CS 172 - Computer Programming II

Course Overview

Where do I find the syllabus and schedule?

- Log into Blackboard Learn
  - Go to your **lab** section
  - On the left-hand side menu click on Course Information and Course Schedule

Course Instructors

Prof. Adelaida A. Medlock

- Email: aalban@drexel.edu
- Office: 3675 Market St., Room 1064
- Office hours:
  - Tuesdays 10:00 am – 12:00 pm
  - And by appointment

Teacher Assistants

TAs should be your first stop for questions on material and grading

Name	Office Hours
Yahmin Haj	Wednesdays 6:00 – 8:00 PM
Uday Jain	Thursdays 3:00 – 5:00 PM
Siwu Li	Thursdays 5:00 - 7:00 PM Fridays 4:00 – 6:00 PM

Office hours schedule can also be found at: www.cs.drexel.edu/clc

Office Hours held at the CLC: 3675 Market St. Room 1066

Course Assistants

TAs should be your first stop for questions on material and grading

Name	Office Hours
Pranav Bhogal	
Tigist Mebrate	Mondays 6:00 – 8:00 PM
Vistrit Pandey	

Office hours schedule can also be found at: www.cs.drexel.edu/clc

Office Hours held at the CLC: 3675 Market St. Room 1066

Office Hours Etiquette

- Please make sure to contact the TAs during the posted office hours
  - Remember that we do not have office hours during the weekends
- You are welcome to post questions in Discord at any time
  - if any TA is available and able to answer the question, they may do so
  - but please be mindful and remember that you may not get an instant answer if the questions are posted outside your course office hours.
- Please remember that your TAs are also students who are taking courses, working on their own assignments, and some may be also working on a co-op job.

How to make the most of office hours

- Seek assistance immediately after you first begin to experience difficulty in class.
- Come prepared: Be able to point out specifically where you are getting stuck.
- Have specific questions, talk about what you have tried, and what resources you have used.
- If you have more than one issue, come with a list.
- Do not send screen-shots of your code
  - TAs/instructors cannot help you debug code sent in an image
- Be patient
  - particularly during busy times such as mid-terms, final exams, etc.
- Assume responsibility
  - A TA's role is to help and guide you, not to do the work for you

Pre-Requisites

- C or better in CS 171
  - Students earning less than a C in CS 171 are advised to re-take CS 171

Description and Learning Goals

- This is the second in a two-term sequence of programming (CS 171- CS 172)
- Required for all majoring in CS, SE and those minoring in CS
- Topics covered include object-oriented design, inheritance hierarchies, information hiding principles, and good programming style.
- Goal is to be able to write a working program using appropriate constructs when presented with a problem description.

Objectives

- By the end of this course, you should be able to
  - Import and use Python modules.
  - Design and Implement Object Oriented Programs.
  - Be familiar with common data structures such as Lists and Stacks.
  - Understand how objects are used to improve code development.

Required Textbooks

- CS 172: Computer Programming II
  - Zybook (learn.zybooks.com)
  - Book code: DREXELCS172MedlockWinter2022
  - Price: $88 (buy directly from Zybooks)
  - Homework, lab, and reading assignments will be given in Zybooks.

- Making Games with Python & PyGame
  - https://inventwithpython.com/pygame/
  - Read online for free

Required Software

- The official language of the course is Python 3
  - Available for free at www.python.org
- Blackboard Learn:
  - https://learn.dcollege.net
  - Lecture notes, examples, video recordings, and announcements will be posted in Bb Learn.
- Discord
  - Join the CS 172 Discord Channel (CCI) ASAP.
    - Instructions are posted in Bb Learn, on the left-hand side menu
  - Official communication tool and TA office hours

Required Software

- IDE (free) either one of these:
  - Thonny: https://thonny.org/
  - PyCharm Community Edition: https://www.jetbrains.com/pycharm

Thonny
Python IDE for beginners

PC PyCharm

Course Assessment

- Your grade will have seven components:
  - Reading Assignments 10%
  - Lab Assignments 15%
  - Individual Programing Assignments 15%
  - Midterm Exam 30%
  - Final Exam 30%

Course Assessment

- Final grades will be assigned via a normal grading scale

Points	Grade	Points	Grade	Points	Grade
97 - 100	A+	83 - 86.99	B	70 - 72.99	C-
93 - 96.99	A	80 - 82.99	B-	65 - 69.99	D+
90 - 92.99	A-	77 - 79.99	C+	60 - 64.99	D
87 - 89.99	B+	73 - 76.99	C	0 - 59.99	F

Reading Assignments

- Every week a chapter from the course Zybook will be assigned.
- These chapters also contain in-line activities.
  - Your (correct) participation in these will be used as your reading grade.
  - Participation and Challenge Activities are required.
- The reading assignment for week 5 is from the Sweigart book (Pygames)
  - You must complete the reading and then take a short quiz on Bb Learn
- Due Date:
  - Sundays 11:59pm
  - No late assignments will be accepted

Lab Assignments

- Each week there will be a lab assignment.
- Lab assignments will be posted in Zybooks and must be submitted via Zybooks
  - Except for Lab 5 which will be posted / submitted via Bb Learn.
- Attendance to the lab is mandatory.
  - You must be present in the lab to get credit.
  - TAs will take attendance at the beginning of the lab, make sure you answer when the TA calls your name.
    - If you arrive later to the lab, please see the TA so they can record your attendance
  - If you miss lab for a valid reason, contact your course instructor ASAP (before you miss the lab is always better)
    - You will be required to attend a lab make-up session
  - If you are late to the lab a late penalty will be apply to your lab grade

Lab Assignments

- You may choose to work alone or with 1 lab partner of your choosing (who is in the same lab section).
  - If you work with a partner, you must put a comment at the beginning of your code with both of your names and user IDs, so we know who is getting credit.
  - Even if you work with a lab partner, each of you must submit the lab via Zybooks
- Lab assignments are due by 11:59 PM on the day of your lab session.
  - No late assignments will be accepted
- Lab assignment grades will have 2 parts:
  - an auto-graded part (test cases)
  - a manual grading part (compliance with requirements)

Programming Assignments

- Homework assignments must be submitted via Zybooks
  - with the exception of Homework 4 which must be submitted via Bb Learn
- Homework assignments are to be **done individually**
- Homework assignments are due on Wednesdays by 11:59PM.
- No late assignments will be accepted
- Homework assignment grades will have 2 parts:
  - an auto-graded part (test cases / unit testing)
  - a manual grading part (compliance with requirements)

Midterm and Final Exams

- The midterm exam will be given during the 6th week of the term.
- The final exam will be given during finals week.
  - Date and Time TBD by the Registrar's Office
- These tests
  - Are individual tests.
  - Are Open-book/Open-notes.
  - Will be given online via Bb Learn and/or Zybooks
- You must take the tests on the date/time prescribed
  - You must be present in the classroom in order to take the exam
  - If you miss a test, you will get zero for it

Due Dates/Late Policy

Due Dates

- Weekly readings: Due on Sundays by 11:59 pm
- Homework assignments: Due on Wednesdays by 11:59 pm.
- Lab assignments: Due by 11:59 PM on the day of your lab session
- Midterm Exam: Week 6, during lecture.
- Final Exam: Finals week – Date/Time/Room TBA by the Registrar's Office

Late Policies:

- Zero if late/missed for everything!
  - However, if you have a valid reason to miss an assignment or test, you must contact the course instructor to see if an exception can be made. You must do this before you miss any assignment/test.
  - Your lowest reading, homework, and lab grades will be dropped from the grade calculation at the end of the term.

Additional Policies

- Any dispute about a grade must be resolved within 5 days of receiving your grade
- We are all bound by the Academic Honesty Policy
- Consequences for breaches of the policy include but are not limited to:
  - A grade of zero for the assignment/test in question
  - Reduction of the final letter grade for the course
  - Failing the course
  - An Academic Misconduct report filed with the University
- To ensure that assignments are done independently
  - In addition to human observation, we will be running all assignments through a plagiarism detection system.
  - This program uses compiler techniques which are invariant of syntax and style. It has a very high accuracy rate.

Additional Policies

- You must observe proper etiquette while online (Discord / Email).
  - Be respectful of your TAs, course instructor, and fellow students.
  - Stay on topic.
  - Don’t post irrelevant links, comments, thoughts, or pictures.
  - Respect the opinions of your classmates.
  - **Do not ever post your assignments' code online**

University Policies

- Students with Disability Statement:
  - http://www.drexel.edu/oed/disabilityResources/students/
- Course Drop Policy:
  - http://www.drexel.edu/provost/policies/course-add-drop/
- Course Withdrawal Policy:
  - http://www.drexel.edu/provost/policies/course-withdrawal/
- Absence from class due to University sponsored activities:
  - http://www.drexel.edu/provost/policies/absence.html

Diversity, Equity and Inclusion Statement

- CCI faculty believes and embraces diversity for it fosters innovative, transformative classrooms where optimal learning for students of all identities and backgrounds can occur.
- For more information on Diversity and Inclusion in CCI, please visit:
  - https://drexel.edu/cci/about/diversity-equity-and-inclusion-council/

Week By Week

- Week 1 - Course Overview. Review Object Usage. Introduction to OOD
- Week 2 – Creating Classes in Python
- Week 3 – Overloading Operators
- Week 4 – Inheritance, Polymorphism and Abstract Base Classes.
- Week 5 – Case Study: Using Objects in Games (PyGame)
- Week 6 – **Midterm Exam**
- Week 7 - Memory Considerations; Stacks and Queues
- Week 8 – Linked Lists
- Week 9 – Hash Tables
- Week 10 – Binary Search Trees; OOP Compared
- Week 11 – **Final Exam**

Note: See syllabus in Bb Learn for full schedule with assignments due dates

Course Resources

- There are lots of resources posted in Bb Learn:
  - Python Documentation
  - Links to download Python, Thonny, PyCharm, PyGame
  - Link to the online book (Making Games with Python and PyGame)
  - TAs office hours and website (CLC)
  - iCommons lab link
    - Borrow laptops, get help with software, etc.
  - Bb Learn help

Syllabus and Course Schedule

- Posted in Blackboard Learn
- Read it!
- You are accountable for everything in it.

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
  - Update 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.
  - 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).
  - This is called the dot-notation
- For example:

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

Static Attributes and Methods

- Sometimes we want an attribute or method to be independent of a 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(25.0)`
- 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

Modules

- Code that is stored in an external file, that can be used in another script, is referred to as a module
- 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`
  - 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
```

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 projects grow, 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()
```

```
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 using 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

- Let's 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 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 instance methods.
- Instance methods require at least one parameter. That parameter is an implicit parameter.
  - This implicit parameter 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
myCar.year = 1492
```

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

- 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
  - But we in CS 172 require you to use attribute mangling
- 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
```

NOTE: As a rule of thumb all 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
  - Methods in the public interface allow us to interact with the objects
- 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 group 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)
- Getter methods do not change the value of any attributes
  - Only return the current value of the attribute
- Setter methods change the value of one (or more) attribute(s)
  - Do not return anything

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 printPoint(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.printPoint()
'(0, 0)'
>>> print(p1.getX())
0
>>> print(p1.getY())
0
>>> p1.setPoint(10 ,10)
>>> p1.printPoint()
'(10, 10)’
>>> p1.reset()
>>> p1.printPoint()
'(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 the math module 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)
7.0710678118654755
```

Static Attributes

- In addition to instance attributes and methods, we can have class attributes and class methods.
  - They belong to the class itself and not an instance of it.
  - Also known as static attributes and static methods.
- 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 have been 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

- Likewise, some methods don't need instance attributes to do their job.
- We call these static, or class methods.
- For a method to be static
  - Make sure you don't attempt to access any instance variables.
  - Do not use the implicit object
- We can use decorators to mark a method a static
  - In Python, a decorator is a special label applied to something to mark it as something special.
  - Decorators start with the `@` symbol.
- To mark a method as static we can use the `@staticmethod` decorator right before the method definition.

Static Methods

We access the class method via the class name

```
class Point():
  __count = 0

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

>>> Point.printPointCount()
Points created: 0
```

Using our class

- We will save our class definition code in a separate file from our main script.
  - The file where we have our class code will be used as a module
- This means that in order to use our class in our script we will have to **`import`** it.
- Example:

```
from point import Point
pointA = Point(10, 5)
pointB = Point(3, 5)
```

The Main Script

- There is sometimes code that we only want to 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 something
```

CS 172 - Computer Programming II

Special methods for Operator Overloading

Objectives

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

Overview

- Imagine that we have a class, called `EL`, that is used to represent English Length objects (yards, feet, inches)
  - Example: 2 yards, 2 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 foot = 12 inches

The EL class

```
class EL:
    # initialization method - constructor
    def __init__(self, inches = 0, feet = 0, yards = 0):
        self.__inches = inches % 12
        self.__feet = feet + inches // 12
        self.__yards = yards + self.__feet // 3
        self.__feet = self.__feet % 3

    # getters
    def getInches(self):
        return self.__inches
    def getFeet(self):
        return self.__feet
    def getYards(self):
        return self.__yards
    def totalInches(self):
        allInches = self.__yards * 36 + self.__feet * 12 + self.__inches
        return allInches

    #setters
    def addInches(self, amountToAdd):
        totalInches = self.totalInches() + amountToAdd
        temp = EL(totalInches)
        self.__inches = temp.getInches()
        self.__feet = temp.getFeet()
        self.__yards = temp.getYards()
```

What else would be needed?

- 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`
- How about comparing two `EL` objects to see if they hold the value?
  - `len1 == len2`
- Or being able to display an EL object by just using the print built-in function:
  - `print(len1)`

Operator Overloading

- By default, Python doesn't know how to handle these operations
  - What does it mean to add together two objects?
- But we can provide "special methods" in our classes to define these behaviors.
- These special methods allow 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

Operator Overloading

- Rules
  - You cannot change the way operators work with existing types
    - For example, you cannot change the meaning of + with numbers.
    - + performs addition when used with numbers
  - You cannot define new operators
    - You can only overload the set of existing operators
  - You cannot alter the rules of precedence
    - Rules of precedence (such as * is performed before +) remain the same

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.getInches()
        newFeet = self.__feet + other.getFeet()
        newYards = self.__yards + other.getYards()
        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
False
```

  - Note: `l1`, `l2`, `l3` are three different objects. The == operator tests whether two objects hold the same values.

The __str__ method

- When we run the built-in function `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)          #this is the preferred way
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 Exception('Invalid index)

>>> 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
- 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

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

UML

- 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 associate getter and setter methods)
  - Dogs can play catch, so we will have a method for that.
- So now we need to initialize `species`, `age` and `breed`
- How do we do this?
- Let's override the **`__init__`** method from `Pet`

Inheritance: creating a Dog class

- The **super()** function allows us to call the parent's constructor within the child's constructor:

```
class Dog(Pet):
    # constructor
    def __init__(self, breed, age = 0):
        # super() calls on Pet's constructor to initialize species and age
        super().__init__('dog', age)

        # the next line initializes the new attribute
        self.__breed = breed   #additional attribute
```

Example: the Dog class

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

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

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

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
```

Overriding methods

- Let's take a closer look at our Pet – Dog example.
- Here are two methods from the `Pet` class:

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

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

- 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

- Inheritance allows us to make use of another class's attributes and methods
  - And we can add new attributes and methods
- But what if we want to **change** a method of the base class?
- Fortunately, we can easily do so by just having our new (child) class define its own version of the method.
- This is called **overriding** a method.

Overriding methods

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

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

Overriding Methods

- For single inheritance we can think of the `super()` function as returning the parent's instance.
  - It's somewhat analogous to the `self` keyword.
- We can access a parent class' attributes and methods via what is returned by `super()`.
- A `super()` call can be made inside any method

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

- Polymorphism is the ability to determine the appropriate version of a method to call on an object
  - 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.

Composition

- Composition is the idea that one object may be made up of other objects.
  - Has-a relationship

Abstract Base Classes

- What happens if we call a method on an 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 have a way to mark which methods are 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 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
- You must take the Reading Quiz:
  - Posted in Bb Learn
  - Due on Sunday

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

- Download Pygame from: https://www.pygame.org/download.shtml
- Open Thonny
- Select Tools —> Manage Packages

- In the window that opens use the search bar to look for pygame

- Click Install

Using Thonny with Pygame

- After you have installed PyGames open Thonny
- From the top menu choose **Tools** and then **Options**

Using Thonny with Pygame

- Go to the Interpreter Tab and from the drop-down menu choose **Alternative Python 3 interpreter or virtual environment**, then click **OK**

Using Thonny with Pygame

- Now you are ready to write and run programs that use PyGame.

Thonny - /Users/aalban/Documents/CS172/Winter 2021/Lectures/Week 5 - Graphics PyGame/DrawableObjects.py @ 130 : 1

Run current script (F5)

DrawableObjects.py

```
#File:      DrawableObjects.py
#Purpose:   Demo of a basic program using pygame
#           Case study for inheritance and Abstract Base classes
#Author:    A. Medlock
#Date:      February 3, 2021

import pygame
from abc import ABC, abstractmethod

# abstract class to represent a drawable obejct
class Drawable(ABC):
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y
```

Shell

```
Python 3.7.2 (/usr/local/bin/python3.7)
>>>
```

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 window in a 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 2-D 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 represented 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>, parameters..)
```

- 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 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

```
from abc import ABC, abstractmethod
class Drawable(ABC):
        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]

        @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)):
      super().__init__(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 enough.
- 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 = 304
catY = 204
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

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

Text

```
class Text(Drawable):
    def __init__(self, message = "Pygame", x = 0, y = 0, color = (0,0,0)):
        super().__init__(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

- If we want our application to do collision detection, we'll need each of our objects to return a rectangle that fits tightly around it.
- This 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 result in larger than expected surface rectangles.

AABB Collision Detection

```
class Drawable(ABC):
    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])
```

CS 172 - Computer Programming II

Memory Considerations

Objectives

- Memory Considerations
- Review of Core CS171 data structures

Data Structures

- A way of organizing, storing, and performing operations on data.
- Operations performed on a data structure include
  - accessing or updating stored data
  - searching for specific data
  - inserting new data
  - removing data.

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 a tuple?

```
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

- In most programming languages, all the things stored in a list must have the same type (homogenous lists).
- That way the computer knows how much contiguous memory to allocate.
- However, Python allows for mixed types in its lists (heterogeneous lists)
- So how does the computer know how much memory to allocate?

```
x = [‘h’, 6, 2.55, ‘I’, 56]
```

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 it is 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, 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

Stacks

Objectives

- Queues
- Stacks
- Priority Queues

More Data Structures

- Tuples and lists are great "general use" data structures.
- 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, the first person in line, is the first person served.
- And once they are 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

- 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 calls
  - 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
  - Python opted to call them put and get

Stacks

- Python provides for us the **`LifoQueue`** class to simulate a stack
- To use this, 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
  - ER at a hospital

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 = tasks.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?
    - A list
  - What should our `__init__` method be able to do?
    - Create an empty list
  - What should our public interface be?
    - `put(),get(), empty(), clear()`
    - `__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 data
- 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 < 0 or index > len(self) - 1:
            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 and current != None :
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if current == None: # not found or empty
            return False
        elif previous == None:  # item was in the first node
            self.__head = current.getNext()
            return True
        else:  # item was somewhere after the first node
            previous.setNext(current.getNext())
            return True
```

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 `log(n)` 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, -1]
```

BST - Search

- Now imagine that we want to search for the value 2
- Again let's start off by doing this manually

BST – Search Example 1

Search for 2

2 < 3 → search on the left

2 > 0 → search on the right

2 == 2 → Found!

BST – Search Example 2

Search for 5

5 > 3 → search on the right

5 > 4 → search on the right

5 == 5 → 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.
- We'll do this as class exercise / demo

Basic Search Algorithm

- If BST is empty, return False
- If BST is not empty: Traverse and see if value is there
- Keep going as long as we haven't reached the last node
  - If search value == current node
    - Return True and we are done
  - If search value < current node
    - Search on the left side
  - If search value > current node
    - Search on the right side
- If we reach the last node and did not find search value: return False

Basic Insert Algorithm

- Create new node with the new value
- If BST is empty, the new node is the root and we are done
- If BST is not empty, find the proper place for the new node
- While we have not found the proper place yet
  - If new value < current node
    - new node is going to be on the left of current node
      - If we have a leaf: insert new node on the left of current and we are done
      - Otherwise: advance to the left
  - If new value > current node
    - new node is going to be on the right of current node
      - If we have a leaf: insert new node on the right of current and we are done
      - Otherwise: advance to the right

CS 172 - Computer Programming II

OOP Compared

Objectives

- Compare Basic Python OOP to C++ and Java
- Compare Overloading Operators in Python to C++ and Java
- Compare Inheritance in Python to C++ and Java
- Compare Abstract Base Classes in Python to C++ and Java

Basic Class

- Create and instantiate a basic class
  - Constructor
  - Accessor/Mutators (getters/setters)
  - Encapsulation
  - Instantiation

Basic Classes: Python

```
# creating MyClass
class MyClass():
    def __init__(self, x = 0):
        self.__x = x

    def getX(self):
        return self.__x

    def setX(self, x):
        self.__x = x

# instantiating an object of MyClass
a = MyClass()
```

Basic Classes: C++

```
class MyClass
{
    private:
        int x;

    public:
        MyClass(const int& x = 0)
        {
            this -> x = x;
        }

        int getX()const
        {
            return x;
        }
        void setX(int x)
        {
            this -> x = x;
        }
};
```

Encapsulation

Constructor

Self

```
int main()
{
    MyClass a = MyClass();
    return 0;
}
```

Basic Classes: Java

Encapsulation

```
public class MyClass
{
    private int x;

    public MyClass(int x)
    {
        this.x = x;
    }

    public int getX()
    {
        return x;
    }

    public void setX(int x)
    {
        this.x = x;
    }
}
```

Constructor

Self

```
public class testMyClass
{
  public static void main(String[] args)
   {
      MyClass a = new MyClass(5);
   }
}
```

Overloading Operators: Python

```
class MyClass():
    def __init__(self, x = 0):
        self.__x = x

    def getX(self):
        return self.__x

    def setX(self, x):
        self.__x = x

    def __plus__(self, other):
        return MyClass(self.getX() + other.getX())

a = MyClass(3)
b = MyClass(4)
c = a + b
```

Overloading Operators: C++

```
class MyClass
{
    private:
      int x;

    public:
      MyClass(const int& x = 0)
      {
         this -> x = x;
      }

      int getX()const
      {
         return x;
      }

      void setX(int x)
      {
         this -> x = x;
      }

      MyClass operator+(const MyClass& other)const
      {
         return MyClass(this->getX() + other.getX());
      }
};
```

```
int main()
{
    MyClass a = MyClass(3);
    MyClass b = MyClass(4);
    MyClass c = a + b;
    return 0;
}
```

Overloading Operators: Java

```
// No support  -- Must do through methods
public MyClass add(MyClass other)
{
   return MyClass(this.x + other.getX());
}
public class testMyClass
{
  public static void main(String[] args)
   {
      MyClass a = new MyClass(3);
      MyClass b = new MyClass(4);
      MyClass c = a.add(b)
   }
}
```

Inheritance: Python

```
class B(MyClass):
    def __init__(self, x = 0, y = 0)
        super().__init__(x);
        self.__y = y

    def getY(self):
        return self.__y

    def setY(self, y):
       self.__y = y

a = MyClass(3)
b = B(4, 2)
```

Inheritance: C++

```
class B: public MyClass
{
   private:
     int y;

   public:
     B(const int& x = 0, const int& y = 0) : MyClass(x)
     {
        this-> y = y;
     }
     int getY()const
     {
        return y;
     }
     int setY(y)
     {
        this -> y = y;
     }
     int getDoubleX()const
     {
            return 2 * MyClass::getX();
     }
};
```

```
int main()
{
    MyClass a = MyClass(3);
    B b  = B(4, 2);
    return 0;
}
```

Inheritance: Java

```
public class B extends MyClass
{
    private int y;

    public B(int x, int y)
    {
        super(x);
        this.y = y;
    }
    public int getY()
    {
          return y;
    }
    public void setY(int y)
    {
          this.y = y;
    }
    public int getDoubleX()
    {
        return 2 * super.getX();
    }
}
```

```
public class testMyClass
{
  public static void main(String[] args)
   {
      MyClass a = new MyClass(5);
      B b = new B(4, 2)
   }
}
```

Polymorphism/ABC: Python

```
#ABC
from abc import ABC, abstractmethod

class MyClass(ABC):
   #prior stuff . . . .

   @abc.abstractmethod
   def printObj(self):
      pass

#polymorphism
a = MyClass(3)
b = B(4, 2)
c = C(5, 6, 7)
x = [a, b, c]

x[1].printObj()
x[2].printObj()
```

Polymorphism/ABC: C++

```
//ABC
class MyClass
{
    //prior stuff . . . .
    virtual void printObj() = 0;

};

//Polymorphism
MyClass* x[3];
x[0] = new MyClass(3);
x[1] = new B(4, 2);
x[2] = new C(5, 6, 7);

x[1].printObj();
x[2].printObj();
```

This creates an abstract method that must be implemented in the derived classes

By declaring a method as *virtual* in the base class, we can allow the computer to determine the method to call in *run time*

Polymorphism/ABC: Java

```
//ABC
public abstract class MyClass
{
    //prior stuff…
    void abstract printObj();

}

// polymorphism
MyClass[] x = new MyClass[3];
x[0] = new MyClass(3)
x[1] = new B(4, 2)
X[2] = new C(5, 6, 7)

x[1].printObj();
x[2].printObj();
```

This designate this class as an abstract class

This creates an abstract method that must be implemented in the derived classes

CS 172 - Computer Programming II

Hash Tables

Objectives

- Understand what hashing is.
- Understand what hash tables are
- Understand how dictionaries work in Python.

Linear Data Structures

- All the previous data structures we studied (tuples, lists, linked-lists, stacks, queues), are linear data structures.
- That is, they can be though of as having two ends.
  - The “left” and the “right”
  - The “front” and the “rear”
  - The “top” and the “bottom”
- What distinguishes one linear structure from another is the way in which items are added and removed
- There are other times when having a linear data structure is not advantageous.
- One such example you should be familiar with are dictionaries.

Dictionaries - Review

- A dictionary is an unordered collection of items.
- A dictionary has key: value pairs.
- Dictionaries are optimized to retrieve values when the key is known
- Example:

```
>>> inventory = {'apples':12, 'pears':5, 'kiwi':8, 'limes':35}
>>> print(inventory['apples’])
12
```

Dictionary Implementation

- Dictionaries allow us to access data using things other than indices.
  - Which can make things quite user-friendly.
- When accessing something in a dictionary (via it's key), there is no point in going from the beginning to the end.
- So how are these implemented?
  - They're implemented using a common data structure called a hash table.

Hash Table

- A hash table is a table (not unlike a tuple or list) that our keys hash into.
- What do we mean by *hash into*?
- The key must be an object that has a method that can turn it into an actual non-negative integer index.
  - Which we can then use to index into the hash table.
  - The method of a class that does the hashing is referred to as its hash function
  - This is a special method **`__hash__`** that returns a non-negative integer.
  - Given an object `X`, whose class implements the **`__hash__`** function, calling **`hash(X)`** implicitly calls **`__hash__`** on `X` (i.e. `X.__hash__()`).
- For example, strings map to integers based on the byte values of the characters in the string, while tuples combine hashes of the items inside the tuple.

Hash Table Example

Hashing Example

- As an example, imagine we have a class that has an attribute `__value` that stores a single character.
- We can get a decimal code from a character using the **`ord()`** function.
- Printable characters range from [32, 126]
- So let's have our hash method get the integer value of a character and shift it back by 32 so that its range is [0, 94]

```
class MyObject():
        def __init__(self, val):
                self.__value = val
        def getValue(self):
                return self.__value
        def __hash__(self):
                return ord(self.__value) - 32
```

Hash Functions

- Many hash function use the modulo operator to map an unbounded integer to one in a finite range.
  - Recall that modulo is essentially remainder.
  - So `24 % 5 = ?`
- Ideally for all possible hash values, there will be only one key that can hash to it.
- In practice this isn't possible or useful.
- But a good hash function will result in each possible hash value having equal numbers of possible keys that hash to it.
- Zybooks shows you a few examples of possible good hash functions.

Hashing

- It's possible that two keys hash to the same index!
  - We call this a collision
- How do we handle this?
- There are two approaches:
  - Open Hashing – Unlimited number of keys can map to the same hash location (Zybooks calls this chaining).
    - At the hash location we have a *list* of (key, value) tuples.
  - Closed Hashing – Only one key can map to a hash location (Zybooks calls this direct hashing).
    - We must define a function or strategy to say where to check next if there's already a key in a location we're trying to hash to.
    - Of course it's possible that the hash table is full, and therefore we can't put the (key, value) pair in it.
- We'll just focus on open hashing.

Open Hash Table

- Open hash tables are often implemented as a list of linked lists.
- At each location (index) within the list we have a linked list.
- Each node in the linked list stores a `(key, value)` tuple

Open Hash Table

- To find something:
  1. Specify a hashable key
  2. Compute the index using the key's hash function
  3. Search in the list located at that index for the key
  4. From the tuple that contains the found key, return the value.

Dictionary Implementation

- So now that we have learned about hashing, hash tables, and linked lists, back to dictionaries.
- When you add or get an entry, the dictionaries first hashes the key to a non-negative integer index.
- Then it searches the linked list for that key
  - And depending on the operation, inserts, removes, or returns it.

Dictionary Implementation