Ceng 111 – Fall 2020
Week 14

ADT, OOP

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.

Today

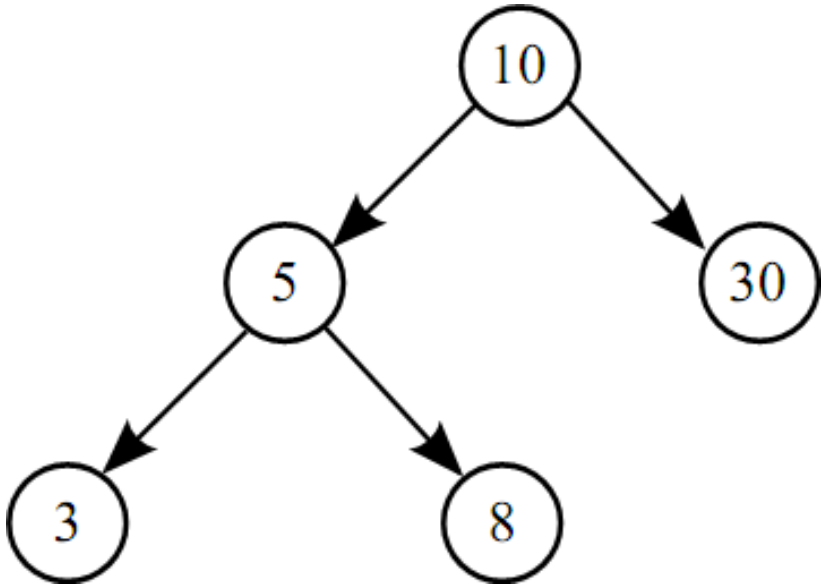
- Finalize ADT with a flat-representation of trees
- Object-oriented programming

Administrative Notes

- Live sessions
  - Tue 13:40 Session
  - Wed 10:40 Session
- Social session
- The labs
- Office hours: Tue 10:30
- Lab Exam 3: 16 January
- Final: 30 January 13:30

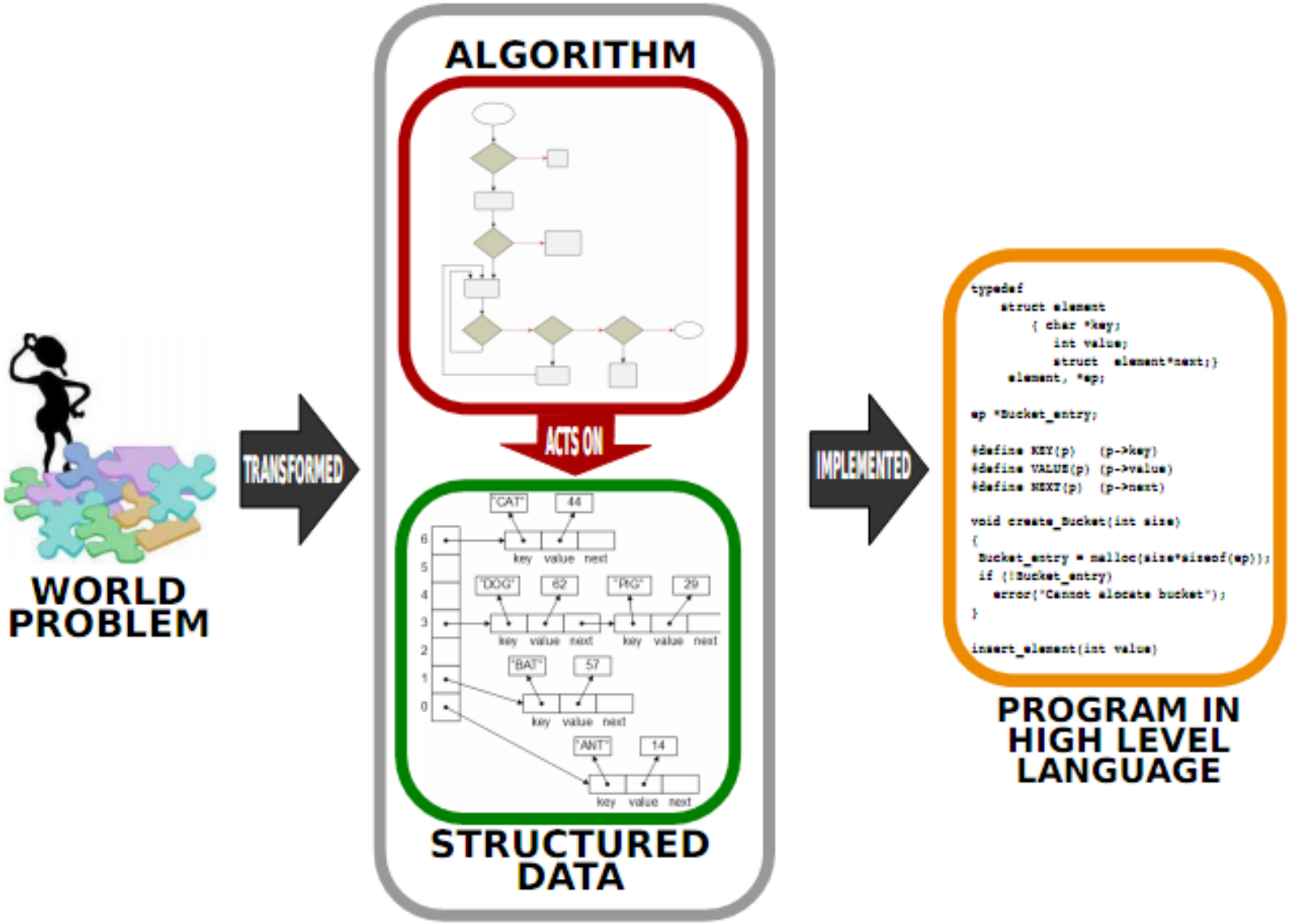
Yet another way to represent trees

- 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) and c = g(n).
- Heap:
  - p = floor(n/2)
  - c = 2n



Node	10	5	30	3	8
Index	1	2	3	4	5

OBJECT-ORIENTED PROGRAMMING


WORLD PROBLEM
TRANSFORMED
ALGORITHM
ACTS ON
STRUCTURED DATA
IMPLEMENTED
typedef
struct element
{ char *key;
int value;
struct element*next;}
element, *ep;
ep *Bucket_entry;
#define KEY(p) (p->key)
#define VALUE(p) (p->value)
#define NEXT(p) (p->next)
void create_Bucket(int size)
{
Bucket_entry = malloc(size*sizeof(ep));
if (!Bucket_entry)
error("Cannot alocate bucket");
}
insert_element(int value)
PROGRAM IN HIGH LEVEL LANGUAGE

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 → R, Θ
    - X, Y, Z → R, Θ, ϕ
  - 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.

```
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 (2nd version)

Point Class: Definition of what a Point object is.

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

    def printPoint(self):
        print "X: ", self.X, "Y: ", self.Y

    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.**

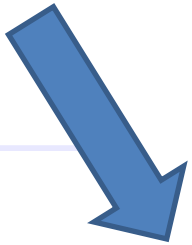
self: **current object/instance.**

X, Y are **member variables.**

printPoint and distance are member functions.

Example (2nd version)

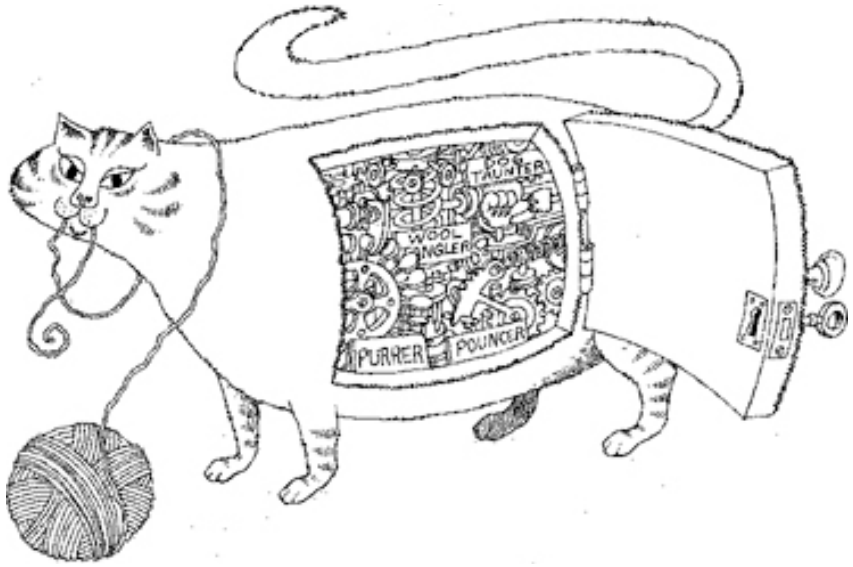

```
class Point:
    def __init__(self, X=0, Y=0):
        self.X = X
        self.Y = Y

    def printPoint(self):
        print "X: ", self.X, "Y: ", self.Y

    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?

- 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.

Figure: G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, K. A. Houston, Object-Oriented Analysis and Design with Applications (3rd Edition), 2007.

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.

```
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?

- Data abstraction: Hiding implementation/representation details

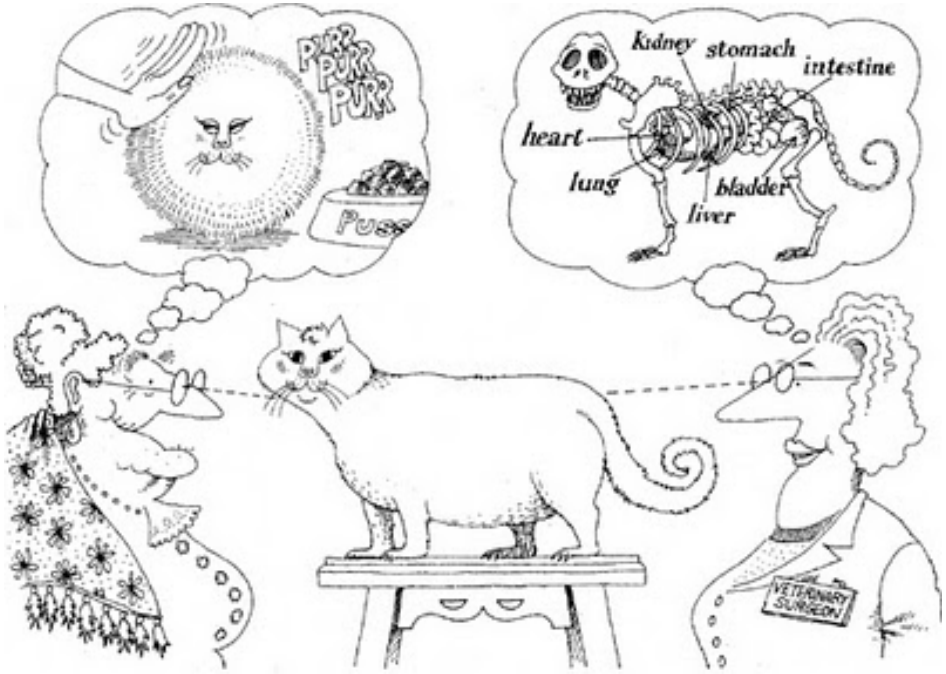


Figure: G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, K. A. Houston, Object-Oriented Analysis and Design with Applications (3rd Edition), 2007.

What does OOP provide?

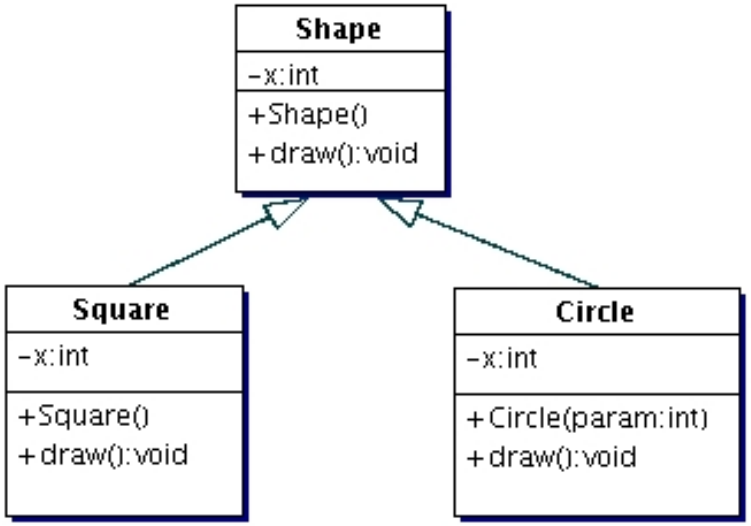
- Modularity: Dividing abstraction into discrete units.

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

Figure: G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, K. A. Houston, Object-Oriented Analysis and Design with Applications (3rd Edition), 2007.

What does OOP provide?

- Inheritance:
  - A class inherits some variables and functions from another one.
  - Square class inherits x and 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.

```
class Animal:
    def __init__(self, name): #Constructor
        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)

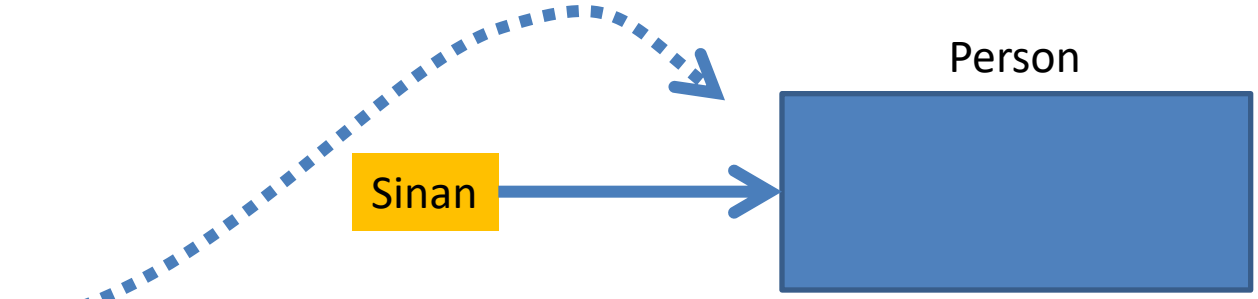
Defining Classes in Python


```
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


```
class Person:
    pass
```



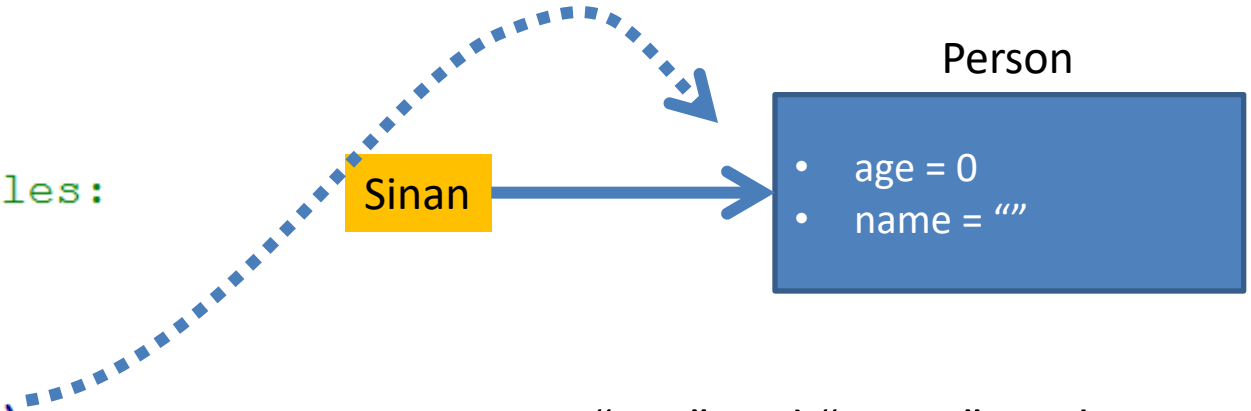

```
>>> Sinan = Person()
>>> Sinan.name = "Sinan Kalkan"
>>> Sinan.age = 20
>>> print Sinan
<__main__.Person instance at 0xb7cfc60c>
>>> print Sinan.age, Sinan.name
20 Sinan Kalkan
```

```
>>> 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
```

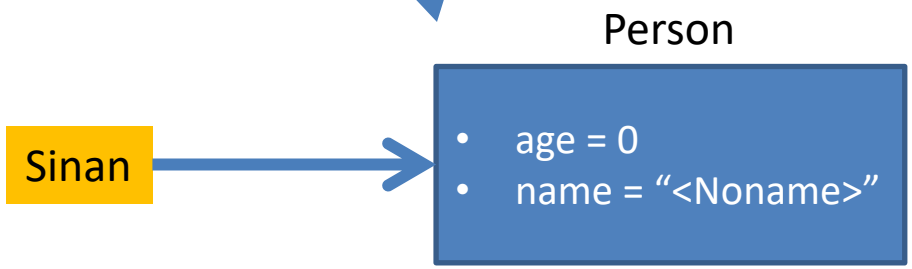


- “age” and “name” are by default the member variables.
- Each instance of the **Person** class will have these two member variables.

Initialization of Objects in Python


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

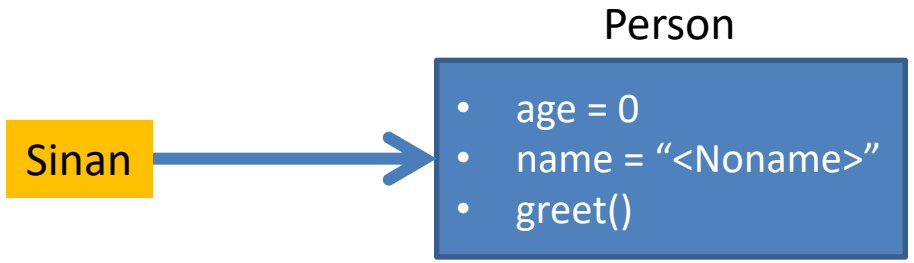
```
>>> Sinan = Person()
>>> Sinan.age
0
>>> Sinan.name
'<Noname>'
```



- _ _init()_ _
  - This is called the constructor.
  - Called whenever an instance of the class is created.
- In this case, “age” and “name” are initialized.

Defining Member functions in Python

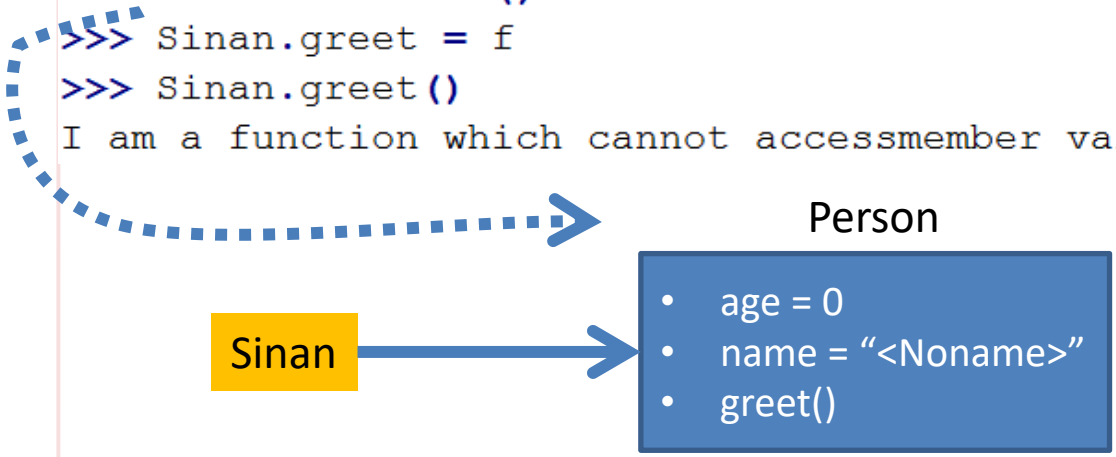

```
class Person:
    def __init__(self, age=0, name="<Noname>"):
        self.age = age
        self.name = name
    def greet(self):
        print "I am ", self.name, ". Nice to meet you."

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



Defining Member functions in 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"
```

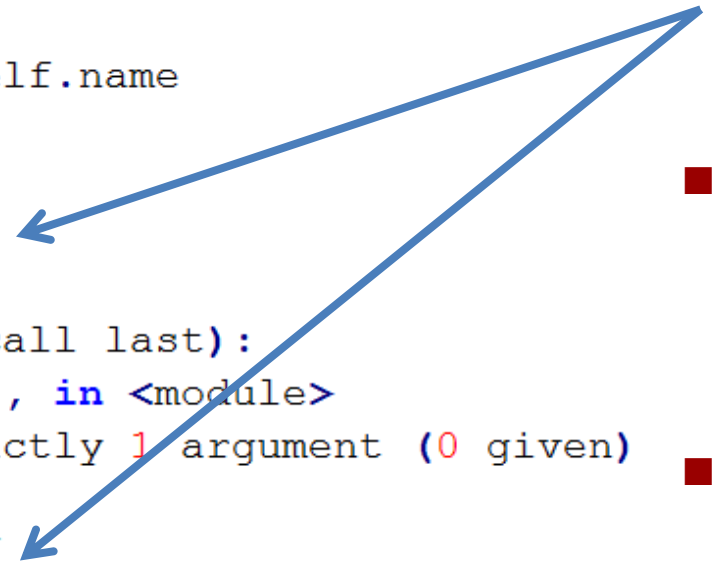
```
>>> Sinan = Person()
>>> Sinan.greet = f
>>> Sinan.greet()
I am a function which cannot accessmember variables
```



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

Defining Member functions in 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.

```
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.

```
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 self._age

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

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

- 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!

Data hiding

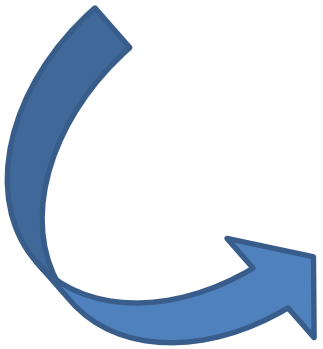
- A single leading underscore indicates weak "internal use"
- Two leading underscores provide some level of privacy.
  - However, this is not strong enough to protect the data from outside

```
1  class deneme:
2      ali = 10
3      _ali = 10
4      __ali = 10
5
6  d = deneme()
7  print d.ali
8  print d._ali
9  print d.__ali # Gives error
10 print d._deneme__ali # Works fine
```

Inheritance

```
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
```

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



```
>>> 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

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

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

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1 == p2
False
```

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

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) &\
            (self._age == other._age)
```

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1 == p2
True
>>> p1._name = "a"
>>> p1 == p2
False
```

- 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


```
>>> p1 = Person()
>>> p2 = p1 # Affected by aliasing

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

>>> p2 = copy.deepcopy(p1) # Deep copying
```

- 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.

Using OOP for Stacks

```
1  class Stack:
2      def __init__(self, items=[]):
3          self._items = items
4      def push(self, item):
5          self._items.append(item)
6      def pop(self):
7          if self.isempty():
8              raise ValueError('pop() operation on empty stack')
9          self._items.pop()
10     def top(self):
11         if self.isempty():
12             raise ValueError('top() operation on empty stack')
13         return self._items[-1]
14     def isempty(self):
15         return self._items == []
```

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