Walk through previous lectures

Tuple

```
tuple_name = (value, value, ..., value)
```

- A way of packing multiple values into a variable

```
>>> x = 3
>>> y = -5
>>> p = (x, y, 42)
>>> p
(3, -5, 42)
```

```
name, name, ..., name = tuple_name
```

- Unpacking a tuple‘s contents in to multiple variables

```
>>> a, b, c = p
>>> a
3
>>> b
-5
>>> c
42
```

Using Tuples

- Useful for storing multi-dimensional data (eg- (x,y) points)

```
>>> p = (42, 39)
```

- Useful for returning more than one value

```
>>> def slope ((x1,y1), (x2, y2)):
...       return (y2 - y1) /(x2 - x1)
... p1 = (2, 5)
... p2 = (4, 11)
... slope(p1, p2)
3
```

Dictionaries

- Hash tables, "associative arrays"

```
d = {"duck": "eend", "water": "water"}
```

- Lookup:

```
d["duck"] -> "eend"
d["back"] # raises KeyError exception
```

- Delete, insert, overwrite:

```
del d["water"] # {"duck": "eend", "back": "rug"}
d["back"] = "rug" # {"duck": "eend", "back": "rug"}
d["duck"] = "duik" # {"duck": "duik", "back": "rug"}
```

Dictionaries

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values

- Keys will be listed in **arbitrary order**
  - again, because of hashing

Web Resources

- Page/Document
  - Typically written in Hyper Text Markup Language (HTML) or Extensible Markup Language (XML)
- File/Data
  - Images
  - Music
  - Executable Objects
  - …

Client Side Technologies

- Document structure / content
  - HTML, XML

- Document styling
  - CSS

- Dynamics
  - Javascript, Java, Flash, Silverlight

HTML

- Markup to support structured documents
- Semantics for . . .
  - Text (headings, paragraphs, lists, tables, etc)
  - Multimedia (images, music, video, etc)
  - Links to other resources
  - Forms
  - Styling
  - Scripting

HTML

Example: (Hello_world.html)

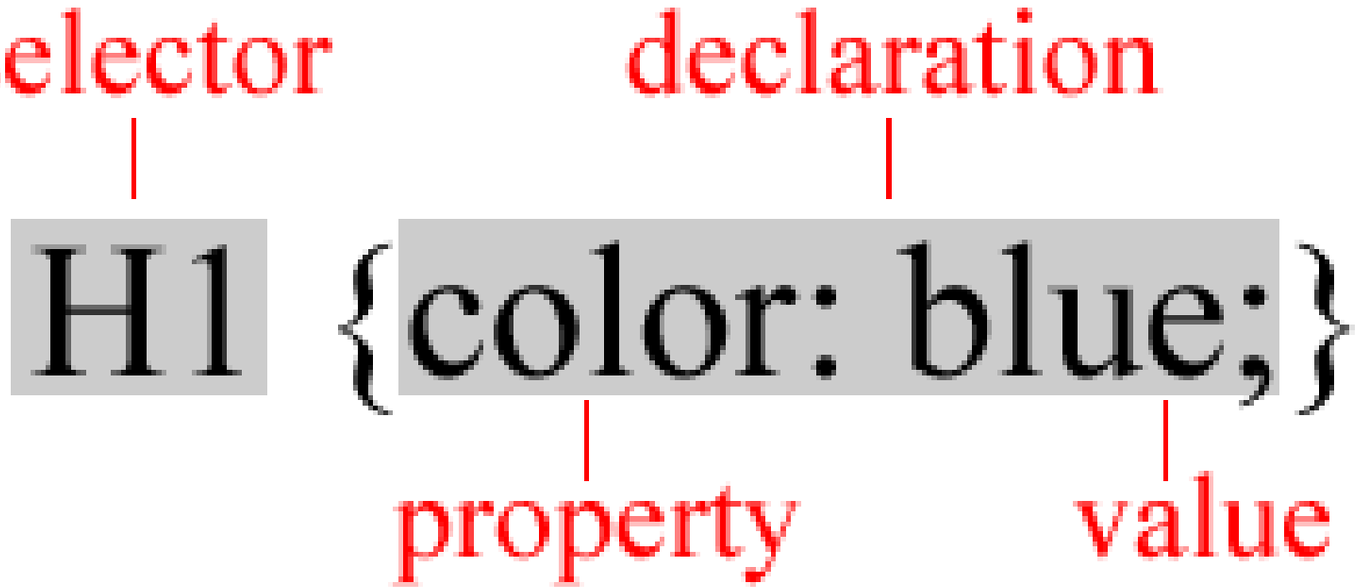
```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-type"
        content="text/html;charset=UTF-8">
        <title>Computer Science </title>
    </head>
    <body>
        <p>hello</p>
        <p>world</p>
    </body>
</html>
```

What are Cascading Style Sheets?

- Separates design elements from structural logic
- Has become the W3C standard for controlling visual presentation of web pages
- You get control and maintain the integrity of your data

Let’s See Some Code

- Rule Structure



Selectors

- **Element Selectors**
  H1 {color: purple;}
  H1, H2, P {color: purple;}

- **Class Selectors**
  <H1 CLASS=“warning”>Danger!</H1>
  <P CLASS=“warning”>Be careful…</P>

What is JavaScript

- Scripting language (object-oriented)
  - Lightweight programming language developed by Netscape
  - Interpreted, not compiled

- Designed to be embedded in browsers
  - Ideal for adding interactivity to HTML pages
  - Detect browser versions
  - Work with info from user via HTML forms
  - Create cookies
  - Validate form data
  - Read and write HTML elements

What can we do with JavaScript?

- To create interactive user interface in a web page (e.g., menu, pop-up alert, windows, etc.)

- Manipulating web content dynamically
  - Change the content and style of an element
  - Replace images on a page without page reload
  - Hide/Show contents

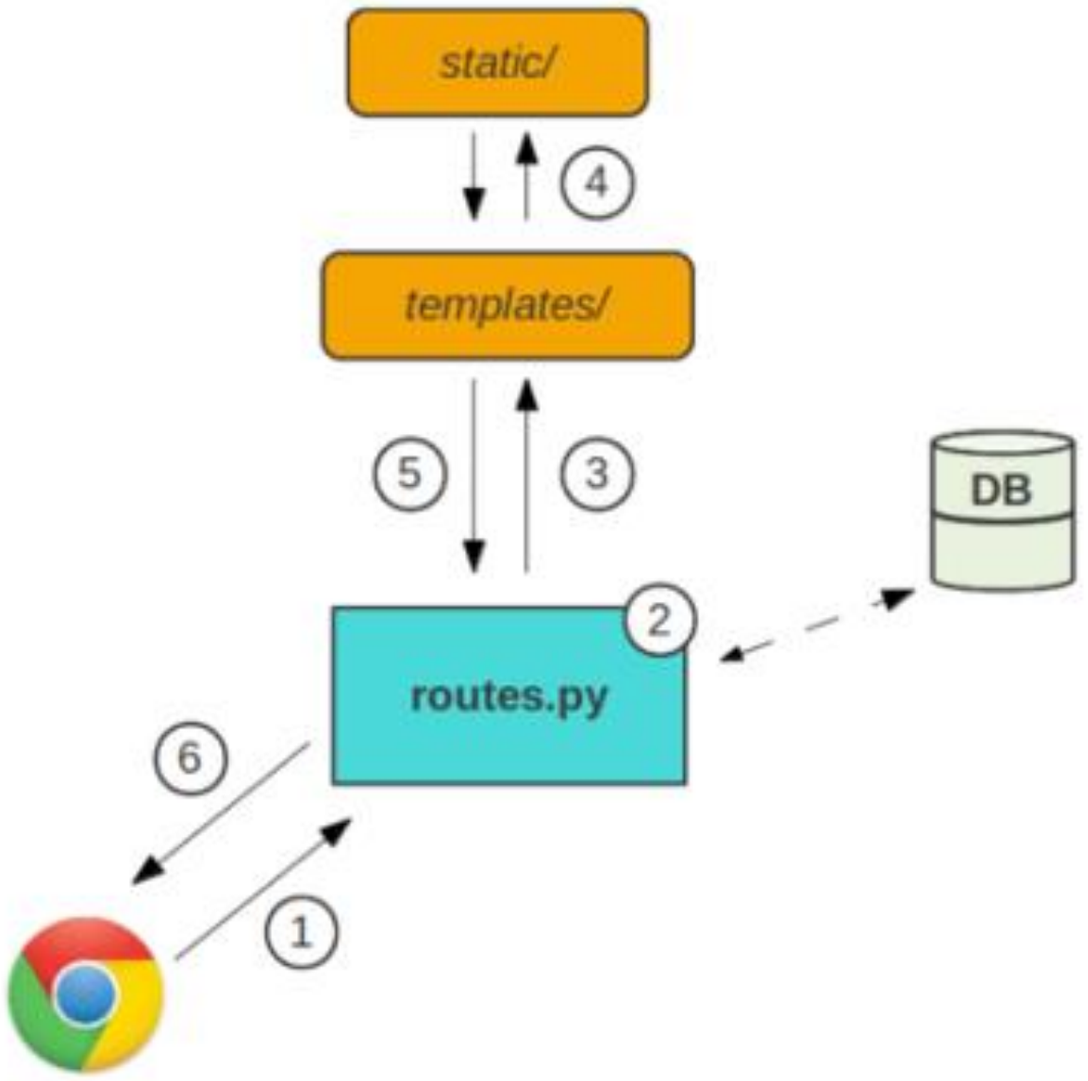
- Generate HTML contents on the fly
- Form validation
- AJAX (e.g. Google complete)
- etc.

Advantages of JavaScript

- **Speed:** JavaScript is executed on the client side.
- **Simplicity: J**avaScript is a relatively easy language
  - The JavaScript language is relatively easy to learn and comprises of syntax that is close to English.
- **Versatility:** JavaScript plays nicely with other languages and can be used in a huge variety of applications.

Flask

1. A user issues a request for a domain's root URL / to go to its home page.
2. routes.py maps the URL / to a Python function.
3. The Python function finds a web template living in the *templates/* folder.
4. A web template will look in the *static/* folder for any images, CSS, or JavaScript files it needs as it renders to HTML
5. Rendered HTML is sent back to routes.py
6. routes.py sends the HTML back to the browser



What is AJAX ?

- Asynchronous Javascript and XML.
- Not a stand-alone language or technology.
- It is a technique that combines a set of known technologies in order to create faster and more user friendly web pages.
- It is a client side technology.

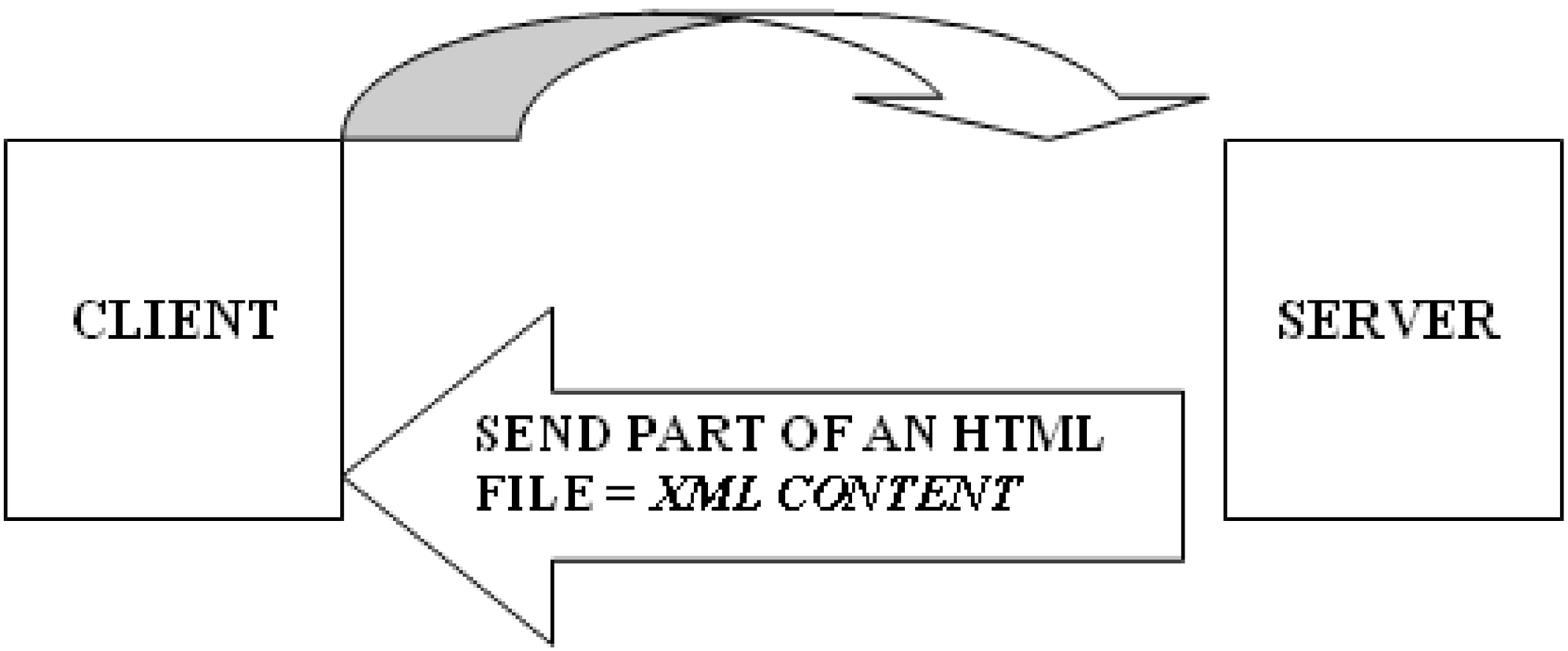
Purpose of AJAX

- Prevents unnecessary reloading of a page.
- When we submit a form, although most of the page remains the same, whole page is reloaded from the server.
- This causes very long waiting times and waste of bandwidth.
- AJAX aims at loading only the necessary information, and making only the necessary changes on the current page without reloading the whole page.

Purpose of AJAX

- Connection between client side script and server side script.
- Better user experience
- More flexibility
- More options

Data Exchange in AJAX



Data Exchange in AJAX

API

- Application Programming Interface
  - A protocol intended to be used as an interface by software components to communicate with each other.

- Source code interface
  - For library or OS
  - Provides services to a program

- At its base, like a header file
  - But, more complete

APIs are Everywhere

- Remote Procedure Calls (RPCs)
- File transfer
- Message delivery
- Java APIs

Characteristics of APIs

- Easy to learn
- Easy to use even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

Classes

```
class name:
    "documentation"
    statements
```

-or-

```
class name(base1, base2, ...):
    ...
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

May also be *class variable* assignments

Using Classes

- To create an instance, simply call the class object:

```
x = Stack()       # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

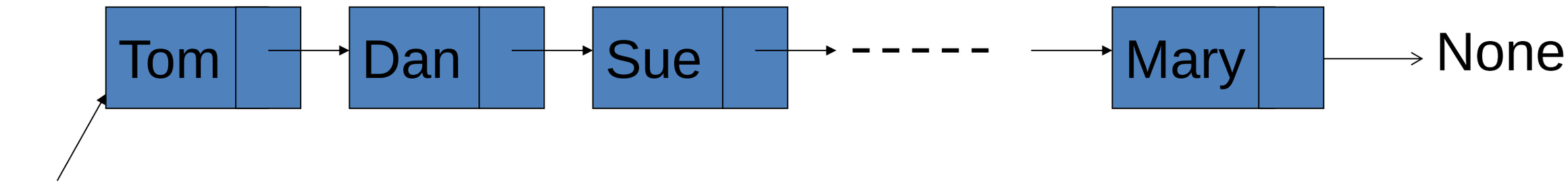
```
x.empty()                 # -> 1
x.push(1)                 # [1]
x.empty()                 # -> 0
x.push("hello")           # [1, "hello"]
x.pop()                   # -> "hello"       # [1]
```

- To inspect instance variables, use dot notation:

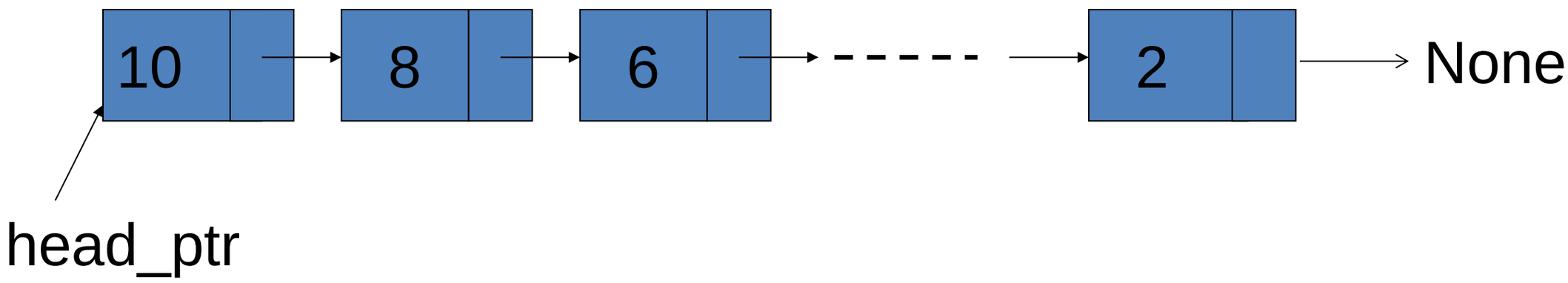
```
x.items           # -> [1]
```

Examples of Linked Lists

- A linked list of strings can represent a waiting line of customers.



- A linked list of integers can represent a stack of numbers.



Node class

- Every Node has a value and a pointer to the next node.
- When a node is first created, its data is set to None and does not point to any node.

```
"""Node class"""
```

```
class Node:
    """By default the data and next are none"""
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

    def __str__(self):
        return str(self.data)
```

LinkedList class

- LinkedList holds a pointer to the first (head) node in the list and an integer that contains the length of the list.
- A linked list is empty when created; thus the "head" node is None and the length is 0.

```
"""LinkedList class"""


class LinkedList:
    """Handler for manipulating list of Node objects"""
    def __init__(self):
        self.length = 0
        self.head = None
```

Operations on Linked Lists

- **Insert** a new item
  - ✓ At the head of the list, or
  - ✓ At the tail of the list, or
  - ✓ Inside the list, in some designated position

- **Delete** an item from the list
  ✓ Search for and locate the item, then remove the item, and finally adjust the surrounding pointers

Insert – at the Tail Example

```
def addnode(self, data):
    """Adds a node to the tail of the List"""
    new_node = Node(data)   # Create a node
    if self.length <= 0:  # if the list is empty
        self.head = new_node
        self.length += 1  # increase the length
    else:
        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node  # Assign the new node
        self.length += 1  # increase the length
```

Delete node at a given Index

```
def removenode(self, index):
    """Removes node at a given index"""
    if self.length <= 0:  # check if the list is empty
        print "The list is empty"
    else:
        prev = None
        current = self.head
        i = 0
        while (current is not None) and (i < index):
            prev = current
            current = current.next
            i += 1
        if prev is None:  # the head element is to be removed
            self.head = current.next
            self.length -= 1  # decrease the length of the list
        else:
            prev.next = current.next
            self.length -= 1  # decrease the length of the list
```

Python Bitwise Operators:

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

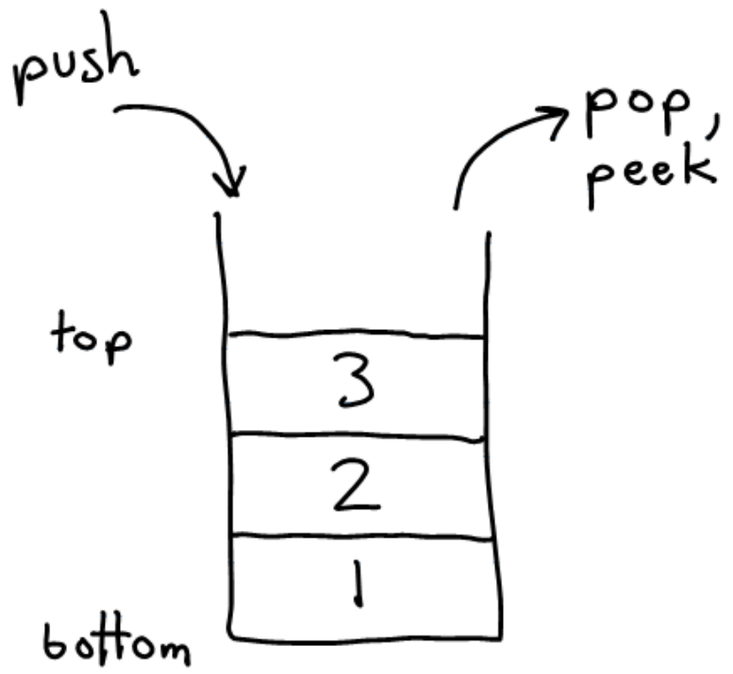
Bitwise Operators Example

```
#!/usr/bin/python
a = 60  # 60 = 0011 1100
b = 13  # 13 = 0000 1101
c = 0
c = a & b  # 12 = 0000 1100
print "Line 1 - Value of c is ", c
c = a | b  # 61 = 0011 1101
print "Line 2 - Value of c is ", c
c = a ^ b  # 49 = 0011 0001
print "Line 3 - Value of c is ", c
c = ~a  # -61 = 1100 0011
print "Line 4 - Value of c is ", c
c = a << 2  # 240 = 1111 0000
print "Line 5 - Value of c is ", c
```

Stack

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **isEmpty**: Check whether the stack is empty.



Stack

- The Stack consists of two classes: the stack, which has a head element and the element, which has a next element.

```
class Element:
    """Element class"""
    def __init__(self, value, next):
        self.value = value
        self.next = next
```

```
class Stack:
    """Stack class"""
    def __init__(self):
        self.items = []
```

Stack: Push and Pop

- Push: To push a new item onto the stack, push appends it onto items list.

```
def push(self, item):
    """Function to push new items on to stack"""
    self.items.append(item)
```

- Pop: To pop an item off the stack, pop removes the item from the items list.

```
def pop(self):
    """Function to pop items off the stack"""
    return self.items.pop()
```

Stack: empty

- Empty: return true if the stack is empty, indicated by checking the items list.

```
def isempty(self):
    """Function to check stack empty"""
    return (self.items == [])
```

Stack: Example

- Main function using the Element and Stack class

```
if __name__ == "__main__":
    S = Stack()
    ELEMENTS = ["first", "second", "third", "fourth"]
    for e in ELEMENTS:
        S.push(e)
    RESULT = []
    while not S.isempty():
        RESULT.append(S.pop())
    assert RESULT == ["fourth", "third", "second", "first"]
```

Queue

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

- basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.

Queue Operations

- The Queue is defined by the following operations:

__init__

✓ Initialize a new empty queue.

insert

✓ Add a new item to the queue.

remove

✓ Remove and return an item from the queue. The item that is returned is the first one that was added.

isempty

✓Check whether the queue is empty.

Queue Class

- The implementation of the Queue is called a **linked queue** because it is made up of linked Node objects.
- A Queue is empty when created ; thus the "head" node is None and the length is 0.

```
"""Queue Class"""

class Queue:
    """Contains the head and the length"""
    def __init__(self):
        self.length = 0
        self.head = None
```

Queue: Insert

```
def insert(self, data):
    """Insert item at the end of list"""
    node = Node(data)  # create a Node
    node.next = None
    if self.head is None:
        # if list is empty the new node goes first
        self.head = node
    else:
        # find the last node in the list
        last = self.head
    while last.next:
        last = last.next
    # append the new node
    last.next = node
    self.length = self.length + 1
```

Queue: remove and isempty

- Removes the first item (head) from the queue and returns the removed item.
- Identical to removing items from head of Linked List.

```
def remove(self):
    """Removes head from list"""
    data = self.head.data
    self.head = self.head.next
    self.length = self.length - 1
    return data
```

- isempty checks if the queue is empty.
- Identical to the LinkedList method.

```
def isempty(self):
    """checks if the Queue is empty"""
    return (self.length == 0)
```

Queue Module

- Useful in threaded programming to exchange information among threads safely.

- The module implements three types of Queues.

✓ FIFO Queue: The first tasks added are the first to be retrieved.

✓ LIFO Queue: The most recently added entry is the first to be retrieved.

✓ Priority Queue: The entries are kept sorted and the lowest valued entry is retrieved first.

Binary Trees

- A binary tree is composed of zero or more nodes in which no node can have more than two children.

- Each node contains:
  - ✓ A value (some sort of data item).
  - ✓ A reference or pointer to a left child (may be null), and
  - ✓ A reference or pointer to a right child (may be null)

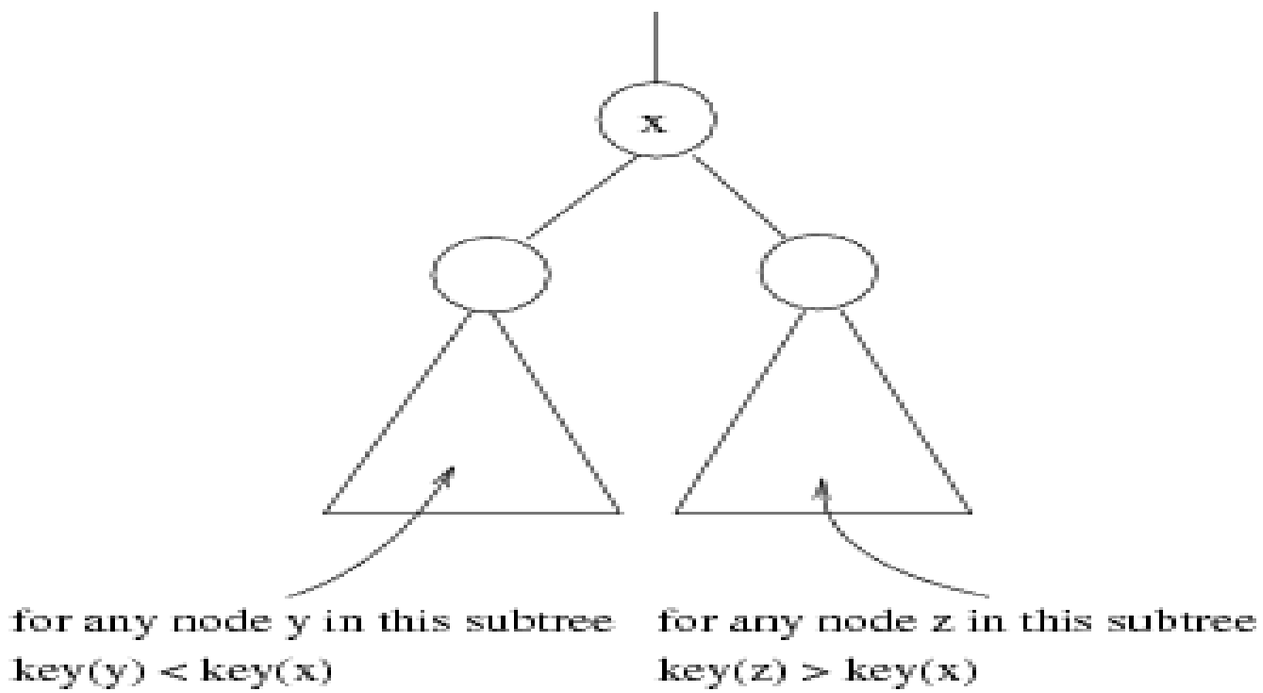
- A binary tree may be empty (contain no nodes).

- If not empty, a binary tree has a root node.
  ✓ Every node in the binary tree is reachable from the root node by a unique path.

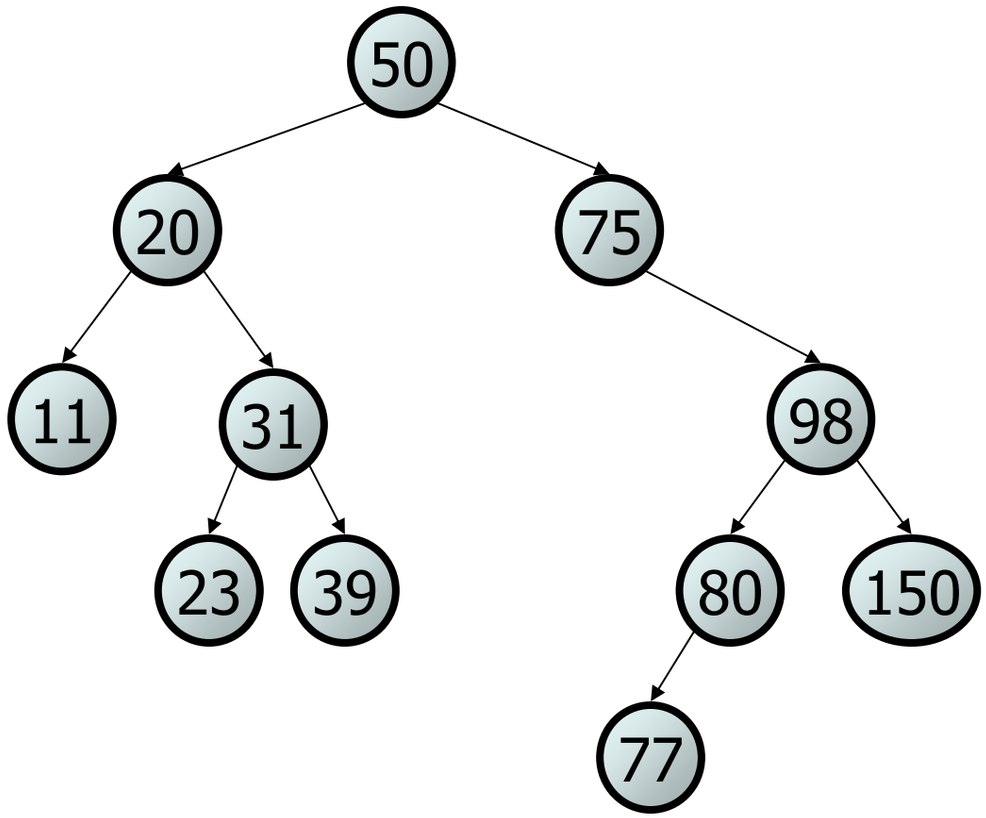
- A node with neither a left child nor a right child is called a leaf.

Binary Search Trees

- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
- Binary search tree property
  - ✓For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



BST: Insert item



- Insert the following items to the binary search tree.

  50
  20
  75
  98
  80
  31
  150
  39
  23
  11
  77

BST: Insert item

- What is the size of the problem?

Ans. Number of nodes in the tree we are examining

- What is the base case(s)?

Ans. The tree is empty

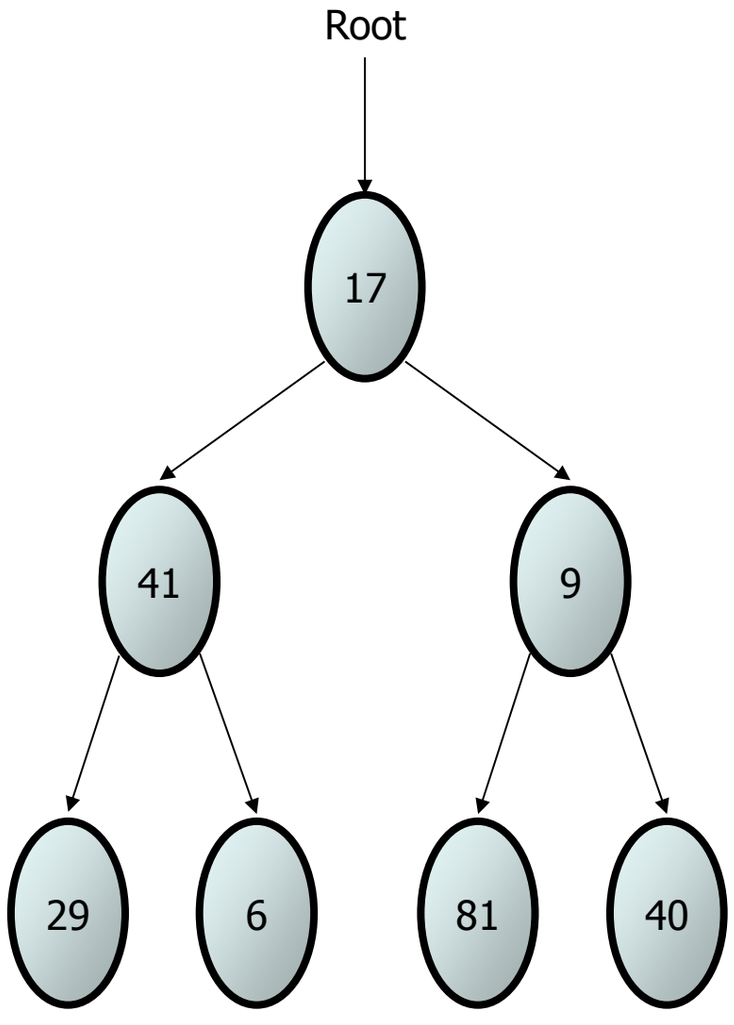
- What is the general case?

Ans. Choose the left or right subtree

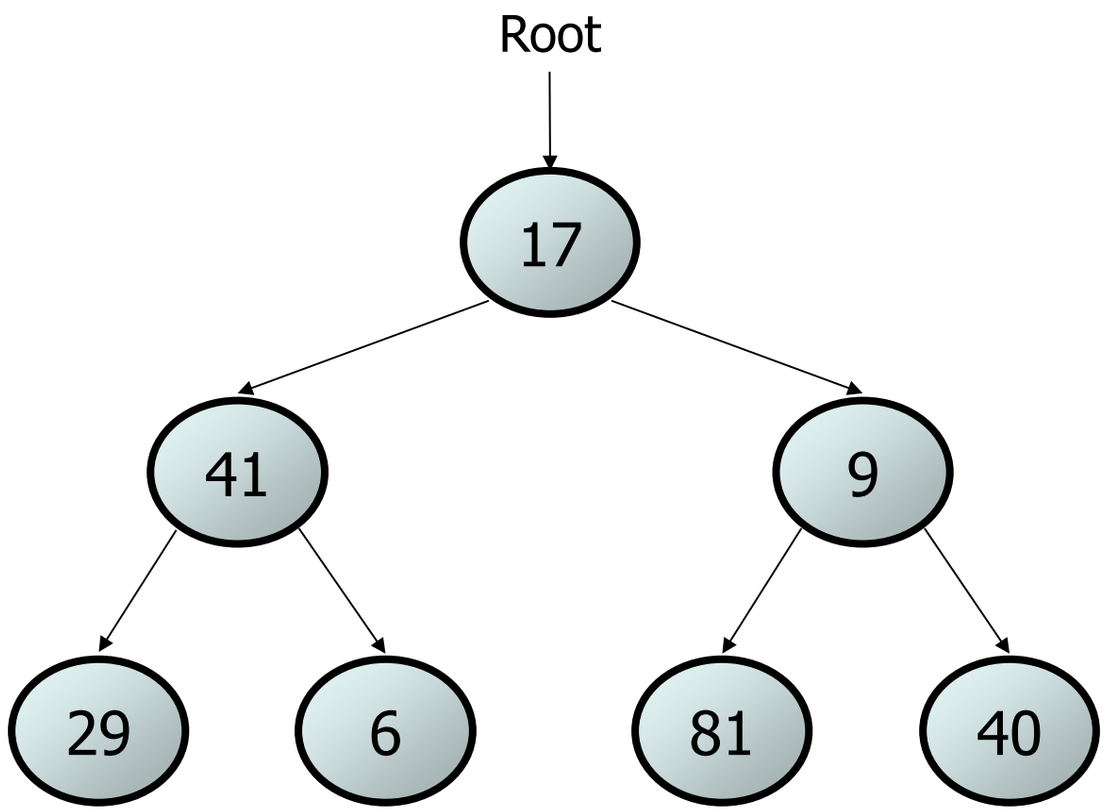
Traversals

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node



Traversal example



- pre-order: 17 41 29 6 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

Huffman Coding

- Huffman codes can be used to compress information
  - Like WinZip – although WinZip doesn't use the Huffman algorithm
  - JPEGs do use Huffman as part of their compression process
- The basic idea is that instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually ½)

Huffman Coding

- Uncompressing works by reading in the file bit by bit
  - Start at the root of the tree
  - If a 0 is read, head left
  - If a 1 is read, head right
  - When a leaf is reached decode that character and start over again at the root of the tree
- Thus, we need to save Huffman table information as a header in the compressed file
  - Doesn’t add a significant amount of size to the file for large files (which are the ones you want to compress anyway)
  - Or we could use a fixed universal set of codes/freqencies

More

Hashing
List Comprehensions
Practice Problems