Python

Classes

Today's Objectives

- To learn about the principles of OOP
  - (Object-Oriented Programming)
  - Encapsulation
  - Abstraction

- To learn about classes (in Python)
  - How they work at a high level
  - Cool stuff like inheritance and overriding

Procedural vs OOP

Procedural Programming

- Procedural programming uses:
  - Data structures (like integers, strings, lists)
  - Functions (like **`printVendingMachine()`**)

- In procedural programming, information must be passed to the function
  - Functions and data structures are <u>not</u> linked

Object-Oriented Programming (OOP)

- Object-Oriented programming uses
  - Classes!

- Classes combine the data and their relevant functions into one entity
  - The data types we use are actually classes!
  - Strings have built-in functions like **`lower()`**, **`join()`**, **`strip()`**, etc.

Procedural vs OOP

- Procedural
  - Calculate the area of a circle given the specified radius
  - Sort this class list given a list of students
  - Calculate the student’s GPA given a list of courses

- Object-Oriented
  - Circle, you know your radius, what is your area?
  - Class list, sort your students
  - Transcript, what is this student’s GPA?

Abstraction and Encapsulation

Abstraction

- All programming languages provide some form of ***abstraction***
  - Hide the details of implementation from the user
  - User doesn't need to know how an engine works in order to drive a car
  - Do you know <u>how</u> append() works?
    - No, but you can still use it!

Image from wikimedia.org

Encapsulation

- ***Encapsulation*** is a form of information hiding and abstraction
  - Data and functions that act on that data are located in the same place (inside a class)

- Class methods are called ***on*** a class object
  - They know everything about that object already
- Remember, classes contain code and data!

Classes

What is a Class?

- According to the dictionary:
  - A set, collection, group, or configuration containing members regarded as **having** certain **attributes or traits in common**
- According to OOP principles:
  - A group of objects with **similar properties**, **common behavior**, **common relationships** with other objects, and **common semantics**

Class Vocabulary

- A ***class*** is a special data type which defines how to build a certain kind of object

- ***Instances*** are objects that are created which follow the definition given inside of the class
  - Every instance of a class has both ***attributes*** and ***methods***

"Method" is just another word for function, often used when talking about classes

Blueprints

- Classes are “blueprints” for creating objects
  - A dog class to create dog objects
  - A car class to create car objects

- The blueprint defines
  - The class’s attributes (properties)
    - As variables
  - The class’s behaviors (functions)
    - As methods

Objects

- Each instance of a class is called an ***object*** of that class type

- You can create as many instances of a class as you want
  - Just like a “regular” data type, like **`int`** or **`float`**
  - There can be more than one dog or one car
    - Multiple dog objects, multiple car objects

Creating a Class

Defining a Class

- When we create a new class, we must define its ***attributes*** and ***methods***
  - Once we’ve done that, we can create ***instances***

- Think about it in terms of parts of speech
  - Objects are nouns (“my dog”, “Ali’s car”)
  - Attributes are adjectives (“big”, “brown”, “old”)
  - Methods are verbs (“speak”, “reverse”, “play”)

Built-In Functions

- Classes have two important built-in functions
  - Have double underscores on either side of name

`__init__`

  - Constructor for the class
  - Initializes and creates attributes

`__str__`

  - Defines how to turn an instance into a string
  - Used when we call **`print()`** with an instance

Familiar Objects

- Objects like integers, lists, and Booleans also have constructors and string representations

- To create an integer, we could use

```
newInt = int()
```

- To print a list, we could use

```
print(myList)
```

  – This will print it out with square brackets

Constructors

- Every class must have a ***constructor***
  - How a new object is created

- A class constructor will
  - Supply default values for attributes
  - Initialize the object and its attributes

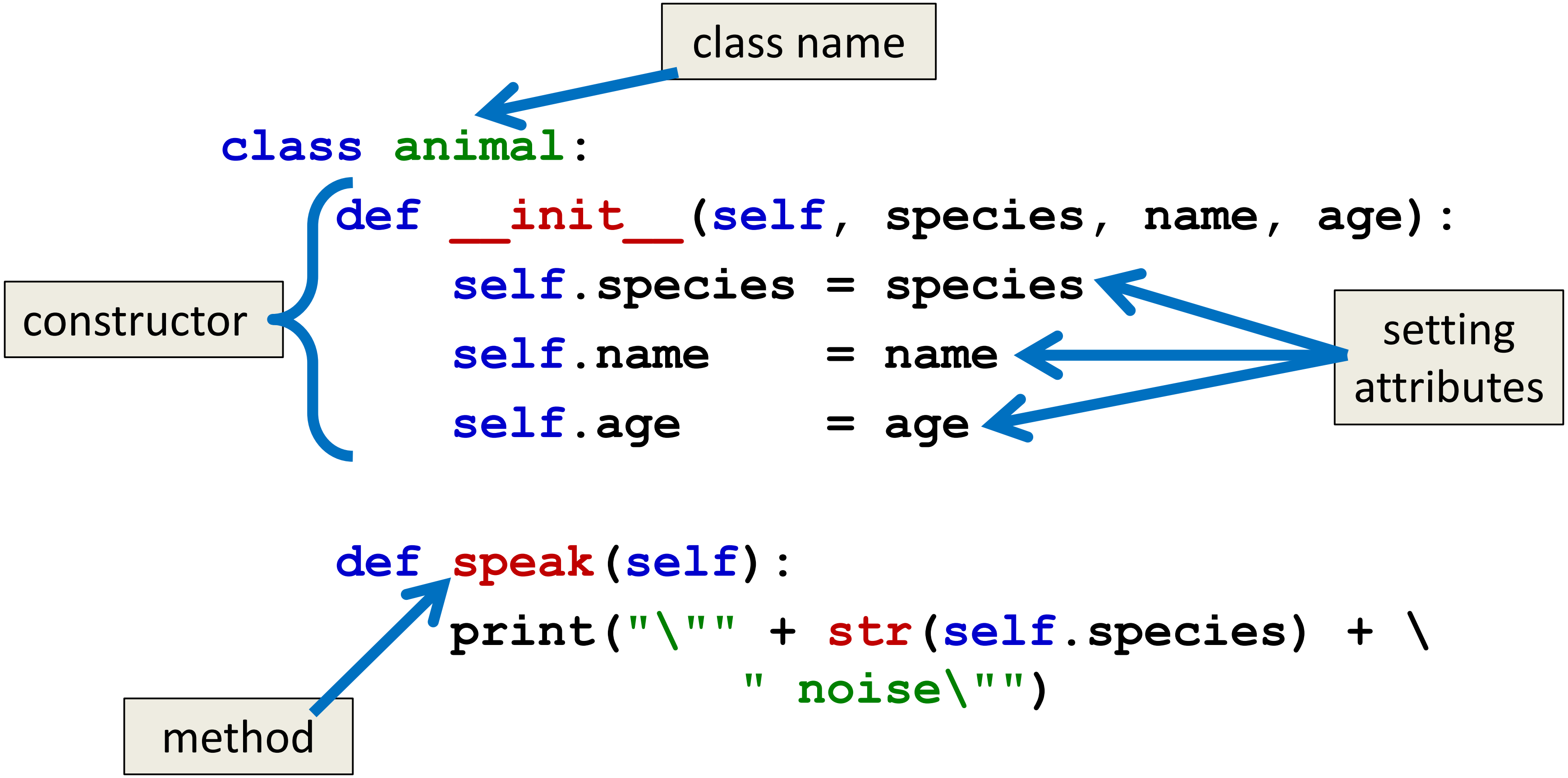
- Constructors are automatically called when an object is created

Class Definition Example

```
class animal:
    def __init__(self, species, name, age):
        self.species = species
        self.name    = name
        self.age     = age

    def speak(self):
        print("\"" + str(self.species) + \
                  " noise\"")
```

Class Definition Example



```
class animal:
    def __init__(self, species, name, age):
        self.species = species
        self.name    = name
        self.age     = age

    def speak(self):
        print("\"" + str(self.species) + \
                  " noise\"")
```

Class Definition Example

```
class animal:
    def __init__(self, species, name, age):
        self.species = species
        self.name    = name
        self.age     = age

    def speak(self):
        print("\"" + str(self.species) + \
                  " noise\"")
```

Notice that everything is <u>indented</u> under the “class animal:” line of code

Class Usage Example

- To create an instance of a class (a class object), use the class name, pass it the values for the attributes, and assign to a variable

```
# create an animal object (species: sheep)
variable1 = animal("sheep", "Dolly", 6)

# create your own animal object!
variable2 = animal("dog", "Fido", 7)
```

The **`self`** Variable

- The **`self`** variable is how we refer to the current instance of the class
  - In **`__init__`**, **`self`** refers to the object that is currently being created
  - In other methods, **`self`** refers to the instance the method was called on

```
def speak(self):
    print("\"" + str(self.species) + " noise\"")
```

Fraction example

```
class Fraction:
    def __init__(self, nominator = 0, denominator = 1):
        self.nominator = nominator
        self.denominator = denominator
    def get_nominator(self):
        return self.nominator
    def get_denominator(self):
        return self.denominator
```

Fraction example

```
def evaluate(self):
    return self.nominator/self.denominator
def set_value(self,n,d):
    self.nominator = n
    self.denominator = d
def show(self):
    print(str(self.nominator) + "/" + str(self.denominator))
```

Fraction Example

```
>>> import frac
>>> f1 = frac.Fraction()
>>> f2 = frac.Fraction(3,5)
>>> f1.get_nominator()
0
>>> f1.get_denominator()
1
>>> f2.get_nominator()
3
>>> f2.get_denominator()
5
```

Fraction example

```
>>> f2.evaluate()
0.6
>>> f1.set_value(2,7)
>>> f1.evaluate()
0.2857142857142857
>>> f1.show()
2/7
>>> f2.show()
3/5
```

Pet example

- Here is a simple class that defines a Pet object.

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet’s name is " + str(self.name)
```

The __str__ built-in function defines what happens when I print an instance of Pet. Here I’m overriding it to print the name.

Pet example

- Here is a simple class that defines a Pet object.

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet’s name is " + str(self.name)
```

```
>>> from pet import
Pet
>>> mypet =
Pet('Ben', '1')
>>> print mypet
This pet's name is
Ben
>>> mypet.get_name()
'Ben'
>>> mypet.get_age()
1
```

Inheritance

Inheritance

- ***Inheritance*** is when one class (the “child” class) is based upon another class (the “parent” class)
- The child class *inherits* most or all of its features from the parent class it is based on

- It is a very powerful tool available to you with Object-Oriented Programming

Inheritance Example

- For example: computer engineering students are a specific type of student
- They share attributes with every other student
- We can use inheritance to use those already defined attributes and methods of students for our computer engineering students

Inheritance Vocabulary

- The class that is inherited ***from*** is called the
  - Parent class
  - Ancestor
  - Superclass
- The class that does the inheriting is called a
  - Child class
  - Descendant
  - Subclass

Inheritance Code

- To create a child class, put the name of the parent class in parentheses when you initially define the class

```
class cengStudent(student):
```

- Now the child class **`cengStudent`** has the properties and functions available to the parent class **`student`**

Extending a Class

- We may also say that the child class is ***extending*** the functionality of the parent class

- Child class inherits all of the methods and data attributes of the parent class
  - Also has its own methods and data attributes
  - We can even redefine parent methods!

Redefining and Extending Methods

Redefining Methods

- ***Redefining*** a method is when a child class implements its own version of that method

- To redefine a method, include a new method definition – **with the same name** as the parent class’s method – in the child class
  - Now child objects will use the new method

Redefining Example

- Here, we have an animal class as the parent and a dog class as the child

```
class animal:
    # rest of class definition
    def speak(self):
        print("\"" + self.species + " noise\"")

class dog(animal):
    def speak(self):
        print("Woof woof bark!")
```

Extending Methods

- Instead of completely overwriting a method, we can also ***extend*** it for the child class

- Want to execute both the original method in the parent class and some new code in the child class
  - To do this, we must explicitly call the parent’s version in the child

Extending Example

- Extending the **`__str__`** method for **`dog`**
  - Used when we **`print()`** an object

```
def __str__(self):
    # get the result from parent __str__
    msg = animal.__str__(self)
    # add information about the breed
    msg += "\n\tTheir breed is " + str(self.breed)
    return msg
```

References

- Dr. Katherine Gibson’s slides