Ceng 111 – Fall 2015
Week 14ab

ADT and
Object-Oriented Programming

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
Let us open here a parenthesis for dictionaries

Python has **dict** data type for storing a set of key-value pairs:

```python
{key_1:value_1, key_2:value_2, ..., key_N:value_N}
```

Key-value pairs are separated by comma and within curly braces.
Dictionaries: Accessing Elements

```python
person = {‘age’: 20, ‘name’: ‘Veli’}

>>> person = {‘age’: 30, ‘name’: ‘Veli’}
>>> person
{‘age’: 30, ‘name’: ‘Veli’}
>>> person[‘ssn’]
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
KeyError: ‘ssn’
>>> person[‘ssn’] = 124
>>> person
{‘age’: 30, ‘name’: ‘Veli’, ‘ssn’: 124}
```
Dictionaries: Accessing Elements

```python
>>> person
dict({'age': 30, 'name': 'Veli', 'ssn': 124})
>>> person.keys()
dict_keys(['deli', 'age', 'name'])
>>> person.values()
dict_values([30, 30, 'Veli'])
```
Creating Dictionaries

Enclosing a set of *key-value* pairs withing curly braces, like 

```
{'age':10, 'name':'Ali'}.
```

- Supplying as argument to the `dict()` function a nested list or a nested tuple of items where each item has two items. For example, `dict([[20, 30], [30, 40], [40, 50]])` creates the dictionary 

```
{40: 50, 20: 30, 30: 40}.
```

- Using the `input()` function as follows:

```
>>> d = input('Give me a tuple:')
Give me a tuple:{'age':20}
>>> d
{'age': 20}
```
Looping Over Dictionaries

We can get:

- Keys
- Values
- Keys & Values

```
P = {'a': 10, 'b': 20, 'c': 0}

# Take keys only
for k in P: # P.keys() is also possible
    print k + " has value " + str(P[k])

# Take values
for v in P.values():
    print str(v) + " is a value"

# Take keys & values
for k, v in P.items():
    print k + " has value " + str(v)
```

Exercise: Find the key with the lowest value.
Modifying Dictionaries

```python
>>> person = {'age': 20, 'name': 'Ali'}
>>> del person['age']
>>> person
{'name': 'Ali'}
```
Now, let us see how we can represent Trees in Python

Using Lists:

- \([10, [5, [3, [], []], [8, [], []]], [30, [], []]].\)
- \([10, [5, [3, '#', '#'], [8, '#', '#']], [30, '#', '#]],\)
  where the empty branches are marked with '#'.
- \([10, [5, [3], [8]], [30]].\)
Now, let us see how we can represent Trees in Python

- Using dictionaries

```
Tree = {
    'value': 10,
    'left': {
        'value': 5,
        'left': {
            'value': 3,
            'left': {},
            'right': {}},
        'right': {
            'value': 8,
            'left': {},
            'right': {}},
    'right': {
        'value': 30,
        'left': {},
        'right': {}},
}
```
Tree operations

- **datum()**
- **isempty()**
- **left()**
- **right()**
- **createNode()**

```python
# Return the value stored in the node
def datum(T):
    return T[0] # Assume nested list rep.

# Check whether the Tree is empty
def isempty(T):
    return T == [] # Assume nested list rep.

# Get the left branch
def left(T):
    # TODO: Throw exception if the tree is empty
    return T[1] # Assume nested list rep.

# Get the right branch
def right(T):
    # TODO: Throw exception if the tree is empty

# Create a node
def createNode(datum, left=[], right=[]):
    return [datum, left, right]
```
Traversing Trees

1. Pre-order Traversal

```
def preorder_traverse(T):
    if isempty(T):
        return

    print datum(T)
    preorder_traverse(left(T))
    preorder_traverse(right(T))
```

```
10, [5, [3, [], []], [8, [], []]], [30, [], []])
```
2. In-order Traversal

```python
def inorder_traverse(T):
    if isempty(T):
        return
    inorder_traverse(left(T))
    print datum(T)
    inorder_traverse(right(T))
```

3 5 8 10 30
3. **Post-order Traversal**

```python
def postorder_traverse(T):
    if isempty(T):
        return
    postorder_traverse(left(T))
    postorder_traverse(right(T))
    print datum(T)
```

Example:

```
3 8 5 30 10
```
Yet another way to represent trees

Can you think of a one-dimensional representation where from the position of a node (call it n), the position of its parent (call it p) and children (call it c) can be calculated?

- In other words:
  - \( p = f(n) \)
  - \( c = g(n) \)

**Heap:**

- \( p = \text{round}(n/2) \)
- \( c = 2n \)

<table>
<thead>
<tr>
<th>Node</th>
<th>10</th>
<th>5</th>
<th>30</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Binary Search Trees
def insert_node(T, value):
    '''Insert a node with value to the binary search tree'''
    if isempty(T):
        T.extend(createNode(value))
    elif datum(T) == value:  # duplicate
        return
    elif value < datum(T):
        insert_node(left(T), value)
    else:
        insert_node(right(T), value)

# The following can construct the tree on the right
Tree = []
insert_node(Tree, 10)
insert_node(Tree, 30)
insert_node(Tree, 5)
insert_node(Tree, 3)
insert_node(Tree, 8)
Binary Search Trees: An example

```python
1 def search_tree(T, value):
2     '''Search 'value' in binary search tree
3     if isempty(T):
4         return False
5     elif datum(T) == value:
6         return True
7     elif value < datum(T):
8         search_tree(left(T), value)
9     else:
10        search_tree(right(T), value)
```
Exercises

1. Write a function to determine the height of a binary tree.

2. Write a function to determine whether a binary tree is balanced (a tree T is balanced if $|\text{height}(\text{left}(T)) - \text{height}(\text{right}(T))| \leq 1$ for every node in T).

3. Write a function to swap left and right branches of every node.

4. Write a function to count the leaves of a binary tree.
Today

- Finish up trees
- Object-oriented programming
```python
def height(T):
    if isEmpty(T):
        return -1
    l_height = height(left(T))
    r_height = height(right(T))
    max = l_height if l_height > r_height else r_height
    return max + 1

def isBalanced(T):
    if isEmpty(T):
        return True
    return abs(height(left(T)) - height(right(T))) <= 1
    and isBalanced(left(T)) and isBalanced(right(T))
```
ALGORITHM

ACTS ON

STRUCTURED DATA

PROGRAM IN HIGH LEVEL LANGUAGE

typedef
struct element
  { char *key;
    int value;
    struct *element[next];
  } element, *ep;

ep *Bucket_entry;

declare KEY(p) (p->key)
declare VALUE(p) (p->value)
declare NEXT(p) (p->next)

void create_Bucket(int size)
{
  Bucket_entry = malloc(size*sizeof(ep));
  if (!Bucket_entry)
    error("Cannot allocate bucket");
}

insert_element(int value)
Object-Oriented Programming (OOP)

As the name suggests, everything is object centered.

- The problem and the solution are modeled based on objects.
- The data and the algorithm are combined in objects.
What is an object?

An object is an entity which has a state and a set of behaviors that, when executed, change the state of the entity or the environment.

- A car object:
  - State:
    - Position, Speed, Gear State, Brake State, Wheel State
  - Behaviors:
    - Rotate, Accelerate, Brake
In Programming..

- Objects are compound types that
  - hold a set of variables to represent the state
  - implement a set of methods that allow the object to change its state and the environment’s state.
Why do we need/have OOP?

Consider a problem of drawing/manipulating geometric objects:

- Points in 2D Cartesian space
- Lines: Made from two points
- Triangle: Made from three lines or three points
- Square/rectangle: Made from four lines or two points
- Circle: A point and a radius
- Polygon: A collection of lines / points.
Why do we need/have OOP? (cont’d)

When we have to draw in 3D:
- Points in 3D Cartesian space
- 3D Lines: Made from two 3D points
- 3D Triangle: Made from three 3D lines or three 3D points
- 3D Square/3D rectangle: Made from four 3D lines or two 3D points
- 3D Circle: A 3D point and a radius
- Prism: A collection of 3D triangles & rectangles & parallelograms.
Why do we need/have OOP? (cont’d)

Now let us look at the representations of the objects

- **Point:**
  - X
  - Y

- **Line:**
  - StartingPoint
  - EndingPoint

- **Triangle:**
  - Point1
  - Point2
  - Point3

- **3D Point:**
  - X
  - Y
  - Z

- **3D Line:**
  - Starting3DPoint
  - Ending3DPoint

- **3D Triangle:**
  - 3DPoint1
  - 3DPoint2
  - 3DPoint3

Do you notice how overlapping they are?
Why do we need/have OOP? (cont’d)

Now let us look at what we need to perform on these geometric objects:

- **Point:**
  - Compute distance to the origin or to another point
  - Change the scale
  - Draw on screen

- **Line:**
  - Compute length, orientation
  - Project on another line
  - Compute distance to a point
  - Draw on screen

- **3D Point:**
  - Compute distance to the origin or to another 3D point
  - Change the scale
  - Draw on screen

- **3D Line:**
  - Compute length, orientation
  - Project on another line
  - Compute distance to a point
  - Draw on screen

Do you notice how overlapping they are?
Why do we need/have OOP? (cont’d)

Now let us look at what we need to perform on these geometric objects:

- **Triangle:**
  - Compute area
  - Check whether a point is in the triangle
  - Check whether a line intersects the triangle
  - Draw on screen

- **Square:**
  - Compute area
  - Check whether a point is in the triangle
  - Check whether a line intersects the triangle
  - Draw on screen

- **3D Triangle:**
  - Compute area
  - Check whether a point is in the triangle
  - Check whether a line intersects the triangle
  - Draw on screen

- **3D Square:**
  - Compute area
  - Check whether a point is in the triangle
  - Check whether a line intersects the triangle
  - Draw on screen

Do you notice how overlapping they are?
Why do we need/have OOP? (cont’d)

- There is so much dependency between the representation of these objects.
  - it is beneficial to model these relationships for:
    - Modularity
    - Encapsulation

- Now, let us assume that we used a non-object centered paradigm for the example problem
  - Let us change the Cartesian representation of points to the Polar representation:
    - \( X, Y \rightarrow R, \Theta \)
    - \( X, Y, Z \rightarrow R, \Theta, \phi \)
  - Such a small change would cause a lot of changes in a non-object centered representation.
**Example**

**Point Class**: Definition of what a Point object is.

```python
class Point:
    pass

>>> p1 = Point()
>>> p1.x = 10
>>> p1.y = 3.4
>>> print p1
<__main__.Point instance at 0x00000000028BB108>
```

- **p1**: An instance of the Point class. `p1` is a Point object.
- **x, y**: Member variables of `p1` that represent `p1`'s current state.
Example (2\textsuperscript{nd} version)

**Point Class**: Definition of what a Point object is.

```python
class Point:
    def __init__(self, X=0, Y=0):
        self.X = X
        self.Y = Y

    def printPoint(self):

    def distance(self):
        x = self.X
        y = self.Y
        return sqrt(x*x + y*y)
```

- Initialization method that sets the initial values for the member variables.
- `self`: current object/instance.
- `X, Y` are member variables.
- `printPoint` and `distance` are member functions.
Example (2\textsuperscript{nd} version)

class Point:
    def __init__(self, X=0, Y=0):
        self.X = X
        self.Y = Y

    def printPoint(self):

    def distance(self):
        x = self.X
        y = self.Y
        return sqrt(x*x + y*y)

>>> P1 = Point(3, 4.5)
>>> P1.printPoint()
X: 3 Y: 4.5
What does OOP provide?

- **Data abstraction:**
  - Compare the definition of Point below with the earlier version.
  - This new definition uses polar coordinates.
  - However, the user of the Point class is not effected by this change at all.

```python
class Point:
    def __init__(self, X=0, Y=0):
        self.R = sqrt(X*X + Y*Y)
        self.Theta = atan(Y/X)

    def printPoint(self):
        print "X: ", self.R * cos(Theta),\
            "Y: ", self.R * sin(Theta)

    def distance(self):
        return R
```

- Data abstraction is the act of hiding the implementation details.
- In our example, the Point class hides in which coordinate space the point is represented.
What does OOP provide?

- **Encapsulation**: Putting data and actions together.
  - The outside world does not need to know how an object implements its functions.
  - An object just provides an interface and the world needs to know what that interface has to offer.

What does OOP provide?

- **Data abstraction**: Hiding implementation/representation details

What does OOP provide?

- **Modularity**: Dividing abstraction into discrete units.

  - A cat has:
    - a head
    - a body
    - 4 legs
    - a tail
    - ...

What does OOP provide?

- **Inheritance:**
  - A class inherits some variables and functions from another one.
  - Square class inherits \( x \) and \( \text{draw()} \) from the Shape class.
  - Shape: Parent class
  - Square: Child class
What does OOP provide?

- **Polymorphism:**
  - The ability of a child class to behave and appear like its parent.

```python
class Animal:
    def __init__(self, name):
        self.name = name
    def talk(self):
        pass  # Overloaded by Child Classes

class Cat(Animal):
    def talk(self):
        return 'Meow'

class Dog(Animal):
    def talk(self):
        return 'Woof'

class Duck(Animal):
    def talk(self):
        return 'Quack'
```
Calling a method of the parent class

- `<parent-class-name>.method(self)`
Defining Classes in Python

```python
class Person:
    pass
```

- **class**: The keyword that tells Python that a class is being declared/defined.
- **Person**: The name of the new class.
- **pass**: The `Person` class does not define anything yet.
  - The `Person` type does not have any member variables or functions yet.
Defining Classes in Python

```python
class Person:
    pass

>>> Sinan = Person()
>>> Sinan.name = "Sinan Kalkan"
>>> Sinan.age = 20
>>> print Sinan
<_main_.Person instance at 0xb7cfc60c>  
>>> print Sinan.age, Sinan.name
20 Sinan Kalkan

>>> del Sinan.age
>>> print Sinan.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Person instance has no attribute 'age'
```

- Objects are mutable. Hence:
  - You can add new member variables.
  - You can delete member variables that you have added.
Defining Classes in Python

```
class Person:
    # Member variables:
    age = 0
    name = ""

>>> Sinan = Person()
>>> print Sinan.age
0
>>> Sinan.name = "Sinan Kalkan"
>>> Sinan.age = 20
>>> print Sinan.age, Sinan.name
20 Sinan Kalkan
```

- "age" and "name" are by default the member variables.
- Each instance of the Person class will have these two member variables.
Initialization of Objects in Python

```python
class Person:
    def __init__(self, age=0, name="<Noname>"):  
        self.age = age
        self.name = name

>>> Sinan = Person()
>>> Sinan.age
0
>>> Sinan.name
'<Noname>'
```

- **__init__()**
  - This is called the constructor.
  - Called whenever an instance of the class is created.
- In this case, "age" and "name" are initialized.

In the diagram:
- **Person**
  - age = 0
  - name = "<Noname>"
- **Sinan**
Defining Member functions in Python

```python
class Person:
    def __init__(self, age=0, name="<Noname>"):
        self.age = age
        self.name = name
    def greet(self):
        print "I am ", self.name, ". Nice to meet you."

>>> Sinan = Person()
>>> Sinan.greet()
I am <Noname> . Nice to meet you.
```

- **Person**
  - age = 0
  - name = "<Noname>"
  - greet()
Defining Member functions in Python

- Like adding new member variables to an object, you can add member functions!
- However, these functions can’t access member variables.

```python
class Person:
    def __init__(self, age=0, name="<Noname>"):  
        self.age = age
        self.name = name

    def f():
        print "I am a function which cannot access"\  
            " member variables"

>>> Sinan = Person()
>>> Sinan.greet = f
>>> Sinan.greet()
I am a function which cannot access member variables
```
Defining Member functions in Python

```python
class Person:
    def __init__(self, age=0, name="<Noname>"):  
        self.age = age
        self.name = name

    def f(self):
        print("Name is:", self.name)

>>> Sinan = Person()
>>> Sinan.printName = f
>>> Sinan.printName()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 1 argument (0 given)

>>> Person.printName = f
>>> Sinan = Person()
>>> Sinan.printName()
Name is: <Noname>
```

- Notice the difference between two cases
- In the first case, we try to add a member function to an object.
- In the second case, we try to add a member function to the class itself.
Data hiding.

```python
class Person:
    def __init__(self, age=0, name="<Noname>"):  
        self.age = age  
        self.name = name

    # Age modifier & accessor
    def setAge(self, age):
        self.age = age
    def getAge(self):
        return age

    # Name modifier & accessor
    def setName(self, name):
        self.name = name
    def getName(self):
        return name

def printPerson(self):
    print "Age=", self.age, "Name=", self.name
```

- Some PLs restrict direct access to member variables or functions; Python don’t have a direct facility for that.
- Data of objects should not be accessed or modified directly!
- Modifier and accessor functions should be defined for that.
Data hiding.

In Python, if you want to hide your data, put an underscore before its name.

However, this is just a convention!

- Python does not protect any member variables!
Inheritance

- Student class inherits member variables & functions from the Person class.

```python
class Person:
    _age = 0
    _name = "NN"

def printPerson(self):
    print "Name:" , self._name, \ 
      "Age:" , self._age

class Student(Person):
    def __init__(self, year=0, grades=[]):
        self._year = year
        self._grades = grades
    def printStudent(self):
        print "Name:" , self._name, \ 
          "Age:" , self._age, \ 
          "Year:" , self._year, \ 
          "Grades:" , self._grades

>>> Sinan = Student()
>>> Sinan._name =
>>> Sinan._name = "Ali"
>>> Sinan.printStudent()
Name: Ali Age: 0 Year: 0 Grades: []
>>> Sinan.printPerson()
Name: Ali Age: 0
Some Remarks

- If a member variable and a member function has the same name, the member variable override the member function.
- In Python, you cannot enforce data hiding.
- The users of a class should be careful about using member variables since they can be deleted or altered easily.
- To arrange proper deletion of an object, you can define a "__del__(self)" function ➔ also called, the destructor.
- The word `self` can be replaced by any other name. However, since "self" is the convention and some code browsers rely on the keyword "self", it is ideal to use "self" all the time.
Equivalence & Copying & Operator Overloading

Python checks equality by checking whether two variables point to the same object.

class Person:
    _age = 0
    _name = "NN"

    def printPerson(self):
        print "Name:", self._name,
        "Age:", self._age

>>> p1 = Person()
>>> p2 = Person()
>>> p1 == p2
False
Equivalence & Copying & Operator Overloading

```
class Person:
    _age = 0
    _name = "NN"

def printPerson(self):
    print "Name:" , self._name, \
        "Age:" , self._age

def __eq__(self, other):
    return (self._name == other._name) &\n        (self._age == other._age)
```

Equivalence can be checked content-wise by overloading the ‘==‘ operator.

In Python, the operators actually call pre-defined member functions:

- < → __lt__
- > → __gt__
- != → __ne__
- == → __eq__
- + → __add__
- * → __mul__
- ...

```
Equivalence & Copying & Operator Overloading

- Simple assignment is affected by aliasing.
- You can use “copy” module to copy the object by value.
- However, if the object has member variables pointing to mutable objects, the member variables are affected by aliasing!
- Use deepcopy() to copy an object without being affected by aliasing at all.

```python
def Person():
    pass
>>> p1 = Person()
>>> p2 = p1  # Affected by aliasing

>>> import copy
>>> p2 = copy.copy(p1)  # Shallow copying

>>> p2 = copy.deepcopy(p1)  # Deep copying
```
```python
class Node:
    _value = ""
    def __init__(self, value=""):  
        self._left = None  
        self._right = None  
        self._value = value

# Left modifier & accessor
def getLeft(self):
    return self._left

def setLeft(self, leftNode):
    self._left = leftNode
# Right modifier & accessor
def getRight(self):
    return self._right

def setRight(self, rightNode):
    self._right = rightNode
# Value modifier & accessor
def getValue(self):
    return self._value

def setValue(self, value):
    self._value = value
```

Using OOP for Trees
Using OOP for Trees

```python
def preorder_traverse(Tree):
    print(Tree.getValue(),
    if Tree.getLeft() != None:
        preorder_traverse(Tree.getLeft())
    if Tree.getRight() != None:
        preorder_traverse(Tree.getRight())

Root = Node(10)
Root.setLeft(Node(5))
Root.setRight(Node(15))
Root.getLeft().setRight(Node(8))
#   10
# / \
# 5   15
#   :
#   8
preorder_traverse(Root) # Prints 10 5 8 15
```
Using OOP for Stacks

class Stack:
    def __init__(self, items=[]):
        self._items = items
    def push(self, item):
        self._items.append(item)
    def pop(self):
        if self.isempty():
            raise ValueError('pop() operation on empty stack')
        self._items.pop()
    def top(self):
        if self.isempty():
            raise ValueError('top() operation on empty stack')
        return self._items[-1]
    def isempty(self):
        return self._items == []

>>> st = Stack()
>>> st.push("A")
>>> st.push("10")
>>> st.push(20)
>>> print st.top()
20
>>> st.pop()
>>> print st.top()
10
>>>