Lab 11 – Classes

Classes

Before we go into Classes, lets learn about Python's scope rules. You need to know how scopes and namespaces work to understand what is going on with Classes.

Some definitions first:

A ***namespace*** is a mapping from names to objects. A ***namespace*** is a space or region, within a program, where a name (variable, class etc) is valid.

It is a system for making names unique.

Everybody knows a namespacing system from daily life, i.e. We actually use this idea in everyday life. Suppose you work in a big company and there is a colleague called Joe. In the accounts department there is another guy called Joe who you see occasionally but not often. In that case you refer to your colleague as "Joe" and the other one as "Joe in Accounts".

They came about because early programming languages (like BASIC) only had *Global Variables*, that is, ones which could be seen throughout the program - even inside functions.
This made maintenance of large programs difficult since it was easy for one section of a program to modify a variable without other parts of the program realizing it.

As you might have heard me say a few times, everything in Python - literals, lists, functions, classes, etc. - is an object.

Such a "name-to-object" mapping allows us to access an object by a name that we've assigned to it.

E.g., if we make a simple string assignment:

string_1 = "Hello string", we created a reference to the "Hello string" **object**, and henceforth we can access via its variable **name** string_1.

Another term used to describe a namespace is ***scope***.

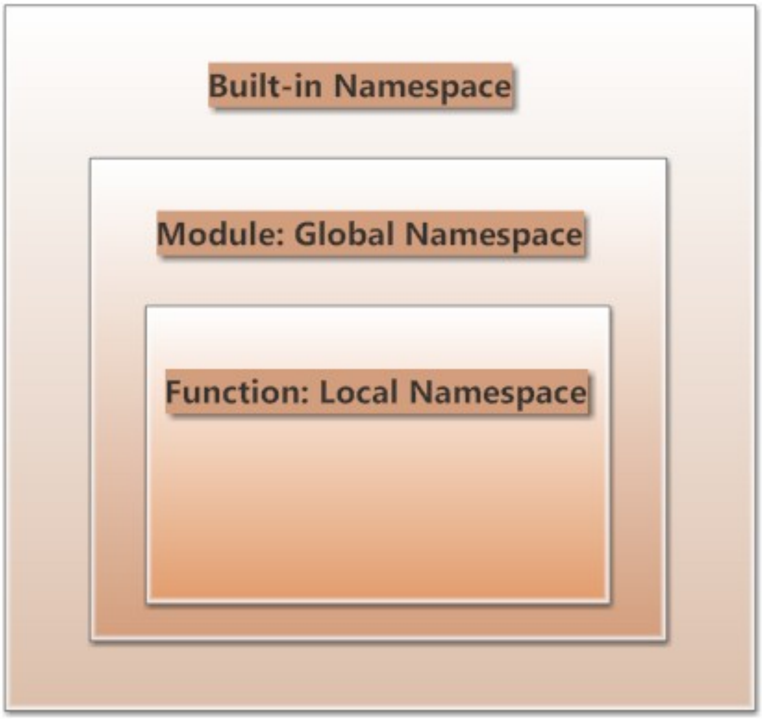
The scope of a name is the extent of a program whereby that name can be unambiguously used, for example inside a function or a module. A textual region of a Python program where a namespace is directly accessible

A name's namespace is exactly the same as it's scope.

There are a few very subtle differences between the terms but only a Computer Scientist would argue with you, and for our purposes namespace and scope are identical.

In Python there are a total of three possible namespaces (or *scopes*):

1. **Built in scope** - names defined within Python itself, these are always available from anywhere in your program.
   - created when the Python interpreter starts up, and is never deleted
2. **Module scope** - names defined, and therefore visible within a file or module, confusingly this is referred to as *global* scope in Python whereas global normally means visible from *anywhere* in other languages.
   - created when the module definition is read in; normally, module namespaces also last until the interpreter quits
3. **Local scope** - names defined within a function or a class method
   - created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function



16

Let's take a look at a piece of code that includes examples of all of these:

```
def square(x):
   return x*x

data = int(raw_input('Type a number to be squared: '))
print data, 'squared is: ', square(data)
```

The following table lists each name and the scope to which it belongs:

Name	Namespace
square	Module/global
x	Local (to square)
data	Module/global
int	Built-in
raw_input	Built-in

Note that we don't count **def** and **print** as names because they are **keywords** or commands forming part of the language, if you try to use them as names of variables of functions you will get an error.

Accessing Names outside the Current Namespace

Here we look in more detail at exactly how Python locates names, even when the names we are using are not in the immediate namespace. It is resolved as follows, Python will look:

1. within it's local namespace (the current function),
2. within the module scope (the current file),
3. the built-in scope.

Remember:

A module is simply a file containing Python code. This code can be in the form of Python classes, functions, or just a list of names.

Each module gets it’s own global namespaces. So you **can’t** have two classes or two functions in the same module with the same name as they share the namespace of the module

However each namespace is also completely isolated. So two modules can have the same names within them.

You can have a module called Integer and a module called FloatingPoint and both could have a function named add(). Once you import the module into your script, you can access the names by prefixing them with the module name: FloatingPoint.add() and Integer.add().

Whenever you run a simple Python script, the interpreter treats it as module called __main__, which gets its own namespace. The builtin functions that you would use also live in a module called __builtin__ and have their own namespace.

Classes and namespaces have special interactions. A class creates a new ***local*** namespace where all its attributes are defined. Attributes are data or functions.

The only way for a class' methods (functions inside classes) to access it's own variables or functions (as names) is to use a reference to itself. This means that the first argument of a method must be a 'self' parameter, if it is to access other class attributes. You need to do this because that while the module has a global namespace, the class itself does **not**.

What is a Class?

Data structures like *lists* and *strings* are extremely useful, but sometimes they aren't enough to represent something you're trying to implement.

For example, let's say we needed to keep track of a bunch of pets.

We could represent a pet using a *list* by specifying the first element of the list as the pet's name and the second element of the list as the pet's species.
This is very arbitrary and nonintuitive, however — how do you know which element is supposed to be which?

Classes give us the ability to create more complicated data structures that contain arbitrary content.
We can create a **Pet** class that keeps track of the name and species of the pet in usefully named attributes called **name** and **species**, respectively.

What is an Instance?

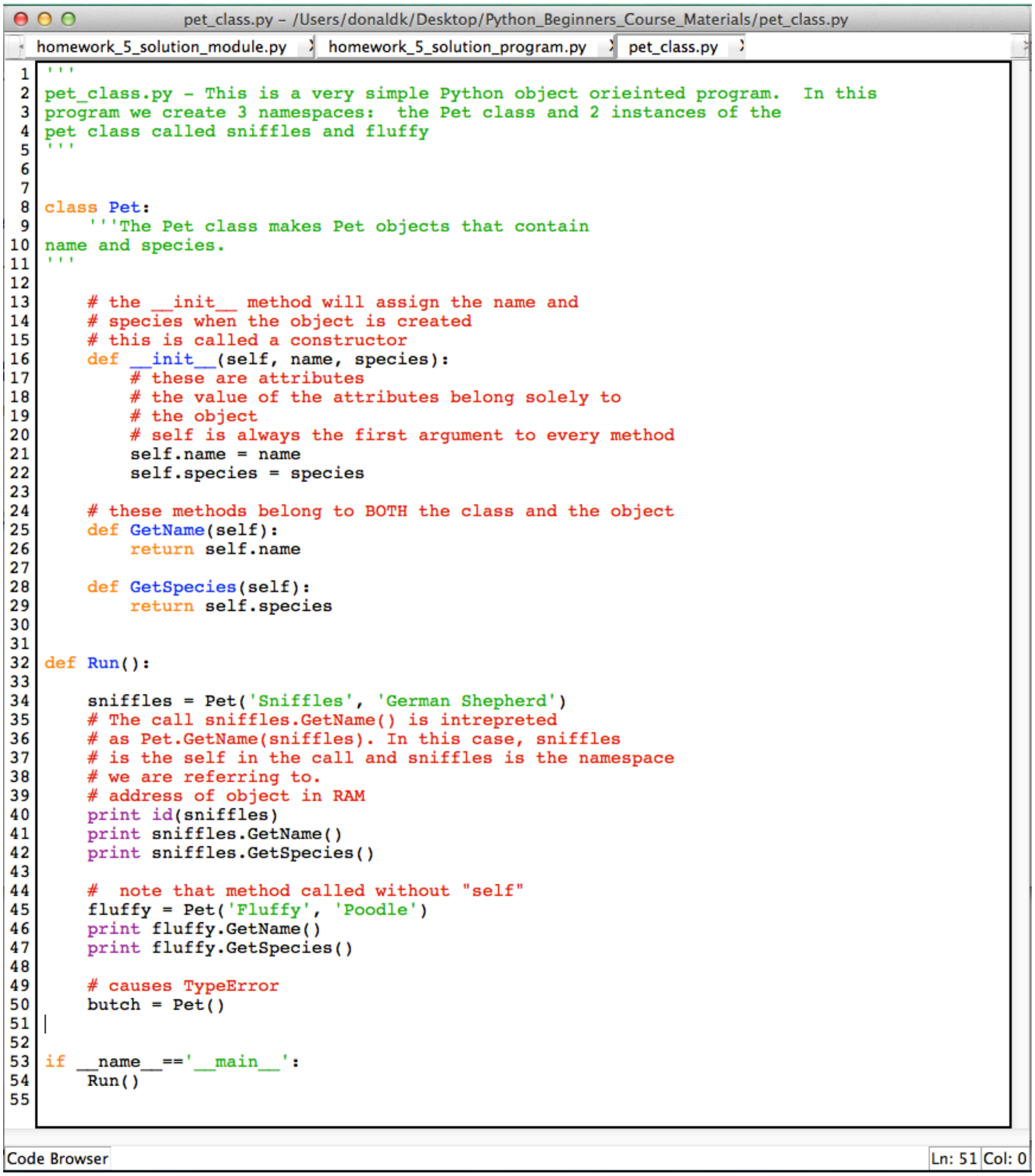
Before we get into creating a class itself, we need to understand an important distinction.

A **class** is something that just contains **structure** — it defines how something should be laid out or structured, but doesn't actually fill in the content.
For example, a Pet class may say that a pet needs to have a name and a species, but it will not actually say what the pet's name or species is.

This is where ***instances*** come in.
An instance is a specific copy of the class that **does** contain all of the content.
For example, if I create a pet polly, with name "Polly" and species "Parrot",
then polly is an **instance** of Pet.

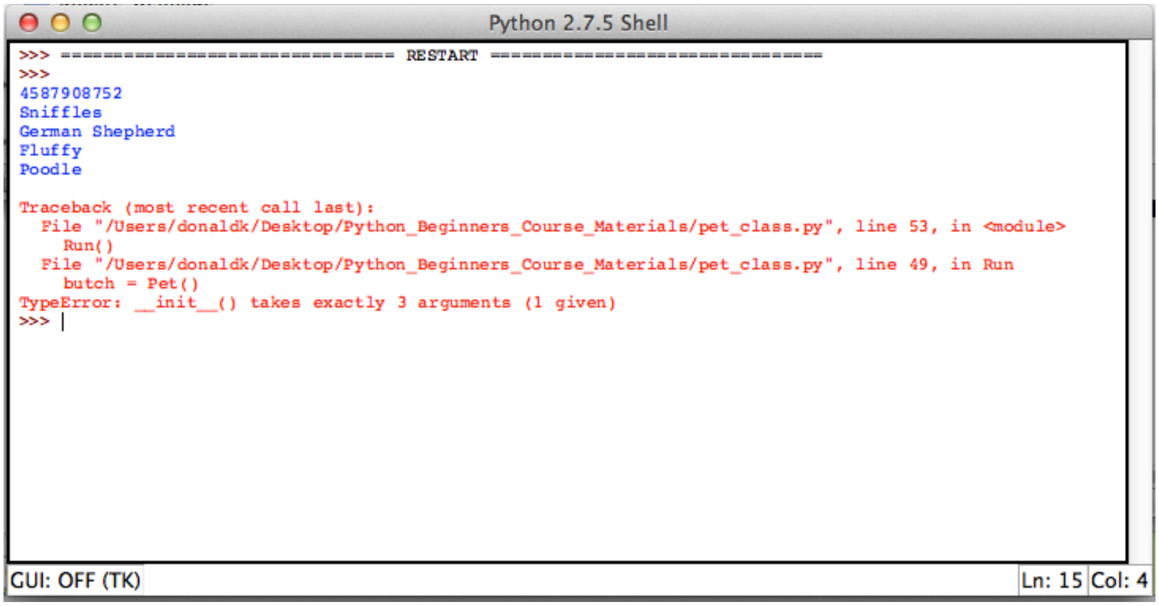
This can sometimes be a very difficult concept to master, so let's look at it from another angle. Let's say that the government has a particular tax form that it requires everybody to fill out. Everybody has to fill out the same *type* of form, but the content that people put into the form differs from person to person.
A **class** is like the **form**: it specifies what content should exist. Your copy of the form with your **specific information** if like an **instance** of the class: it specifies what the content actually is.

Now that we know the difference between a class and an instance of the class, let's look a real class:



pet_class.py – /Users/donaldk/Desktop/Python_Beginners_Course_Materials/pet_class.py
homework_5_solution_module.py
homework_5_solution_program.py
pet_class.py
Code Browser
Ln: 51
Col: 0

```python
'''
pet_class.py - This is a very simple Python object orieinted program.  In this
program we create 3 namespaces:  the Pet class and 2 instances of the
pet class called sniffles and fluffy
'''


class Pet:
    '''The Pet class makes Pet objects that contain
name and species.
'''

    # the __init__ method will assign the name and
    # species when the object is created
    # this is called a constructor
    def __init__(self, name, species):
        # these are attributes
        # the value of the attributes belong solely to
        # the object
        # self is always the first argument to every method
        self.name = name
        self.species = species

    # these methods belong to BOTH the class and the object
    def GetName(self):
        return self.name

    def GetSpecies(self):
        return self.species


def Run():

    sniffles = Pet('Sniffles', 'German Shepherd')
    # The call sniffles.GetName() is intrepreted
    # as Pet.GetName(sniffles). In this case, sniffles
    # is the self in the call and sniffles is the namespace
    # we are referring to.
    # address of object in RAM
    print id(sniffles)
    print sniffles.GetName()
    print sniffles.GetSpecies()

    #  note that method called without "self"
    fluffy = Pet('Fluffy', 'Poodle')
    print fluffy.GetName()
    print fluffy.GetSpecies()

    # causes TypeError
    butch = Pet()


if __name__=='__main__':
    Run()

```



```
>>> ============================== RESTART ==============================
>>> 
4587908752
Sniffles
German Shepherd
Fluffy
Poodle

Traceback (most recent call last):
  File "/Users/donaldk/Desktop/Python_Beginners_Course_Materials/pet_class.py", line 53, in <module>
    Run()
  File "/Users/donaldk/Desktop/Python_Beginners_Course_Materials/pet_class.py", line 49, in Run
    butch = Pet()
TypeError: __init__() takes exactly 3 arguments (1 given)
>>> |
```

Exercises – Lab 11:

1. Write a module called die.py. In this module create a function called RollDie( ) that will roll a 6 sided die and will return the value. Make sure to add all necessary code to make this a module that you can also execute from the commandline. Add a Run( ) function that will allow for testing this module and its function.
2. Write a class called Die (save file as die_class.py) that will import the module die.py and will have one method called RollDice( ) that will roll the die calling the function in the die module. Add all code necessary to allow for this Class to be executed as a python script. Write a testing Run( ) function that will test the rolling of a die 4 times.
3. Write a Python Program (save it as dice_game_class.py) that will import the module (remember Class can be module) created in 2. This program will have a function that will ask the players how many players are playing. The program will then create an instance of a class defined in this program called Game that has a constructor that has 3 attributes: num_players, and 2 instances of object defined in 2. above. It will then call a method of the class called PlayGame( ) that will loop over the number of players and roll the two die attributes created in the constructor. PlayGame( ) will call another method of the Class called Roll( ) that will return a list of the values of the 2 dice. The function in the program will then call method GetTotal( ) that will sum the players results and create another list of lists that has as elements [sum, player number]. The programs function will then call class method DetermineWinner( ) that will sort the list created in GetTotal( ). Finally, the program function will call class method PrintWinners( ) to print out the players in order of best score to worse.

Lab 12 – Inheritance

Inheritance

One of the biggest advantages of **Object Oriented Programming** (OOP) is that you can take a base class and add complexity to it while leaving the original base class as it is. This is the basic idea behind **inheritance**.

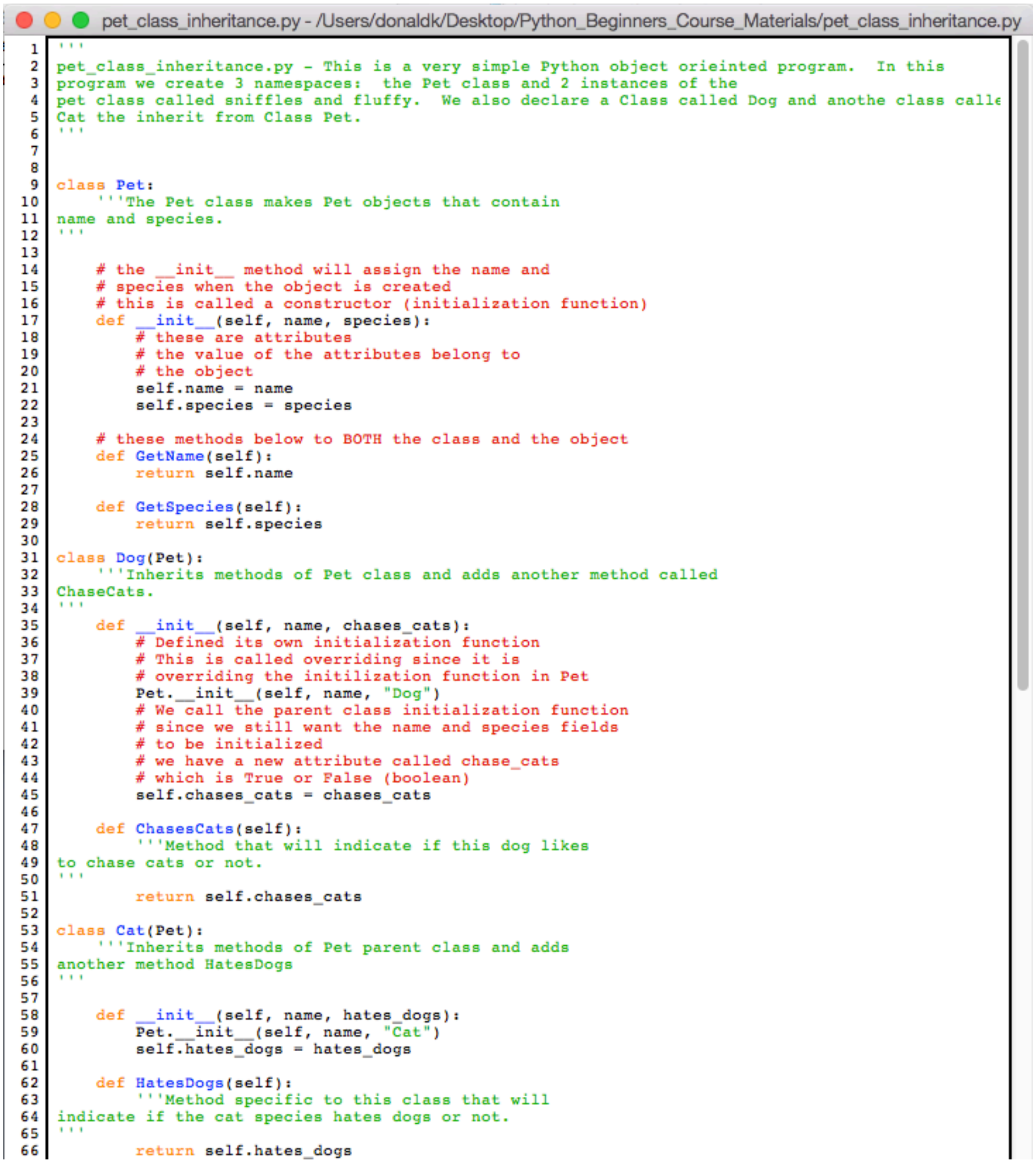
Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class.

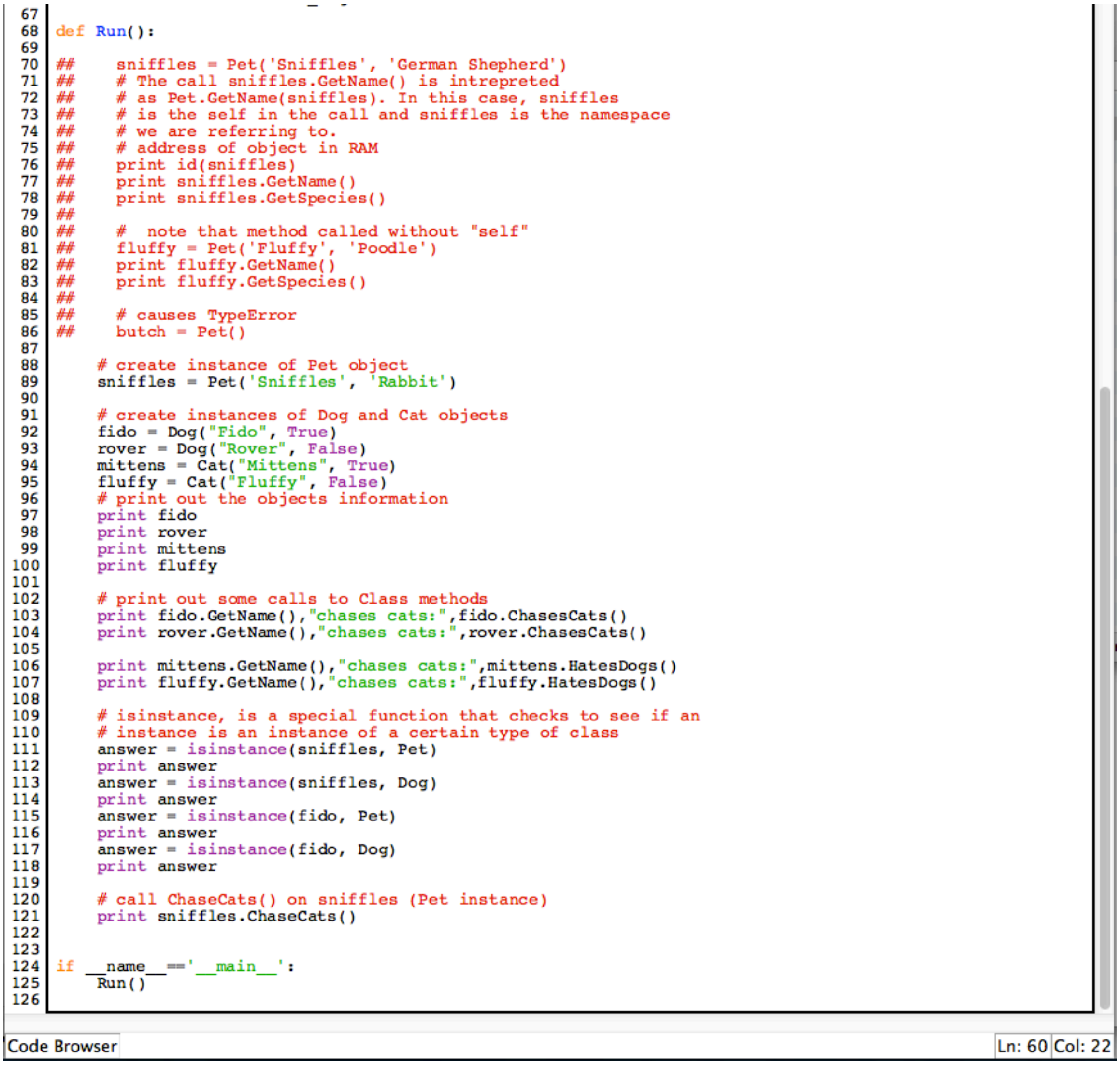
A child class can also override data members and methods from the parent.

Lets look at our class Pets from the previous lab.

Some pets are dogs and most dogs like to chase cats, and maybe we want to keep track of which dogs do or do not like to chase cats. Birds are also pets but they generally don't like to chase cats.


pet_class_inheritance.py - /Users/donaldk/Desktop/Python_Beginners_Course_Materials/pet_class_inheritance.py

```python
'''
pet_class_inheritance.py - This is a very simple Python object orieinted program.  In this
program we create 3 namespaces:  the Pet class and 2 instances of the
pet class called sniffles and fluffy.  We also declare a Class called Dog and anothe class calle
Cat the inherit from Class Pet.
'''


class Pet:
    '''The Pet class makes Pet objects that contain
name and species.
'''

    # the __init__ method will assign the name and
    # species when the object is created
    # this is called a constructor (initialization function)
    def __init__(self, name, species):
        # these are attributes
        # the value of the attributes belong to
        # the object
        self.name = name
        self.species = species

    # these methods below to BOTH the class and the object
    def GetName(self):
        return self.name

    def GetSpecies(self):
        return self.species

class Dog(Pet):
    '''Inherits methods of Pet class and adds another method called
ChaseCats.
'''
    def __init__(self, name, chases_cats):
        # Defined its own initialization function
        # This is called overriding since it is
        # overriding the initilization function in Pet
        Pet.__init__(self, name, "Dog")
        # We call the parent class initialization function
        # since we still want the name and species fields
        # to be initialized
        # we have a new attribute called chase_cats
        # which is True or False (boolean)
        self.chases_cats = chases_cats

    def ChasesCats(self):
        '''Method that will indicate if this dog likes
to chase cats or not.
'''
        return self.chases_cats

class Cat(Pet):
    '''Inherits methods of Pet parent class and adds
another method HatesDogs
'''

    def __init__(self, name, hates_dogs):
        Pet.__init__(self, name, "Cat")
        self.hates_dogs = hates_dogs

    def HatesDogs(self):
        '''Method specific to this class that will
indicate if the cat species hates dogs or not.
'''
        return self.hates_dogs
```



```

def Run():

##    sniffles = Pet('Sniffles', 'German Shepherd')
##    # The call sniffles.GetName() is intrepreted
##    # as Pet.GetName(sniffles). In this case, sniffles
##    # is the self in the call and sniffles is the namespace
##    # we are referring to.
##    # address of object in RAM
##    print id(sniffles)
##    print sniffles.GetName()
##    print sniffles.GetSpecies()
##
##    #  note that method called without "self"
##    fluffy = Pet('Fluffy', 'Poodle')
##    print fluffy.GetName()
##    print fluffy.GetSpecies()
##
##    # causes TypeError
##    butch = Pet()

    # create instance of Pet object
    sniffles = Pet('Sniffles', 'Rabbit')

    # create instances of Dog and Cat objects
    fido = Dog("Fido", True)
    rover = Dog("Rover", False)
    mittens = Cat("Mittens", True)
    fluffy = Cat("Fluffy", False)
    # print out the objects information
    print fido
    print rover
    print mittens
    print fluffy

    # print out some calls to Class methods
    print fido.GetName(),"chases cats:",fido.ChasesCats()
    print rover.GetName(),"chases cats:",rover.ChasesCats()

    print mittens.GetName(),"chases cats:",mittens.HatesDogs()
    print fluffy.GetName(),"chases cats:",fluffy.HatesDogs()

    # isinstance, is a special function that checks to see if an
    # instance is an instance of a certain type of class
    answer = isinstance(sniffles, Pet)
    print answer
    answer = isinstance(sniffles, Dog)
    print answer
    answer = isinstance(fido, Pet)
    print answer
    answer = isinstance(fido, Dog)
    print answer

    # call ChaseCats() on sniffles (Pet instance)
    print sniffles.ChaseCats()


if __name__=='__main__':
    Run()

```

Lab 12 – Exercises

1. Import the Game class created in Lab 11 (dice_game_class.py) so that you can use it to create a new class called CrapsDice that inherits Game class. Add this new class to the same file and save it with a new name (e.g. dice_game_class_inheritance.py). This new CrapsDice class will have an additional method called: Roll( ). Roll( ) will return the value of both die if the roll was a successful hard ways roll and will return zero for both die if it was not. A hard way roll is one where the shooter bets that both dice show identical values. This Roll( ) method will override the one from Game class. Modify the StartGame( ) function (this is the program function that asks for number of players and creates instances of objects) in the program to create an instance of CrapsDice( ) and pass to it the number of players that the user input. All the other calls should be the same.