Constructors

There is one more element to a typical class definition -- the constructor method.

Consider the following program:

```
class Person:
        def setName( self, myName):
                self.name = myName

def main():
        x = Person()
        x.setName( "bob" )
        print( x.name )
        y = Person()
        print( y.name )
```

This crashes on the call to print y.name because variable name for object y hasn't been created; it is only created when the setName( ) method is called.

This is unacceptable; we don't want the instance variables of an object to exist only when methods are called in the right order.

Instead of this, almost all classes use a "constructor method". The job of a constructor is to give initial values to each of the instance variables of the class.

In Python, the constructor method has the (weird) name
_ _ init_ _(self, ...)

This is the function that is called when we construct new object (remember that is done by using the class name as a function).

The constructor method is allowed to take arguments in addition to self. For example, a constructor method for a class Person that has instance variables name and age might be

```
def _ _init_ _ (self, myName ):
        self.name = myName
        self.age = 0
```

The call that constructs a new object needs to give a value for each parameter of _ _init_ _( ) other than self.

For example with the constructor above we would create a new Person with

```
        x = Person( "bob" )
```

Consider ProgramA. What will it print?

```
class A:
    def set(n):
        value = n

    def get():
        return value

def main():
    x = A()
    x.set(1)
    print( x.get() )

main()
```

A. Nothing
B. It gets an error message
C. 1

Consider ProgramB. What will it print?

```
class B:
    def __init__(self, n):
        self.value = n

    def get(self):
        return self.value

def main():
    x = B(1)
    print( x.get() )

main()
```

A. Nothing
B. It gets an error message
C. 1