

# Python Objects

Alex Seong

# Lets Start with Programs



```
inp = input('Europe floor? ')
usf = int(inp) + 1
print('US floor', usf)
```

Europe floor? 0  
US floor 1



# Object Oriented

- A program is made up of many cooperating objects
- Instead of being the “whole program” - each object is a little “island” within the program and cooperatively working with other objects
- A program is made up of one or more objects working together - objects make use of each other’s capabilities

# Object

- An Object is a bit of self-contained Code and Data
- A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)
- Objects have boundaries that allow us to ignore un-needed detail
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...

Input

Object

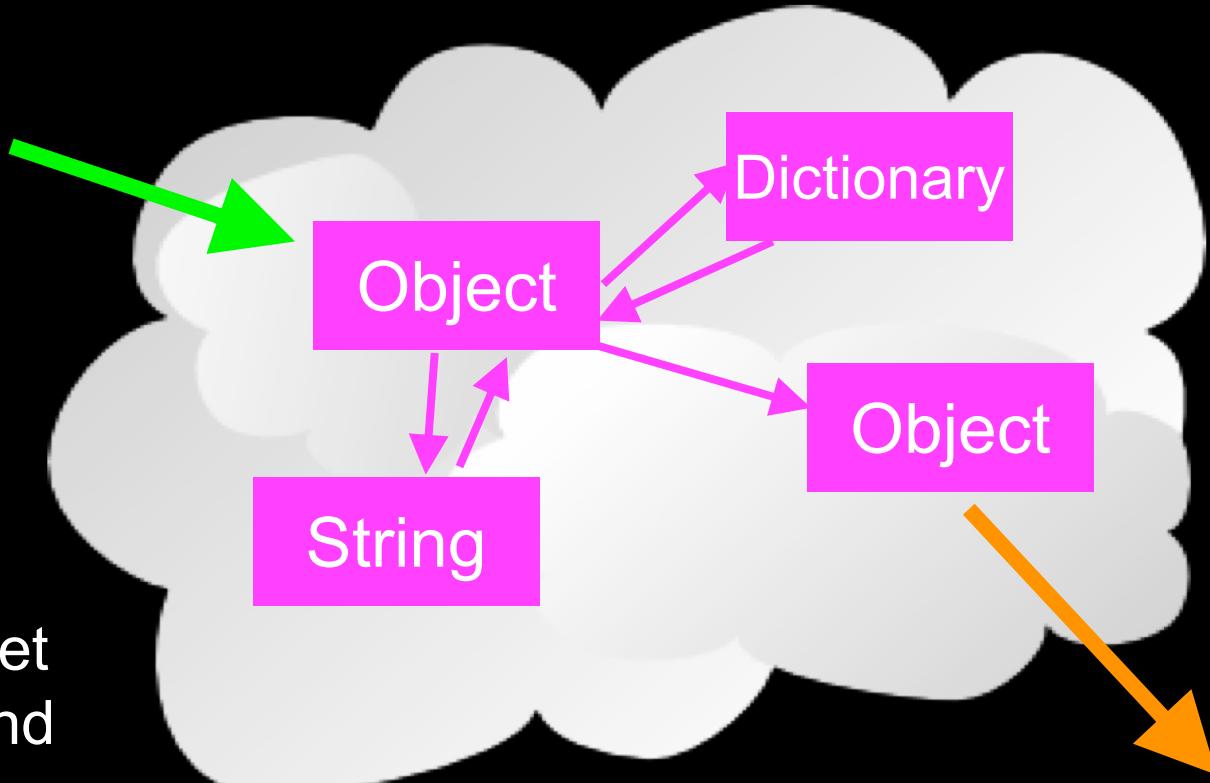
String

Dictionary

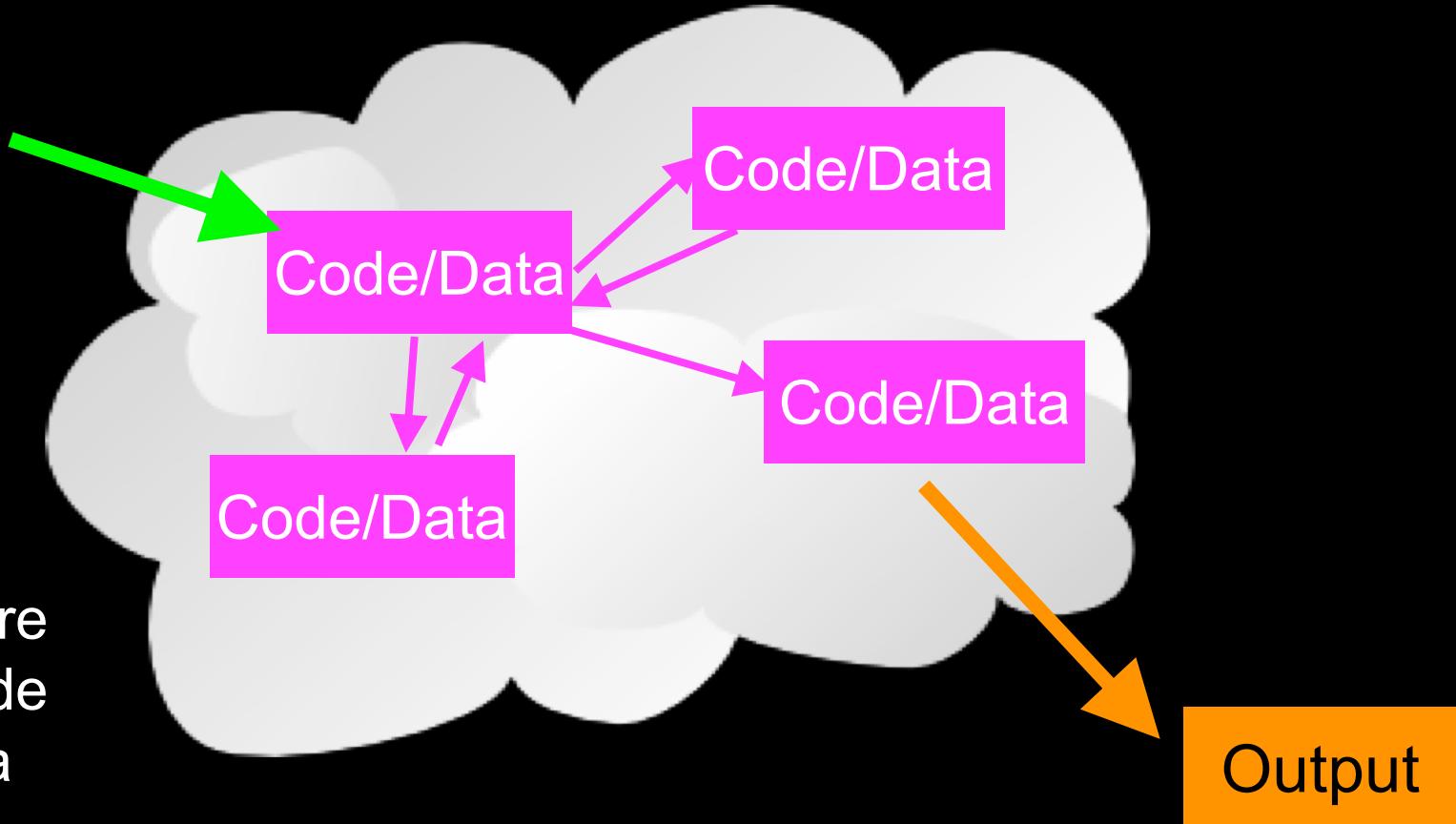
Object

Objects get  
created and  
used

Output



Input



Objects are  
bits of code  
and data

Input

Code/Data

Code/Data

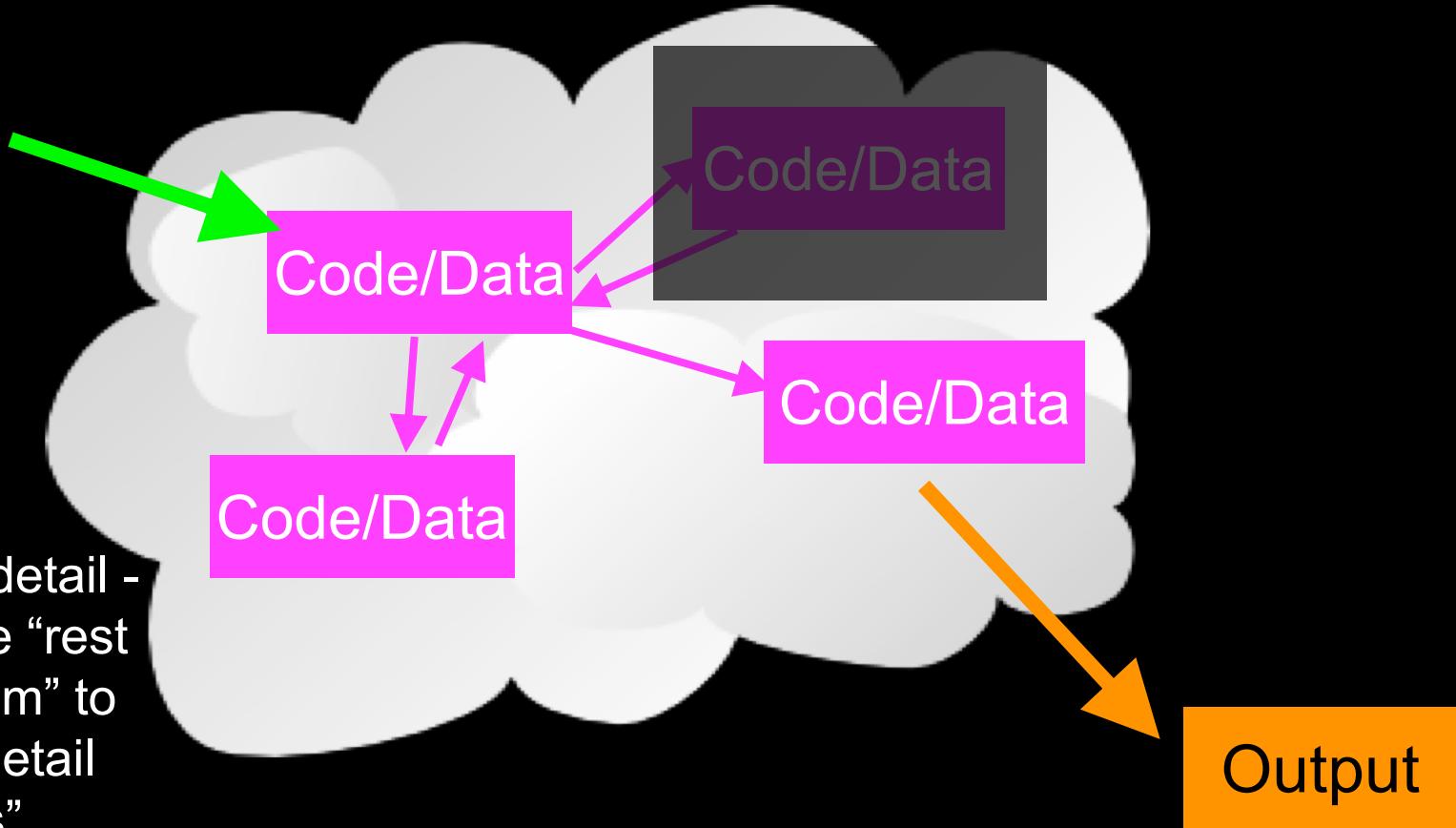
Code/Data

Code/Data

Output

Objects hide detail  
- they allow us to  
ignore the detail of  
the “rest of the  
program”.

Input



Objects hide detail -  
they allow the “rest  
of the program” to  
ignore the detail  
about “us”.

# Definitions



- **Class** - a template
- **Method or Message** - A defined capability of a class
- **Field or attribute**- A bit of data in a class
- **Object or Instance** - A particular instance of a class

# Terminology: Class



Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, **fields** or **properties**) and the thing's behaviors (the things it can do, or **methods**, operations or features). One might say that a **class** is a **blueprint** or factory that describes the nature of something. For example, the **class** Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors).

# Terminology: Instance



One can have an **instance** of a class or a particular object.

The **instance** is the actual object created at runtime. In programmer jargon, the Lassie object is an **instance** of the Dog class. The set of values of the attributes of a particular object is called its **state**. The **object** consists of state and the behavior that's defined in the object's class.

Object and Instance are often used interchangeably.

# Terminology: Method



An object's abilities. In language, **methods** are verbs. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other **methods** as well, for example sit() or eat() or walk() or save\_timmy(). Within the program, using a **method** usually affects only one particular object; all Dogs can bark, but you need only one particular dog to do the barking

Method and Message are often used interchangeably.

# Some Python Objects

```
>>> x = 'abc'  
>>> type(x)  
<class 'str'>  
>>> type(2.5)  
<class 'float'>  
>>> type(2)  
<class 'int'>  
>>> y = list()  
>>> type(y)  
<class 'list'>  
>>> z = dict()  
>>> type(z)  
<class 'dict'>  
  
>>> dir(x)  
[ ... 'capitalize', 'casefold', 'center', 'count',  
  'encode', 'endswith', 'expandtabs', 'find',  
  'format', ... 'lower', 'lstrip', 'maketrans',  
  'partition', 'replace', 'rfind', 'rindex', 'rjust',  
  'rpartition', 'rsplit', 'rstrip', 'split',  
  'splitlines', 'startswith', 'strip', 'swapcase',  
  'title', 'translate', 'upper', 'zfill']  
>>> dir(y)  
[... 'append', 'clear', 'copy', 'count', 'extend',  
  'index', 'insert', 'pop', 'remove', 'reverse',  
  'sort']  
>>> dir(z)  
[..., 'clear', 'copy', 'fromkeys', 'get', 'items',  
  'keys', 'pop', 'popitem', 'setdefault', 'update',  
  'values']
```

# A Sample Class



class is a reserved word

Each PartyAnimal object has a bit of code

Tell the an object to run the party() code within it

```
class PartyAnimal:  
    x = 0  
  
    def party(self) :  
        self.x = self.x + 1  
        print("So far",self.x)  
  
an = PartyAnimal()  
  
an.party()  
an.party()  
an.party()
```

This is the template for making PartyAnimal objects

Each PartyAnimal object has a bit of data

Construct a PartyAnimal object and store in an

PartyAnimal.party(an)

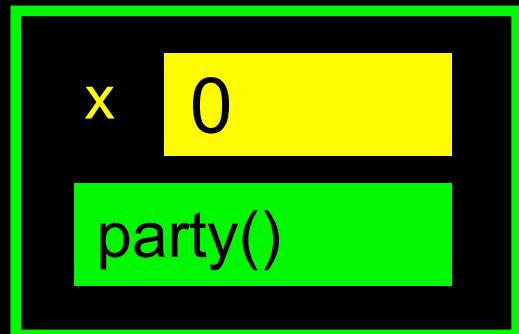
```
class PartyAnimal:  
    x = 0  
  
    def party(self) :  
        self.x = self.x + 1  
        print("So far",self.x)  
  
an = PartyAnimal()  
  
an.party()  
an.party()  
an.party()
```

\$ python party1.py

```
class PartyAnimal:  
    x = 0  
  
    def party(self) :  
        self.x = self.x + 1  
        print("So far",self.x)  
  
an = PartyAnimal()  
  
an.party()  
an.party()  
an.party()
```

\$ python party1.py

an



```
class PartyAnimal:  
    x = 0  
  
    def party(self) :  
        self.x = self.x + 1  
        print("So far",self.x)  
  
an = PartyAnimal()  
  
an.party()  
an.party()  
an.party()
```

```
$ python party1.py  
So far 1  
So far 2  
So far 3
```

an  
self



PartyAnimal.party(an)

# Playing with dir() and type()

# A Nerdy Way to Find Capabilities

- The `dir()` command lists capabilities
- Ignore the ones with underscores - these are used by Python itself
- The rest are real operations that the object can perform
- It is like `type()` - it tells us something \*about\* a variable

```
>>> y = list()
>>> type(y)
<class 'list'>
>>> dir(x)
[ '__add__', '__class__',
  '__contains__', '__delattr__',
  '__delitem__', '__delslice__',
  '__doc__', ..., '__setitem__',
  '__setslice__', '__str__',
  'append', 'clear', 'copy',
  'count', 'extend', 'index',
  'insert', 'pop', 'remove',
  'reverse', 'sort']
>>>
```

```
class PartyAnimal:  
    x = 0  
  
    def party(self) :  
        self.x = self.x + 1  
        print("So far",self.x)  
  
an = PartyAnimal()  
  
print("Type", type(an))  
print("Dir ", dir(an))
```

We can use `dir()` to find the “capabilities” of our newly created class.

```
$ python party3.py  
Type <class '__main__.PartyAnimal'>  
Dir  ['__class__', ... 'party', 'x']
```

# Try dir() with a String

```
>>> x = 'Hello there'  
>>> dir(x)  
['__add__', '__class__', '__contains__', '__delattr__',  
'__doc__', '__eq__', '__ge__', '__getattribute__',  
'__getitem__', '__getnewargs__', '__getslice__', '__gt__',  
'__hash__', '__init__', '__le__', '__len__', '__lt__',  
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__',  
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',  
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit',  
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',  
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',  
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',  
'splitlines', 'startswith', 'strip', 'swapcase', 'title',  
'translate', 'upper', 'zfill']
```

# Object Lifecycle

[http://en.wikipedia.org/wiki/Constructor\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Constructor_(computer_science))

# Object Lifecycle

- Objects are created, used, and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

# Constructor

The primary purpose of the constructor is to set up some instance variables to have the proper initial values when the object is created

```
class PartyAnimal:  
    x = 0  
  
    def __init__(self):  
        print('I am constructed')  
  
    def party(self) :  
        self.x = self.x + 1  
        print('So far',self.x)  
  
    def __del__(self):  
        print('I am destructured', self.x)  
  
an = PartyAnimal()  
an.party()  
an.party()  
an = 42  
print('an contains',an)
```

```
$ python party4.py  
I am constructed  
So far 1  
So far 2  
I am destructured 2  
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

# Constructor



In **object oriented programming**, a **constructor** in a class is a special block of statements called when an **object** is created

[http://en.wikipedia.org/wiki/Constructor\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Constructor_(computer_science))

# Many Instances

- We can create **lots of objects** - the class is the template for the object
- We can store each **distinct object** in its own variable
- We call this having multiple **instances** of the same class
- Each **instance** has its own copy of the **instance variables**

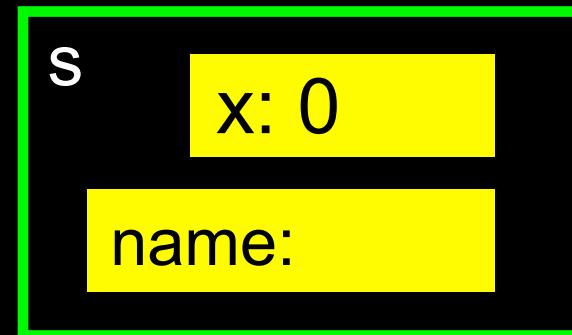
```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, z):  
        self.name = z  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
s = PartyAnimal("Sally")  
j = PartyAnimal("Jim")  
  
s.party()  
j.party()  
s.party()
```

Constructors can have additional parameters. These can be used to set up instance variables for the particular instance of the class (i.e., for the particular object).

party5.py

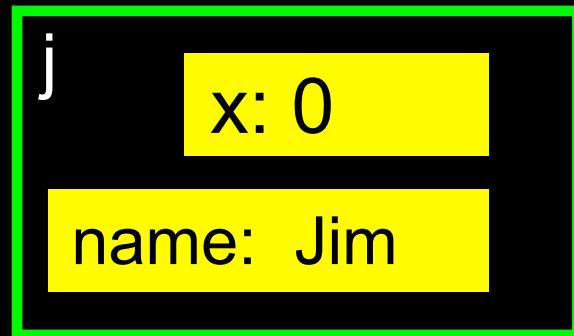
```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, z):  
        self.name = z  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
s = PartyAnimal("Sally")  
j = PartyAnimal("Jim")  
  
s.party()  
j.party()  
s.party()
```

```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, z):  
        self.name = z  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
s = PartyAnimal("Sally")  
j = PartyAnimal("Jim")  
  
s.party()  
j.party()  
s.party()
```



```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, z):  
        self.name = z  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
s = PartyAnimal("Sally")  
j = PartyAnimal("Jim")  
  
s.party()  
j.party()  
s.party()
```

We have two  
independent  
instances



```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, z):  
        self.name = z  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
s = PartyAnimal("Sally")  
j = PartyAnimal("Jim")  
  
s.party()  
j.party()  
s.party()
```

Sally constructed  
Jim constructed  
Sally party count 1  
Jim party count 1  
Sally party count 2

# Inheritance

<http://www.ibiblio.org/g2swap/bytewofpython/read/inheritance.html>

# Inheritance

- When we make a new class - we can reuse an existing class and **inherit** all the capabilities of an existing class and then add our own little bit to make our new class
- Another form of store and reuse
- Write once - reuse many times
- The new class (child) has all the capabilities of the old class (parent) - and then some more

# Terminology: Inheritance



‘Subclasses’ are more specialized versions of a class, which **inherit** attributes and behaviors from their parent classes, and can introduce their own.

[http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming)

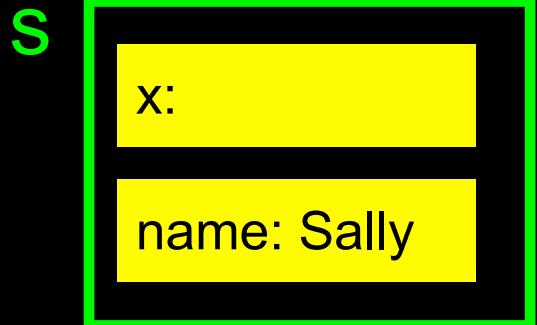
```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, nam):  
        self.name = nam  
        print(self.name,"constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name,"party count",self.x)  
  
class FootballFan(PartyAnimal):  
    points = 0  
    def touchdown(self):  
        self.points = self.points + 7  
        self.party()  
        print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")  
s.party()  
  
j = FootballFan("Jim")  
j.party()  
j.touchdown()
```

**FootballFan** is a class which extends **PartyAnimal**. It has all the capabilities of **PartyAnimal** and more.

```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, nam):  
        self.name = nam  
        print(self.name, "constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name, "party count", self.x)  
  
class FootballFan(PartyAnimal):  
    points = 0  
    def touchdown(self):  
        self.points = self.points + 7  
        self.party()  
        print(self.name, "points", self.points)
```

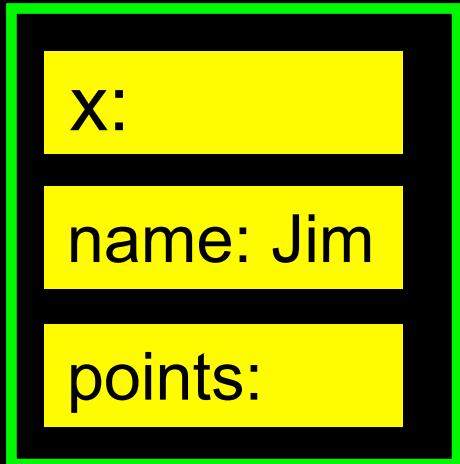
```
s = PartyAnimal("Sally")  
s.party()  
  
j = FootballFan("Jim")  
j.party()  
j.touchdown()
```



```
class PartyAnimal:  
    x = 0  
    name = ""  
    def __init__(self, nam):  
        self.name = nam  
        print(self.name, "constructed")  
  
    def party(self) :  
        self.x = self.x + 1  
        print(self.name, "party count", self.x)  
  
class FootballFan(PartyAnimal):  
    points = 0  
    def touchdown(self):  
        self.points = self.points + 7  
        self.party()  
        print(self.name, "points", self.points)
```

```
s = PartyAnimal("Sally")  
s.party()  
  
j = FootballFan("Jim")  
j.party()  
j.touchdown()
```

j



# Definitions

- **Class** - a template
- **Attribute** – A variable within a class
- **Method** - A function within a class
- **Object** - A particular instance of a class
- **Constructor** – Code that runs when an object is created
- **Inheritance** - The ability to extend a class to make a new class.



# Summary

- Object Oriented programming is a very structured approach to code reuse
- We can group data and functionality together and create many independent instances of a class