# CSC 271 - Software I: Utilities and Internals

## Lecture #11 – Objects and Classes in Python

# Recap

- Python is a general purpose interpreted language using indentation as block delineation.
- Variables are implicitly defined, dynamically typed and data types are dynamically bound to variables.
- The language is case sensitive.
- Atomic data types include integer, floating point number, Boolean and string.
- Composite data types include list, set and dictionary.

## Classes and Objects

- We know from our classes in object-oriented programming that an object is a tangible instance of a class.
- Objects encapsulate data with operations.
- Data is represented by attributes and operations are implemented as methods.

## Classes and Objects

- In pure OO languages, classes can specialize super classes (or, alternatively phrases, super classes generalize sub classes).
- For example:
  - It is fair to say that all squares are rectangles, and that tall rectangles are shapes.
  - Rectangle is a generalization of square and shape is a generalization of rectangle.
  - Circle is also sub-class of shape, but along a different path.

# Classes

- Although it is completely possible to ignore it, Python is a true object-oriented language.
- By convention, classes are named with an initial uppercase letter.
- Methods are functions and always take a reference to self as their first parameter. self is assigned a value when the class is instantiated.
- In this example, the Card class contains two methods (func1 and func2).

# Card.py

```
#!/usr/bin/python

class Card:
        """ A simple example class """
        def func1(self):
                """ Function 1 """

        def func2(self):
                """ Function 2 """

if __name__ == "__main__":
        card = Card()
```

# Constructors

- If a class contains instance variables, they are defined and initialized within the constructor.
- The constructor method is called **`__init__`** and, in addition to self, may contain additional parameters.

# **card.py**

```
#!/usr/bin/python

class Card:
        """ A simple example class """
        def __init__(self, suit, value):
                """ the constructor method sets
up instances by
                initializing initial vale to
instance variables."""
                self.suit = suit
                self.value = value

if __name__ == "__main__":
        card = Card("hearts", 2)
```

# Class Variables

- Class variables are variables that are shared by all instances of the class.
  - Other languages refer to them as static variables.
- All variables defined in a class, but outside a function, are class variables.

## `card2.py`

```
#!/usr/bin/python

class Card:
        """ A simple example class """

        # Class variables
        suits = ["hearts", "clubs", "spades",\
                  "diamonds"]
        jack = 11
        queen = 12
        king =13
        ace = 14
```

```
      """ the constructor method sets up
      instances by initializing initial
      value to instance variables."""
                self.suit = suit
                self.value = value

if __name__ == "__main__":
      card = Card("hearts", Card.ace)
```

# Visibility of variables

- Python does not have built-in for visibility modifiers.
    - As such, it is not possible to define a variable as private.
- However, by convention, all names that start with single underscore (_) should be considered private.
- This applies to functions, as well as variables.

# card3.py

```
!/usr/bin/python

class Card:
        """ A simple example class """

        # Class variables
        suits = ["hearts", "clubs", "spades", \
                "diamonds"]
        jack = 11
        queen = 12
        king =13
        ace = 14
```

# card3.py

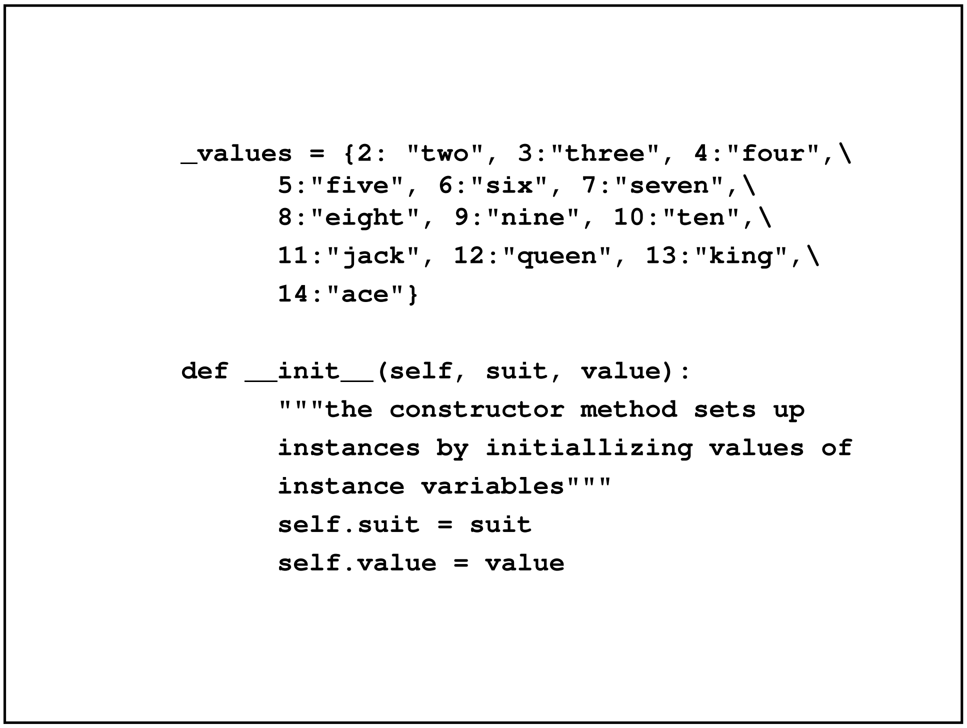
```
            _values = {2: "two", 3: "three",
                  4: "four", 5:"five", 6: "six",
                  7: "seven", 8: "eight",
                  9: "nine", 10: "ten",
                  11: "jack", 12: "queen",
                  13:"king", 14:"ace" }

        def __init__(self, suit, value):
                """ the constructor method sets
up instances by
                initializing initial value to
instance variables."""
                self.suit = suit
                self.value = value
```

# Functions

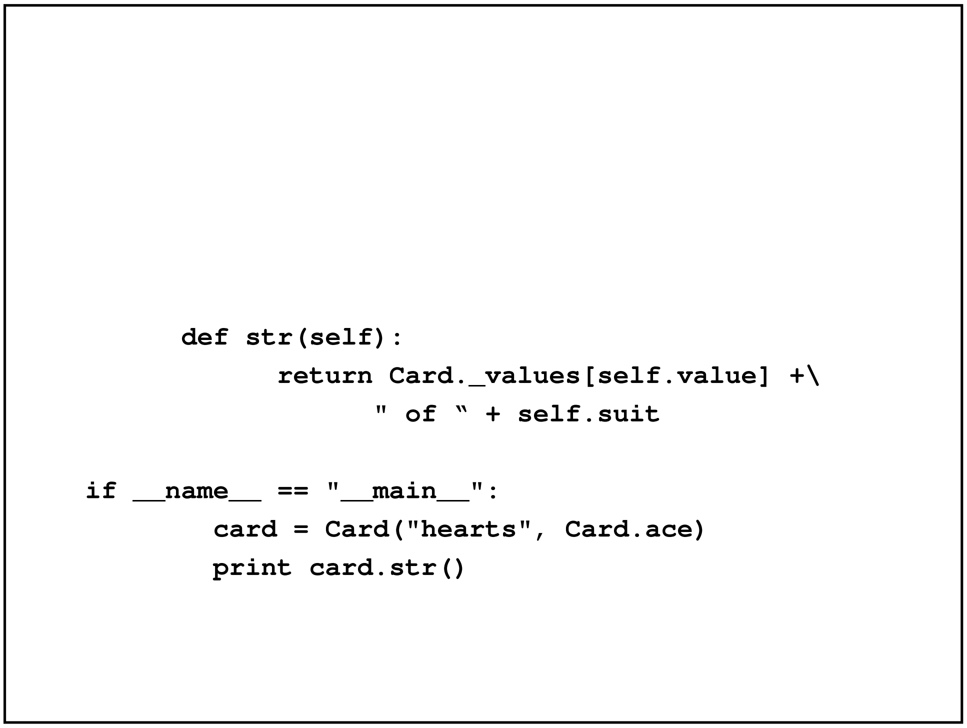
- There is nothing special about functions in a class.
  - They behave exactly the same as other functions.
  - The only difference is that self must be defined as the first argument to the function.
  - When calling the function, it can be omitted.
- Note that functions have no special visibility; to access class variables, they need to be called with fully qualified names (i.e., Card._values in Card.str)

# card4.py

```
#!/usr/bin/python

class Card:
      """ A simple example class """

      # class variables
      suits = \
      ["hearts", "clubs", "spades", "diamonds"]
      jack = 11
      queen = 12
      king = 13
      ace = 14
```



```
      _values = {2: "two", 3:"three", 4:"four",\
            5:"five", 6:"six", 7:"seven",\
            8:"eight", 9:"nine", 10:"ten",\
            11:"jack", 12:"queen", 13:"king",\
            14:"ace"}

      def __init__(self, suit, value):
            """the constructor method sets up
            instances by initiallizing values of
            instance variables"""
            self.suit = suit
            self.value = value
```



```
      def str(self):
            return Card._values[self.value] +\
                  " of “ + self.suit

if __name__ == "__main__":
        card = Card("hearts", Card.ace)
        print card.str()
```

# Exceptions

- Exceptions are objects.
- Exceptions are raised using the raise keyword.
- Exceptions can be caught using the try: ... except ... syntax.

# card5.py

```
#!/usr/bin/python

class InvalidSuitException:
        pass

class InvalidValueException:
        pass

class Card:
        """ A simple example class"""

        # class variables
        jack = 11
        queen = 12
        king = 13
        ace = 14
        _suits = ["hearts", "clubs", "spades",\
            "diamonds"]

        _values = {2:"two", 3:"three", 4:"four",
            5:"five", 6:"six", 7:"seven",
            8:"eight", 9:"nine", 10:"ten",
            11:"jack", 12:"queen", 13:"king",
            14:"ace"}

        def __init__(self, suit, value):
            """The constructor method sets up
            instances by initializing values of
            instance variable """
            if not suit in Card._suits:
                  raise InvalidSuitException
            self.suit = suit

            if not value in Card._values:
                  raise InvalidValueException
            self.value = value
```

```
        def str(self):
            return Card._values[self.value]
                  + " of " + self.suit

if __name__ == "__main__":
        try:
                card = Card("hearts", Card.ace)
        except InvalidValueException:
                print "Bad value"
        except InvalidSuitException:
                print "Bad suit"
        print card.str()
```

# Inheritance

- Unlike Java, Python understands multiple inheritance.
  - By using multiple inheritance, objects can acquire properties of other classes without having to worry.
- An example in which multiple inheritance can be useful is when making a GUI.
  - A RectangularButton can inherit from Rectangle and Button.

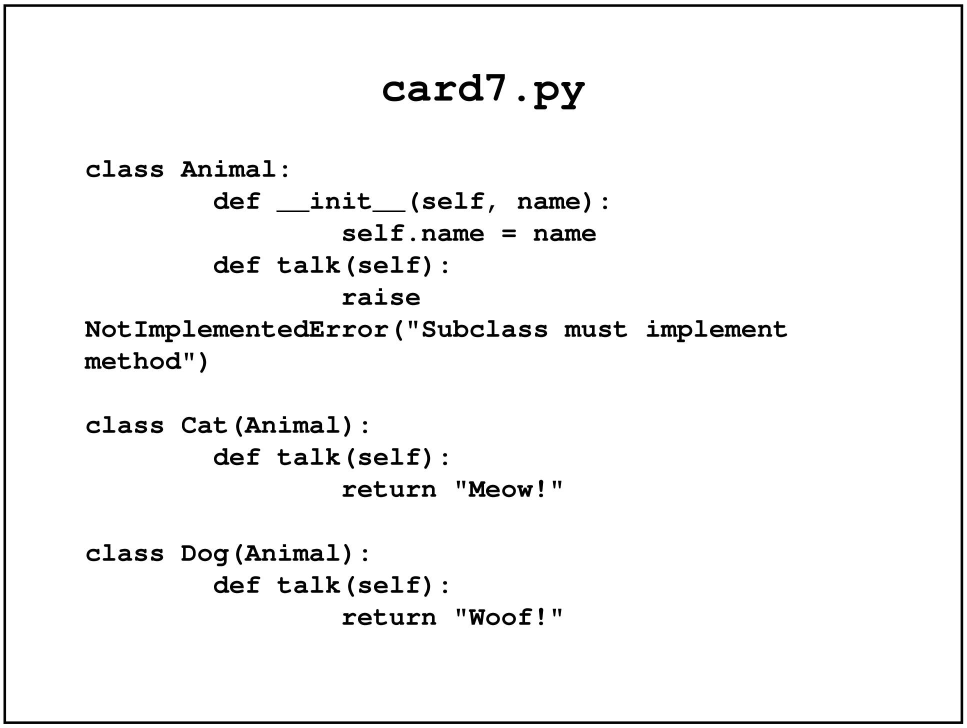
# card6.py

```
class ClassA:
        """the first class """

        def hello(self):
                return "Hello"

class ClassB:
        """The second class"""
        def world(self):
                return "World"

class ClassC(ClassA, ClassB):
        """The composite classr"""
        def helloWorld(self):
                return self.hello() + " " +
                  self.world()
```

```
if __name__ == "__main__":
        c = ClassC()
        print c.hello()
        print c.world()
        print c.helloWorld()
```

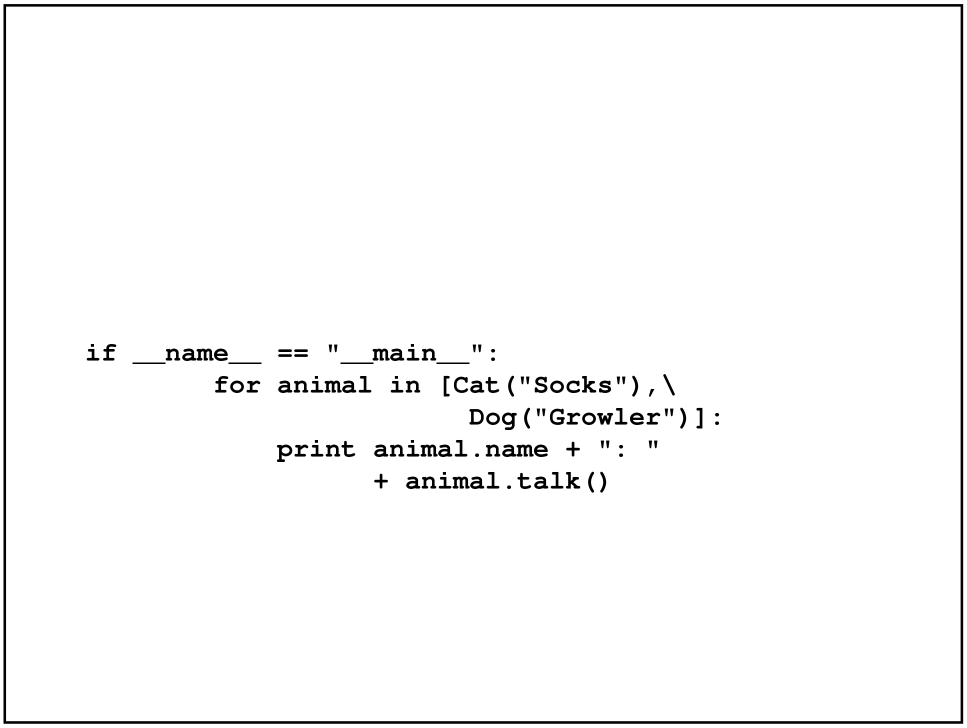
# Abstract methods

- Python does not support abstract functions, because it does not need it.
- The recommended way in Python is by raising an exception in the superclass that does not implement it.



## card7.py

```
class Animal:
        def __init__(self, name):
                self.name = name
        def talk(self):
                raise
NotImplementedError("Subclass must implement
method")

class Cat(Animal):
        def talk(self):
                return "Meow!"

class Dog(Animal):
        def talk(self):
                return "Woof!"
```



```
if __name__ == "__main__":
        for animal in [Cat("Socks"),\
                        Dog("Growler")]:
            print animal.name + ": "
                  + animal.talk()
```

# Polymorphism

- Python does not support polymorphism within a class.
  - In other words, if the same function is defined more than once in a class, subsequent definitions hide the first one.

# Polymorphism

- There are two good reasons for polymorphism within a namespace:
  1. To define a method with default parameter values
  2. To allow a method to operate on different data types.
- Both reasons do not apply in Python.
- Parameters support default values and data types are dynamically bound to variables.

## card8.py

```
SIEGFRIE@panther:~/python$ cat card8.py
def method(a = 10, b = 20, c = 30):
        return a, b, c

print method()
print method(1)
print method(1, 2)
print method(1, 2, 3)
SIEGFRIE@panther:~/python$ python card8.py
(10, 20, 30)
(1, 20, 30)
(1, 2, 30)
(1, 2, 3)
SIEGFRIE@panther:~/python$
```

## Name mangling

- To avoid name clashes in subclasses, Python supports name mangling.
- All names (functions and variables) that begin with a double underscore (__) are implicitly translated to the form `_classname__name`.
- In some case, name mangling is used to simulate private names.

## card9.py

```
SIEGFRIE@panther:~/python$ cat card9.py
class ClassA:
        def myfunc(self):
                return "myfunc A"

        def __myfunc(self):
                return "__myfunc A"

class ClassB(ClassA):
        def myfunc(self):
                return "myfunc B"

        def __myfunc(self):
                return "__myfunc B"
```

```
if __name__ == "__main__":
        b = ClassB()
        print b.myfunc() # myfunc in ClassA is
inaccessible
        print b._ClassA__myfunc()
        print b._ClassB__myfunc()
SIEGFRIE@panther:~/python$ p
myfunc B
__myfunc A
__myfunc B
SIEGFRIE@panther:~/python$
```