterative normal function approach

Let's look at a program that *that would do the same recursive execution*, by counting the number of words in a sentence before the **full-stop** ( . )

```
#!/usr/bin/env python
"""
A program to demonstrate the recursive function in non-recursive
    function
"""
def main():
    words = input("Type in some text, ending with a '.': ")
    words = words.split() # create a list
    words = iter(words)
    count = int(countWords(words))
    print("You have typed in ", count , "words.")
```

# Using Traditional Function

Example: using simple iterative normal function approach

```
def countWords(words):
    # an iterative solution couting the words in a sentence
    count = 0
    for word in words:
        count+=1
        for text in word:
            if(text.endswith(".")):
                return count
main()
```

# Recursive Function Steps

## Example: explaining recursive function depth

Let’s look at how this recursive function works step-by-step. Suppose we run the wordCounts program and type in the following sentence: ***Danny likes recursion.***

```
#!/usr/bin/env python
"""
A program to test and demonstrate the steps in a recursive function
"""
def main():
    words = input("Type in some text, ending with a ‘.’: ")
    words = words.split()
    words = iter(words)
    count = int(countWords(words))
    print("You have typed in " , count , " words. ")
```

# Recursive Function Steps

## Example: explaining recursive function depth

Let's take the first word in the Sentence: ***Danny likes recursion.***

```
def countWords(words):
    for word in words:
        for text in word:
            # >> STEP 1: if (text == "Danny"):
    # base case
            if(text.endswith(".")):
                return 1
    # standard case
        return 1 + countWords(words) # >> ***RECURSIVE CALL***
main()
```

# Recursive Function Steps

Example: explaining recursive function depth

Let's take the second word in the Sentence: ***Danny likes recursion.***

```
def countWords(words):
    for word in words:
        for text in word:
            # >> STEP 2: if (text == "likes"):
    # base case
            if(text.endswith(".")):
                return 1
    # standard case
        return 1 + countWords(words) # >> ***RECURSIVE CALL***
main()
```

# Recursive Function Steps

## Example: explaining recursive function depth

Let's take the second word in the Sentence: ***Danny likes recursion.***

```
def countWords(words):
    for word in words:
        for text in word:
            # >> STEP 3: if (text == "recursion."):
    # base case
            if(text.endswith(".")):
                return 1 # >> ***THE BASED CASE IS REACHED AT THE LAST
DEPTH***
    # standard case
        return 1 + countWords(words)
main()
```

The **depth** of recursion within the *countWords()* function is **3**.

# Recursive Function Steps

## Example: explaining recursive function depth

Let's take the second word in the Sentence: ***Danny likes recursion.*** Let's explain the operation of the **standard case**.

```
def countWords(words):
    for word in words:
        for text in word:
            # >> AT STEP 2: if (text == "likes"):
    # base case
            if(text.endswith(".")):
                return 1
    # standard case
        return 1 + 1 # >> ***ADD 1 TO THE STANDARD CASE RETURN STATEMENT ***
main()
```

# Recursive Function Steps

## Example: explaining recursive function depth

Let's take the first word in the Sentence: ***Danny likes recursion.***

```
def countWords(words):
    for word in words:
        for text in word:
            # >> AT STEP 1: if (text == "Danny"):
    # base case
            if(text.endswith(".")):
                return 1
    # standard case
        return 1 + 2 # >> ***ADD 1 TO THE STANDARD CASE RETURN STATEMENT ***
main()
```

# Recursive Function Steps

## Example: explaining recursive function depth

The main() function then call the recursive function to get the computation of the number of words in the sentence.

```python
def main():
    words = input("Type in some text, ending with a '.': ")
    words = words.split()
    words = iter(words)
    # THE PROGRAM THEN PRESENT THE COUNT OF THE RECURSIVE FUNCTION CALL
    count = int(countWords(words))
    print("You have typed in " , count , " words. ")
```

# Depth of Recursion

## Recursive procedure depth ...

The **depth of recursive procedure** can be said to be *the number of steps a program iterate over a given condition until it gets to the base case or until the base case is* TRUE. The maximum depth of recursion refers to the **number of levels of activation of a procedure** which exist during the deepest call of the procedure.

# Factorials

## Recursive factorial problem ...

**n!**: the factorial of a non-negative integer, n, is the product of all positive integers less than or equal to n. Let's look at a simple factorial process:

1! = 1
2! = 2 x 1
3! = 3 x 2 x 1
4! = 4 x 3 x 2 x 1
5! = 5 x 4 x 3 x 2 x 1

*We can rewrite these factorials in terms of smaller or modular unit of factorials*. In general:
**n! = n x (n -1)!**
A recursive definition! What are the **base** and **standard case**?

# Recursive Factorial Function

## Example: recursive factorial ...

Let's look at an example of recursive factorial.

```python
#!/usr/bin/env python
"""
A program to demonstrate a factorial function
"""
def main():
    number = int(input("Type in an integer: "))
    result = int(fact(number))
    #print(number,"!=", result)
    print("{}! = {}".format(number, result))
```

# Recursive Factorial Function

Example: recursive factorial ...

*In what order should we put the **standard case**, and **base case?*** First, we have to test for the **base case** why?. Because ... we **first check the conditional statement** to ascertain whether this is has been met or not.

```
def fact(number):
    # base case
    if(number == 1):
        return 1
    # standard case
    return (number * fact(number - 1))

main()
```

# Importance of Recursive Function

## Significance of a recursive function call in Python ...

Python (like many programming languages) supports recursion:

- A recursive function is one that includes a call to itself
- Python does not treat recursive functions any different from any other functions
  - ... but recursion can be a very useful problem-solving technique
- Recursive functions can **always** be written without recursion
  - ... but for some problems an alternative solution is much difficulty and harder to find (and much less elegant)
- It is not absolutely compulsory that you should always write your program with iterative nature with ***only a recursive function***.

# Raising to Integer Powers

## Raising to the integer powers ... recursive

$y^n$ : A real number **y** raised to a positive integer power **n** is the product of **n y's** This means multiplying **y** in **n** times. Let's look at an example or integer raise to power:

$y^0 = 1$ (by definition)

$y^1 =$ y x 1

$y^2 =$ y x y x 1

$y^3 =$ y x y x y x 1

$y^4 =$ y x y x y x y x 1

*What are the **base** and **standard** cases?*

# Raising to Integer Powers

## Raising to the integer powers ... recursive

$y^n$ : A real number **y** raised to a positive integer power **n** is the product of **n y's** This means multiplying **y** in **n** times. Let's look at an example or integer raise to power:

$y^0 = 1$ (by definition)

$y^1 =$ y x 1 » = y x $y^0$

$y^2 =$ y x y x 1 » = y x $y^1$

$y^3 =$ y x y x y x 1 » = y x $y^2$

$y^4 =$ y x y x y x y x 1 » = y x $y^3$

*What are the **base** and **standard** cases?*

In general for us to represent this using recursive definition in Python, we could simply use this expression:

$y^n =$ y x $y^{(n-1)}$

# Raising to Integer Powers

Example: recursive raise to powers ...

Let's look at an example about recursive raise to power program in Python.

```python
#!/usr/bin/env python
"""
A program to demonstrate recursive integer power function
"""

def main():
    number_1 = int(input("Type in a number: ")) # real number
    number_2 = int(input("Type in raise to power number: ")) # power integer
    result = pow(number_1, number_2)
    print("The number", number_1, "raised to the power of", number_2, " is =", result)
```

# Raising to Integer Powers

## Example: recursive raise to powers ...

```
def pow(number1, number2):
    # base case
    if(number2 == 0): # we first test the base case
        return 1
    # standard case
    return number1 * pow(number1, number2 - 1)
main()
```

Again, we first test for the base case.

# Recursive Binary Search

## Recursive binary search ... a bigger problem example

Let's look at a simple search problem:

- I have a number of words written on separate bits of paper
- They're ordered **lexicographically** (like a dictionary or text presented in alphabetical order as used in dictionary)
- You know how many words there are
- You can read one word at a time based on its index (its address in the stack)
- Find the word I am looking for in the fewest steps
- What if you can replace the whole stack with only part of it after each lookup?

# Binary Search Procedure

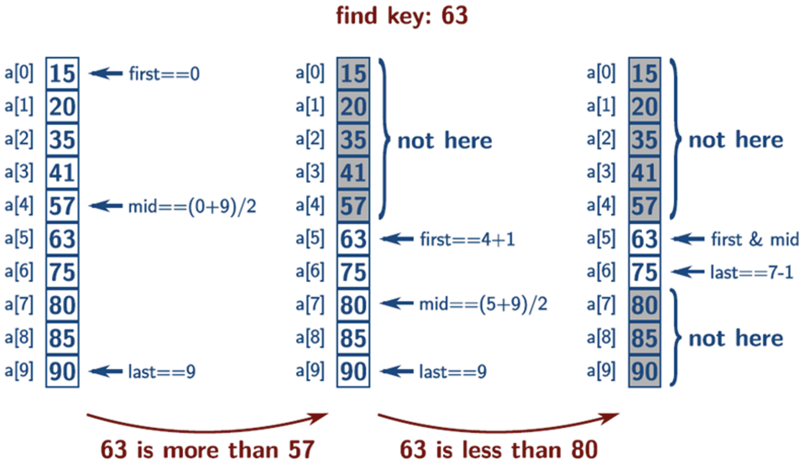
## Procedure for binary search ...

Binary search is a particularly efficient recursive algorithm for finding the position of an item in an ordered list of items. To perform Binary Search look at item approximately midway along the list:

- if this is what you are looking for then **stop**
- if the item you're looking for is:
  - less than the midway item, then perform Binary Search on the first half of the list
  - more than the midway item, then perform Binary Search on the second half of the list.

# Binary Search (Visual Representation)

Visually presenting binary search process ...



# Binary Representation

## Representation of binary search process ...

In everyday life, we tend to write numbers in their decimal representation.
From **right-to-left** we write a count for:

```
... thousands   hundreds   tens  units
```

... where each digit must be less than ten

An alternative is binary representation, where from **right-to-left** we write a count for:

```
... eights  four  two  units
```

... where each digit must be less than two

# Binary Representation

## Representation of binary search process ...

Some examples of decimal versus binary representations:

| decimal | binary |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |

| decimal | binary |
|---|---|
| 5 | 101 |
| 8 | 1000 |
| 11 | 1011 |
| 16 | 10000 |
| 17 | 10001 |

# Binary Representation

## Recursive binary algorithm ...

Let's write a recursive algorithm to convert integers into binary.
To convert non-negative decimal number to binary String:

- If number is **0** or **1** return String value
- Otherwise, divide number by two
  - convert result to binary
  - convert remainder to binary
- Then stick one to the other.

# Recursive Binary Conversion

## Example: recursive binary conversion ...

Let's look at an example of binary conversion in Python.

```python
#!/usr/bin/env python
"""
A program to demonstrate binary conversion
"""
def main():
    number = int(input("Type in an integer: "))
    resultInBinary = convertToBinary(number)
    print("In converting to binary is: ", resultInBinary)
def convertToBinary(number):
    # two base cases
    if(number == 0): # first base case
        return "0"
    elif (number == 1): # second base case
        return "1"
```

# Recursive Binary Conversion

Example: recursive binary conversion ...

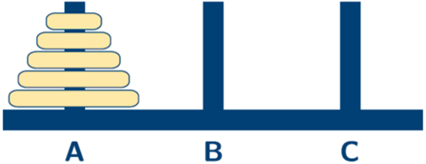
```
 # standard case with two recursive calls
    output = int(number/2)
    remainder = int(number%2)
    return convertToBinary(output) + convertToBinary(remainder)
main()
```

Notice the two base cases.

The Legend of Tower of Hanoi: Priests of Brahma are working on 64 disk version in a temple at the centre of the world. The believe is that when the priests finish, the world will end. *Moving one disk in a second, it would take the Priests over **584 billion years** to accomplish moving the entire disks and completing the task*.

# Tower of Hanoi

## Recursive tower of Hanoi ...




Goal: move all the yellow disks from peg A to peg C, using peg B as a temporary place to put disks.

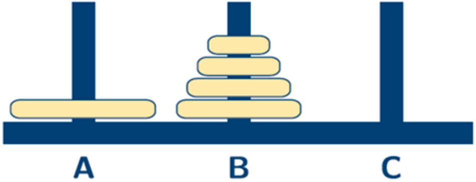
Rules:

- Move only one disk at a time.
- Do not put a larger disk on top of a smaller one.
- Disks cannot be left anywhere except on a peg.

# Tower of Hanoi

## Recursive solution to tower of Hanoi ...

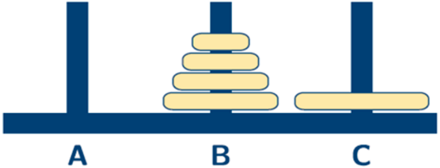
1. Solve the 4-Disk
   - **A - to - B puzzle:**



# Tower of Hanoi

## Recursive solution to tower of Hanoi ...

2. Move the largest disk from
   - **A - to - C:**



# Tower of Hanoi

## Recursive solution to tower of Hanoi ...

3. Solve the 4-Disk
   - **B - to - C puzzle:**

# Tower of Hanoi Recursive Program

## Example: A recursive solution to Tower of Hanoi problem ...

Let's look at the solution to the **Tower of Hanoi** problem in Python.

```
#!/usr/bin/env python
"""
A program to demonstrate solving the legend of the Tower of Hanoi
    recursively
"""
def TowersOfHanoi():
    numberOfDisks = int(input("How many disks?: "))
    moveToTower(numberOfDisks, 'A', 'C', 'B')
```

# Tower of Hanoi Recursive Program

Example: A recursive solution to Tower of Hanoi problem ...

```python
def moveToTower(numberOfDisks, start, end, spareDisk):
    # base case
    if (numberOfDisks == 1):
        moveOneDisk(start, end)
    else:
        # two recursive calls in standard case
        moveToTower(numberOfDisks - 1, start, spareDisk, end)
        moveOneDisk(start, end)
        moveToTower(numberOfDisks - 1, spareDisk, end, start)

def moveOneDisk(start, end):
    print("Move the top disk on ", start , " to ", end)

TowersOfHanoi()
```

# Tower of Hanoi Recursive Program

## Output solution ...

```
How many disks?: 5
Move the top disk on  A  to  C
Move the top disk on  A  to  B
Move the top disk on  C  to  B
Move the top disk on  A  to  C
Move the top disk on  B  to  A
Move the top disk on  B  to  C
Move the top disk on  A  to  C
Move the top disk on  A  to  B
Move the top disk on  C  to  B
Move the top disk on  C  to  A
Move the top disk on  B  to  A
Move the top disk on  C  to  B
Move the top disk on  A  to  C
Move the top disk on  A  to  B
Move the top disk on  C  to  B
```

# Tower of Hanoi Recursive Program

## Output Solution ...

```
Move the top disk on  A  to  C
Move the top disk on  B  to  A
Move the top disk on  B  to  C
Move the top disk on  A  to  C
Move the top disk on  B  to  A
Move the top disk on  C  to  B
Move the top disk on  C  to  A
Move the top disk on  B  to  A
Move the top disk on  B  to  C
Move the top disk on  A  to  C
Move the top disk on  A  to  B
Move the top disk on  C  to  B
Move the top disk on  A  to  C
Move the top disk on  B  to  A
Move the top disk on  B  to  C
Move the top disk on  A  to  C
```


Summary

# Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following:

- A recursive function includes a call to itself somewhere in its definition. It is a function that *calls itself either directly or indirectly in order to loop or iterate over a given condition*.
- If a problem can be reduced to smaller instances of the two kinds of cases:
  - *one or more base cases without recursive calls*
  - *one or more cases that include at least one recursive call or standard case*

**Warning Notice:** Be careful when writing a recursive function definition, always check to see that the function will not produce an ***infinite recursion***.

**Important Notice:** Every recursive program should always reach its base case **eventually at the end of the recursive depth**. Therefore ensure your recursive program have a *properly defined base case condition*.

# Further Reading

## chapters to find further reading on the concepts

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

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 11 Recursion*
- Learning Python ($5^{th}$ Edition): **By Mark Lutz** - *Chapter 19 Advanced Function Topics*

# Next Lecture 7
In week 7

**Lecture 7: More on Objects and Classes**