CS 172 - Computer Programming II

Introduction to Object Oriented Programming

Objectives

- Basic Object Oriented Analysis (OOA), Object Oriented Design (OOD) and Programming (OOP) Concepts
  - Understand what object oriented analysis and design is
  - Understand the relationship between objects and classes
  - Attributes and behaviors of objects
- Using Objects
- Modules and Packages

Intro to Object Oriented Design

- In many applications, bundling together variables to form an "object" makes sense.
- For instance, if our application requires keeping track of many cars, it may make sense to define the concept of a car
  - A car contains a make, model, color, year, mileage, etc.

OOA, OOD and OOP

- In addition we may want to define what *actions* can be performed on those objects
  - Change color
  - Set mileage
  - Speed up
- The process of taking a problem and identifying the objects needed and the interactions between the objects is called *object-oriented analysis*
- The outcome of the analysis is a set of requirements.

OOA, OOD and OOP

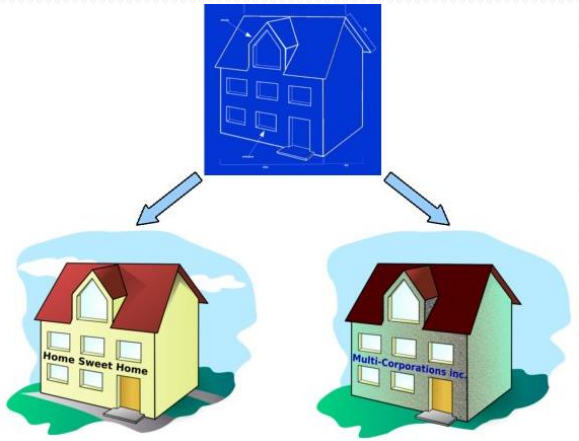
- The process of converting the requirements into an implementation specification is called *object-oriented design*.
- A lot goes into the design process
  - What variables should there be?
  - What actions should there be?
- *Object Oriented Programming* (OOP) is a programming technique that takes advantage of the concept of objects.
  - Converts the OOD into a working program

So… what's an object?

- An object is
  - A collection of data and
  - A set of actions that can be performed on the object
- In OOP terminology we usually call
  - The collection of data that defines a type of object its *attributes* (or *members* or *properties)*
  - The collection of actions its *methods*

What is a class?

- We call a particular *type* of object a *class*
- Classes describe objects
  - All objects of the same class have the same attributes and methods
  - Like a blueprint for creating an object
  - Like a cookie cutter



Instantiating a class

- If we want to create an instance of a type of object, we call it *instantiating a class*
  - Or creating an instance of a class
- We've already used some classes that are provided for us by the developers of Python *(built-ins)*:
  - `string`
  - `file`
  - `list`

Using Objects

- Let's start off by revisiting using existing classes.
- The first thing we need to do is create an instance of a class.
- Then we can call its methods!

Constructing an Object

- Typically, to create an instance of a class, we call a special function that is just the name of the class.
- This function returns a new instance of the class (an object) that we can store in a variable.
- As we'll see a bit later, this process can also involve an implicit call to a special method called `__init__` to which we can pass parameters to help initialize the attributes of the object. → every class must have this
  - This method is often referred to as the *constructor*

Python `list` class

- To demonstrate this process, let's look at Python's built-in `list` class
- To create a new `list` we can do:

```
myList = list()
```

- Often there are several ways to call this function, with a varying number of parameters:

```
myList = list("123")
myList = list([0, 1, "two"])
```

- Note: `myList = [0, 1, "two"]` is just shorthand/syntactic sugar.

Calling Methods

- Once we have an instance of a class, we can call methods on it.
- We call a method on an object using the name of the object variable, followed by the dot operator, followed by the method's name, and its parameters (if needed).
- For example:

```
myList.append(4)
myList.count(4)
myList.sort()
```

⇒ add something to the end of a list

Static Attributes and Methods

- Sometimes we want an attribute or method to be independent of the particular instance (object).
- We can think of it as being "shared" with all instances of the class.
- These attributes and methods are called *static*
  - **zyBooks calls these class attributes and class methods** (not to be confused with instance attributes and instance methods).
- To access as static attribute or method you just use the class name, followed by the dot operator, followed by the attribute or method name.
- One such class that has many static attributes and methods is the `math` class.

Static Attributes and Methods

- For example, let's get access to the π static *attribute*
  - `math.pi #shared/static attribute`
- Static *methods* can be useful
  - Some actions pertain to a class, not an instance of it
    - That is, they do not depend on any instance data.
    - Although they can be used by non-static methods.
- Example:
  - `math.sqrt(5)`
- Why does it make sense for this method to be static?

Using Custom Classes

- Are all built-in classes automatically included into all Python scripts?
  - Of course not! That would be wasteful.
  - Just a few are (like `list`).
- What if we want to use a class that isn't automatically included?
  - For example, although the `math` class is provided for us, it isn't automatically included when you start Python, since only some programs may need it.
- We could either
  - Add that class code to our script
  - Store the class code in a different file and **`import`** it.
    - This is the preferred approach

sometimes you have to import a class!

Modules

- Code that is stored in an external file, that can be used in another script, is referred to as a **modules**
- We refer to a module by the file name without its extension.
- To use a module, we must *import* it.
- Let's imagine we have a class called `Database` that's within a file called `database.py` → One of the classes in database.py is database
  - For now, let's assume it's in the same directory as your main script.
- We could import the entire module as

```
import database
```

- Then to access the Database class we could type:

```
database.Database
```
→ This is your class database

Modules

- Or we could just import the class from the module: `from database import Database`
- And now we can make use of that class!

```
db = Database()
```

- To avoid name conflicts we can import **`as`** some other name:

```
from database import Database as DB
```

- We can also import several classes from a module:

```
from database import Database, Query
```

Packages

- As our project grows, we'll likely include more and more modules.
  - It might be a pain to have to include each module
- A **package** is a collection of modules in a folder
  - The name of the package is the name of the folder
- Then we can just import this folder/package
  - As long as there is a "special" file in it to indicate this directory is part of a package
  - This file must be called `__init__.py` and is typically just empty

Packages

- Imagine an ecommerce project that wants to make use of a lot of modules:
  - database
  - products
  - square payments
  - stripe payments
- We could organize these hierarchically in directories
  - Each of which needs that special `__init__.py` file

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

Packages

- Imagine that `Product` is a class defined in the `products` module.
- Now we can import from parts of packages:
  - ```
    from ecommerce import products
    prod = products.Product()
    ```
  - ```
    from ecommerce import Product
    prod = Product()
    ```
- There's plenty more you can read about packages, but we'll leave it at this, for now.

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

CS 172

- In this course we'll focus on OOP and OOD
- We want to look at interesting problems and identify the use of OOP (after all, not all problems call for OOP)
  - Then use OOD to come up with the class designs.
- Along the way we'll leverage key concepts of OOP/OOD like interfaces, composition and inheritance for some fun and useful examples

CS 172 - Computer Programming II

Designing and Creating Classes

Objectives

- Defining a basic class
  - Constructors
  - Adding instance attributes and methods
  - Adding class (static) attributes and methods
  - Defining a public interface
- Executing modules as scripts

Should I use OOP?

- How do we determine if we should use OOP?
- Here some ideas and questions to ask yourself:
  - Do we want to carry around groups of variables?
  - Might we want to have several groups of the same types of variables?
  - Conceptually does this group of variables represent/describe something?

Should I use OOP?

- If we identify that creating a class could be useful, then the next step is to design it.
- This design cycle is usually:
  - Identify the set of useful attributes and their default values.
  - Identify what you want your constructor to do
  - Identify useful methods
- More likely than not you'll end up adding attributes and methods as you go.

Creating a Class

- To define a class called use the following syntax:

```
class ClassName():
    #some more code here, indented
    #parenthesis are optional (for now)
```

- Good style suggests to use the camel case style for the class name:
  - The first letter of each word is capitalized.
- All the rules for identifier names still apply:
  - Letters (a – z), digits, underscores
  - No special characters
  - Do not begin with a digit

Creating a Class - Example

- Lets define a very basic class called `MyFirstClass`:

```
class MyFirstClass():
    pass
```

- Note: **`pass`** is a Python keyword indicating no further work is to be done in this block
- So we created a class that does not do anything

Creating a Class - Example

- But wait, where should we put this code?
- If we suspect we'll only use it for this application, we can actually have it in the same file as our main script
- However, often the purpose of defining classes is to create re-usable object
  - So it may make more sense to put it in its own file and import it.

Instantiating a Class

- Now we can use our class!
- Assuming the `MyFirstClass` definition was put within the file `lecture2.py`

```
from lecture2 import MyFirstClass
a = MyFirstClass()
b = MyFirstClass()
print(a)
print(b)
```

- Here is the output:

```
>>> %Run myFirstClassTest.py
<lecture2.MyFirstClass object at 0x1021f9e48>
<lecture2.MyFirstClass object at 0x1021f9ef0>
```

Instantiating a Class

- What's up with the weird numbers/letters?

```
<lecture2.MyFirstClass object at 0x1021f9e48>
<lecture2.MyFirstClass object at 0x1021f9ef0>
```

- Well, `a` and `b` are just objects, so what does it mean to print out an entire object?
- By default Python will just print out its location in memory
  - Kind of cool
  - But useful?
- Soon we'll see how to *override* this behavior if we should choose.

Constructor Method

- Although technically we don't need to define a constructor method, it's good practice to have one
  - This way we define ahead of time which attributes exist
  - And provide a way to initialize them.
- Therefore one of the first methods that most classes have is a special one called `__init__`
- This method gets called when an object is created, and typically defines and initializes the attributes

Constructor Method

- For example, maybe we want to create an object to represent points in a plane
  - What should the attributes be?
  - What should their default values be?
  - Here's some code:

```
class Point:
        def __init__ (self):
                self.__x = 0
                self.__y = 0

>>> p = Point()
```

- Note that we never actually passed in anything for the `self` parameter

The Implicit Parameter

- Methods that are called on instances of a class are called (shocker), *instance* methods.
- Instance methods require at least one parameter. That parameter is an *implicit parameter*.
  - This implicit parameters refers to the object itself.
    - Therefore, it is commonplace to name this parameter **`self`**
  - We use **`self`** to access methods and attributes of the class within the class itself
  - We use the dot notation with **`self`**
- Without **`self`**, we would have created **local** variables called `__x` and `__y`

Constructor Method

- We can require that certain information is provided (passed via parameters) when an instance of a class is created

```
class Point:
        def __init__ (self, x, y):
                self.__x = x
                self.__y = y

>>> p1 = Point()  # will cause an error
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'

>>> p2 = Point(10, 5)
```

Constructor Method

- We could use the default arguments technique:

```
def __init__ (self, x = 0, y = 0):
        self.__x = x
        self.__y = y
```

- All three of the following statements are valid:

```
>>> p1 = Point()        # x and y are 0
>>> p2 = Point(2, 3)    # x = 2 and y = 3
>>> p3 = Point(10)      # x = 10 and y = 0
```

Encapsulation

- For better or worse Python lets programmer have direct access to attributes via the dot operator
  - Most other languages do not allow this by default
- This can make life easier for someone who uses a class, but can result in incorrect usage.
- For example suppose we want to create a `Car` class that has attributes:

```
make, model, mileage, year
```

- What's to stop a programmer from doing:

```
myCar = Car()
myCar.mileage = -1234
```

Encapsulation

- Ideally, outside of the class itself, people would only interface with attributes indirectly, via methods.
- This process of forcing a user to interface with a class via methods is the process of *information hiding* or *encapsulation*
  - We're "encapsulating" (protecting) an object and only providing some "safe" public interface for them to work with
  - Only showing users what they need to see provides a level of *abstraction*

Attribute Mangling

- As previously mentioned, Python doesn't allow for different types of access permissions.
- As a "work around", however, Python will "mangle" the name of any attributes that start with a double underscore, __
  - **Unfortunately, Zybooks doesn't encourage this ☹**
- Mangling means that Python appends an underscore followed by the name of the class in front of the attribute name when we attempt to access the attribute outside the class.

Attribute Mangling

- Example:

```
def __init__ (self, x = 0, y = 0):
    self.__x = x
    self.__y = y
```

```
>>> p2 = Point(2, 3)
>>> print (p2.x)      #ERROR!
>>> print (p2.__x)    #ERROR!
>>> print (p2._Point__x). #this works
2
```

- **As a rule of thumb all of our attribute names will begin with a double underscore.**

Adding More Methods

- We already have a method to initialize the attributes
  - `__init__`
- Next we typically add methods that define a *public interface*
- Deciding on the public interface for a class is extremely important.
- You'll likely want to update your class at some point
  - But don't want to "break" code that uses it
- So the public interface should remain the same
  - But its implementation can change

Public Interface

- The first part of the public interface that we'll want to make is a set of methods for reading and writing data from/to the attributes.
- We call these
  - Getter, inspector, or accessor methods (for reading)
  - Setter, mutator methods (for writing)

Public Interface

- We may not want to supply getters and setters for all the attributes
  - We should identify which attributes need getters and setters and define them
  - Example: the model of a car does not need a setter
- There may be inspectors and mutators that are more complicated than just getting and setting an attribute

Getters/Setters Example

```
class Point:
    def __init__ (self, x = 0, y = 0):
        self.__x = x
        self.__y = y

    def getX(self):
        return self.__x

    def getY(self):
        return self.__y

    def print(self):
        return "(" + str(self.__x) + ", " + str(self.__y) + ")"

    def setPoint(self, x, y):
        self.__x = x
        self.__y = y

    def reset(self):
        self.setPoint(0,0)
```

Getters/Setters Example

```
>>> p1 = Point()
>>> p1.print()
"(0, 0)"
>>> p1.getX()
0
>>> p1.getY()
0
>>> p1.setPoint(10 ,10)
>>> p1.print()
"(10, 10)"
>>> p1.reset()
>>> p1.print()
"(0, 0)"
```

Adding More Methods…

- It might be nice to have a method that takes in another point and computes the distance between the two:

```
#assume this code is in the Point class and that math has been imported
def distance(self, p):
    return math.sqrt(
            (self.getX() - p.getX()) ** 2 +
            (self.getY() - p.getY()) ** 2 )
>>> a = Point()
>>> b = Point(5, 5)
>>> dist = a.distance(b)
>>> print(dist)
7.071067811865475 5
```

Static Attributes

- In addition to instance attributes and methods, we can have *class* attributes and methods.
  - They belong to the class itself and not an instance of it.
- For an attribute to be static (class), just define it outside of any methods.
- Now we can access it just using

```
ClassName.attribute_name
```

Static Attributes

```
class Point:
    # static attribute
    __count = 0  # to count how many points created

    # initialization method
    def __init__ (self, x = 0, y = 0):
        self.__x = x
        self.__y = y
        Point.__count += 1  #update the number of points created
```

Static Methods

- Recall that a static method can do its job with just the parameters it was given.
  - It doesn't need the implicit object.
- Techniqually, any method can be called in a static context if we call it on the class name, and pass in the implicit object explicitly:

```
pt1 = Point()
Point2D.setPoint(pt1,10,10)
```

- But this is confusing and a bit unlike other languages.

Static Methods

- Alternatively (and preferred), we can mark a method as static so that it can ***only*** be called in a static way.
- To do this we *decorate* the method.
- In Python, a decorator is a special label applied to something to mark it as something special.
- These start with the @ symbol.
- To mark a method as static we can use the @staticmethod decorator right before the method definition.
- And again we access it via the classname

```
class Point():
    __count

    @staticmethod
    def printPointCount():
       print("Points created:", Point.__count)

Point.printPointCount()
```

The Main Script

- There is sometimes code that we only want run if the file/module is the entry-point ("main") module
- This can be useful if we want to have code in a module to test it, but then want to include that module in other ones later.
- To check if a module is the entry-point/main one, we can check against the **__name__** variable

```
if __name__ == "__main__":
    #do whatever
```

CS 172 - Computer Programming II

"Special" methods

Objectives

- Overload the basic special methods
  - Arithmetic Operators
  - Boolean operators, math operators

Overview

- Imagine that we have a class, called `EL`, that is used to represent English Length objects (yards, feet, inches)
  - Example: 2 yards, 3 feet, 5 inches
- First off, what attributes and methods should there be?
- Let's quickly implement this.
- Remember that:
  - 1 yard = 3 feet = 36 inches
  - 1 feet = 12 inches

Overview

- Wouldn't it be neat if we could create two `EL` objects and add them together to create a new one?
- Sure we could create a method, `add()`
  - `len3 = len1.add(len2)`
- But it would be more user-friendly to allow the users to use the + operator:
  - `len1 = EL(1,2,3)`
  - `len2 = EL(3,4,5)`
  - `len3 = len1 + len2`

Overview

- By default Python doesn't know how to handle this
  - What does it mean to add together two objects?
- But we can provide "special methods" in our classes to define this behavior.
- This special methods allows us to overload operators such as: `+` `-` `*` `/` etc.
- What does overloading mean?
  - The ability to create multiple functions of the same name with different implementations
  - We have seen this with operators such as `+` and `*`
  - `+` and `*` can be used with number and strings

Special Methods

OPERATOR	FUNCTION	METHOD DESCRIPTION
`+`	`__add__(self, other)`	Addition
`*`	`__mul__(self, other)`	Multiplication
`-`	`__sub__(self, other)`	Subtraction
`%`	`__mod__(self, other)`	Remainder
`/`	`__truediv__(self, other)`	Division
`<`	`__lt__(self, other)`	Less than
`<=`	`__le__(self, other)`	Less than or equal to
`==`	`__eq__(self, other)`	Equal to
`!=`	`__ne__(self, other)`	Not equal to
`>`	`__gt__(self, other)`	Greater than
`>=`	`__ge__(self, other)`	Greater than or equal to
`[index]`	`__getitem__(self, index)`	Index operator
`in`	`__contains__(self, value)`	Check membership
`len`	`__len__(self)`	The number of elements
`str`	`__str__(self)`	The string representation

Overloading Arithmetic Operators

- Let's implement the `__add__` method for our `EL` class
- How many arguments does it need to do its job?
  - Remember that + is a binary operator (requires two operands)
- Should any of the arguments (operands/objects) change after it's done?
  - Or should it return a new English Length?

Overloading Arithmetic Operators

```
def __add__(self, other):
        newInches = self.__inches + other.__inches
        newFeet = self.__feet + other.__feet
        newYards = self.__yards + other.__yards
        return EL(newInches, newFeet, newYards)

>>> len1 = EL(1, 1, 1)
>>> len2 = EL(1, 1, 1)
>>> len3 = len1 + len2
```

Overloading Comparison Operators

- Let's implement the `__lt__` method for our `EL` class
  - This method will allow us to use the < operator
- How many arguments does it need to do its job?
  - < is also a binary operator
- What should be the return value for this operator?

```
def __lt__(self, other):
    return (self.totalInches() < other.totalInches())

>>> len1 = EL(1, 1, 1)
>>> len2 = EL(0, 0, 1)
>>> len2 < lens1
True
>>> len1 < len2
False
```

Overloading Comparison Operators

- Let's implement the `__eq__` method for our `EL` class
  - This method will allow us to use the == operator

```
def __eq__(self, other):
        return (self.totalInches() == other.totalInches())

>>> l1 = EL(1, 1, 1)
>>> l2 = EL (2, 1, 0)
>>> l3 = EL (1, 1, 1)
>>> l1 == l2
```

The __str__ method

- When we run the commend `print(obj)`, Python is *implicitly* first calling a method on `obj` to get a string representation of it.
- This method is **`__str__`** and should return a string.
- Remember what happens if we don't implement this!?

The __str__ method

- Let's implement the **__str__** method which will allow us to put all the data from an object into string that can then be printed.

```
def __str__(self):
        mystr = "An English Length object that is "
        if self.__inches == 1:
            mystr += "1 inch, "
        else:
            mystr += str(self.__inches) + " inches, "

        if self.__feet == 1:
            mystr += "1 foot, "
        else:
            mystr += str(self.__feet) +" feet, and "

        if self.__yards == 1:
            mystr += "1 yard long."
        else:
            mystr += str(self.__yards) + "yards long"

        return mystr
```

The __str__ method

```
>>> len1 = EL(1, 2, 3)
>>> print(len1.__str__())
An English Length object that is 1 inch, 2 feet, and 3yards long
>>> print(len1)
An English Length object that is 1 inch, 2 feet, and 3yards long
```

Overloading the Subscript Operator

- The `[]` operator is a binary operator generally used for subscripting
- If your class contains a list of elements, it might be useful to overload the `[]` operator to refer attributes.
- For example, maybe we want the `[]` operator to behave on our `EL` example so that
  - `len1[0]` refers to `__inches`
  - `len2[1]` refers to `__feet`
  - `len3[2]` refers to `__yards`
- This behavior can be defined via the `__getitem__` method
  - How many arguments does this method need?

Overloading the Subscript Operator

```
def __getitem__(self,loc):
        if loc == 0:
                return self.__inches
        elif loc == 1:
                return self.__feet
        elif loc == 2:
                return self.__yards
        else:
                raise IndexError

>>> len1 = EL(2, 0, 5)
>>> len1[0]
2
>>> len1[1]
0
>>> len1[2]
5
```

CS 172 - Computer Programming II

Inheritance

Objectives

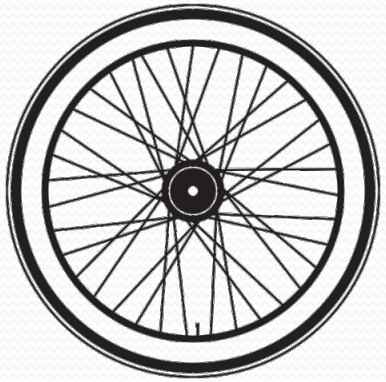
- Understand what Inheritance is and how to use it
- Understand what Polymorphism is
- Understand what Abstract Base Classes are

The 4 pillars of OOP

- Encapsulation
  - Hiding a module's internal data and all other implementation details/mechanism from other modules.
  - Also a way of restricting access to certain properties or component.
- Abstraction
  - The process of exposing essential feature of an entity while hiding other irrelevant detail
- Inheritance
- Polymorphism

Inheritance

- Inheritance allows us to create is-a relationships between two or more classes.
  - It allows us to put common logic into super-classes and managing specific details in the subclass.
- Inheritance facilitates code reuse
- Once again this comes back to “why reinvent the wheel”?



Inheritance

- The idea behind inheritance is that some classes can be built upon others
  - *Inherit* from others
- This saves time, avoids redundancy, and errors
  - Have someone do it right once!
- We call the class inherited from the *base class*
  - *Parent class* or *Super class*
- We call the class that inherits a *derived class*
  - *Child class* or *sub-class*

Inheritance

Super Class

Animals

Child Class

Amphibians

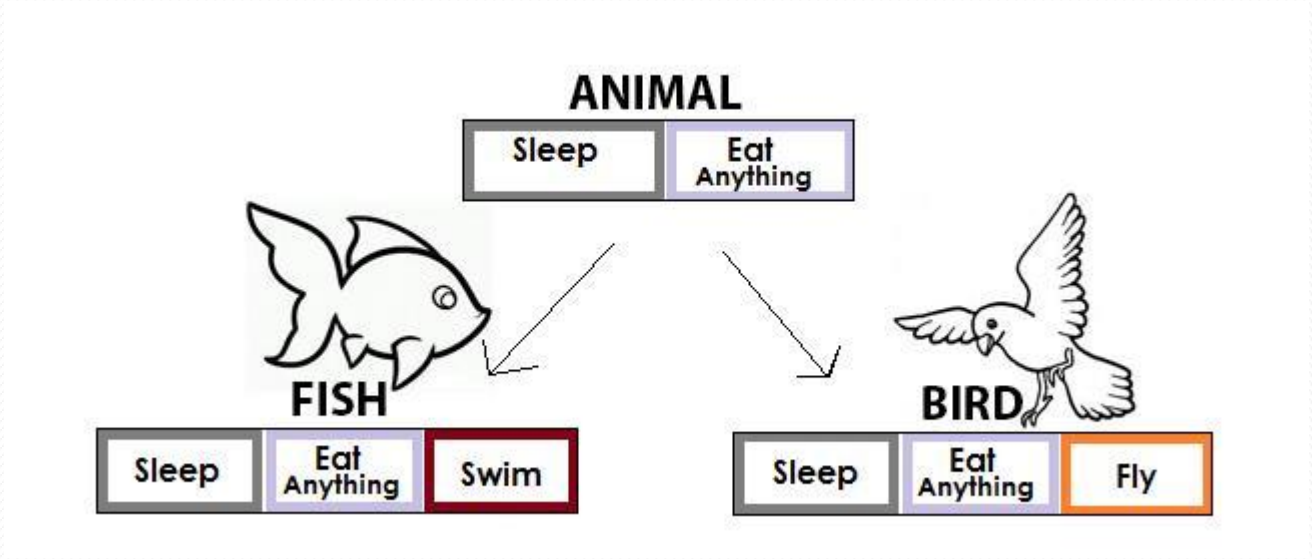
Reptiles

Mammals

Birds

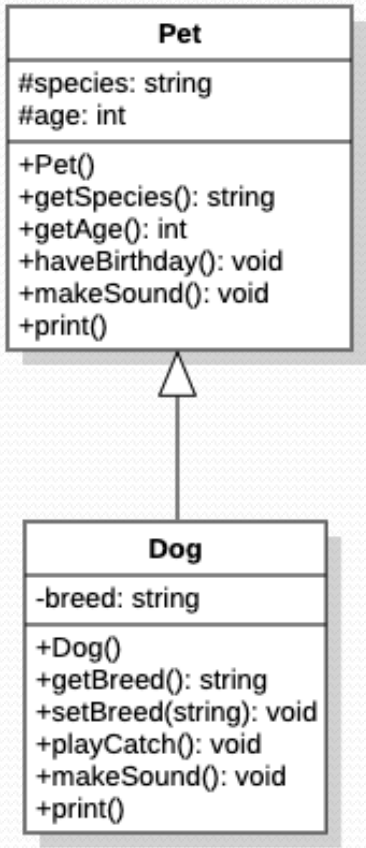
Inheritance

- A derived class has access to all the attributes and methods of the base class
- A derived class has additional attributes and methods



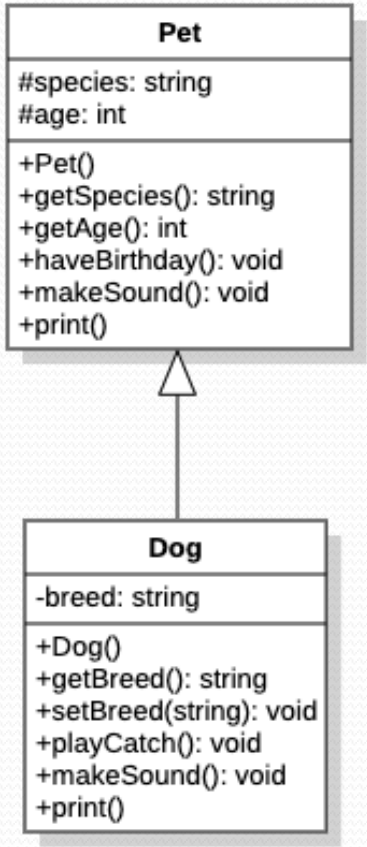
UML

- Often it is useful to be able to visualize the relationship between classes.
- A standard way to do this is via the Unified Modeling Language (UML)
  - Modeling language used in software engineering to visualize the way a system has been designed.



UML

- Class diagrams represent classes and their relationships
  - A class is represented by a rectangle divided in 3 parts:
    - Class name
    - Attributes
    - Methods
  - Attributes and methods have symbols to represent the access level:
    - - means private
    - + means public
    - # means protected
- The inheritance relationship (is-a) is depicted with a closed-arrowhead pointing from the child class to the parent class.



Inheritance

- Technically, every class in Python inherits from a special class called **`object`**.
- This is **implicitly** done.
- But we could also explicitly do so:

```
class MySubClass(object):
    pass
```

- Here we pass as a parameter the name of the class that we are inheriting from

Inheritance - syntax

```
class DerivedClass(BaseClass):
    #add any additional features
```

- As an example, let's create a Pet class and then inherit from it to make other related classes!

Example: the `Pet` class

```
class Pet:
    def __init__(self, species, age = 0):
        self.__species = species
        self.__age = age

    def getSpecies(self):
        return self.__species

    def getAge(self):
        return self.__age
    def haveBirthday(self):
        self.__age = self.__age + 1

    # The makeSound method is the pet's way of making a generic sound
    def makeSound(self):
        print("Grrrrr")

    def __str__(self):
        myStr = "A " + str(self.__age) + "-year old " + self.__species
        return myStr
```

Inheritance: creating a `Dog` class

- What if we wanted to have different types of pets that can have their own special features but also have the core features pets have?
- Let's create a class for dogs that builds upon the `Pet` class
  - Our `Dog` class will have an attribute to represent the dog's `breed` (and associated getter and setter methods).
  - Dogs can play catch, so we will have method for that as well

Example: the Dog class

```
class Dog(Pet):
    # constructor
    def __init__(self, age = 0, breed = “”):
        self.__breed = breed   #additional attribute

    # additional getters
    def getBreed(self):
        return self.__breed

    # additional setters
    def setBreed(self, breed):
        self.__breed = breed

    # additional behavior
    def playCatch(self) :
        print(“Running and catching the stick. Good doggie!”)
```

Inheritance and Overriding Methods

- The good news:
  - We'll inherit methods from the `Pet` class for free!
- The bad news:
  - We still need to initialize its attributes.
  - And we may want to change their behavior for our Dog class.
- The good news again!:
  - We can **override** out parent's method by creating one of our own!
  - And if we need to access its version, we just call it on the object returned by the function `super()`!
    - We can access a parent class' attributes and methods via what is returned by **`super()`**.
    - A **`super()`** call can be made inside any method

Example: the Dog class

```
class Dog(Pet):
    # constructor
    def __init__(self, age = 0, breed = “”):
        super().__init__(“dog”, age)
        self.__breed = breed   #additional attribute

    # additional getters
    def getBreed(self):
        return self.__breed

    # additional setters
    def setBreed(self, breed):
        self.__breed = breed

    # additional behavior
    def playCatch(self) :
        print(“Running and catching the stick. Good doggie!”)
```

Inheritance and Overriding Methods

- And for free from the `Pet` class we got:

```
def makeSound(self):
     print('Grrrr')

def __str__(self):
     myStr = "A " + str(self.__age) + "-year old " + self.__species
        return myStr
```

- But it would make sense to have these two methods updated to work with the `Dog` class:
  - A dog makes a woof sound instead of Grrrr
  - We would like to be able to print all the information about a dog, including the breed.

Overriding methods

```
# override makeSound
def makeSound(self):
    print("Woof! Woof!")

# override __str__
def __str__(self):
    myStr = super().__str__() +  ", " + self.__breed
    return myStr
```

Example: `Pet` and `Dog` classes

```
>>> fred = Pet("frog", 2)
>>> print(fred)
A 2-year old frog

>>> sparky = Dog(1, "poodle")
>>> sparky.getBreed()
poodle
>>> sparky.getAge()
1
>>> sparky.getSpecies()
dog
>>> sparky.haveBirthday()
>>> print(sparky)
A 2-year old dog
```

Overloading vs. Overriding

- Overloading:
  - Two or more methods/functions have the same name but different parameters.
  - Body of the methods/functions are different
- Overriding:
  - Two methods with the same method name and parameters (i.e., method signature).
  - One of the methods is in the parent class and the other is in the child class.

Polymorphism

- The ability for a program to find the correct version of a method to run is called polymorphism.
  - Different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is.

Example

```
>>> myPet = Pet("unknown")
>>> myPet.makeSound()
Grrrrr

>>> snoopy = Dog(3, "Beagle")
>>> snoopy.makeSound()
Woof! Woof!

>>> fluffy = Cat(5, "double coat")
>>> fluffy.makeSound()
Meow
```

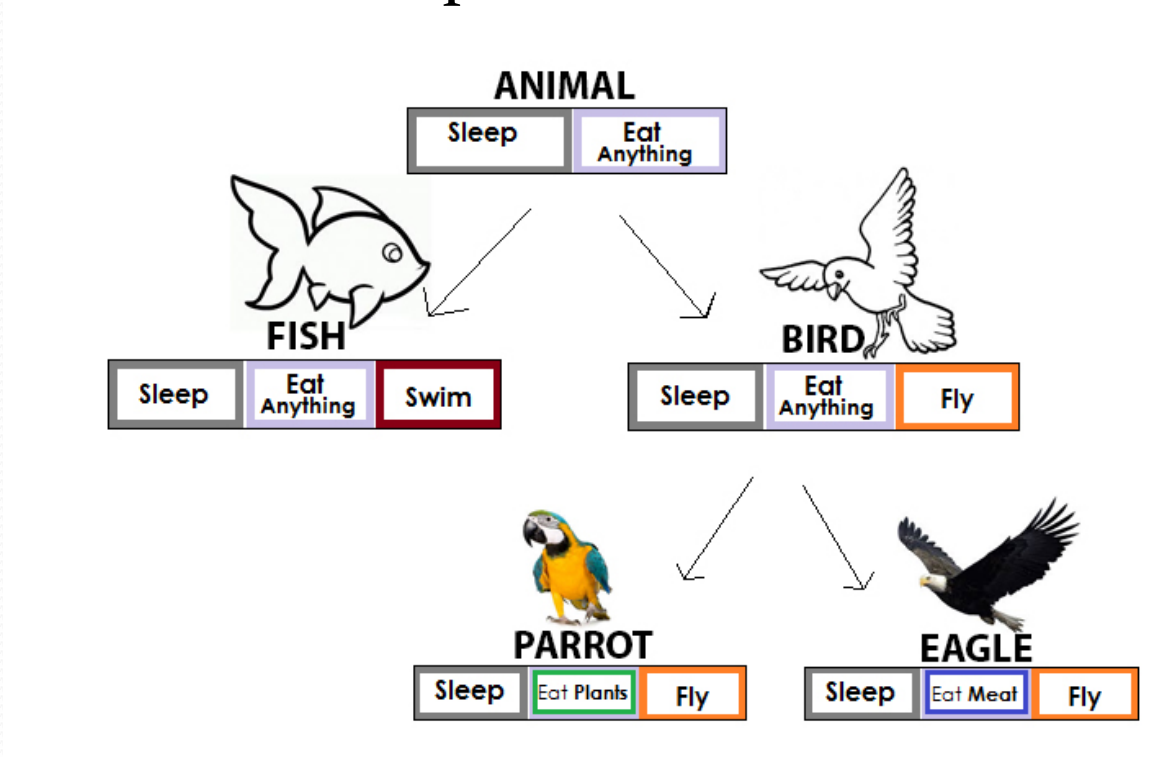
Polymorphism

- We can explicitly determine the type of an object and/or determine if one class is a subclass of another.
  - **`isinstance(object, class)`**
    ```
    #returns True if object is of type class
    ```
  - **`issubclass(class1, class2)`**
    ```
    #returns True if class1 is a subclass of class2
    ```
- Example:

```
if isinstance(snoopy, Dog) :
    # do something
if issubclass(Dog, Pet) :
    # do something
```

Inheritance Hierarchies

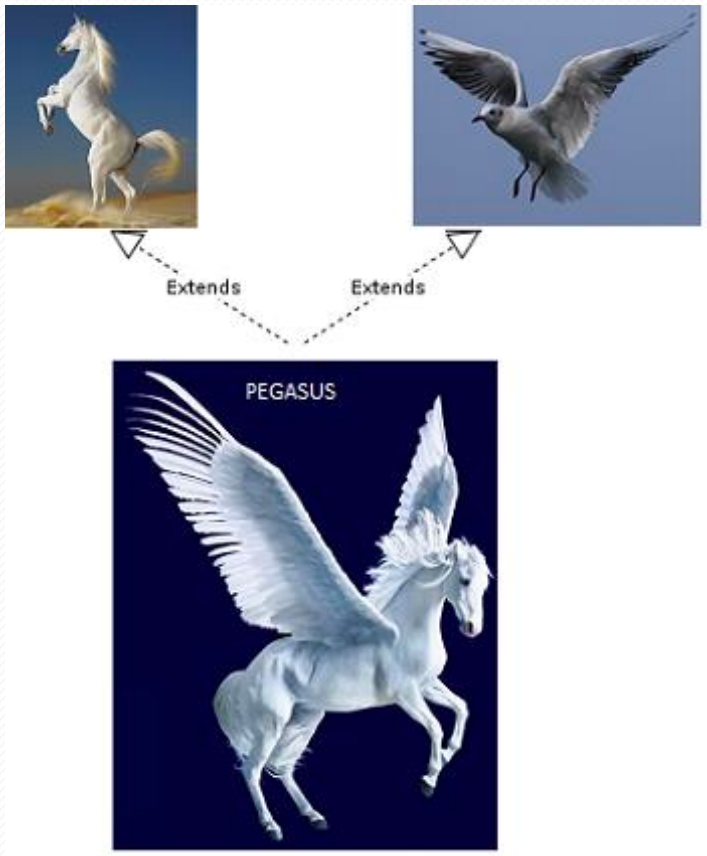
- We can create hierarchies by using inheritance
- A child class can become parent to one or more derived classes



Multiple Inheritance

- Python allows a class to inherit from several super-classes
- Syntax:
  - just provide a comma separated list of the parent classes.

```
class Pegasus(Horse, Bird):
    pass
```



Multiple Inheritance

```
class Pegasus(Horse, Bird):
    pass
```

- So now what does `super()` refer to?
- The `Horse` class
- But what about accessing stuff of the `Bird` class?
- If a method or attribute is unique in that class, Python will find it.
- But what if it has the same name as an attribute or method in `Pegasus` or `Horse`?
- For example `__init__`?
- The easiest way is to access it using the class name instead of `super()`, i.e.

```
Horse.__init__(self,….)
Bird.__init__(self,…)
```

- Note that now we have to ***explicitly*** pass in the first parameter object when doing this.

Abstract Base Classes

- What happens if we call a method on a object and the interpreter can't find it?
  - Run-time error!!
- Is there some way to **require** a class to implement a method?
  - Yes!
- We create an *abstract base class* (`abc`) that defines what we want to require derived classes to implement.
- Then any class that *derives* from that abstract base class **must** implement the listed methods.

Abstract Base Classes

- As an example, maybe we have an application where we want to be able to play different audio files.
  - `.wav, .mp3, etc.`
- Each of these file formats have different rules from reading and playing the file.
- Our application doesn't care about the details, it just wants to make sure that any media type must have a `play()` method that can be called.
- So let's create an ABC called `AudioFile` that requires those who derive from it to implement the method `play()`

Abstract Base Classes

- First, we need to tell Python that our class is supposed to be an abstract base class.
  - This way it will know to enforce the rules for others to inherit from it.
- To do this we have our class inherit from the **`ABC`** class of the **`abc`** module.

```
from abc import ABC
class Audiofile(ABC):
    pass
```

Abstract Base Classes

- Next, we need to mark which methods are to be abstract.
- To do this we add a **decorator** before any method that is meant to be abstract.
  - In Python, decorators are keywords that start with @
- The **`abc`** module also has an **`abstractmethod`** decorator:

```
from abc import ABC, abstractmethod
class AudioFile(ABC):

    @abstractmethod
    def play(self):
        pass
```

Abstract Base Classes

- Now if we want something to be able to be derived from `AudioFile`, it needs to have a method `play()`

```
class MP3(AudioFile):
    def play(self):
        #code here to play mp3 files
```

CS 172 - Computer Programming II

Pygame and Computer Graphics, Part 1

Objectives

- Computer Graphics Intro
- What is Pygame
- Installing Pygame
- Basics of Pygame
- Drawing on a Surface

Reading

- Readings from “Making Games with Python & Pygame”
  - https://inventwithpython.com/pygame/
- Chapters 1 and 2

CLI vs. GUI

- CLI: Command Line Interface:
  - User interacts with the program via text
    - Displayed on the screen / Entered via keyboard
  - No graphics, colors, or mouse input
- GUI: Graphical User Interface:
  - Windows with graphics, colors, images, sounds
  - User can interact with the program via the mouse, keyboard, and other devices.

Computer Graphics

- Going from a command-line interface (CLI) to a graphical user interface (GUI) is non-trivial!
- It involves things like:
  - Determining how to render pixels on a screen given primitive representation
    - i.e. How to make the pixels of a line given its endpoints
    - Covered in CS 430
  - How to respond to events
    - Where was a mouse clicked? And on what?
- And a lot of these things are hardware and software dependent
  - Graphics card capabilities
  - Operating System

Computer Graphics

- As computers became more mainstream in the 1980's, the desire for a more graphical interface became necessary
- Graphics APIs (application programming interface) try to take away some of the "under the hood" details of graphics at different levels of abstraction
  - Covered in CS 432 – OpenGL Programming

Computer Graphics

- Computer Graphics are a great application of object oriented programming since intrinsically there are various “objects” on the screen.
- As a sort of case study for this course, we’ll look at making graphic applications using a Python 2D graphics API, Pygame

What is Pygame?

- Pygame is a Free and Open Source Python library for making multimedia applications like games and animations.

http://pygame.org

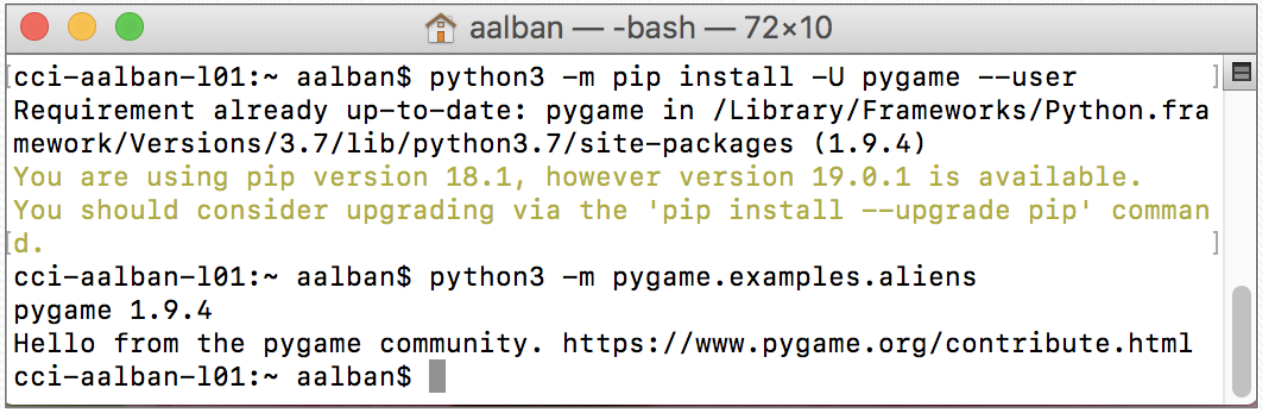
- Includes several modules with functions for drawing graphics, playing sounds, handling mouse input, etc.

Pygame - Installation

- To install pygame we'll use the python-supplied `pip3` program, run from the command line:

```
pip3 install pygame
```

- Note: You may have to navigate to where you installed Python to find the pip executable and run the command from in there.



Pygame

- The following lecture material is based on the online book:
  https://inventwithpython.com/pygame/
- In addition the official Pygame website:
  http://pygame.org/docs/

The Imports

- The first step to use Pygame will be to `import` Pygame.
- We could just import sub-packages of Pygame, but for simplicity lets just import everything:

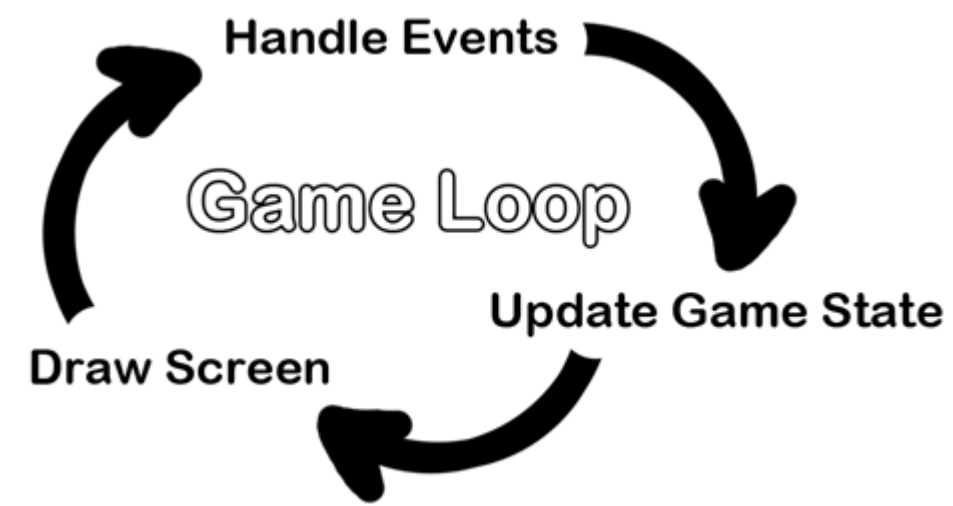
```
import pygame
```

Initialization

- Once we have our imports set, we need to do some initialization.
- This usually includes
  - Initializing the API (in this case, Pygame)
  - Setting some display mode
- With Pygame this will be done via:
  - `pygame.init()`
  - `surface = pygame.display.set_mode((width, height))`
    - Where `width` and `height` are the integer dimensions of the window.
      - Passed as a tuple
      - Width and height are given in pixels
    - What is returned is the "surface" that we'll draw on
    - In many graphics applications we refer to this as the *frame buffer*

Game Loop

- Most graphics APIs are event-driven
  - That is, they have an infinite loop running, and as events occur, they are added to an event queue to be served.
- Common events include:
  - Redraw
  - Mouse click
  - Mouse move
  - Keyboard click
  - Timer



Game Loop

- So at the very least we need to create an infinite loop.
- And at each iteration of the loop, process all the events in the event queue, and then ask Pygame to update the display window

```
while(True):
    for event in pygame.event.get():
        pass
    pygame.display.update()
```

- But as is, our program will never quit!
- To quit the program, we'll allow the user to either
  - Attempt to close the window by clicking on the close-window button
  - Type the `'q'` key

Events

- Every **`Event`** object has a **`type`** which tells us what kind of event the object represents.
- In the image below, the left column lists the different types, all of which are constants accessible through `pygame.<type>`
- For instance, we can see if an event is a **`QUIT`** type by:

```
if event.type == pygame.QUIT :
```

QUIT	none
ACTIVEEVENT	gain, state
KEYDOWN	unicode, key, mod
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy, axis, value
JOYBALLMOTION	joy, ball, rel
JOYHATMOTION	joy, hat, value
JOYBUTTONUP	joy, button
JOYBUTTONDOWN	joy, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	none
USEREVENT	code

Events

- The right column in the image lists attributes of a particular type of event that are accessible through the **`__dict__`** dictionary attribute.
- For instance, if I want to see if an event is a **`KEYDOWN`** event and the key that is pressed is the `'q'` key we can do:

```
if (event.type == pygame.KEYDOWN) and (event.__dict__['key'] == pygame.K_q):
```

QUIT	none
ACTIVEEVENT	gain, state
KEYDOWN	unicode, key, mod
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy, axis, value
JOYBALLMOTION	joy, ball, rel
JOYHATMOTION	joy, hat, value
JOYBUTTONUP	joy, button
JOYBUTTONDOWN	joy, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	none
USEREVENT	code

Game Loop with events

Ok so now we can at least quit our application

```
while(True):
    for event in pygame.event.get():
            if ((event.type == pygame.QUIT) or
                (event.type == pygame.KEYDOWN and event.__dict__['key'] == pygame.K_q)):
                pygame.quit()
                exit()
    pygame.display.update()
```

Keyboard Events

- From the previous example, it should be relatively clear how to check the key's code using the Pygame constants like `K_q`
- For keyboard events, these codes each start with `K_` followed by the key
- Here is a reference of the various keys:

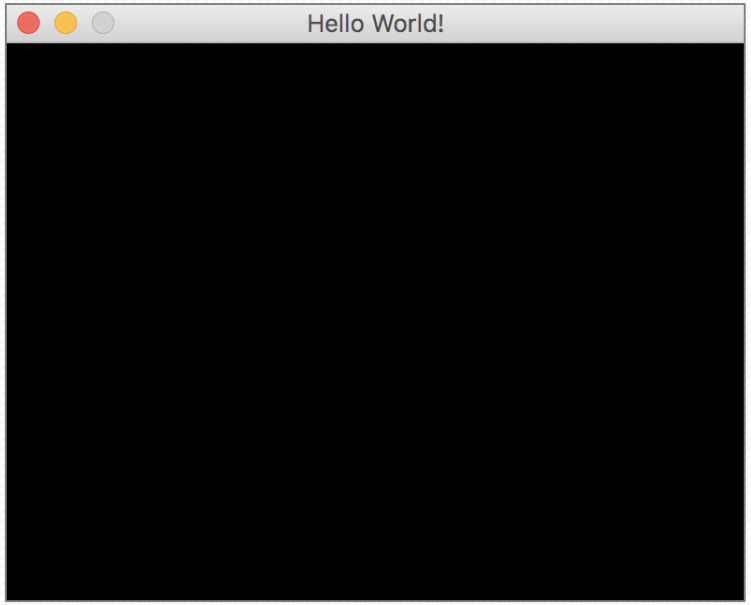
https://www.pygame.org/docs/ref/key.html

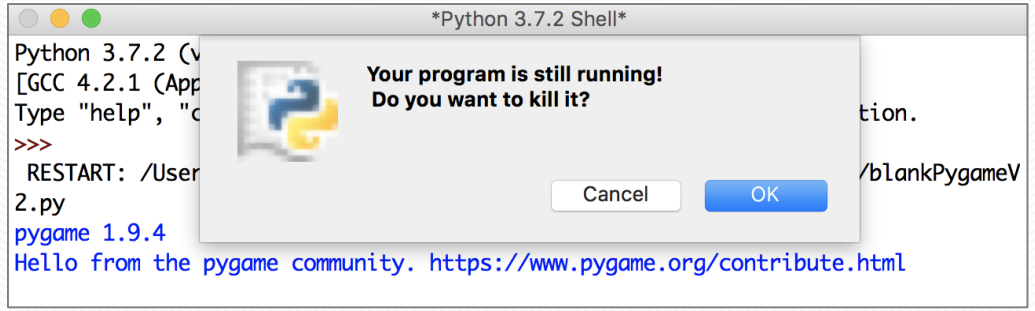
Let’s see our full program

```
import pygame

#initialization
pygame.init()
surface = pygame.display.set_mode((400, 300))
pygame.display.set_caption('Hello World!’) #give window a title

#main game loop
while True:
    for event in pygame.event.get():
        if ((event.type == pygame.QUIT) or (event.type == \
            pygame.KEYDOWN and event.__dict__['key'] == pygame.K_q)):
            pygame.quit()
            exit()
    pygame.display.update()
```

Let's see our full program

- To exit we need to either press q or close the window





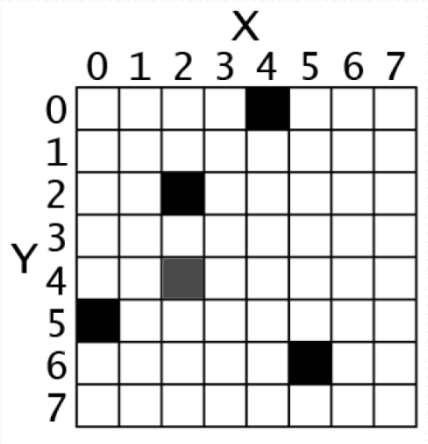
Mouse Events

- The other important events we may want to respond to are mouse events.
  - In particular, `MOUSEBUTTONDOWN` and `MOUSEMOTION`
- Each of these events have in their dictionary attribute the position of the mouse as a tuple

```
QUIT            none
ACTIVEEVENT     gain, state
KEYDOWN         unicode, key, mod
KEYUP           key, mod
MOUSEMOTION     pos, rel, buttons
MOUSEBUTTONUP   pos, button
MOUSEBUTTONDOWN pos, button
JOYAXISMOTION   joy, axis, value
JOYBALLMOTION   joy, ball, rel
JOYHATMOTION    joy, hat, value
JOYBUTTONUP     joy, button
JOYBUTTONDOWN   joy, button
VIDEORESIZE     size, w, h
VIDEOEXPOSE     none
USEREVENT       code
```

Pixels

- The windows in graphics program is made of of little square dots called *pixels*
- For example, if our window is 10 × 10 it will have 100 pixels.
- It's useful to think of our window as a pixel grid or matrix whose origin is at the top-left of the window
- Therefore it's natural to think of this as a 2D Cartesian coordinate system and refer to a given pixel location as (x, y)



Colors

- At each of these pixel locations, a color can be stored.
- In most graphics applications, this color is a tuple, (red, green, blue), where each component can be a value between 0 and 255.
  - RGB: 3 color channels, 1 byte each
- We can use the `fill` method to fill a surface with a particular color.
- This can be useful to clear the drawing surface to some color at time we draw

```
white = (255, 255, 255)
surface.fill(white)
```

Colors

- About 16 million+ colors can be represent with the RGB system.
- Here is a resource to find the RGB codes for many colors:
  https://www.rapidtables.com/web/color/RGB_Color.html

Primitives

- The most basic set of things we want to put on our surface are called *primitives*.
- Each API supports its own set of primitives, but most include at least:
  - Points
  - Lines
  - Triangles
- Any other polygon can be made up of triangles, so many APIs limit polygon types to triangles.

Primitives

- Pygame's set of primitives includes:
  - Line
  - Rectangle
  - Circle
  - Ellipse
  - Polygon (with an arbitrary number of sides)
- See: https://www.pygame.org/docs/ref/draw.html?highlight=draw#module-pygame.draw

Primitives

- To draw a primitive we just ask Pygame to draw a type of primitive on our surface via a function call:

```
pygame.draw.<type>(<surface>, <color>, params..)
```

- For instance, maybe we want to draw a line segment.
- What do we need to do this?
  - Two endpoints

```
GREEN = (0, 255, 0)
pygame.draw.line(surface, GREEN, (0,0),(50,50))
```

Additional Primitives

- Likewise we can add additional primitives to our display surface:

```
pygame.draw.rect(surface, color,(xmin, ymin, width, height))
pygame.draw.circle(surface, color,(xcenter, ycenter),radius)
pygame.draw.ellipse(surface, color, (xmin, ymin, width, height))
pygame.draw.polygon(surface, color, pointlist)
```

- To any of these we can also add an optional **`thickness`** parameter
  - If omitted (or set to zero), the shape will be filled in.
  - If included (and non-zero), then it will be outlined with the desired thickness (and not filled in)

CS 172 - Computer Programming II

Pygame and Computer Graphics, Part 2

Objectives

- Drawable Objects
- Animation
- Collision detection

Drawable Objects

- In real graphics applications, there tend to be many things drawn.
- Each of these may be a kind of particular type of drawable object.
- And each type of thing must have a way for it to be drawn.
- Sounds like a perfect application of polymorphism, abstract base classes, and inheritance!

Drawable Objects

- At the very least, all of our objects should have a location and a method to draw onto a surface.
- So let's start off with an abstract base class that has a location for its object and requires others who derive from it to implement a draw method
  - We may want to have more methods in our public interface

Drawable Objects

```
import abc
class Drawable(metaclass = abc.ABCMeta):
        def __init__(self, x = 0, y = 0):
                self.__x = x
                self.__y = y

        def getLocation(self):
                return (self.__x, self.__y)

        def setLocation(self, point):
                self.__x = point[0]
                self.__y = point[1]

        @abc.abstractmethod
        def draw(self,surface):
                pass
```

Drawable Derived Object

- Now let's create a class that:
  - derives from `Drawable`
  - is used to store (and render) a *house* object (as a rectangle and triangle) at a specified location and in a specified color.

Drawable Derived Object

```
class House(Drawable):
   def __init__(self, x = 0, y = 0, color = (0,0,0)):
      Drawable.__init__(self, x, y)
      self.__color = color

   def draw(self, surface):
      location = self.getLocation()
      pygame.draw.rect(surface, self.__color, [location[0]-15, location[1]-10, 30, 20])
      pygame.draw.polygon(surface, self.__color, [(location[0]-15, location[1]-10), \
                     (location[0]+15, location[1]-10), (location[0], location[1]-20)])
```

Main Script

- In our main program we can have a list of `Drawable` objects, appending new objects as needed.
- Then in our main loop, we just loop through this list, calling the `draw` method of each

Main Script

```
pygame.init()
surface = pygame.display.set_mode((400, 300))
pygame.display.set_caption('My Drawable Objects’)

drawables = []
newHouse = House(100, 100, (255, 0, 0))
drawables.append(newHouse)

while True:
    for event in pygame.event.get():
        if (event.type == pygame.QUIT) or (event.type == pygame.KEYDOWN \
                                    and event.__dict__['key'] == pygame.K_q):
            pygame.quit()
            exit()

    surface.fill((255, 255, 255))
    for drawable in drawables:
        drawable.draw(surface)
    pygame.display.update()
```

Animation

- So far, all we've done is create a static images.
- For some applications, this is sufficient.
- But for many others, including interactive graphics, this is not.
- To do animation, each time our infinite loop runs:
  1. Fill our surface with some clear color
  2. Make some updates to the items to be drawn.
  3. Draw the items on our surface
  4. Make a display update request

     `pygame.display.update()`

Animation

- As is, this loop will run as fast as possible.
- So that we have smooth animation (and don't kill our CPU) we typically want to set the frame rate.
- To do this we create a clock:

```
fpsClock = pygame.time.Clock()
```

- And then at the end of our main loop, call its **`tick`** method with a desired *frame per second* rate.
  - This is typically around 30

```
fpsClock.tick(30)
```

Example

```
while True:
    for event in pygame.event.get():
        if (event.type == pygame.QUIT) or (event.type == pygame.KEYDOWN \
                                and event.__dict__['key'] == pygame.K_q):
            pygame.quit()
            exit()

    surface.fill((204, 229, 255))
    for drawable in drawables:
        drawable.moveRight()
        drawable.draw(surface)
    pygame.display.update()
    fpsclock.tick(30)
```

Images

- Pygame also allows us to (relatively) easily copy an image or text to a surface.
- To load an image we use **`load`**:

```
imgSurface = pygame.image.load(filepath)
```

- This function returns a surface itself
- So we'll need to copy this surface to our main surface.
- Pygame calls this process *blitting*
- You can copy (**`blit`**) one surface to a location on another as:

```
targetsurface.blit(sourcesurface,(xloc, yloc))
```

- Pygame is able to load images onto Surface objects from PNG, JPG, GIF, and BMP image files

Example

```
catImg = pygame.image.load('cat.png')
catX = 10
catY = 10
surface.blit(catImg, (catX, catY))
```

Text

- Text is done a similar way
  - After all a font is really just a bunch of images, one per character.
- So to create a text "image"
  1. Create a Font object:

```
fontObj = pygame.font.Font(fontName, fontSize)
```

  2. Render a text object using the font

```
textSurface = fontObj.render(text, True, color)
```

  3. Copy (blit) the text object to a drawing surface

Text

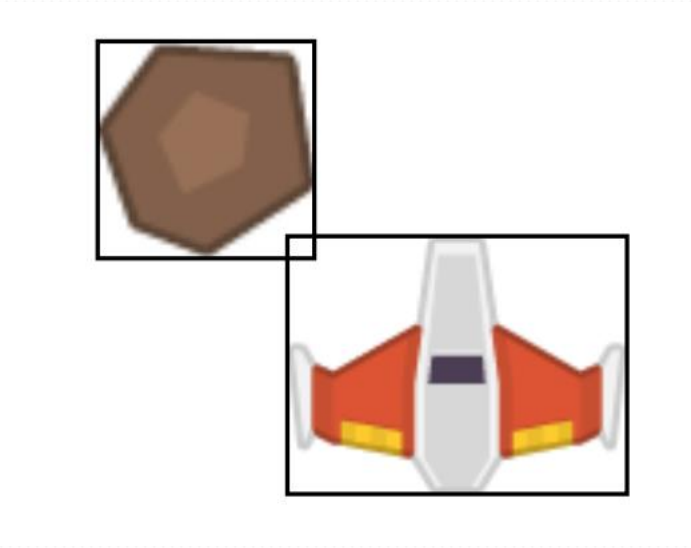
```
class Text(Drawable):
    def __init__(self, message = "Pygame", x = 0, y = 0, color = (0,0,0)):
        Drawable.__init__(self, x, y)
        fontObj = pygame.font.Font("freesansbold.ttf", 32)
        self.__surface = fontObj.render(message, True, color)

    def draw(self, surface):
        surface.blit(self.__surface, self.getLocation())
```

Collision Detection

- The last thing we'll talk about is collision detection.
- In real-time interactive graphics application, collision detection is usually done in a course-to-fine resolution approach.
  - That is, we first do a "crude" check to see if two things could potentially collide.
  - If they pass that, then we go to a more refined test.
  - And so on….
- One "middle resolution" test is to see if the *axis aligned bounding boxes* (AABB) around objects collide.
- Let's look at how to do this.

AABB Collision Detection

- An axis-aligned bounding box (AABB) is a rectangle will be *axis aligned* meaning that one side is along the x-axis and one is along the y-axis.



AABB Collision Detection

- Given two AABB rectangles (each of which have an `x, y, width, height` attribute), we can easily test if their bounding boxes intersect:

```
def intersect(rect1, rect2):
    if (rect1.x < rect2.x + rect2.width) and
       (rect1.x + rect1.width > rect2.x) and
       (rect1.y < rect2.y + rect2.height) and
        (rect1.height + rect1.y > rect2.y) :
         return True
    return False
```

AABB Collision Detection

- So if we want to test if two objects in our list intersect, we need to do pair-wise tests:

```
for i in range(len(drawables)):
   for j in range(i + 1, len(drawables)):
     if(intersect(drawables[i].get_rect(),drawables[j].get_rect())):
         print('Collision detected!')
```

AABB Collision Detection

- So maybe our `Drawable` class should require a method called `get_rect()`
- Good news: surfaces have their own `getRect()` method that we can leverage if our object is copying a surface to our main surface
  - Like we did with text.
  - However, they are axis aligned, so rotations may results in larger than expected surface rectangles.

AABB Collision Detection

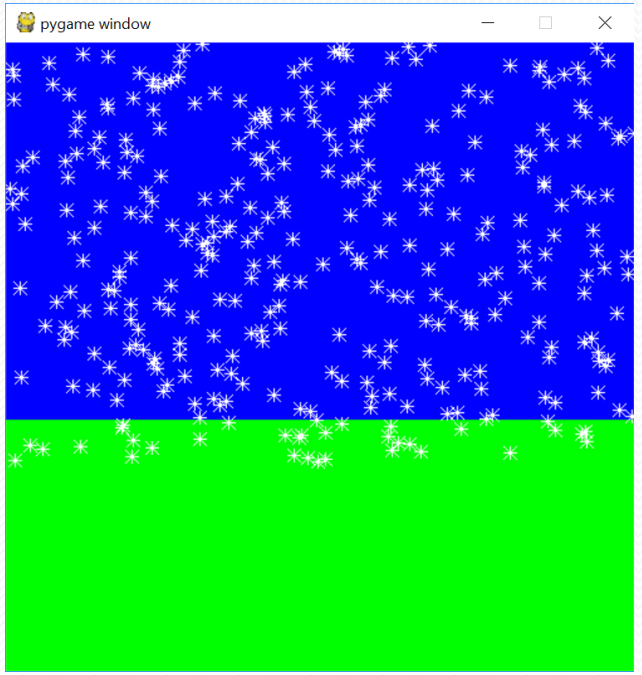
```
class Drawable(metaclass = abc.ABCMeta):
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y

        #other methods here

    @abc.abstractmethod
    def draw(self, surface):
        pass

    @abc.abstractmethod
    def get_rect(self):
        pass
```

AABB Collision Detection

```
class House(Drawable):
    #the other methods go here

    def get_rec(self):
        location = self.getLocation()
        return pygame.Rect([location[0]-15, location[1]-20, 30, 30])
```

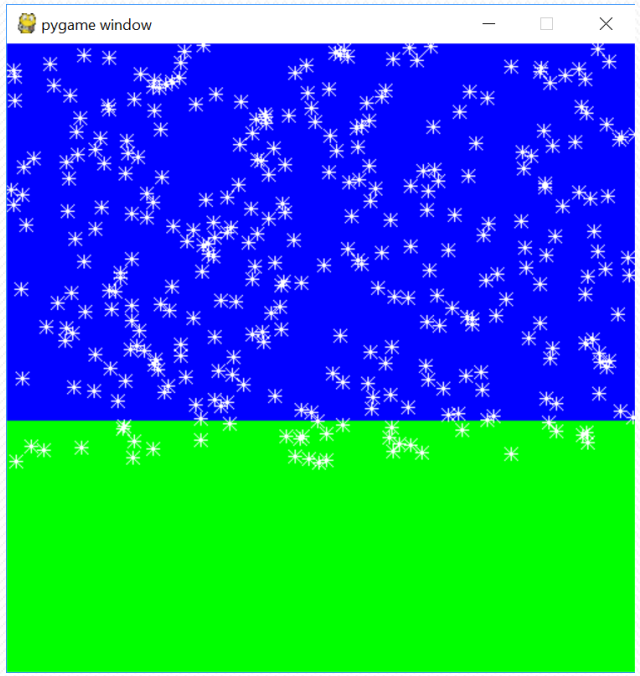
Lab 5

- Winter Wonderland!
- We'll give you an abstract base class that has:
  - Constructor to initialize the `(x,y)` location of object
  - Accessor and mutator methods for these attributes
  - An abstract method `draw`, that takes a surface as a parameter.
- You need to create two derived classes:
  - `Rectangle`
    - At a location, with some dimensions and color
  - `Snowflake`
    - At some location



Lab 5

- In the main application:
  - Draw a “ground plane” using your rectangle.
  - Draw a “sky plane”
  - On each iteration of your display loop
    - Draw all the existing drawable objects.
    - Increment the y-coordinate of all snowflakes
    - Create (and add to the list of drawables) a new snowflake with y=0 and x being some random location on the window
- Bonus!
  - When a snowflake is made, assign it a max y-value and only increment the y-value of snowflakes that haven’t hit their max yet.



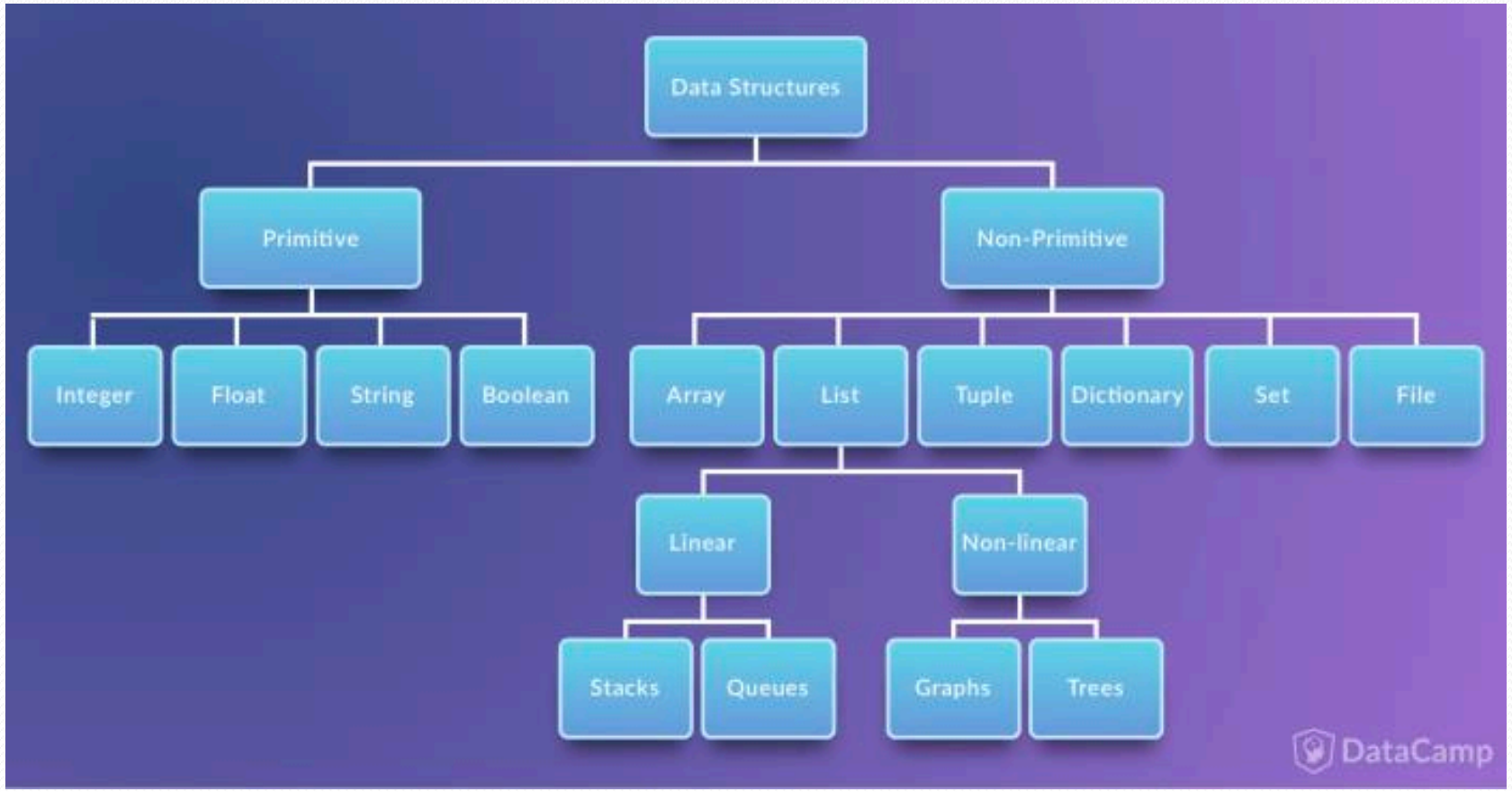
CS 172 - Computer Programming II

Memory Considerations

Objectives

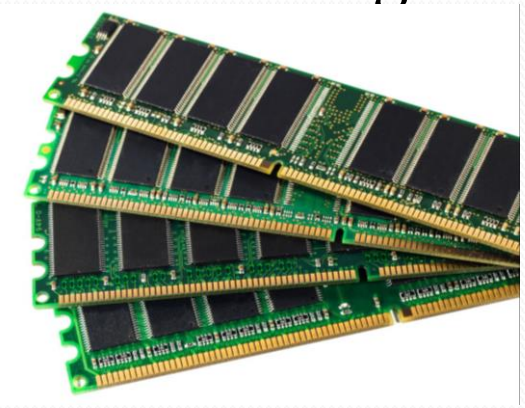
- Memory Considerations
- Core CS171 data structures

Data Structures



Data Structures

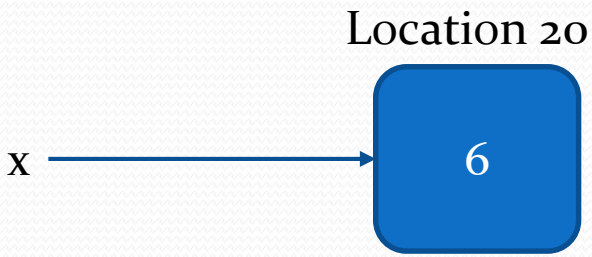
- Choosing the "right tool for the job" is a critical element of Computer Science.
- And here, by "tool", we mean data structure.
- In choosing the right data structure we want to take a few things into account:
  - Memory cost.
  - Computational cost.
- Often there's a trade-off between these things.



- In order to start thinking about this, let's first take a look at how Python deals with memory.

Memory

- A variable is really just storing a memory address.
- So when a variable is created, the program asks the computer for some available memory.
- In the case of a single value (like an integer), this is pretty straight forward
- Imagine the line of code `x = 6`
  - The program asks the computer for enough memory to store an integer.
    - Let's imagine that the computer finds memory location `20`
  - Now `x` is really storing memory address `20`
    - And at that memory location the value `6` is placed
  - We can visualize this with blocks and arrows



Contiguous Memory

- But what happens if we want to store the data for a list (or tuple, or queue)?

```
x = [2, 4, 5]
```

- We need memory for all of those elements!
- Then what does `x` refer to?
  - The memory address of the first element.
- So how do we know where the second element is?
- What makes tuples and lists a good default data structure is that *all of the elements are placed after each other in memory*
  - We call this **contiguous** memory.
- This allows for fast access to locations within the list/tuple.
- Why?
  - We can use the starting memory location and the index to **compute** the memory locations.

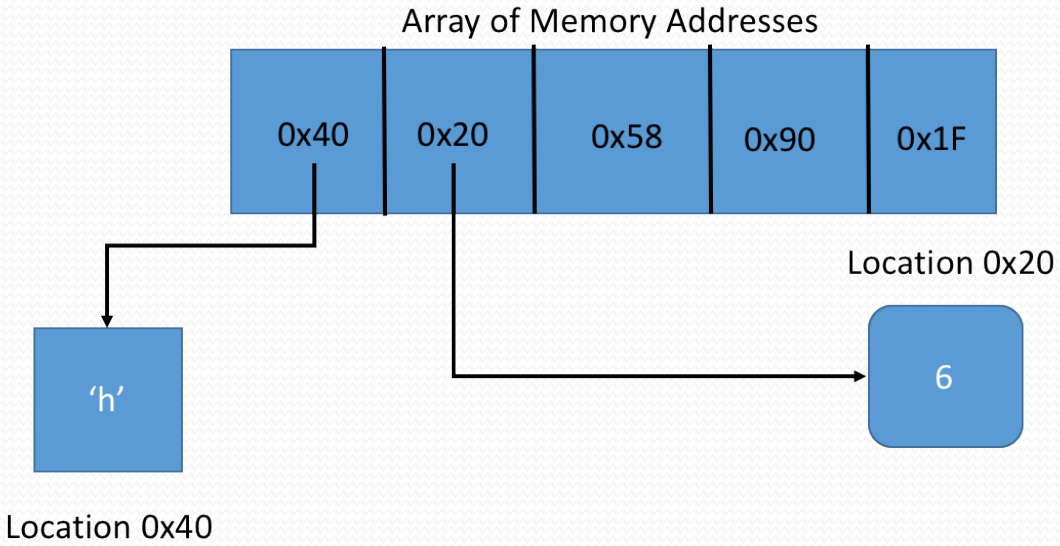
Contiguous Memory

- For example, imagine that in order to store each element we need 2 bytes.
- If we have 5 elements, then we need 10 total bytes.
- When the program runs the line `x = [78, 45, 12, 89, 56]`, it asks the computer for the necessary chunk of contiguous memory.
  - And now `x` refers to the beginning of this contiguous memory.
  - Let's imagine this is memory location `48252`
- Then if we want to access the second element, `x[2]`, we just compute

$$48252 + 2 * 2 = 48256$$

Contiguous Memory

- So how does Python know how much room to allocate if it allows for mixed types in its lists/tuples?
- Python stores memory address in each location, each of which point to the memory address storing its actual data!



Memory Management

- So we know when/how our program requests memory from the operating system.
- But what about releasing it?
  - After all, we can't just let our programs ask for more and more memory without ever releasing/freeing memory
- The operating system keeps track of who owns what memory
- So at the very least the operating system *should* release all the memory pertaining to a program when that program terminates.
- However, often programs will run a long time, with potential heavy memory requirements.
- If we (or the programming language) doesn't release memory when its done with it, this could result in a huge performance impact on our computer!

Memory Management

- When might we be done with some memory?
- At the very least, when a variable goes *out of scope*
- When might this happen?
  - When we leave a function/method where it was created.
- But sometimes memory is referred to by several variables.
  - How do we handle this?
- Languages like C make us manage this ourselves.
  - Which is powerful, but dangerous!

Memory Management

- Java, and Python, do automatic *garbage collection* and *reference counting*
  - Python keeps track of how many times a memory address is used by a program.
  - Each time a variable that is referencing a memory address leaves scope, the count is reduced by one.
  - If the count becomes zero (or less) then the memory address is released by the program.

Tuples and Lists

- Now that we know a little more about how Python handles memory, let's think a bit more about those tuple and list objects/classes!

List Memory

- When a `list` is created, some amount of memory is allocated for it
  - At least enough to store the memory addresses for the data requested.
  - But typically also "a little extra"
- As a list grows, if it outgrows its allocated memory:
  - a new, larger chunk of memory is requested
  - the memory addresses are copied over to that new location
  - the memory address of the start of this new chunk is now stored in our list variable.
- As a list shrinks, if it's wasting too much space, a new area will be allocated, and the memory addresses will be copied.

Tuple Memory

- Since tuples are immutable, the operating system knows exactly how much memory to allocate for them when they're created.
  - So no memory is wasted
  - Therefore tuples are light weight

CS 172 - Computer Programming II

Queues and Stacks

Objectives

- Queues
- Stacks
- Priority Queues

More Data Structures

- Tuples and lists are great "general use" data structures.
- And linked lists are another good general use data structure with different pros and cons.
- However, for certain applications and algorithms a different data structure may be appropriate
  - Although it may be built upon one of these aforementioned data structures
- One data structure that can be built on top of those general use data structures are *queues.*

Queues

- The word queue is related to the English word for a line.
- If we're at the supermarket, we'd hope that the first person in line, is the first person served.
- And once he/she is served, they are removed from the front of the line.
- If anyone new shows up, they're added to the end of the line.
- Therefore a queue is also referred to as a FIFO data structure
  - First In – First Out

Queues

- Hopefully we can think of a lot of good computer-science applications for Queues:
  - Storing (and servicing) messages received on a network
  - Storing (and servicing) requests to use the CPU
  - Etc.
- So if we were to design a queue data structure, we would want it to efficiently allow for:
  - Adding things to the end
  - Removing things from the front.
- These operations are typically called *enqueue* and *dequeue*, respectively
  - However in Python they're referred to as *put* and *get*, respectively

Queues in Python

- Python provides for us the **`Queue`** class.
- Actually the **`Queue`** class is within the **`queue`** package

```
from queue import Queue
```

- We may want to do this if we primarily care about quickly getting (and removing) things from the front of our collection and adding things to the rear.
- The **`Queue`** public interface is rather simple.
- The primary methods are
  - **`put()`** `# add an element to the back of the queue`
  - **`get()`** `# retrieve and remove element at the front of the queue`
  - **`empty()`** `# returns True if the queue is empty`

Example

```
julie = Customer("Julie", 18)
harry = Customer("Harry", 20)
marie = Customer("Marie", 15)

# using the Python Queue class
q = Queue()
q.put(julie)
q.put(harry)
q.put(marie)

print('Servicing customers in the order they arrived (FIFO):')
while not q.empty():
    print(q.get())
```

Stacks

- To think about a stack, think about a pile of pancakes.
- Typically, the last one place on top of that pile, will be the first one served (and removed).
- In addition, if something needs to be added to the pile, it gets added on top.
- Therefore a stack is referred to as a LIFO data structure
  - Last In – First Out

Stacks

- Examples computer-science applications for Stacks:
  - Syntax Parsing
  - Functions call
  - Etc.
- So if we were to design a stack data structure, we would want it to efficiently allow for:
  - Adding things to the top
  - Removing things from the top.
- These operations are typically called *push* and *pop*, respectively
  - Of course again, Python opted to call them *put* and *get*

Stacks

- Python provides for us the **`LifoQueue`** class to simulate a stack
- To use these we need to:

```
from queue import LifoQueue
```

- As with all the queue derived data structures, **`LifoQueue`** has **`put`**, **`get`** and **`empty`** methods:
  - **`put()`** `# add an element to the top of the stack`
  - **`get()`** `# retrieve and remove element at the top of the stack`
  - **`empty()`** `# returns True if the stack is empty`

Example

```
julie = Customer("Julie", 18)
harry = Customer("Harry", 20)
marie = Customer("Marie", 15)

# using the Python Stack class --> LifoQueue
s = LifoQueue()
s.put(julie)
s.put(harry)
s.put(marie)

print('Servicing customers starting with one that arrived last (LIFO):')
while not s.empty():
    print(s.get())
```

Priority Queues

- A priority queue is a queue that has a priority and an object pair in each location
  - (priority, object).
- An element with high priority is served before an element with low priority.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.
- Examples of priority queue applications
  - CPU Scheduling
  - Graph algorithms

Priority Queues

- To use a priority queue

```
from queue import PriorityQueue
```

- And when we put things on it give it a (priority, data) pair:

```
mypq = PriorityQueue()
mypq.put((4, myData))
```

- And then we can get the tuple out as:

```
(priority, value) = mypq.get()
```

Example

```
tasks = PriorityQueue()

tasks.put((2, “code”))
tasks.put((1, “eat”))
tasks.put((3, “sleep”))

while not tasks.empty():
    next_item = pq.get()
    print(next_item)
```

Implement a Queue

- As an in-class exercise, lets implement our own Queue class, which we'll call `MyQueue`
- Design questions…
  - What attributes do we initially think we need?
  - What should our `__init__` method be able to do?
  - What should our public interface be?
    - `put()`, `get()`, `empty()`
    - `__str__`

CS 172 - Computer Programming II

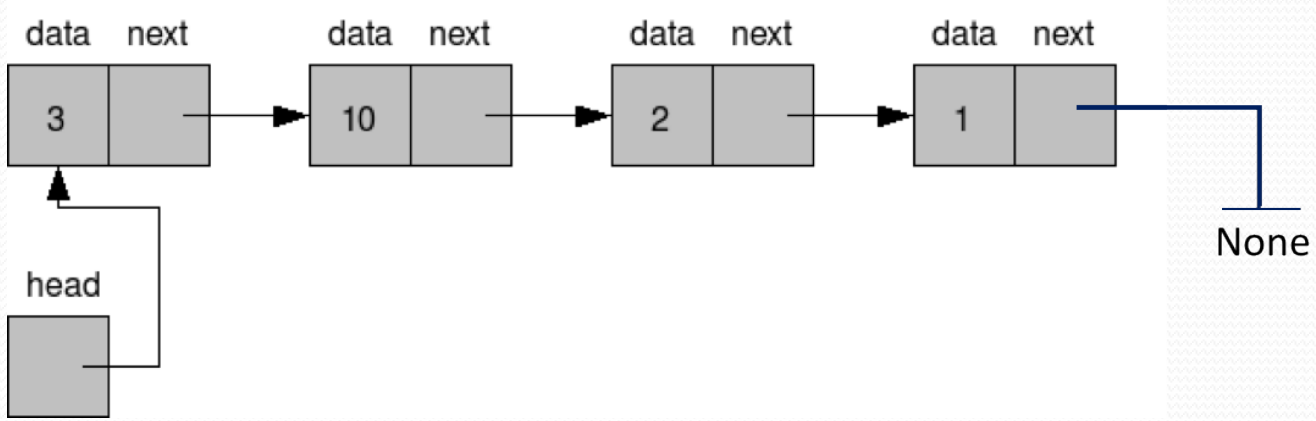
Linked Lists

Objectives

- Understand the need for linked lists.
- Understand the design of a node and how it can be used to make simple linked lists.
- Perform simple linked list manipulations
- Use iterative or recursive algorithms to traverse a linked list.
- Be able to write code that performs basic list operations including:
  - Inserting elements into a list at the front, rear, or in order
  - Search a list for an element
  - Remove a selected element from a list
- Implement useful operators.

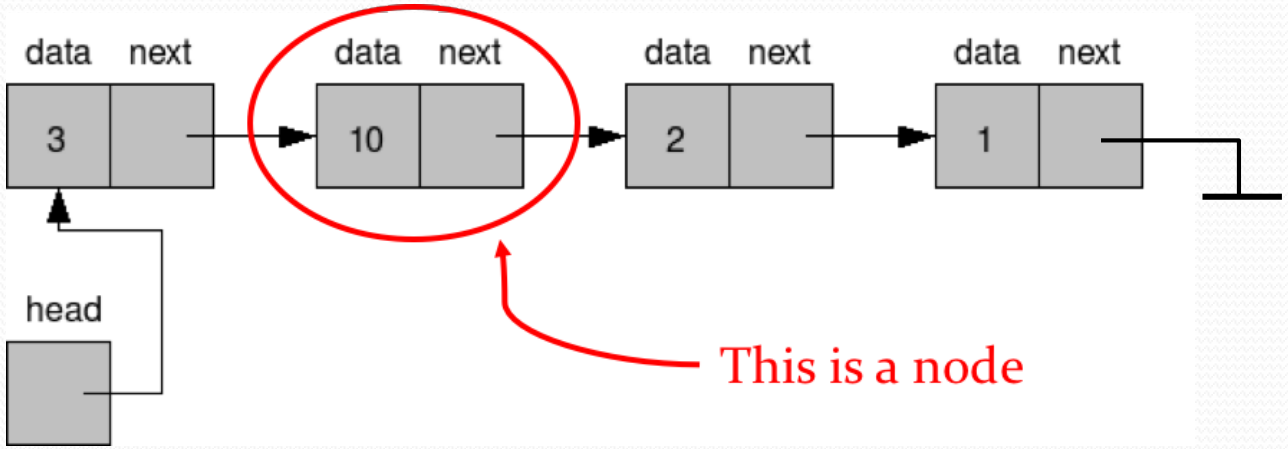
Link Lists

- There are times when using a data structure that relies on contiguous memory isn't appropriate.
  - In particular, when we need to store a lot of stuff
- To get around this limitation, we'll explore (and loosely implement) another type of data structure, linked lists!



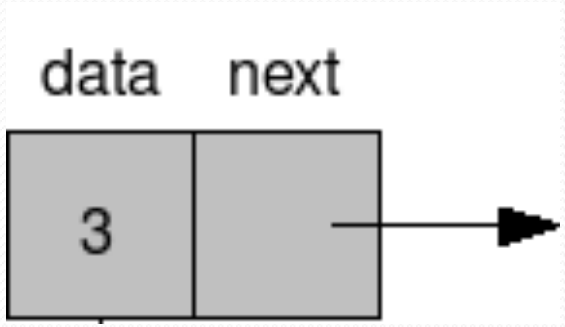
Nodes

- The basic element in a linked list is called a *node*.
- A node typically has two attributes:
  1. Some data to be stored
  2. A reference to the next element in the list



Nodes

- Let's start off by designing and creating a Node class.
  - What attributes do we need?
    - Data
    - Reference to next node
  - So what should be the public interface for this class?
    - Constructor, Getters and Setters for both attributes
  - Any useful operators to overload?
    - Most likely: `__str__`

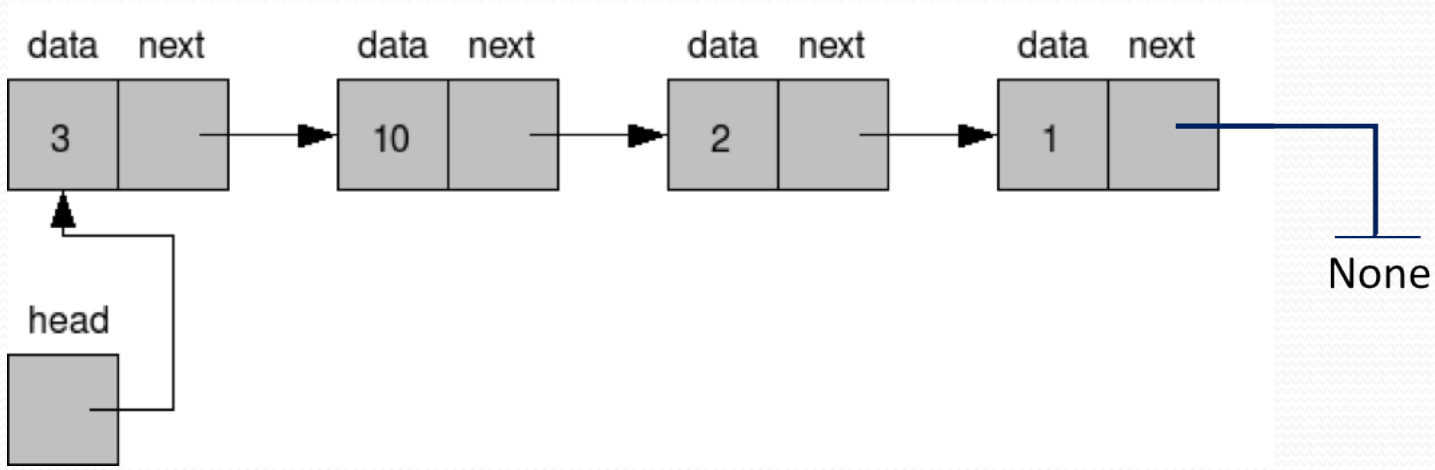


The Node class

```
class Node():
   def __init__(self, data, next = None):
         self.__data = data
         self.__next = next

   def getData(self):
         return self.__data

   def getNext(self):
         return self.__next

   def setData(self,d):
         self.__data = d

   def setNext(self,n):
         self.__next = n

   def __str__(self):
         return str(self.__data) + " whose next node is " + str(self.__next)
```

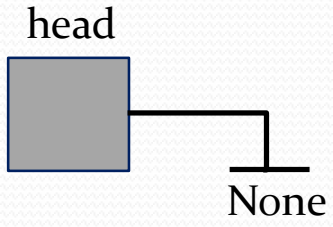
Linked List

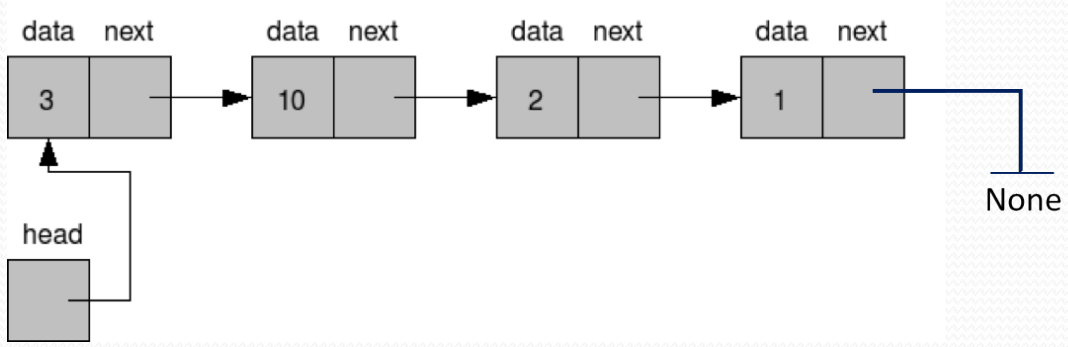
- We can “link” several nodes together.
- Then, given a node, we can *traverse* the list.



Linked List

- All this behavior (linking, traversing, etc.) is usually *encapsulated* within another class.
- This class (which is our actual Linked List class), typically has a single attribute, the *head* of the linked list.
  - Initially this may be `None`
  - But once something is put on the linked list, then this will refer to the first node of the linked list.





Linked List Class

- Let's create our own linked list class!
- It needs at least one attribute
  - Head: point to the first item in the list or None if empty
- How about the public interface?
  - Constructor
  - Add node
  - Remove node
- What additional methods?
  - We may want to check if the list is empty or not
- What operators to overload?
  - We may want to print our linked list, get a node from a given location, find out the size of the list.

Linked List Class

- Methods:
  - `__init__`
  - `isEmpty()`
  - `append(Node)`
  - `remove()`
- Operators to overload
  - `__getitem__`
  - `__str__`
  - `__len__`

`__init__` and `isEmpty()`

- For the constructor:
  - There is only one attribute: `head` and at the start the list is empty so `head` points to **`None`**.
- For the `isEmpty()`:
  - If `head` is pointing to **`None`**, then the list is empty (no nodes)

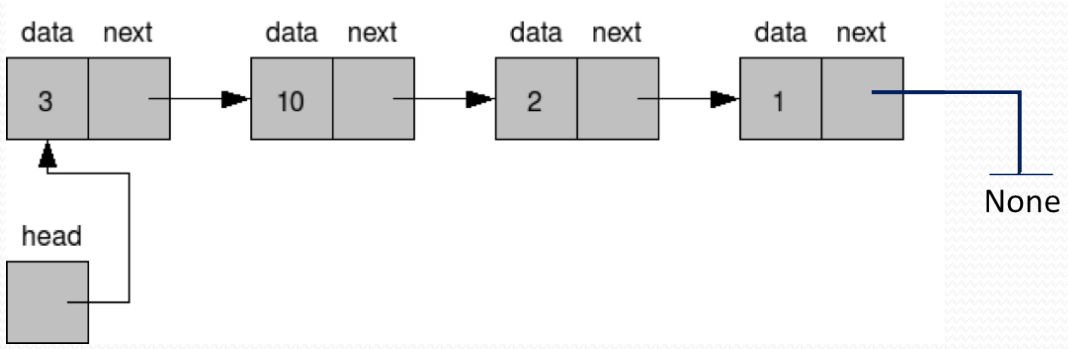
```
class LinkedList():

    def __init__(self):
        self.__head = None

    def isEmpty(self):
        return self.__head == None
```

Overriding `__len__`

- The length of a link list is the number of nodes in it.
- But we only have direct access the first node.
  - Via `head`
- As you'll see with a lot of these methods, we'll need to *traverse* the link lists by following the link
  - i.e. the `next` attribute
- And we're done when we hit a node that has **`None`** as the value of its `next` attribute.

Overriding `__len__`

```
def __len__(self):
        if self.__head == None:
            return 0

        current = self.__head
        counter = 1

        while current.getNext() != None:
            counter += 1
            current = current.getNext()
        return counter
```



The `append()` method

- Process:
  1. Create a new `Node` object
  2. If the linked list is empty, the `head` will point to this object.
  3. Otherwise find the last `Node` and set its `next` attribute to this `Node`.

The append() method

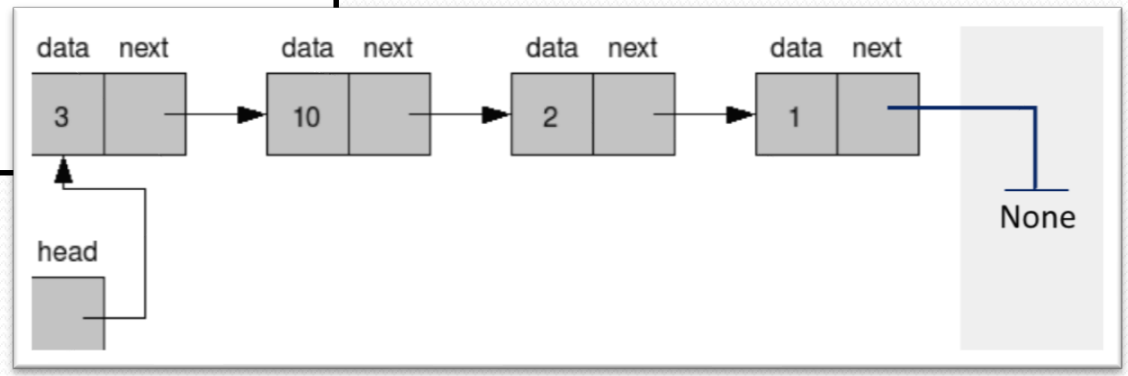
```
def append(self, data):
    newNode = Node(data)

    if self.isEmpty() :
        self.__head = newNode

    else:
        current = self.__head
        while current.getNext() != None:
            current = current.getNext()

        current.setNext(newNode)
```

Optional attribute: `tail`

- Since it's so common to add things at the end of a linked list, instead of having to traverse to the last node, we could keep track of the last node
- We could do that by adding a `tail` attribute to our Linked List.
  - `tail` will keep track of the last element in the list.
- Now we'll have to think about what to do for the different cases:
  - Empty list: head and tail point to None
  - Length is one: head and tail point to the same node and it's not None
  - Length > 1: head and tail point to different nodes

Overriding `__str__`

```
def __str__(self):
        mystr = ''
        current = self.__head

        while current !=  None:
            mystr += str(current.getData()) + ' --> '
            current = current.getNext()

        return mystr
```



Overriding `__getitem__`

- It would be a good idea to overload the `[]` operator so we can access Nodes from the list.
- We can do this via the `__getitem__` method

```
def __getitem__(self, index):
    if index > len(self):
        raise IndexError
    current = self.__head
    for i in range(index):
        current = current.getNext()
    return current.getData()
```

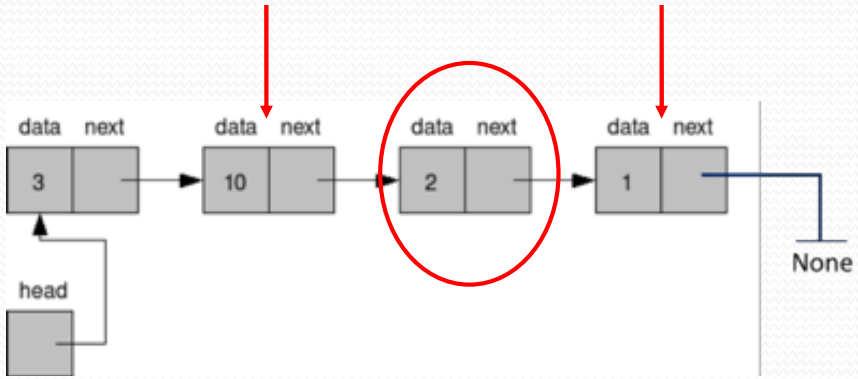
Searching the linked list

- It may be useful to have a search method

```
# search for item in linked list
    def search(self, item):
        current = self.__head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()

        return found
```

The remove() method

- Probably the toughest operation is removal
- We need to find keep track of two nodes
  - The node before the one to be removed
  - The one after the one to be removed
- And we need to think of the different cases:
  - No elements
  - Length is one
  - More than one



The `remove()` method

```
def remove(self, item):
        current = self.__head
        previous = None
        found = False

        # first find item in the list
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:  # item was in the first node
            self.__head = current.getNext()
        else:  # item was somewhere after the first node
            previous.setNext(current.getNext())
```

Limitations

- Linked lists give us the benefit of
  - Quick insertion without big memory load
- But aren't good at inspection
  - lists/arrays are constant time to get `x[4]`
  - A linked list would have to go from the head
- For searching
  - If a list/array is sorted, then we can search in `logn` time via Binary Search
  - But even if a linked list is sorted, we still need to search from the head.

CS 172 - Computer Programming II

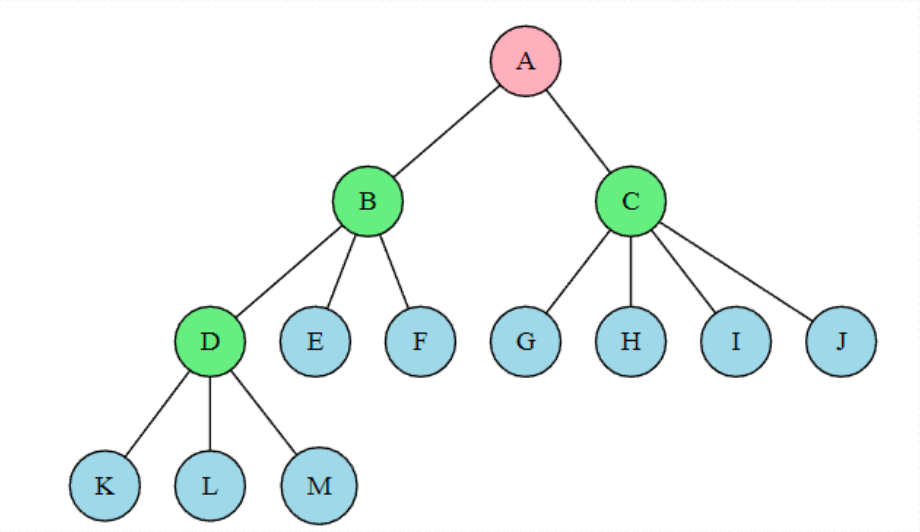
Binary Search Trees

Objectives

- Define Trees
- Define Binary Search Trees
- Describe how to traverse Binary Search Trees

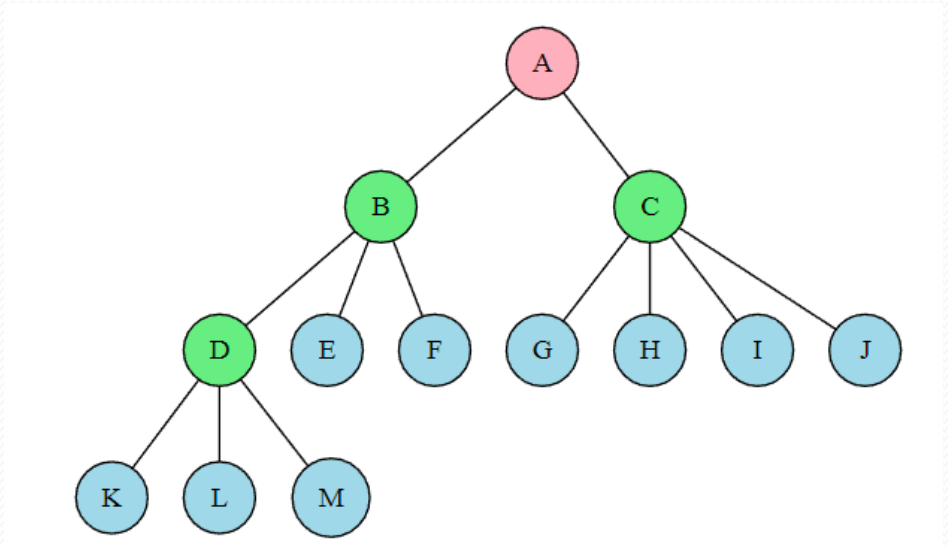
Trees

- Another common type of data structure is a tree
- Like linked lists, trees are made up of nodes.
- And nodes store both data and links to other nodes.
- However, unlike linked lists, a node can connect to more than one or two other nodes.

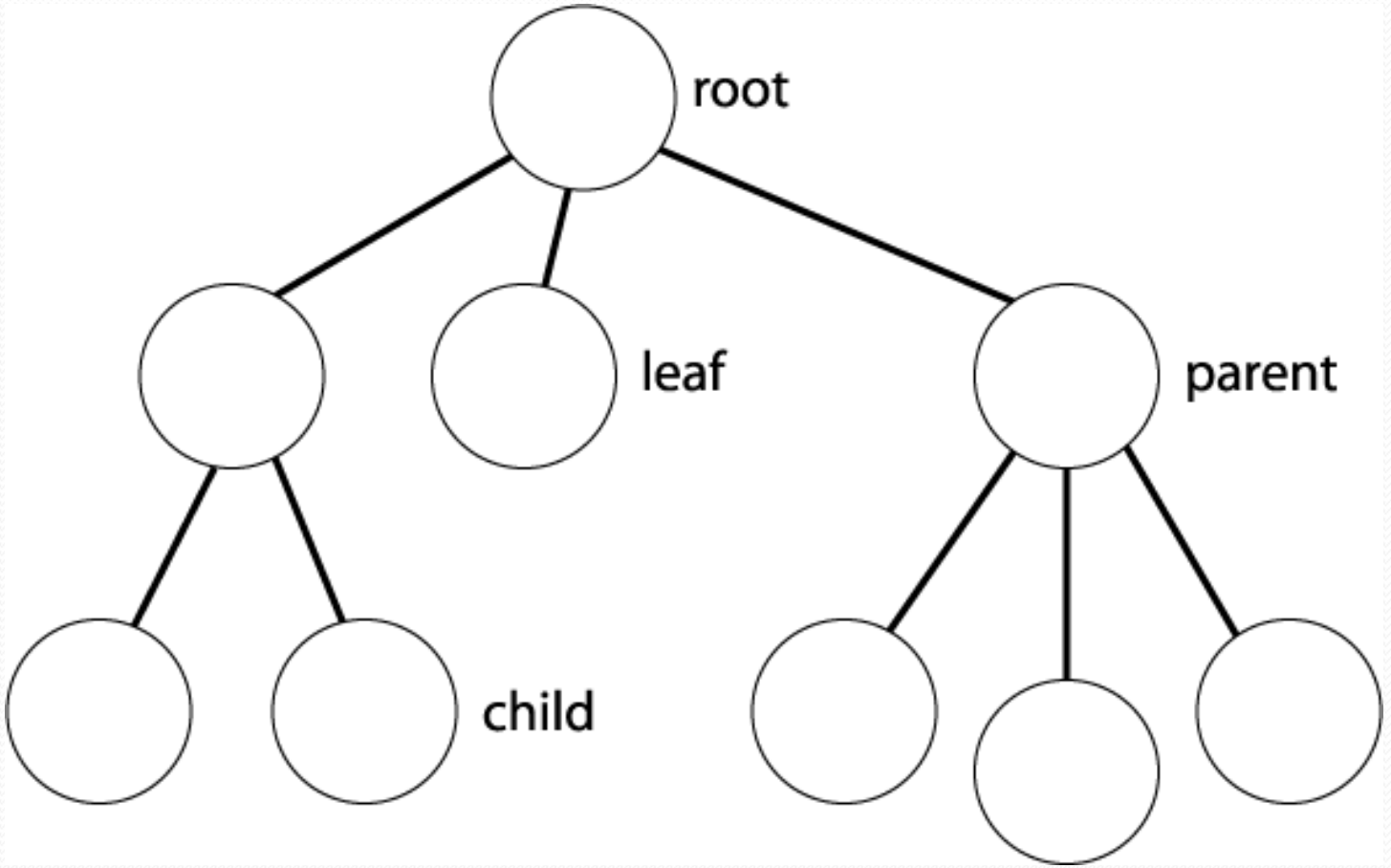


Trees

- The first node in a tree is referred to as the root
- The root's links are called its children.
- The children of the root refer to the root as its parent.
- This relationship between parents and children exists for each node
  - Technically the root's parent is `None`
- A node with no children is called a leaf

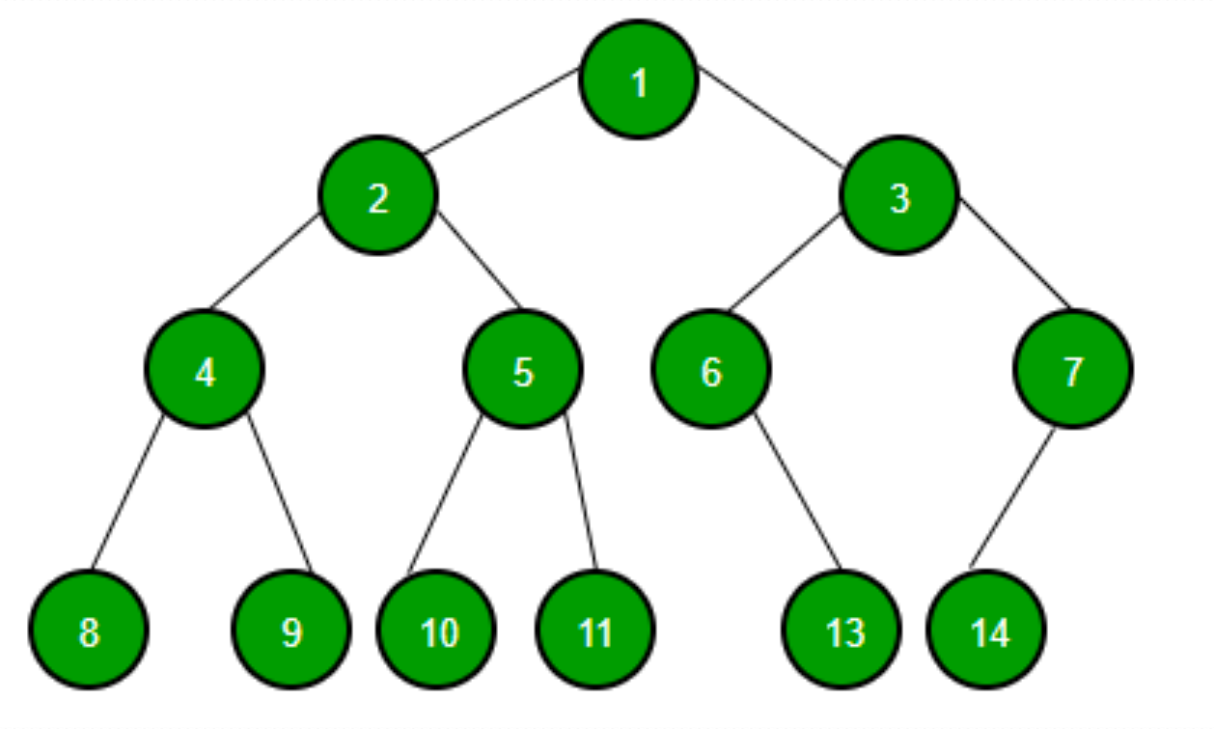


Trees



Binary Trees

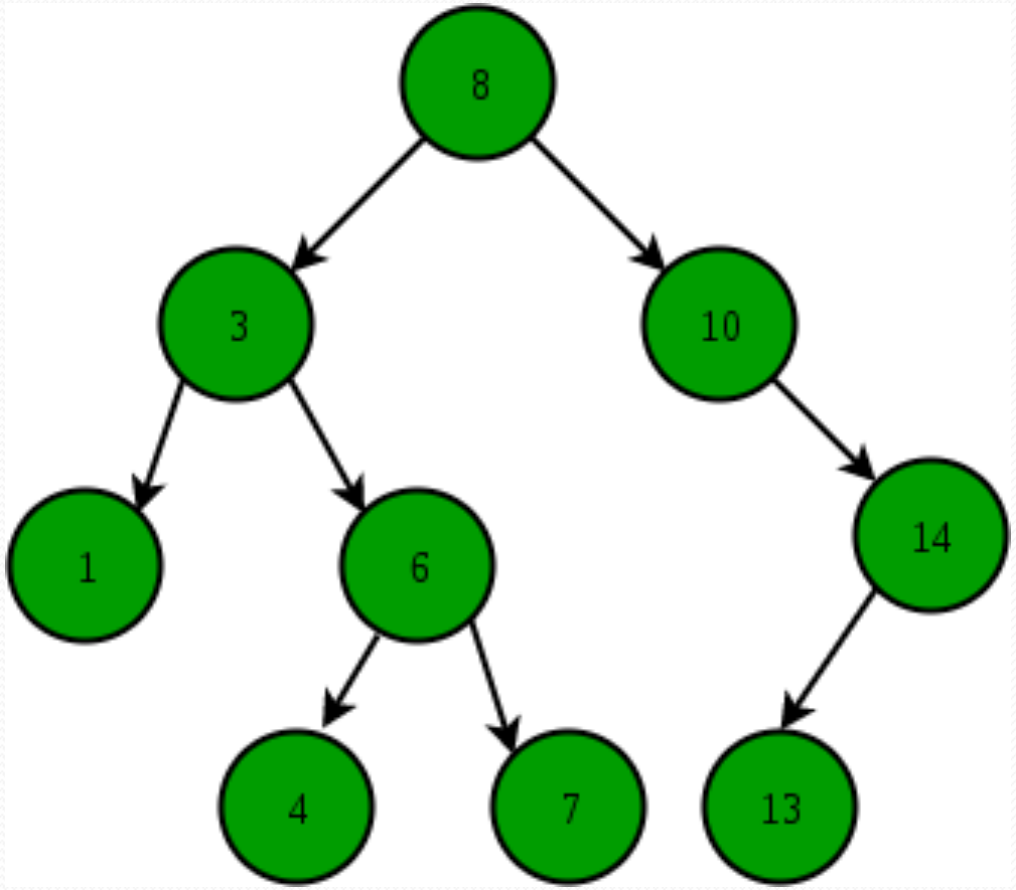
- One type of tree is a binary tree
- A binary tree's nodes can have at most two children.



Binary Search Trees (BST)

- A special type of binary tree is a binary search tree
- Binary search trees (BSTs) have the benefit of linked-lists but are far more efficient at finding things.
  - We don't need to search from head to tail through all the nodes
- The requirement for a binary tree to be a binary search tree is that for all nodes:
  - its left child must have a value less than its parent (or be `None`), and
  - its right child must have a value greater than its parent (or be `None`).

Binary Search Trees (BST)

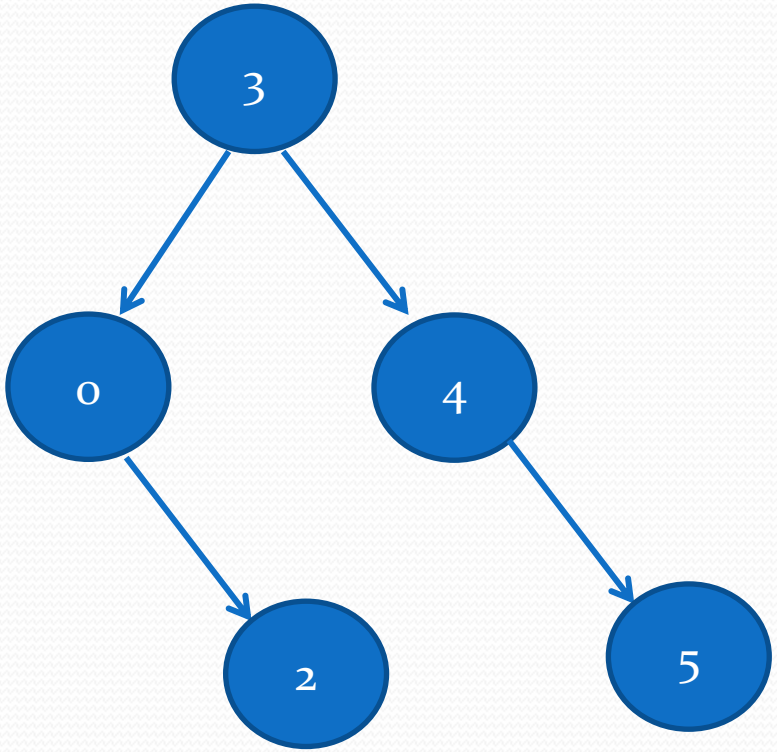


BST – Inserting nodes

- Let's first look at "manually" creating a binary search tree
- Places nodes `[3, 4, 5, 0, 2]`, in that order, in an empty BST

BST – Inserting nodes

```
[3, 4, 5, 0, 2]
```



BST - Search

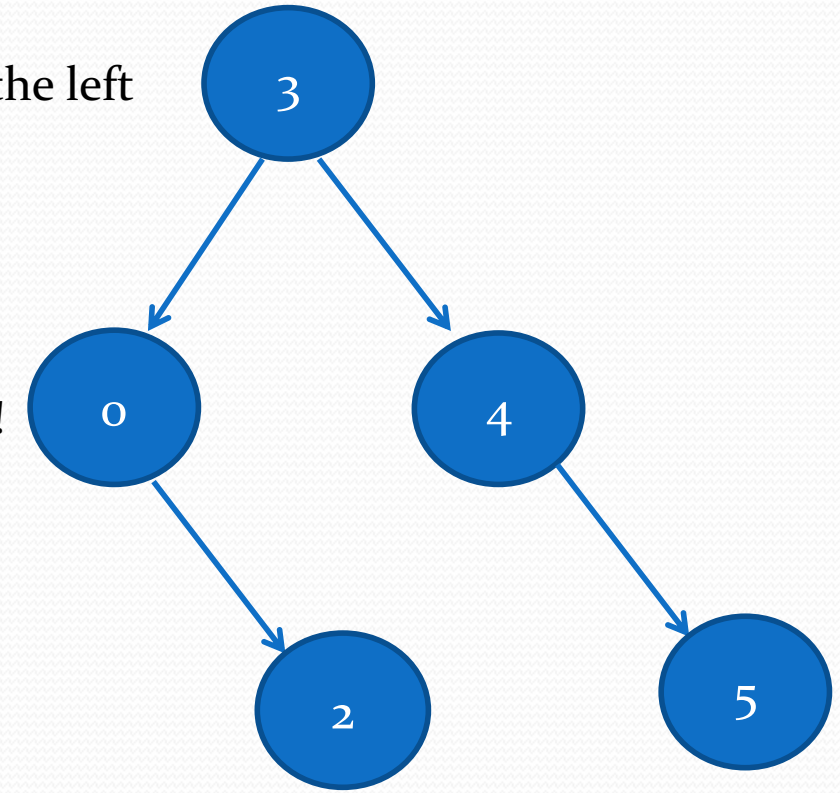
- Now imagine that we want to search for the value `0`
- Again let's start off by doing this manually

BST - Search

Search for 0

0 < 3 → search on the left

0 == 0 → Found!

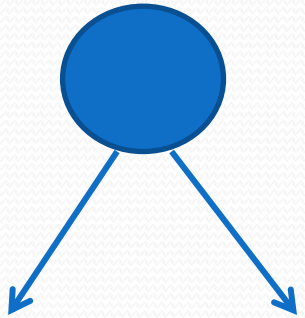


Why would we use this?

- What are some real-world examples of BST's?
  - Trees are often used in search, game logic, autocomplete tasks, and graphics.
  - Manipulate hierarchical data.
  - Make information easy to search
  - Manipulate sorted lists of data
  - As a workflow for compositing digital images for visual effects.
  - Router algorithms

BST In Python

- Ok, let's try to code some of this!
- Let's start off with our nodes
  - What attributes do they need?
    - Data
    - Link to left
    - Link to right
  - How about a public interface?
    - Constructor
    - Getter for the data
    - Getters for left and right links
    - Setters for left and right links

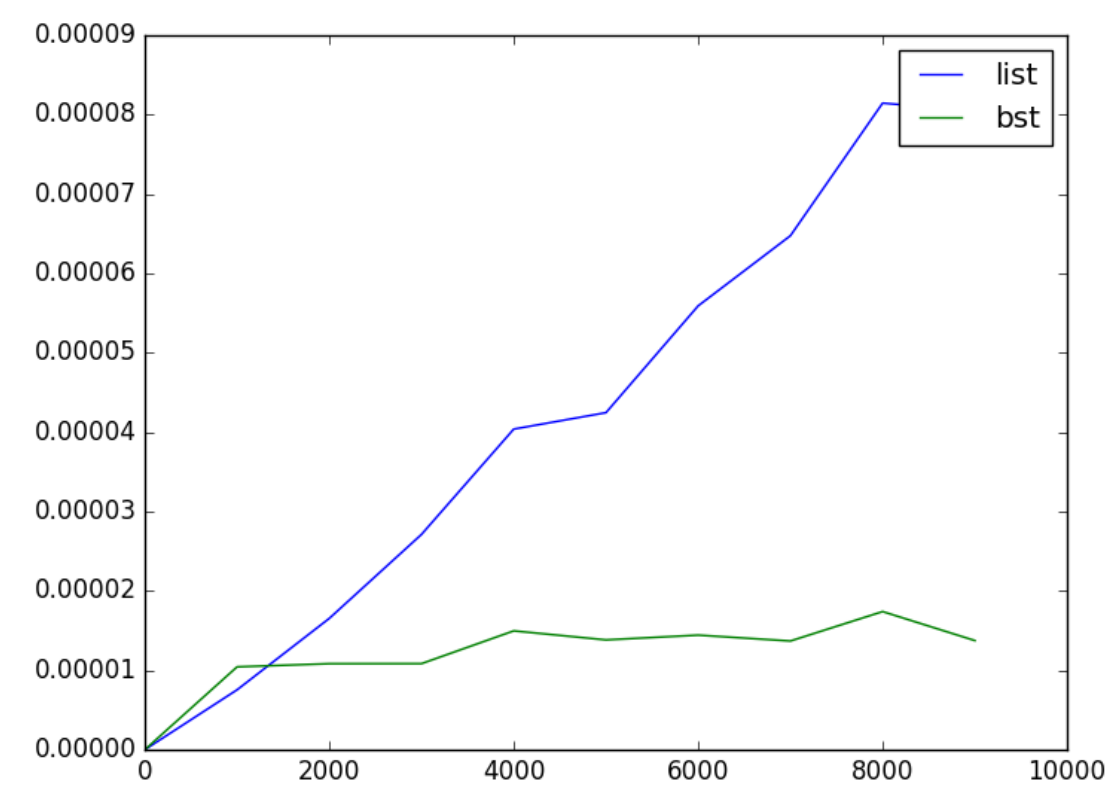


BST In Python

- Now our BST class
  - What does it need?
    - A way to track the root
  - What should we have in the public interface?
    - Insert nodes
    - Delete nodes
    - Search for a value
    - Print the tree
- Let's code up `search` and `insert` methods
  - We could do this iteratively or recursively.

Lab

- In this lab you'll explore efficiency in searching an array vs. a binary search tree (BST)
- In particular, you will
  - Populate a (regular) list and a BST with n random numbers.
    - We will provide the code for inserting and searching in a BST
  - Search for n items within your list and BST and compute the overall time to do so.
  - Vary the size n and plot average search time vs n



Time to Play!

- Go to kahoot.it on your laptop
  - Or Use the app on your phone if you have installed
- Type the game pin and press Enter
- For your nickname use your Drexel user id in the form **abc123**