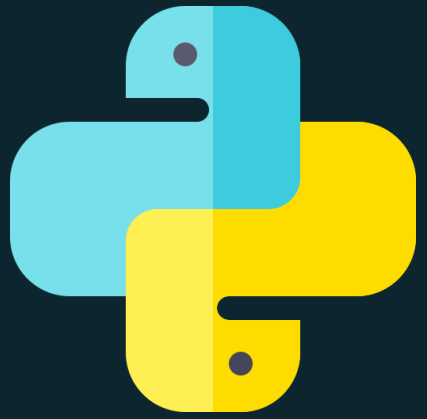
Brief Python

Python Course for Programmers

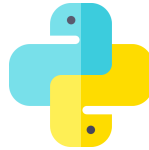


Learn Python Language for Data Science

CHAPTER 10: OBJECT-ORIENTED PROGRAMMING

DR. ERIC CHOU

IEEE SENIOR MEMBER

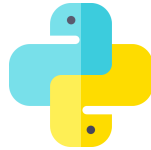


Objectives

- Learn basic class design using party animal.
- Learn detailed class design using Atom class
- Learn magic function design using Molecule and TestMethane

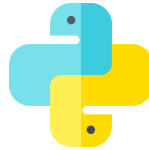
Python Objects

LECTURE 1



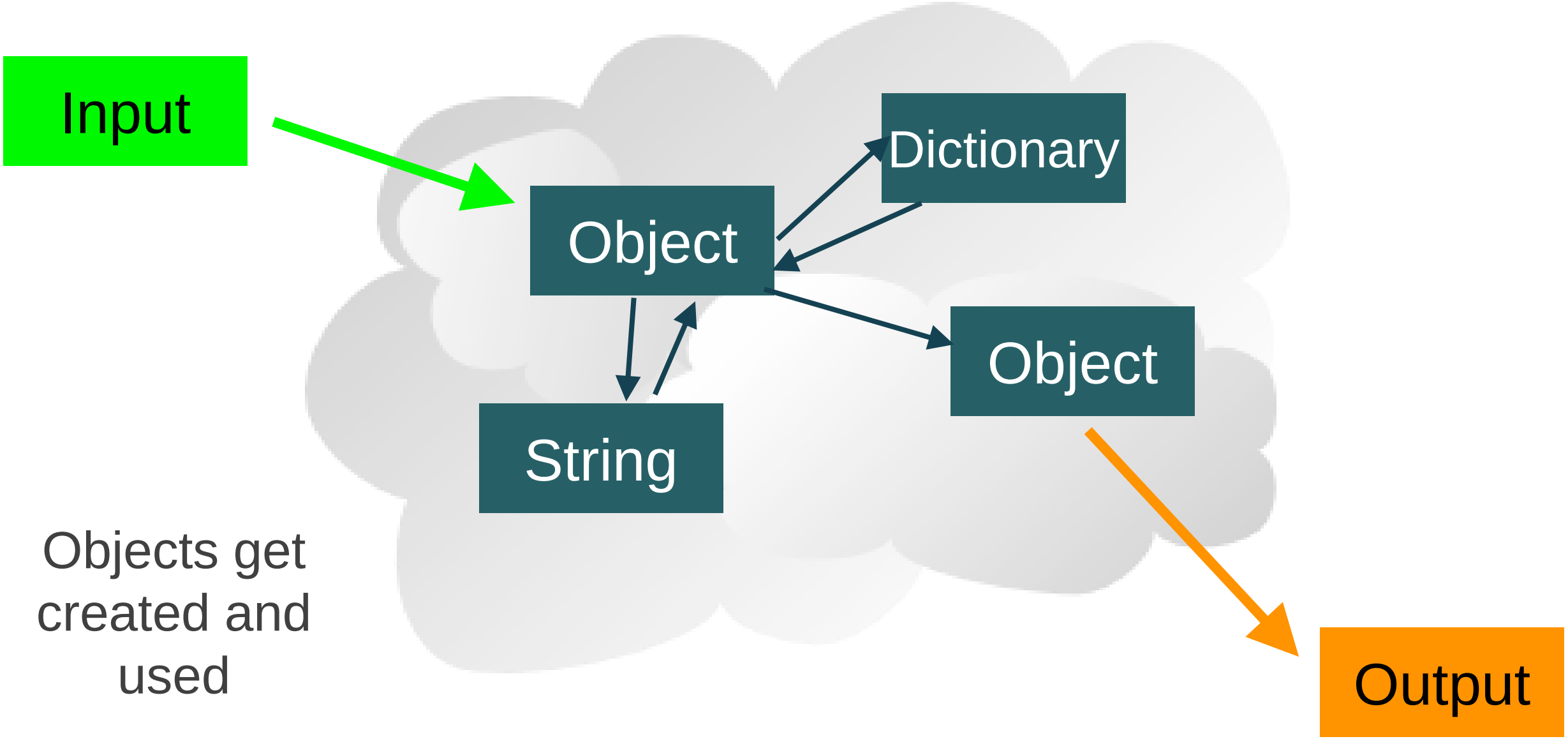
Object Oriented

- 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

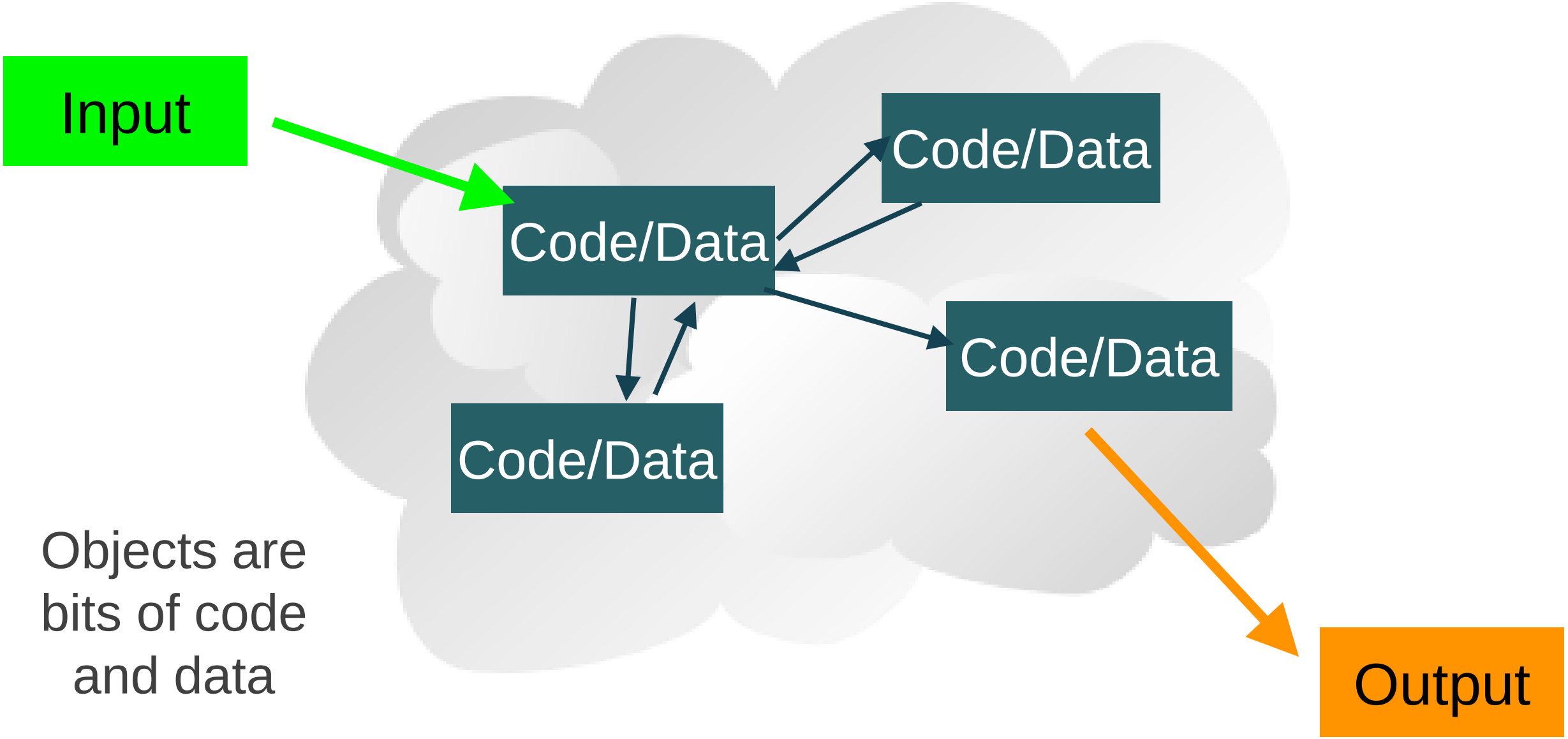


Object

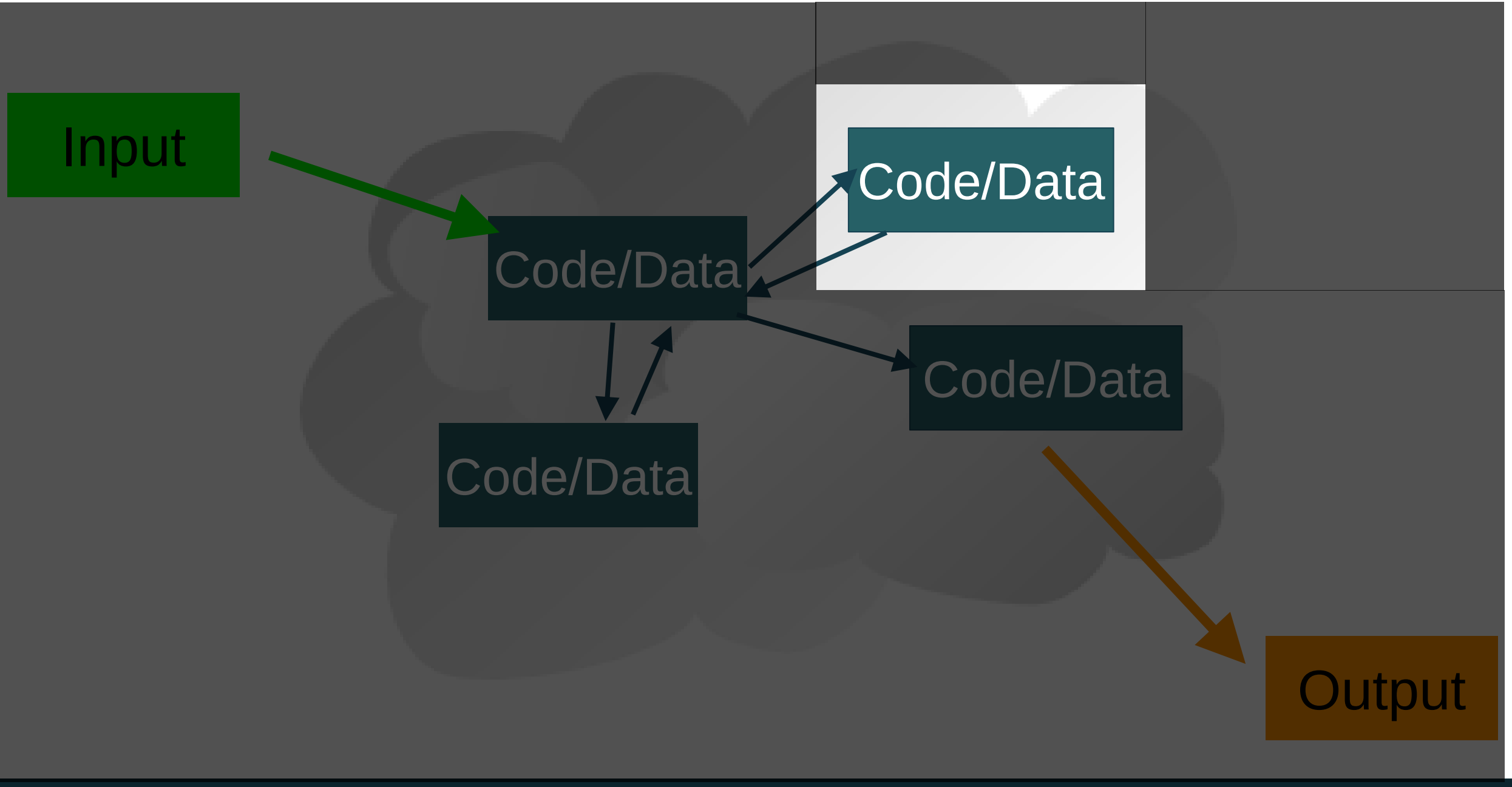
- An Object is a bit of self-contained Code and Data
- A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)
- Objects have boundaries that allow us to ignore un-needed detail
- We have been using objects all along: String Objects, Integer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

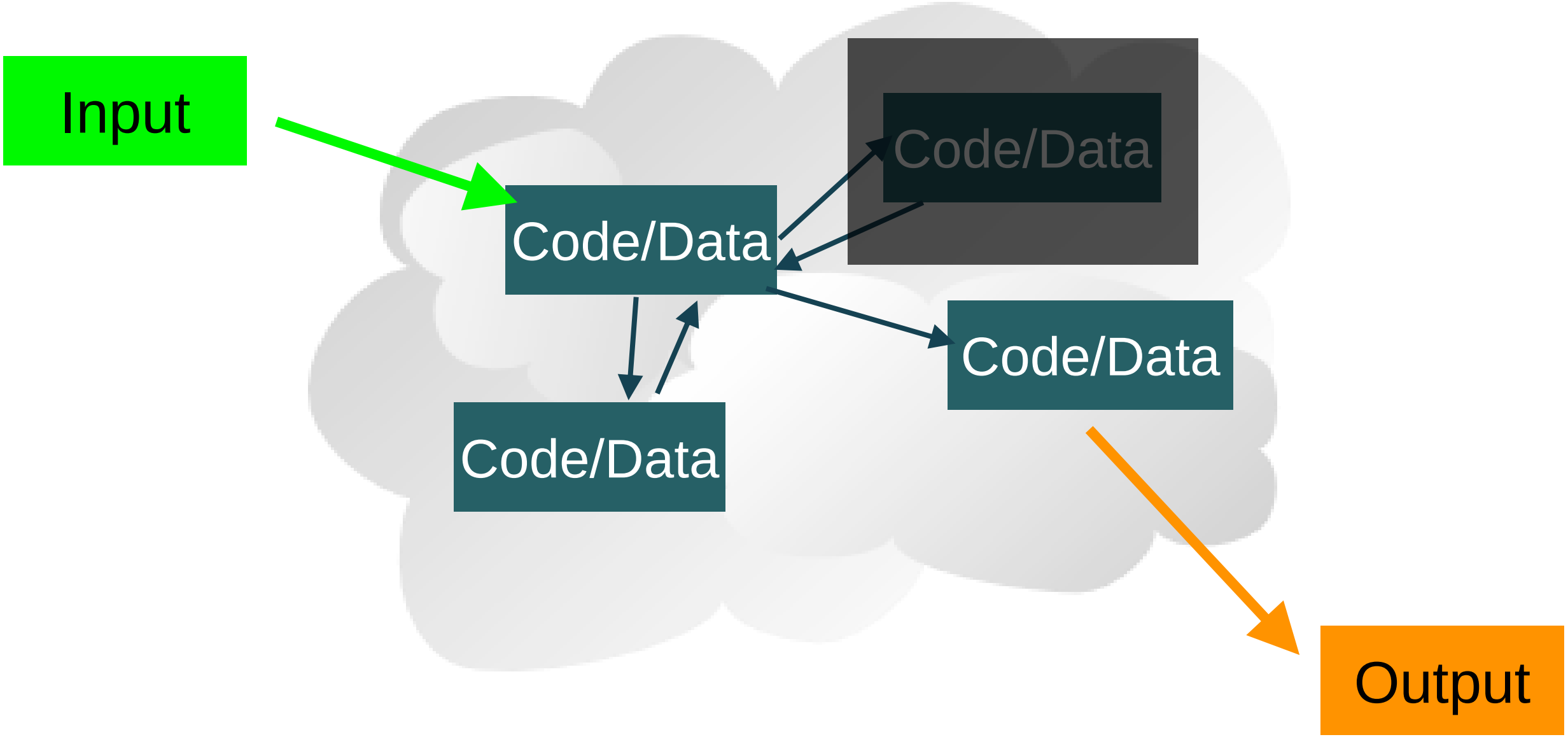


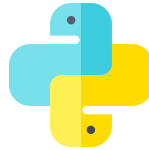
Objects get created and used



Objects are bits of code and data

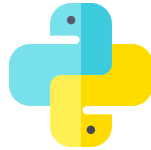

Input
Code/Data
Code/Data
Code/Data
Code/Data
Output


Input
Code/Data
Code/Data
Code/Data
Code/Data
Output



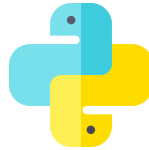
Definitions

- **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



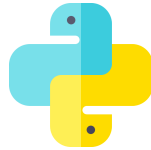
Terminology: Class

- 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).



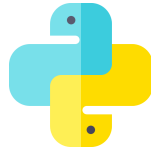
Terminology: Instance

- 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']
```

Python Class

LECTURE 2

class is a reserved word

Each PartyAnimal object has a bit of code

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

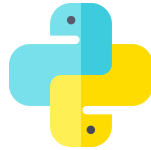
```
class PartyAnimal:
  x = 0

  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)



Simple Class

Demo Program: partyanimal1.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()
```

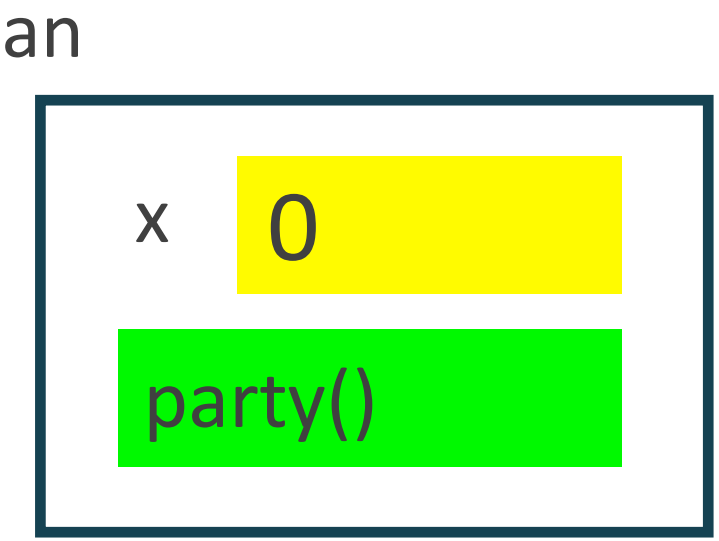
```
So far 1
So far 2
So far 3
```

```
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 partyanimal1.py

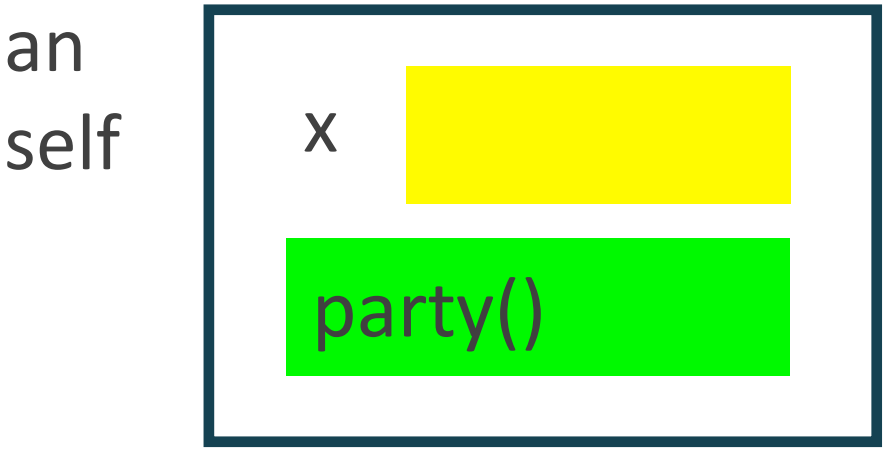
```
class PartyAnimal:
  x = 0

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

an = PartyAnimal()

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

$ python partyanimal1.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
So far 1
So far 2
So far 3
```



PartyAnimal.party(an)

Playing with dir() and type()

ACTIVITY

A Nerdy Way to Find Capabilities

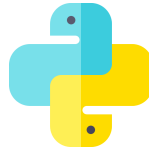
- The dir() command lists capabilities
- Ignore the ones with underscores - these are used by Python itself
- The rest are real operations that the object can perform
- It is like type() - it tells us something *about* a variable

```
>>> 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 partyanimal2.py
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', ... 'party', 'x']
```

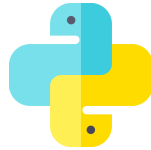


Type and Dir

Demo Program: partyanimal2.py

```
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))
```

```
Type <class '__main__.PartyAnimal'>

Dir  ['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'party', 'x']
```

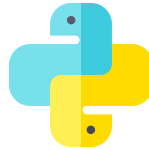


Try dir() with a String

```
>>> 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']
```

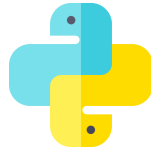
Object Lifecycle

ACTIVITY



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



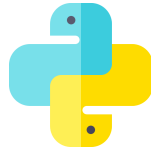
Constructor

- 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 destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

```
$ python partyanimal3.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

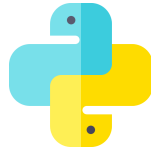
Demo Program: partyanimal3.py

```
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 destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

Constructor

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



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

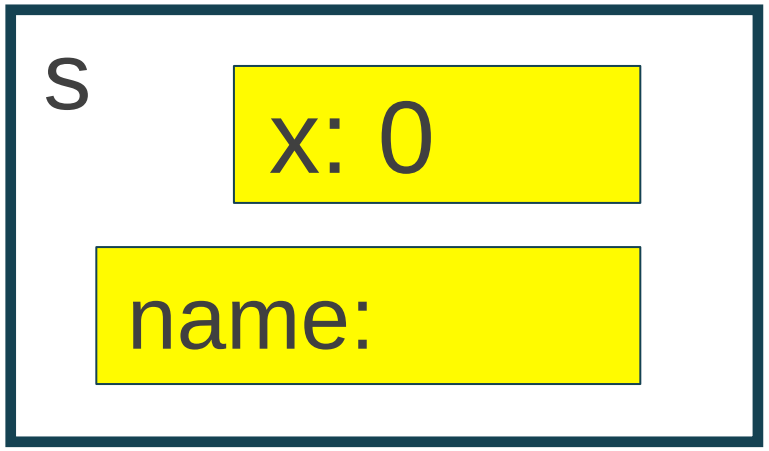
```
class PartyAnimal:
    x = 0
    name = ""
    def __init__(self, z):
      self.name = z
      print(self.name,"constructed")

    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).

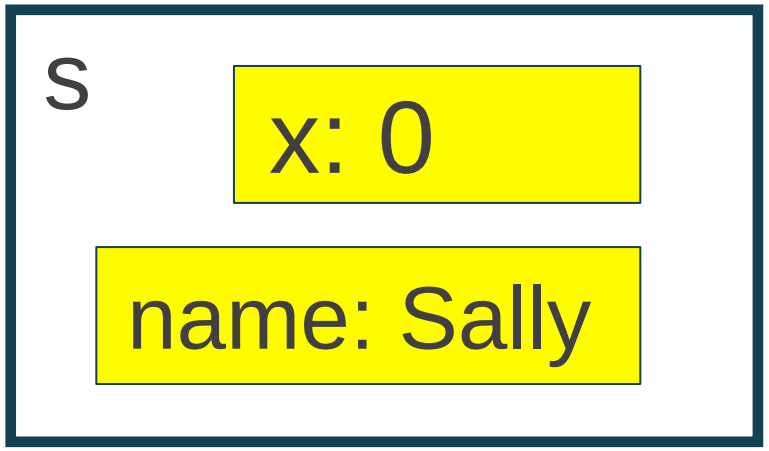
party5.py

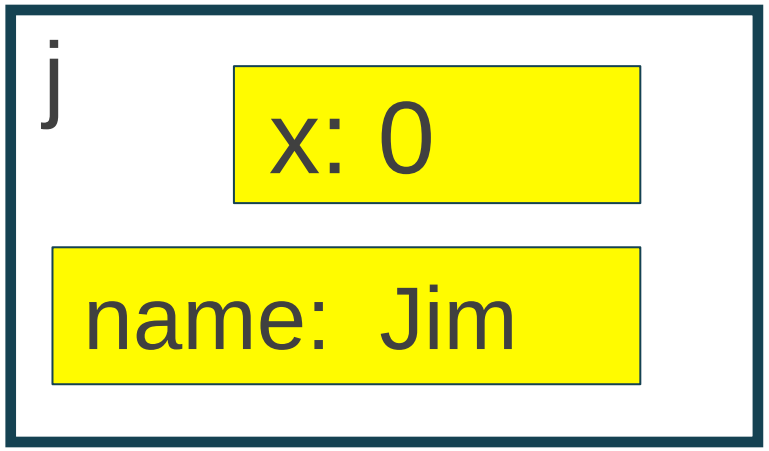
```
class PartyAnimal:
    x = 0
    name = ""
    def __init__(self, z):
      self.name = z
      print(self.name,"constructed")

    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()
```

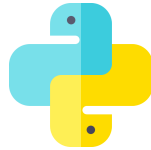




We have two independent instances

```
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
```



Multiple Instance

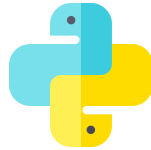
Demo Program: partyanimal4.py

```
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()
```

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

Inheritance

ACTIVITY



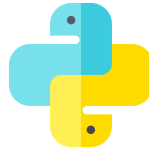
Inheritance

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.

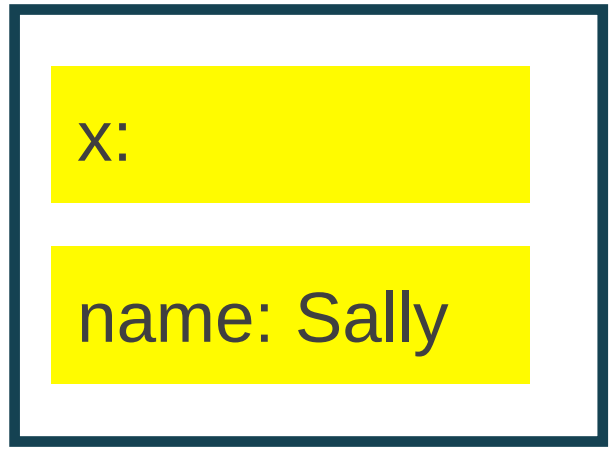
```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   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.

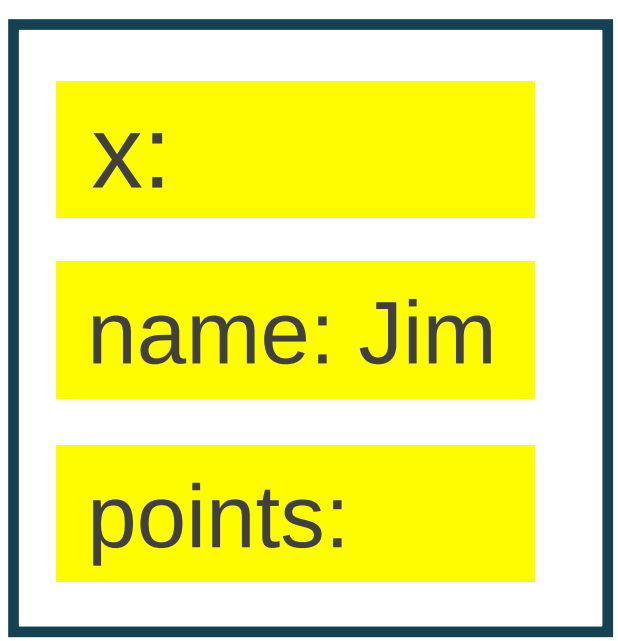
```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   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)
```

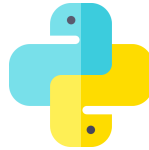
```
s = PartyAnimal("Sally")
s.party()

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)
```

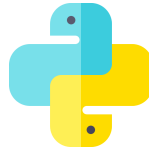
```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```





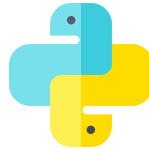
Party Animal 5 (I)

```
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)
```



Party Animal 5 (II)

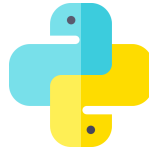
```
class FootballFan(PartyAnimal):
   points = 0
   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```



Party Animal 5 (III)

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 7
```



Definitions

Class - a template

Attribute – A variable within a class

Method - A function within a class

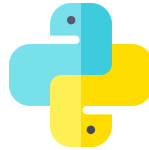
Object - A particular instance of a class

Constructor – Code that runs when an object is created

Inheritance - The ability to extend a class to make a new class.

Atom Class

LECTURE 3

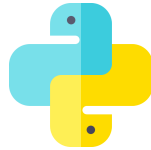


What is an Object?

A software item that contains variables and methods

Object Oriented Design focuses on

- Encapsulation:
  - dividing the code into a public interface, and a private implementation of that interface
- Polymorphism:
  - the ability to overload standard operators so that they have appropriate behavior based on their context
- Inheritance:
  - the ability to create subclasses that contain specializations of their parents

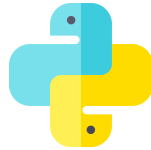


Namespaces

At the simplest level, classes are simply namespaces

```
class myfunctions:
    def exp():
        return 0


>>> math.exp(1)
2.71828...
>>> myfunctions.exp(1)
0
```

It can sometimes be useful to put groups of functions in their own namespace to differentiate these functions from other similarly named ones.



Python Classes

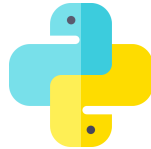
Python contains classes that define objects

- Objects are instances of classes

__init__ is the default constructor

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
```

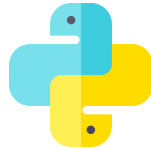
self refers to the object itself,
like *this* in Java.



Atom

Demo Program: atom1.py

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)

c = atom(6, 0, 1.0, 2.0)

print(c)
```



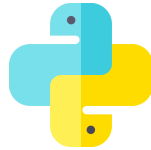
Example: Atom Class

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
    def symbol(self):    # a class method
        return Atno_to_Symbol[self.atno]
    def __repr__(self): # overloads printing
        return '%d %10.4f %10.4f %10.4f' %
                (self.atno, self.position[0],
                 self.position[1],self.position[2])
>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

Atom Symbols

Data File: atom.txt

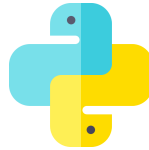
```
Atno_to_Symbol=["", \
"H", "He", "Li", "Be", "B", "C", "N", "O", "F", "Ne", \
"Na", "Mg", "Al", "Si", "P", "S", "Cl", "Ar", "K", "Ca", \
"Sc", "Ti", "V", "Cr", "Mn", "Fe", "Co", "Ni", "Cu", "Zn", \
"Ga", "Ge", "As", "Se", "Br", "Kr", "Rb", "Sr", "Y", "Zr", \
"Nb", "Mo", "Tc", "Ru", "Rh", "Pd", "Ag", "Cd", "In", "Sn", \
"Sb", "Te", "I", "Xe", "Cs", "Ba", "La", "Ce", "Pr", "Nd", \
"Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho", "Er", "Tm", "Yb", \
"Lu", "Hf", "Ta", "W", "Re", "Os", "Ir", "Pt", "Au", "Hg", \
"Tl", "Pb", "Bi", "Po", "At", "Rn", "Fr", "Ra", "Ac", "Th", \
"Pa", "U", "Np", "Pu","Am", "Cm", "Bk", "Cf", "Es", "Fm", \
"Md", "No", "Lr", "Rf", "Db", "Sg", "Bh", "Hs", "Mt", "Ds", \
"Rg", "Cn", "Nh", "Fl", "Mc", "Lv", "Ts", "Og"]
```



Atom

Demo Program: atom2.py

```
class atom:                                                    C
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
    def symbol(self):    # a class method
        return Atno_to_Symbol[self.atno]
    def __repr__(self): # overloads printing
        return '%d %10.4f %10.4f %10.4f' % \
                (self.atno, self.position[0], \
                 self.position[1],self.position[2])

c = atom(6, 0, 1.0, 2.0)
print(c.symbol())
```



Atom class

Overloaded the default constructor

Defined class variables (atno,position) that are persistent and local to the atom object

Good way to manage shared memory:

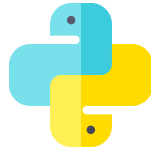
- instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
- much cleaner programs result

Overloaded the print operator

We now want to use the atom class to build molecules...

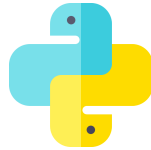
Molecule Class

LECTURE 4



Molecule Class

```
class molecule:
    def __init__(self,name='Generic'):
        self.name = name
        self.atomlist = []
    def addatom(self,atom):
        self.atomlist.append(atom)
    def __repr__(self):
        str = 'This is a molecule named %s\n' % self.name
        str = str+'It has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            str = str + `atom` + '\n'
        return str
```



Using Molecule Class

```
>>> mol = molecule('Water')
>>> at = atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print mol
This is a molecule named Water
It has 3 atoms
8  0.000 0.000 0.000
1  0.000 0.000 1.000
1  0.000 1.000 0.000
```

Note that the print function calls the atoms print function

- Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have one place to update.

Has_A Relationship

Demo Program: molecule.py

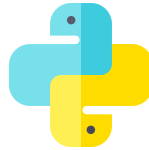
```
from atom2 import *    #using in C++, or import in Java

class molecule:
    def __init__(self,name='Generic'):
        self.name = name
        self.atomlist = []
    def addatom(self,atom):
        self.atomlist.append(atom)
    def __repr__(self):
        str = "his is a molecule named %s\n" % self.name
        str = str+"It has %d atoms\n" % len(self.atomlist)
        for atom in self.atomlist:
            str = str + atom.symbol() + " " + atom.__repr__() + '\n'
        return str
```

Has_A Relationship

Demo Program: molecule.py

```
def main():
    mol = molecule("Water")
    at = atom(8, 0.0, 0.0, 0.0)
    mol.addatom(at)
    mol.addatom(atom(1, 0.0, 0.0, 1.0))
    mol.addatom(atom(1, 0.0, 1.0, 0.0))
    print(mol)

if __name__ == "__main__":    # public static void main(String[] args) in Java
    main()
```

```
C
This is a molecule named Water
It has 3 atoms
O 8    0.0000    0.0000    0.0000
H 1    0.0000    0.0000    1.0000
H 1    0.0000    1.0000    0.0000
```



Inheritance

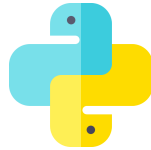
```
class qm_molecule(molecule):
    def addbasis(self):
        self.basis = []
        for atom in self.atomlist:
            self.basis = add_bf(atom,self.basis)
```

__init__, __repr__, and __addatom__ are taken from the parent class (molecule)

Added a new function addbasis() to add a basis set

Another example of code reuse

- Basic functions don't have to be retyped, just inherited
- Less to rewrite when specifications change

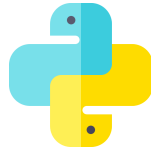


Overloading parent functions

```
class qm_molecule(molecule):
    def __repr__(self):
        str = 'QM Rules!\n'
        for atom in self.atomlist:
            str = str + `atom` + '\n'
        return str
```

Now we only inherit __init__ and addatom from the parent

We define a new version of __repr__ specially for QM



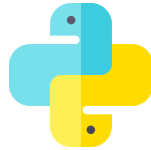
Adding to parent functions

Sometimes you want to extend, rather than replace, the parent functions.

```
class qm_molecule(molecule):
    def __init__(self,name="Generic",basis="6-31G**"):
        self.basis = basis
        molecule.__init__(self,name)
```

add additional functionality to the constructor

call the constructor for the parent function



Public and Private Data

Currently everything in atom/molecule is public, thus we could do something really stupid like

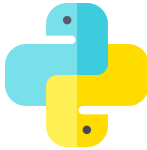
```
>>> at = atom(6,0.,0.,0.)
>>> at.position = 'Grape Jelly'
```

that would break any function that used at.poisition

We therefore need to protect the at.position and provide accessors to this data

- Encapsulation or Data Hiding
- accessors are "gettors" and "settors"

Encapsulation is particularly important when other people use your class



Public and Private Data, Cont.

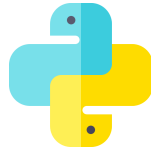
In Python anything with two leading underscores is private

```
__a, __my_variable
```

Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.

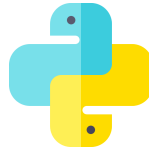
```
_b
```

- Sometimes useful as an intermediate step to making data private



Encapsulated Atom

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.__position = (x,y,z) #position is private
    def getposition(self):
        return self.__position
    def setposition(self,x,y,z):
        self.__position = (x,y,z) #typecheck first!
    def translate(self,x,y,z):
        x0,y0,z0 = self.__position
        self.__position = (x0+x,y0+y,z0+z)
```

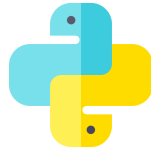


Why Encapsulate?

By defining a specific interface you can keep other modules from doing anything incorrect to your data

By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users

- Write to the Interface, not the the Implementation
- Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions



Classes that look like arrays

- Overload __getitem__(self,index) to make a class act like an array

```
class molecule:
    def __getitem__(self,index):
        return self.atomlist[index]

>>> mol = molecule('Water') #defined as before
>>> for atom in mol:        #use like a list!
        print atom
>>> mol[0].translate(1.,1.,1.)
```

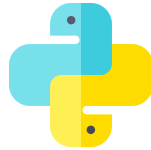
- Previous lectures defined molecules to be arrays of atoms.
- This allows us to use the same routines, but using the molecule class instead of the old arrays.
- An example of focusing on the interface!

Container Class

Demo: TestMolecule.py

```
def main():
    mol = molecule('Water')  # defined as before
    at = atom(8, 0.0, 0.0, 0.0)
    mol.addatom(at)
    mol.addatom(atom(1, 0.0, 0.0, 1.0))
    mol.addatom(atom(1, 0.0, 1.0, 0.0))
    print("Water:")
    for a in mol:  # use like a list!
        print("Atom", a.symbol(), ":")
        print("  ", a)
    print("")
    mol[0].translate(1., 1., 1.)
    print(mol[0])
if __name__ == "__main__":
    main()
```

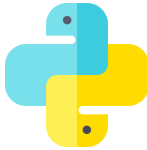
```
Water:
Atom O :
  8    0.0000    0.0000    0.0000
Atom H :
  1    0.0000    0.0000    1.0000
Atom H :
  1    0.0000    1.0000    0.0000

8    1.0000    1.0000    1.0000
```



Classes that look like functions

Overload __call__(self,arg) to make a class behave like a function

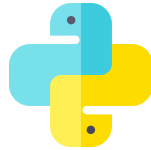
```
class gaussian:
    def __init__(self,exponent):
        self.exponent = exponent
    def __call__(self,arg):
        return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.)
>>> func(3.)
0.0001234
```



Callable Class (Functional Closure)

Demo Program: gaussian.py

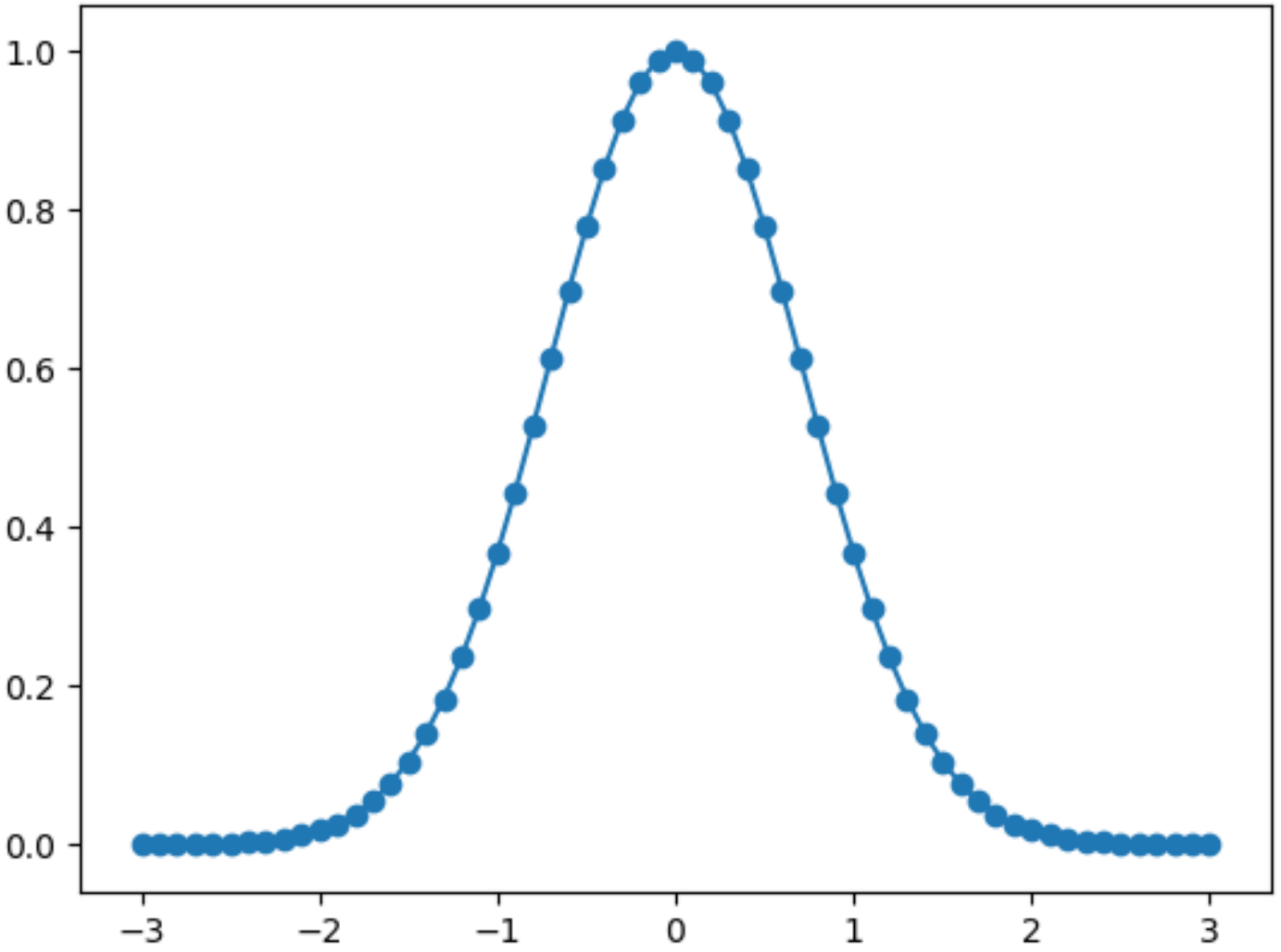
```
from pylab import *
class gaussian:
    def __init__(self, exponent):
        self.exponent = exponent
    def __call__(self, arg):
        return math.exp(-self.exponent*arg*arg)

x = np.linspace(-3.0, 3.0, 61)
y = []
for t in x:
    g = gaussian(1.0)
    y.append(g(t))
```

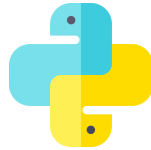


Callable Class (Functional Closure)

Demo Program: gaussian.py

```
y = array(y)
figure()
plot(x, y, '-')
scatter(x, y)
show()
```


1.0
0.8
0.6
0.4
0.2
0.0
−3
−2
−1
0
1
2
3



Other things to overload

- __setitem__(self,index,value)
  - Another function for making a class look like an array/dictionary

```
a[index] = value
```

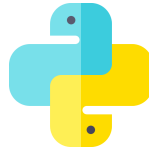
- __add__(self,other)
  - Overload the "+" operator

```
molecule = molecule + atom
```

- __mul__(self,number)
  - Overload the "*" operator

```
zeros = 3*[0]
```

- __getattr__(self,name)
  - Overload attribute calls
  - We could have done atom.symbol() this way



Other things to overload, cont.

- __del__(self)
  - Overload the default destructor

```
del temp_atom
```

- __len__(self)
  - Overload the len() command

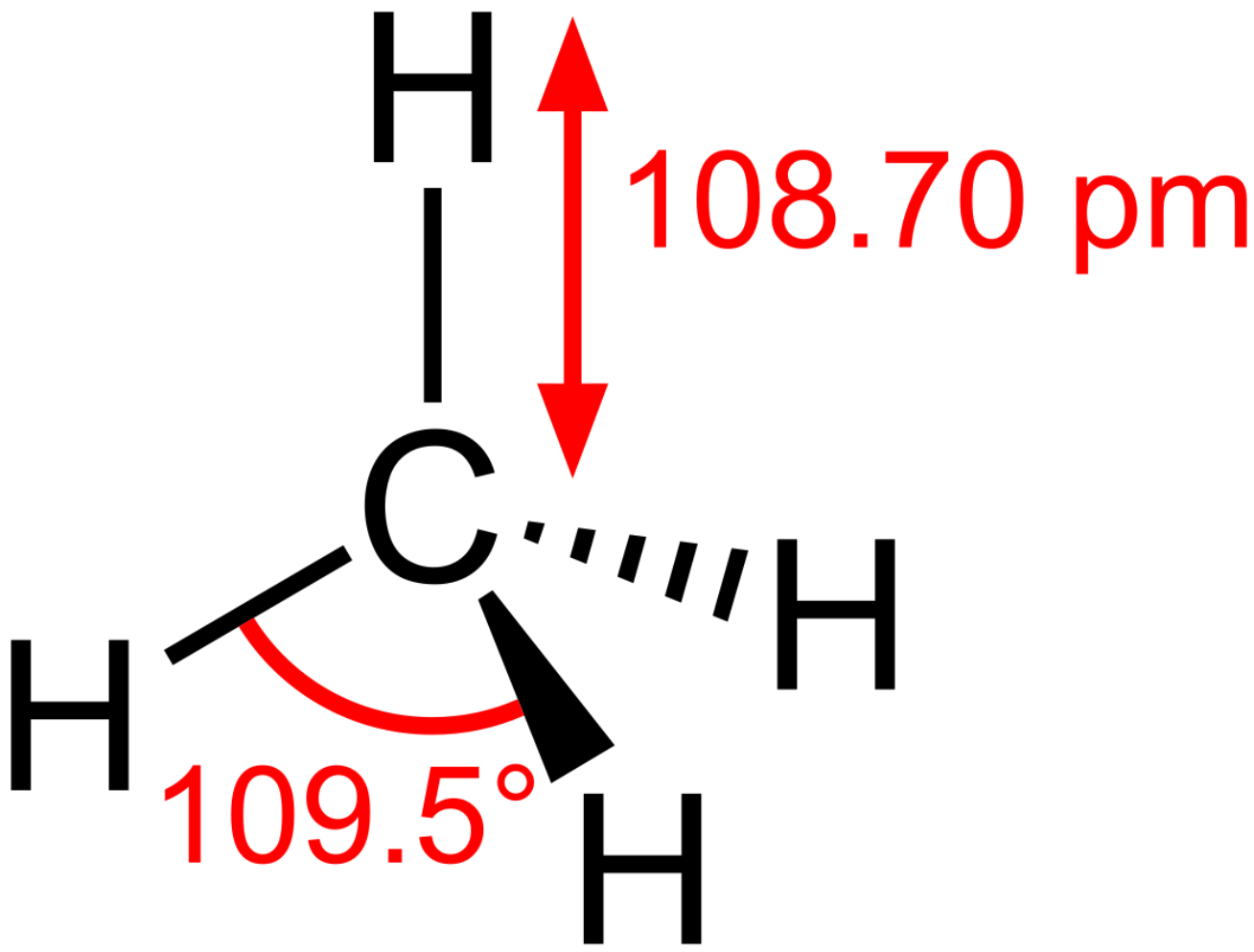
```
natoms = len(mol)
```

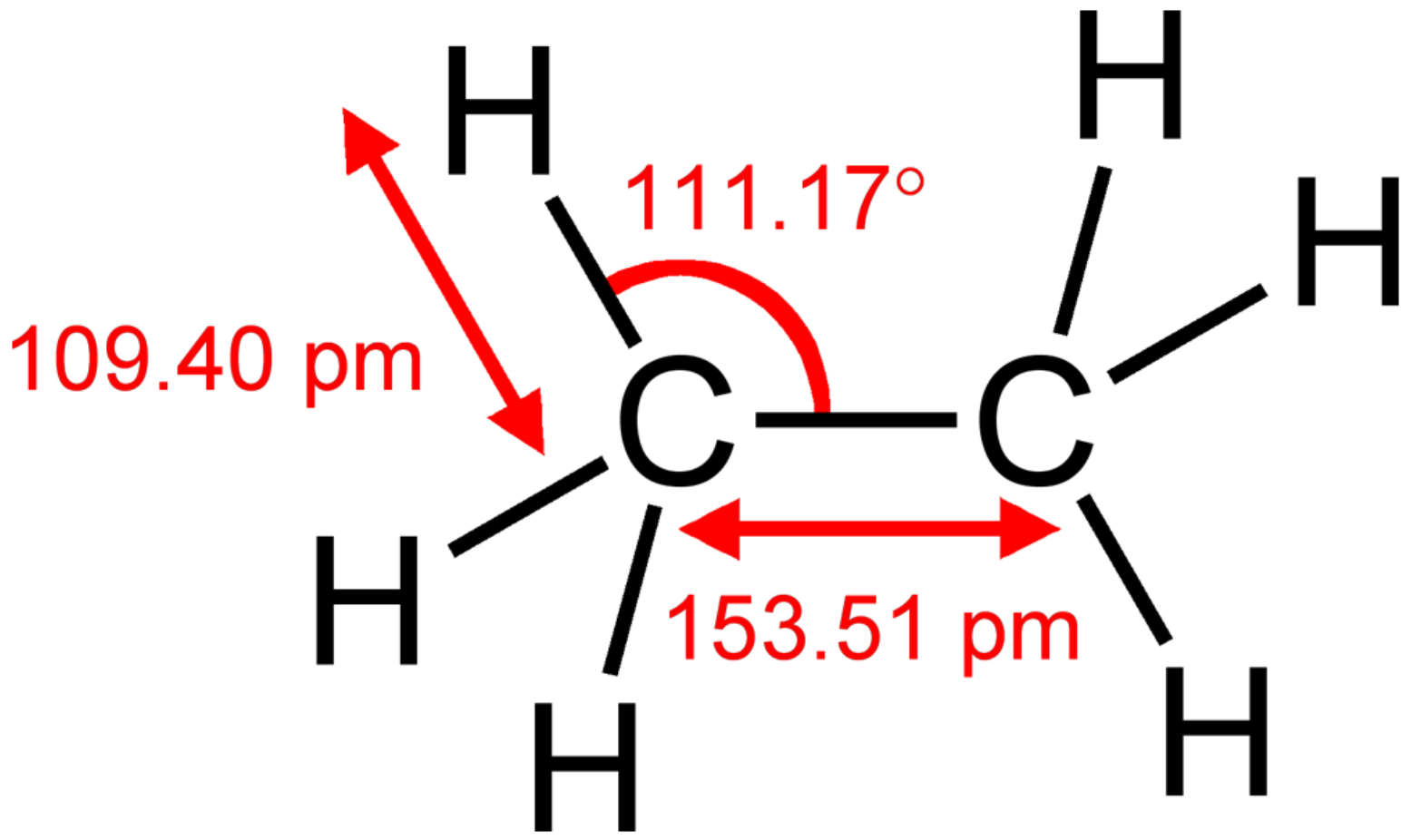
- __getslice__(self,low,high)
  - Overload slicing

```
glycine = protein[0:9]
```

TestMethane Class

LECTURE 5


H
108.70 pm
C
H
H
109.5°
H


H
H
H
H
H
H
C
C
111.17°
109.40 pm
153.51 pm

```
import molecule as me
import atom as at
def main():
    methane = me.molecule("Methane")
    methane.addatom(at.atom(6, 0, 0, 0))
    methane.addatom(at.atom(1, 1, 0, 0))
    methane.addatom(at.atom(1, 0, 1, 0))
    methane.addatom(at.atom(1, 0, 1, 1))
    methane.addatom(at.atom(1, -1, 0, 0))

    print("Indexing: ")
    print("methane[0]=%s" % methane[0])
    print("methane[2]=%s" % methane[2])
    print()
    print("methane list:\n %s" % methane)
    print()
```

```
print("Setting: ")
methane[4] = at.atom(1, 0, -1, 0)
print("methane list:\n %s" % methane)
print()
print("Length: ")
print("The length of Methane is : %d\n" % len(methane))
print("For-loop:")
for i in range(len(methane)):  # methane is not iterable
    print(methane[i])
print()
print("Slicing:")
for a in methane[2:4]:
    print(a)
print()
```

```
    ethane = me.molecule("Ethane")
    ethane.addatom(at.atom(6, 0, 0, 0))
    ethane.addatom(at.atom(1, 1, 0, 0))
    ethane.addatom(at.atom(1, 0, 0, 1))
    ethane.addatom(at.atom(1, 0, -1, 0))
    ethane.addatom(at.atom(6, 0, 1, 0))
    ethane.addatom(at.atom(1, 1, 1, 0))
    ethane.addatom(at.atom(1, 0, 1, 1))
    ethane.addatom(at.atom(1, 0, 2, 0))
    print("Comparison:")
    print("Ethan:\n", ethane)
    print()
    print("Methane >  Ethane = %s" % (methane > ethane))
    print("Methane >= Ethane = %s" % (methane >= ethane))
    print("Methane <  Ethane = %s" % (methane < ethane))
    print("Methane <= Ethane = %s" % (methane <= ethane))
    print("Methane == Ethane = %s" % (methane == ethane))
    print("Methane != Ethane = %s" % (methane != ethane))

if __name__ == "__main__":
    main()
```

```
Indexing:
methane[0]=6     0.0000     0.0000     0.0000
methane[2]=1     0.0000     1.0000     0.0000

methane list:
 This is a molecule named Methane
It has 5 atoms
C 6     0.0000     0.0000     0.0000
H 1     1.0000     0.0000     0.0000
H 1     0.0000     1.0000     0.0000
H 1     0.0000     1.0000     1.0000
H 1    -1.0000     0.0000     0.0000

Setting:
methane list:
 This is a molecule named Methane
It has 5 atoms
C 6     0.0000     0.0000     0.0000
H 1     1.0000     0.0000     0.0000
H 1     0.0000     1.0000     0.0000
H 1     0.0000     1.0000     1.0000
H 1     0.0000    -1.0000     0.0000
```

```
Length:
The length of Methane is : 5
For-loop:
6     0.0000     0.0000     0.0000
1     1.0000     0.0000     0.0000
1     0.0000     1.0000     0.0000
1     0.0000     1.0000     1.0000
1     0.0000    -1.0000     0.0000
Slicing:
1     0.0000     1.0000     0.0000
1     0.0000     1.0000     1.0000
Comparison:
Ethan:
 This is a molecule named Ethane
It has 8 atoms
C 6     0.0000     0.0000     0.0000
H 1     1.0000     0.0000     0.0000
H 1     0.0000     0.0000     1.0000
H 1     0.0000    -1.0000     0.0000
C 6     0.0000     1.0000     0.0000
H 1     1.0000     1.0000     0.0000
H 1     0.0000     1.0000     1.0000
H 1     0.0000     2.0000     0.0000
```

```
Methane >  Ethane = False
Methane >= Ethane = False
Methane <  Ethane = True
Methane <= Ethane = True
Methane == Ethane = False
Methane != Ethane = True
```