JOHNS HOPKINS
WHITING SCHOOL
*of* ENGINEERING

# EN.540.635
# Software Carpentry

## Lecture 7
## Variable Scope | Recursion | Data Structures

# Variable Scope

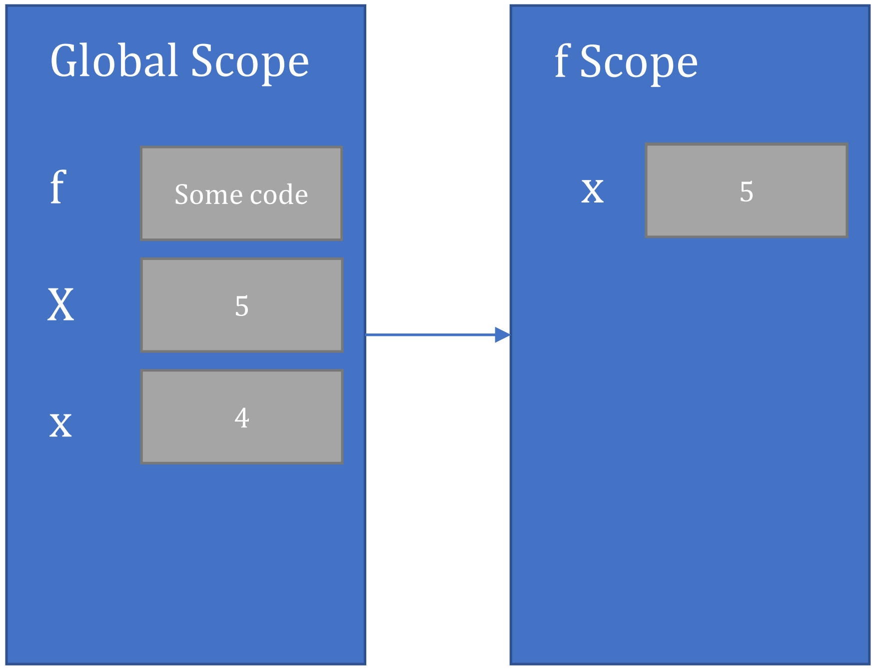


- Scope is the environment in a program from which a particular Python object is accessible
- When you enter a function, a new ‘scope’ is created

```
def f(x):
    x = x + 1
    print('Within f, x =', x)
    return x
        Within f, x = 5
        5
if __nar4               _':
    x = 4
    print(f(x))
    print(x)
```

# Variable Scope

```python
def f(x):
    x = x + 1
    print(x)
    print(X)
    return x


if __name__ == '__main__':
    X = 5
    x = 4
    f(x)
```

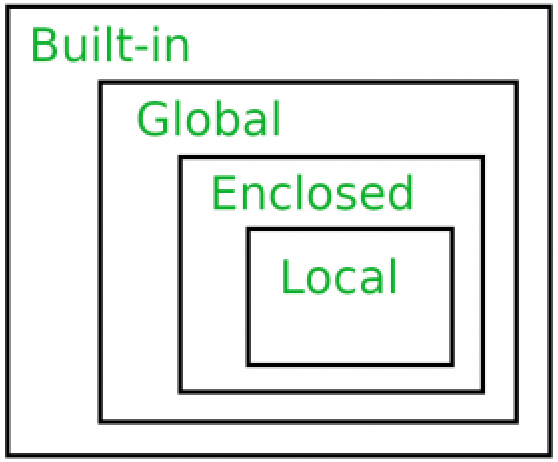



- Use of global variables is frowned upon. You can use them as constants (GLOBAL_CONSTANT)
- https://stackoverflow.com/questions/19158339/why-are-global-variables-evil

# Variable Scope

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

- In python, the LEGB rule is used to decide the order in which the namespaces are checked:
  - Local: Inside function/ class
  - Enclosed : Defined inside enclosing functions (parent/ nesting) function
  - Global: Uppermost Level
  - Built-In
- Python tutor:
  - [http://pythontutor.com/](http://pythontutor.com/)



# Recursion

- A programming technique in which a program calls itself
- An iterative solution to a problem can also be solved recursively

```python
def iter_factorial(n):
    prod = 1

    while n > 0:
        prod = prod * n
        n = n - 1

    return prod
```

```python
def rec_factorial(n):
    if n == 1:
        return 1
    else:
        return n * rec_factorial(n - 1)
```

Base case

# Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

```python
def rec_fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return rec_fibonacci(n - 1) + rec_fibonacci(n - 2)
```

```python
def iter_fibonacci(n):
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n - 1):
        old, new = new, old + new

    return new
```

A recursion approach is sometimes more intuitive than the iterative approach to solve a problem. Towers of Hanoi is one such problem.

# Recursion vs Iteration

```
t0 = time.time()
print(rec_fibonacci(30))
t1 = time.time()
print(t1 - t0)
print(iter_fibonacci(30))
t2 = time.time()
print(t2 - t1)
```

```
832040
0.5660800933837891
832040
0.0
```

```
t0 = time.time()
print(rec_fibonacci(49))
t1 = time.time()
print(t1 - t0)
print(iter_fibonacci(49))
t2 = time.time()
print(t2 - t1)
```

```
165580141
146.2495937347412
165580141
0.0
```

- Recursion is almost definitely always slower than the iterative solution as the size of the input increases

# Why Data Structures?



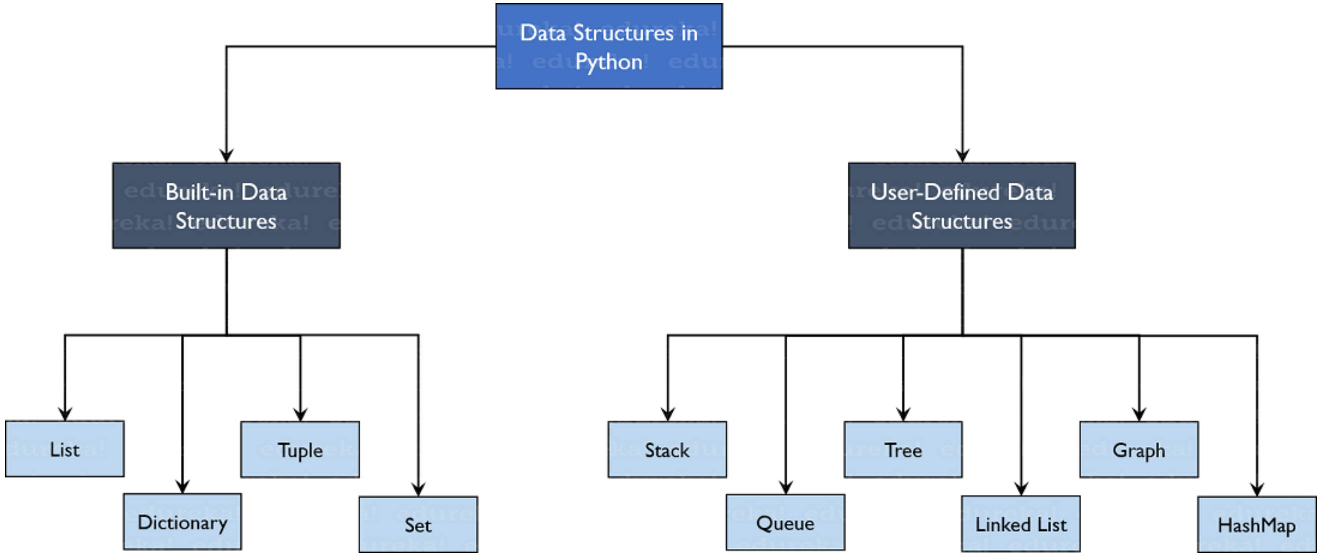
- Optimize the processing of data via:
  - Algorithms
  - Data Structures

- Data Structures help in:
  - Organization: Instead of having N variables, we can have one variable that holds N values
  - Speed: It can be MUCH faster to search (ex. smallest value) in a data structure
  - Math: Custom data structures (classes and objects) and default/in-built functions can help in mathematical operations (ex: Matrices)

# Common Data Structures

- Things we've already seen:
    - Lists
    - Tuples
    - Dictionaries
    - Strings

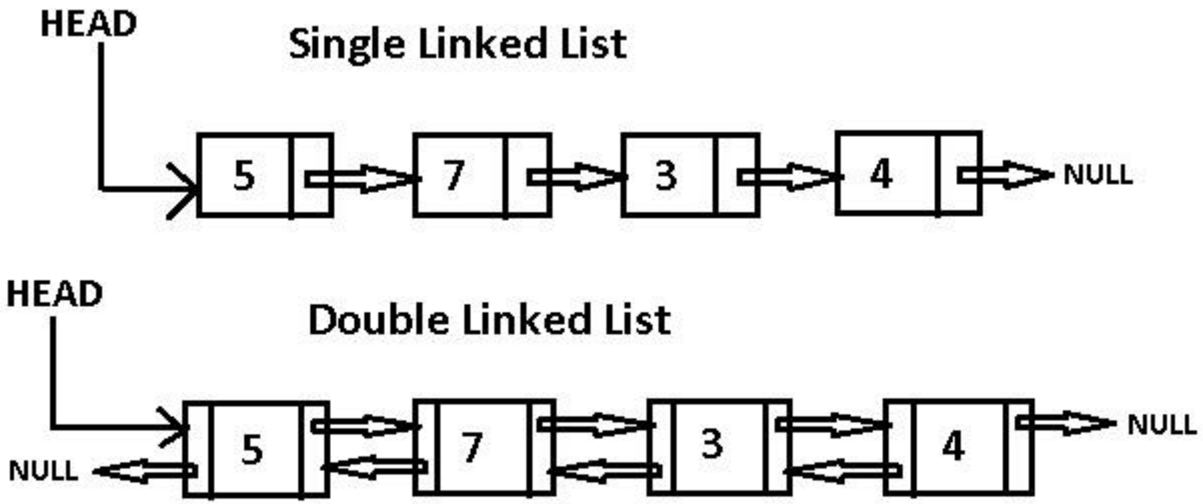
- New concepts in this class:
    - Linked Lists
    - Binary Trees
    - Stacks and Queues




https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2019/10/TreeStructure-Data-Structures-in-Python-Edureka1.png

# Linked Lists

- A list of objects, with each element “pointing” to the next
- A double linked list also points backwards
- Not commonly used in python



# Linked Lists

Johns Hopkins Whiting School of Engineering

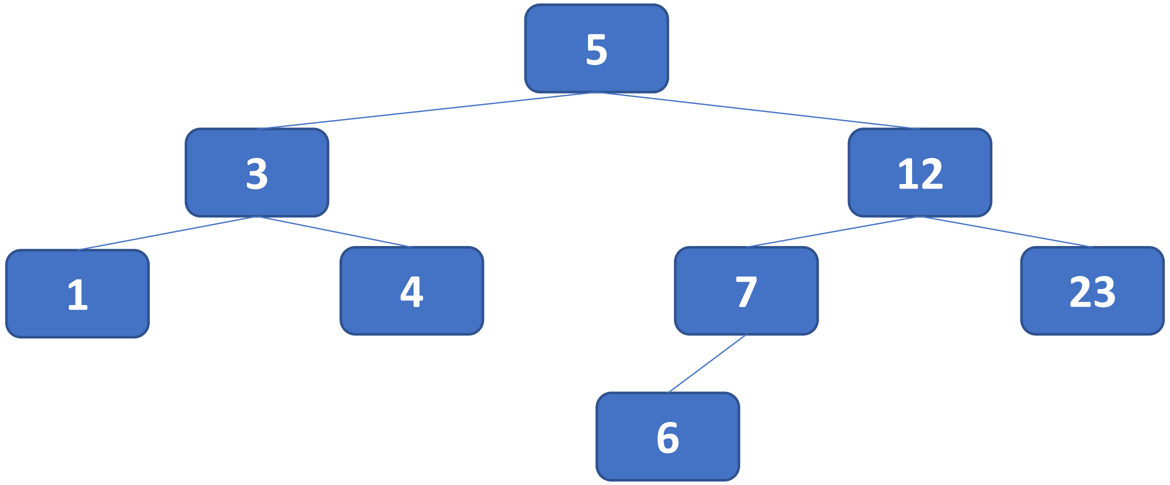
```python
class Node:
    '''
    The Node class is the building block for the user-defined linked-list
    data structure. Each node (for a singly linked list) holds the data
    corresponding to each node and also points to the next element in the
    linked list

    **Attributes**
        val: *int*
            The data/ cargo held by the node, with which it
            must be initialized
        next: *class: Node, optional*
            A node object that the node may or may not point
            to

    **Returns**
        Node: *class: Node*
            The Node class container
    '''

    def __init__(self, x, point=None):
        '''
        Initialze a Node object

        ** Parameters**
            x: **int**
                Data/Cargo for Node
            point: **class: Node, optional**
                Node to which to current Node points to
        '''
        self.val = x
        self.next = point
```

```python
class Linked_List:
    '''
    The Linked_list class is a user defined implementation of a singly-linked
    list, consisting of nodes, with each node pointing to the next. The class
    is initialized using a head 'Node' object. Can be initlaized as empty

    **Attributes**
        head: **class: Node, optional**
            Starting Node for the Linked List

    **Returns**
        Linked_List: **class: Linked_list**
    '''

    def __init__(self, node=None):
        '''
        **Parameters:
            node: **class: Node**
                Node to initialize the Linked List with
        '''
        self.head = node

    def insert_node(self, node_data):
        '''
        Insert the next node in a Linked List, at the end of the
        existing one. If the Linked List is empty, the node is assigned to be
        the head. Each added node points to None, as it is added to the end.

        **Parameters**
            node_data: **int**
                Data/Cargo for the node that is to be added to the linked list.
                A new Node is created with this data and then add to the
                linked list
        '''
        new_node = Node(node_data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while (current.next):
                current = current.next
            current.next = new_node

    def __str__(self):
        ll = ''
        for i in self:
            ll = ll + str(i)

        return ll

    def __iter__(self):
        current = self.head
        while(current):
            yield current.val
            current = current.next
```

# Binary Trees



- A form of data organization
- Each element has a maximum of 2 **children** nodes
- A binary tree can **insert**, **delete**, and **traverse** nodes.



# Binary Trees

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

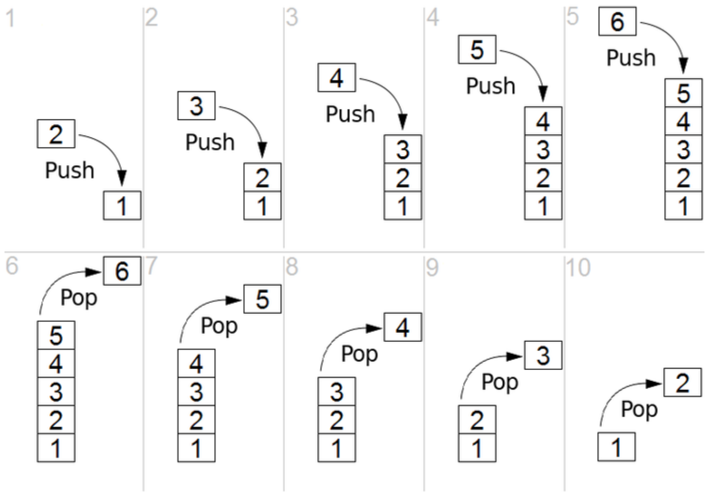
```python
class Node:
    '''
    This Node class is the building block for the user-defined binary-tree
    data structure. Each node holds the data corresponding to each node and
    also points to the next elements in the tree, the child node to its left,
    and the child node to its right

    **Attributes**
        val: *int*
            The data/ cargo held by the node, with which it
            must be initialized
        left: *class: Node, optional*
            A child node object to the left of the current node object
        right: *class: Node, optional*
            A child node to the right of the current node object

    **Returns**
        Node: *class: Node*
            The Node class container
    '''

    def __init__(self, data, left=None, right=None):
        '''
        Initialize a Node object

        **Parameters**
            data: *int*
                Data/Cargo held by Node
            left: *class: Node, optional*
                Child Node to the left of the current node
            right: *class: Node, optional*
                Child Node to the right of the current node
        '''
        self.right = right
        self.left = left
        self.val = data
```

```python
class Binary_Tree:
    '''
    This Binary Tree is a user-defined data-structure, that consists of a
    parent Node, and children nodes are added to either the left of right
    sub-tree after comapring their values to the parent. Lesser values go
    to the left and the greater values go to the right of the Binary Tree

    **Attributes**
        root: *class: Node, optional*
            The parent Node of the Binary Tree
    **Returns**
        Binary_Tree: *class: Binary_Tree*
            The class contained for the Binary_Tree
    '''

    def __init__(self, node=None):
        '''
        **Parameters**
            node: *class: Node, optional*
        '''

        self.root = node

    def add_node(self, key, node=None):
        '''
        Insert a node to either the left or the right side of a binary tree,
        after comparing the values with the parent Node, This function works
        recursively, and keeps calling itself (while traversing) either the
        left or the right sub-tree and treating each node as a parent Node,
        until it reaches the correct position in the tree

        **Parameters**
            key: *int*
                value to be added to the Binary tree as a Node
            node: *class: Node, optional*
                Node that is being considered as parent, while the Binary
                Tree is being traversed so that the new node can be added
        '''
        if not node:
            node = self.root
        if not self.head_node:
            self.root = Node(key)
        else:
            if key < node.val:
                if not node.left:
                    node.left = Node(key)
                else:
                    self.add_node(key, node.left)
            else:
                if not node.right:
                    node.right = Node(key)
                else:
                    self.add_node(key, node.right)
```

# Stacks and Queues

## STACK

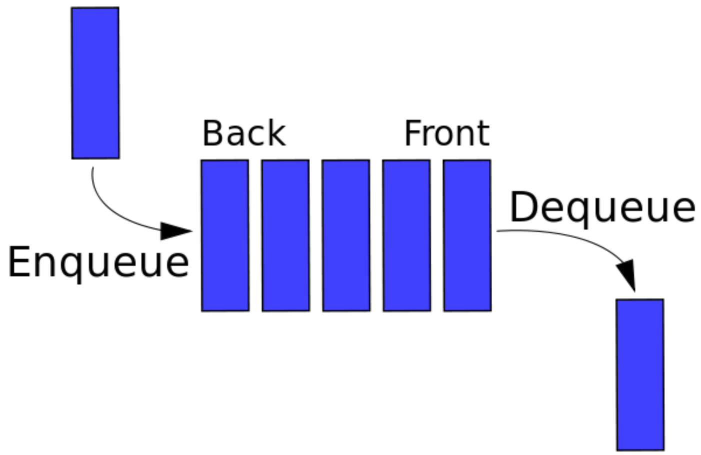
- Referred to as First In Last Out(FILO)
- Operations that can be performed are ‘push’ and ‘pop’




https://miro.medium.com/max/875/1*4Pn00ch_p4DTCb4r3naCDQ.png

## QUEUE

- Referred to as First In First Out(FILO)
- Operations that can be performed are ‘enqueue’ and ‘dequeue’




https://miro.medium.com/max/750/1*FwL7mJ4qpQWZnommC5tsFQ.png

# As Lists

- STACK

```python
stack = ['Andrew', 'Haili', 'Isaiah', 'Divya', 'Aaron']
stack.append('Nikita')
stack.append('Seun')
stack.pop()
stack.pop()
```

- QUEUE

```python
queue = ['Andrew', 'Haili', 'Isaiah', 'Divya', 'Aaron']
queue.append('Nikita')
queue.append('Seun')
queue.pop(0)
queue.pop(0)
```