

Python Objects

Charles Severance



Python for Everybody
www.py4e.com



Warning

- This lecture is very much about definitions and mechanics for objects
- This lecture is a lot more about “how it works” and less about “how you use it”
- You won’t get the entire picture until this is all looked at in the context of a real problem
- So please suspend disbelief and learn technique for the next 40 or so slides...

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list **objects**:

`list.append(x)`

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.extend(L)`

Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.remove(x)`

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

`list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

12.6. `sqlite3` — DB-API 2.0 interface for SQLite databases

Source code: [Lib/sqlite3/](#)

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The `sqlite3` module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by [PEP 249](#).

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('CREATE TABLE stocks
          (date text, trans text, symbol text, qty real, price real)')
```

<https://docs.python.org/3/library/sqlite3.html>

Lets Start with Programs



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

Europe floor? 0
US floor 1



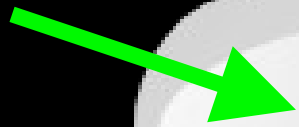
Object Oriented

- 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

Dictionary

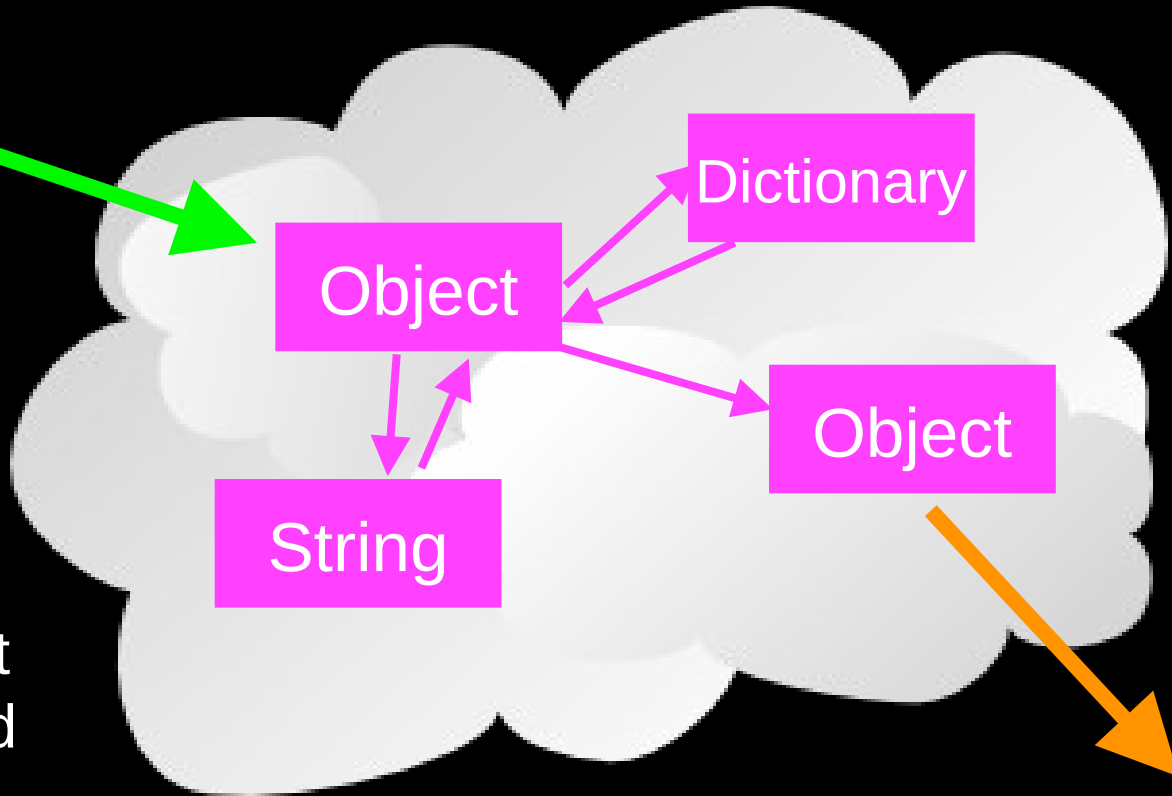
String

Object

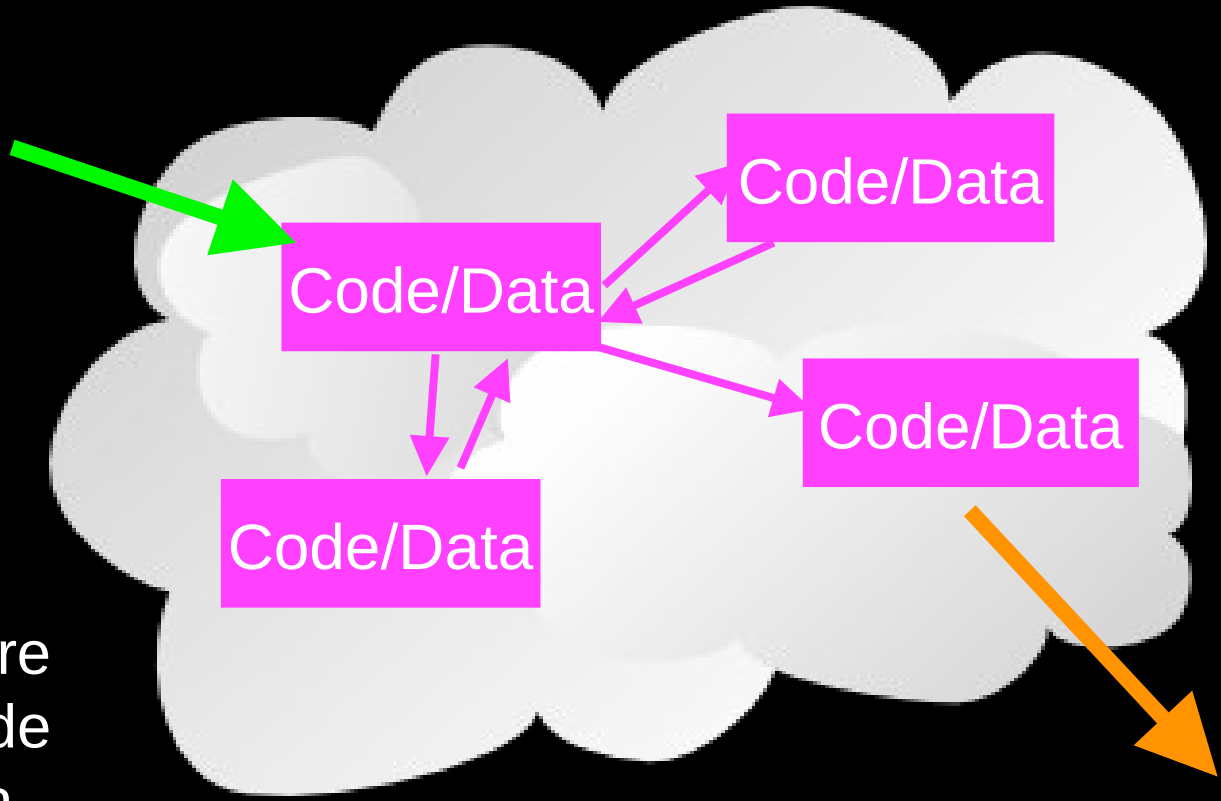
Objects get created and used



Output



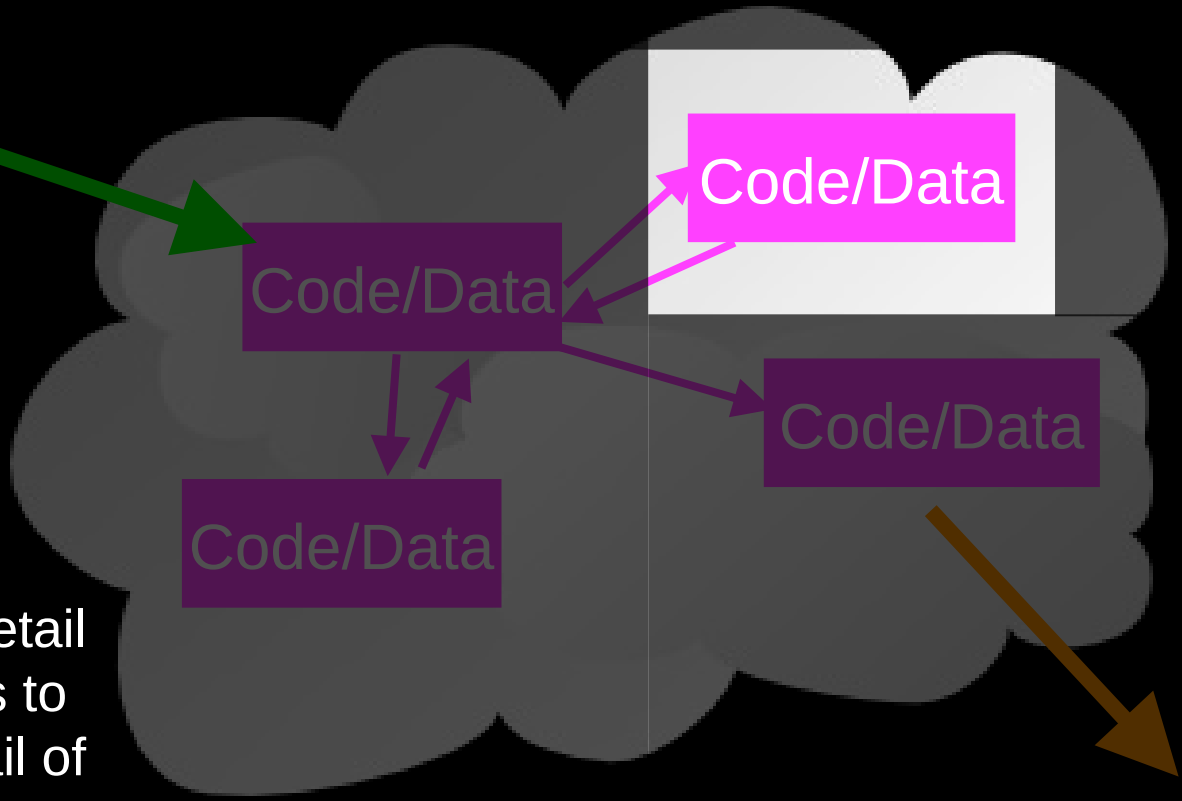
Input



Objects are bits of code and data

Output

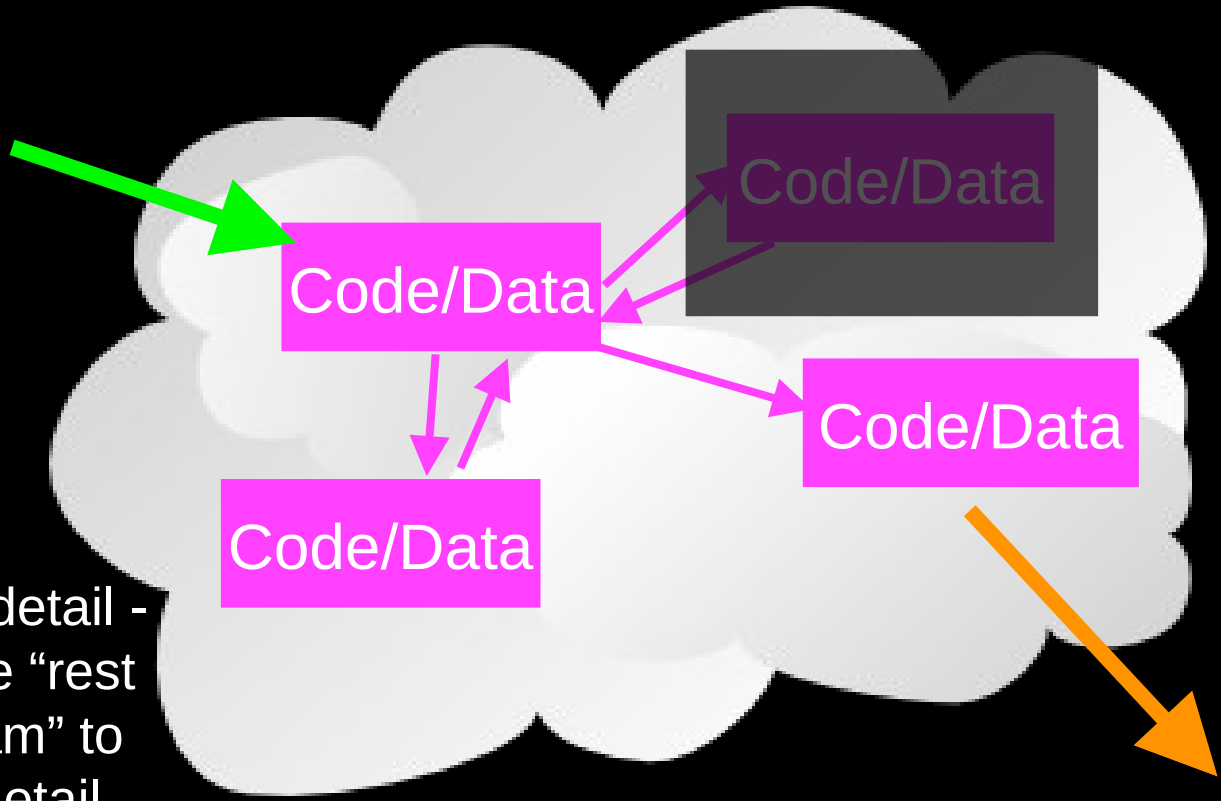
Input



Output

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

Input



Code/Data

Code/Data

Code/Data

Code/Data

Output

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).

http://en.wikipedia.org/wiki/Object-oriented_programming

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.

http://en.wikipedia.org/wiki/Object-oriented_programming

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.

http://en.wikipedia.org/wiki/Object-oriented_programming

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',
'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 that defines a template for making objects

Each PartyAnimal object has a bit of code

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

```
class PartyAnimal:

    def __init__(self):
        self.x = 0

    def party(self) :
        self.x = self.x + 1
        print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

When the object is constructed, a specially named method is called to allocate and initialize attributes.

Construct a PartyAnimal object and store in an

PartyAnimal.party(an)

```
class PartyAnimal:
```

```
    def __init__(self):  
        self.x = 0
```

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

```
an = PartyAnimal()
```

```
an.party()
```

```
an.party()
```

```
an.party()
```

```
$ python party2.py
```

```
class PartyAnimal:
```

```
    def __init__(self):  
        self.x = 0
```

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

```
an = PartyAnimal()
```

```
an.party()  
an.party()  
an.party()
```

```
$ python party2.py
```

an

x

0

party()

```
class PartyAnimal:

    def __init__(self):
        self.x = 0

    def party(self) :
        self.x = self.x + 1
        print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

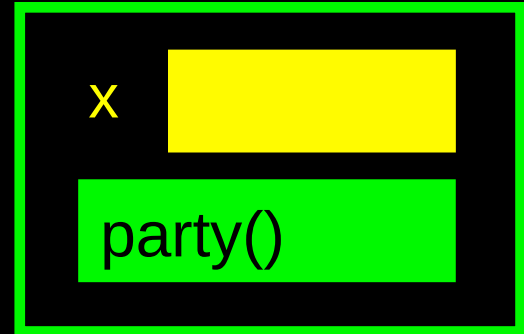
\$ python party2.py

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(y)
['__add__', '__class__',
 '__class_getitem__',
 '__contains__', '__delattr__',
 '__delitem__', '__dir__',
 '__doc__', '__eq__', ...
 'append', 'clear', 'copy',
 'count', 'extend', 'index',
 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>>
```

```
class PartyAnimal:

    def __init__(self):
        self.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))
print ("Type", type(an.x))
print ("Type", type(an.party))
```

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

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

party3.py

Try dir() with a String

```
>>> x = 'Hello there'
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__',
 '__getstate__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', ...'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'removesuffix',
'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:

    def __init__(self):
        self.x = 0
        print('I am constructed')

    def party(self) :
        self.x = self.x + 1
        print('So far',self.x)

    def __del__(self):
        print('I am destructed', 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 destructed 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:

    def __init__(self, z):
        self.x = 0
        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")
s.party()
j = PartyAnimal("Jim")

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).

```
class PartyAnimal:

    def __init__(self, z):
        self.x = 0
        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")
s.party()

j = PartyAnimal("Jim")

j.party()
s.party()
```

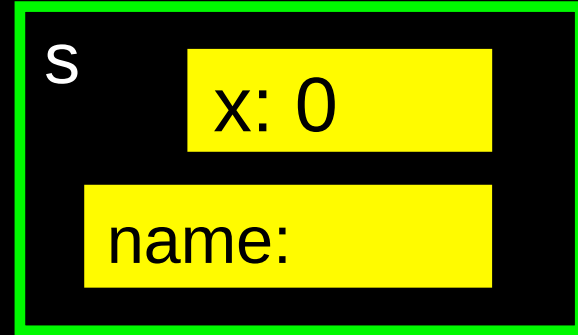
```
class PartyAnimal:

    def __init__(self, z):
        self.x = 0
        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")
s.party()
j = PartyAnimal("Jim")

j.party()
s.party()
```



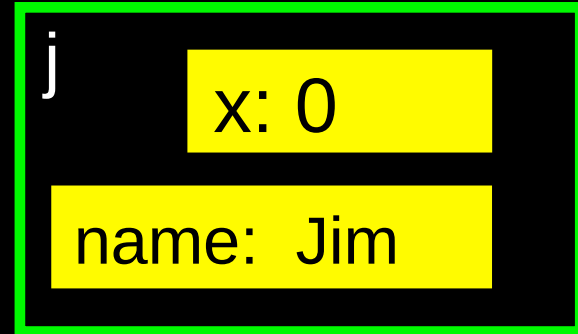
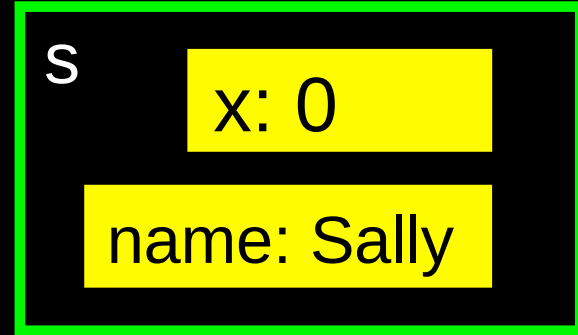
```
class PartyAnimal:

    def __init__(self, z):
        self.x = 0
        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")
s.party()
j = PartyAnimal("Jim")

j.party()
s.party()
```



```
class PartyAnimal:
```

```
    def __init__(self, z):
```

```
        self.x = 0
```

```
        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")
```

```
s.party()
```

```
j = PartyAnimal("Jim")
```

```
j.party()
```

```
s.party()
```

```
Sally constructed
```

```
Sally party count 1
```

```
Jim constructed
```

```
Jim party count 1
```

```
Sally party count 2
```

Inheritance

<http://www.ibiblio.org/g2swap/byteofpython/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

```
class PartyAnimal:
```

```
    def __init__(self, nam):  
        self.x = 0  
        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):
```

```
    def __init__(self, nam) :  
        super().__init__(nam)  
        self.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.

party7.py

```
class PartyAnimal:
```

```
    def __init__(self, nam):  
        self.x = 0  
        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):
```

```
    def __init__(self, nam) :  
        super().__init__(nam)  
        self.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()
```

S

x:

name: Sally

```
class PartyAnimal:
```

```
    def __init__(self, nam):
        self.x = 0
        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):
```

```
    def __init__(self, nam) :
        super().__init__(nam)
        self.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

x:

name: Jim

points:

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



Acknowledgements / Contributions



These slides are Copyright 2010- Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School of Information

... Insert new Contributors here

Additional Source Information

- "Snowman Cookie Cutter" by Didriks is licensed under CC
<https://www.flickr.com/photos/dinnerseries/23570475099>
- Photo from the television program *Lassie*. Lassie watches as Jeff (Tommy Rettig) works on his bike is
https://en.wikipedia.org/wiki/Lassie#/media/File:Lassie_and_Tommy_Rettig_1956.JPG