CS 172 – Homework 4

Purpose

After completing this assignment, you will have practiced using the Pygame graphics API to create a fun, interactive 2D game! Graphics are a great application of object-oriented programming.

Description

For this assignment you follow a tutorial to simulate a ball which bounces around the screen, then add elements to the simulation to turn the animation into a simple game. You will add original elements to the game and provide basic user-facing documentation.

Specifications

Below are the base specifications for the minimum classes required to complete this activity. Following the base specifications are additional instructions to help you implement the basic elements of a game. Once you complete the base requirements, you will then implement extensions to the project as described below.

The Drawable Class

In lecture we discussed how you can use object-oriented programming concepts to make graphics applications more organized and easier to create. Several of the things that we'll leverage are inheritance, polymorphism, and abstract base classes.

Your first task will be to create your own version of the Drawable abstract base class (similar to the one shown in lecture). This class should have the following attributes, at a minimum:

- Position: x and y location for the center of the object
- Visible: a Boolean (True/False) value such that if True, the object is drawn. Although you may not choose to use this attribute, it could be useful

and the following abstract methods:

- draw: Takes as a parameter a surface to draw on
- get_rect: Returns a Pygame Rect object that is the bounding rectangle that fits tightly around the object

Feel free to implement constructors, inspectors and mutator methods, along with additional attributes and methods as you see fit.

Put all this code in a file named drawable.py

The Ball Class

The Ball class inherits from Drawable and will draw a circle at its current location. You must implement at the very least, the required methods of the base class (draw and get_rect), as well as a constructor and a method to move the ball. You may need to implement other methods as part of the public interface.

Put all this code in a file named ball.py

The Paddle Class

The Paddle class inherits from Drawable and it will be used to allow player interaction by catching the ball. The paddle's position will be updated based on the horizontal position of the user's mouse. You will check for and respond to instances when the ball and the paddle touch.

Put all this code in a file named paddle.py

The Text Class

The Text class inherits from Drawable and it will be used to display information to the player such as instructions or a score. You must implement at the very least, the required methods of the base class (draw and get_rect), as well as a constructor. You may need to implement other methods as part of the public interface.

Put all this code in a file named text.py

Additional Classes

Your project may require the creation of additional classes. Put each class in its own file and use the appropriate import statements to include them in your program. Common additions are goal objects which you are trying to reach, enemy objects which are trying to avoid, or neutral objects such as bricks.

Instructions

Your completed project must include a readme.txt file which tells the user anything they need to know in order to play your game. See the details below in the Grading section.

Step-By-Step "Getting Started" Guide

The tutorial below will provide you with the starting point for an original game. Submitting only the code provided in this tutorial will not result in a passing grade for this assignment. Be sure you pay careful attention to the concepts presented. Simply copying and pasting the code into your IDE will result in a program that runs, but you should read and understand the accompanying information so you will have the knowledge and skills required to complete the rest of this assignment.

Main and Drawable

Begin by creating the main program in hw4.py. Write any basic code that runs, such as a simple print statement. One goal is to always have code that runs so you can check your work as you go in small increments.

hw4.py

```
print("Hello, Homework 4!")
```

Add a drawable.py file and implement the start of the Abstract Base Class in it by importing ABC and sketching out the required constructor and the two prescribed methods. In the example below, we provided default values for the x and y parameters in the constructor.

drawable.py

```
from abc import ABC

class Drawable(ABC):

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

  def draw(self, surface):
    pass

  def get_rect(self):
    pass
```

Set Up The Window

In hw4.py, import pygame and create a rectangular window with a white background. Add the code necessary to allow the user to exit the program by closing the window. This will involve doing some initialization of the pygame system followed by a loop which continually checks for events. You can also remove your print statement if you'd like.

hw4.py

```
import pygame

pygame.init()
surface = pygame.display.set_mode((800, 600))
running  = True
while running:
  surface.fill((255, 255, 255))
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
       running = False
  pygame.display.update()
exit()
~~print("Hello, Homework 4!")~~
```

Check Your Work

If you are on the right track, running hw4.py should display a rectangular window with a white background which disappears when you close it using the operating system's close button (typically at the top of the window).

Add Some Drawables

While the ultimate goal is to simulate a ball which bounces around the screen, let's start by getting a circle to appear on the screen. Since we cannot create instances of Drawable directly because it's abstract, create a Ball class in a file called ball.py which inherits from the Drawable abstract base class.

Define some reasonable parameters for the constructor, such as the Ball's position, size, and color. Pass the position information along to the Drawable constructor and store the remaining Ball-specific details into class variables.

Implement the draw() method in Ball so we can draw ball instances on a provided surface. We will need to import the Drawable class as well as the pygame library. To use the Pygame Circle primitive to draw our ball, we need the following information:

- **surface** – the surface to draw on
- **color** (a *tuple of three integers representing (R, G, B) values*) – the color to fill the circle with
- **center** (a *tuple of two numbers representing (x, y) values)* – the center point of the circle
- **radius** (*int or float*) – the radius of the circle, measured from the center position

Most of this information is stored within the Ball instance, but the position information is stored in Drawable, the Ball's parent class. Since the Drawable class uses attribute mangling, code outside the Drawable class cannot directly access the __x and __y values.

In drawable.py, add a method called getLoc() which returns a tuple of the Drawable's current position, which we can then pass off to the Pygame's circle() constructor, which is invoked inside the Ball's draw() method.

drawable.py

```
...

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

...
```

Finally, complete draw() in ball.py so it uses the information available to it to draw a circle on the surface specified.

ball.py

```
from drawable import Drawable
import pygame

class Ball(Drawable):

  def __init__(self, x=0, y=0, radius=10, color=(0, 0, 0)):
    super().__init__(x, y)
    self.__color = color
    self.__radius = radius

  def draw(self, surface):
    pygame.draw.circle(surface, self.__color, self.getLoc(), self.__radius)
```

In hw4.py, import the Ball class, create a new Ball instance, and try to draw the Ball on your Pygame surface.

hw4.py

```
import pygame
from ball import Ball

pygame.init()
surface = pygame.display.set_mode((800, 600))
DREXEL_BLUE = (7, 41, 77)
myBall = Ball(400, 300, 25, DREXEL_BLUE)
running  = True
while running:
  surface.fill((255, 255, 255))
  myBall.draw(surface)
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
  pygame.display.update()
exit()
```

Check Your Work

If you are on the right track, running hw4.py should now display a rectangular window with a white background with a circle in it. Before proceeding, resolve any bugs you may be experiencing. If text is **in bold**, it was added to a file. If text is ~~struck through~~, it was deleted from a file. If you see three dots (…), that means that part of a file was not shown in the code snippet.

Scoot the Ball

To make the ball appear to move, we will need to update the Ball's position and then redraw the screen. To do this, add a method to the Ball class called move() which will be responsible for updating the Ball's position.

ball.py

```
...

  def move(self):
    # Increase __x and __y by some amount

...
```

As we know, the __x and __y values are not stored in the Ball object, but in its parent class, Drawable. Again, revisit Drawable and add code that will allow you to change the Drawable's position. You could write a setLoc() method which lets you specify a whole new position, you could write setX() and setY() methods that let you change the coordinate values independently, or you could write increaseX() and increaseY() methods which increment the given values by the amount specified. These are not mutually exclusive, so you could write them all. For now, let's write setX() and setY() in Drawable.

drawable.py

```
...

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

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

...
```

Back in Ball, we now have the tools required to update the Ball's position. For now, every time someone calls move(), add 1 pixel to both the x and y coordinates.

ball.py

```
...

  def move(self):
    # Increase __x and __y by some amount
    currentX, currentY = self.getLoc()
    newX = currentX + 1
    newY = currentY + 1
    self.setX(newX)
    self.setY(newY)
...
```

Finally, the game loop in hw4.py needs to be revised to call move() right after the ball is drawn so the ball's position will be different on the next iteration of the loop.

hw4.py

```
...
while running:
  surface.fill((255, 255, 255))
  myBall.draw(surface)
  myBall.move()
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
  pygame.display.update()
...
```

Check Your Work

If you are on the right track, running hw4.py should now display a rectangular window with a white background with a circle in it that zips down and to the right, disappearing after less than a second.

Set the Frame Rate

One way to slow down the animation is to tell Pygame our desired number of frames per second. The more frames per second, the faster things animate. The fewer the number of frames per second, the slower things go.

In the main program, create a variable to reference Pygame's clock outside the while loop. Inside the while loop, near the bottom, add an instruction that will pause animation long enough to achieve the desired number of frames per second.

hw4.py

```
...
fpsClock = pygame.time.Clock()
while running:
  surface.fill((255, 255, 255))
  myBall.draw(surface)
  myBall.move()
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
  pygame.display.update()
  fpsClock.tick(30) # Approx. 30 Frames Per Second
...
```

Now, the Ball glides gracefully off the bottom of the screen. This isn't the end goal, but it's a great start.

Bouncing Off The Walls

Rather than moving down and to the right forever, we need to recognize when the Ball object has touched the boundary of the display surface. In order to do that, we need to ask the display surface what its dimensions are. In order to do that, we need to have access to the display surface.

The draw() method is provided with the display surface as a parameter. We could revise move() to also require that the display surface be provided as input. Another way is to ask Pygame for access to the current display surface.

Calling pygame.display.get_surface() returns the current surface; Calling get_size() on that surface instance will return a tuple containing its width and height. To confirm this, add some code to Ball's move() method to get the current surface and print its width and height to the console.

ball.py

```
...

  def move(self):
    # Increase __x and __y by some amount
    currentX, currentY = self.getLoc()
    newX = currentX + 1
    newY = currentY + 1
    self.setX(newX)
    self.setY(newY)

    surface = pygame.display.get_surface()
    width, height = surface.get_size()

    print (width, height)
...
```

If the console repeatedly prints integers which match the size of your display surface, you are on the right track! Now, to determine whether the ball's updated position is touching the boundary.

Since there are four edges we could run into, we need to check four possible conditions – we have hit the top, bottom, left, or right edge of the surface. If we have hit the top or bottom edge, we should reverse our movement in the y direction. If we have hit the left or right edge, we should reverse our movement in the x direction.

Since we are currently moving "1" pixel in both x and y directions, let's pull out the hard-coded values and replace them with variables which can be reversed by negating the values.

ball.py

```
...
  def __init__(self, x=0, y=0, radius=10, color=(0, 0, 0)):
    super().__init__(x, y)
    self.__color = color
    self.__radius = radius
    self.__xSpeed = 1
    self.__ySpeed = 1
...
  def move(self):
    # Increase __x and __y by some amount
    currentX, currentY = self.getLoc()
~~    newX = currentX + 1~~
~~    newY = currentY + 1~~
    newX = currentX + self.__xSpeed
    newY = currentY + self.__ySpeed
    self.setX(newX)
    self.setY(newY)

    surface = pygame.display.get_surface()
    width, height = surface.get_size()

~~    print (width, height)~~
...
```

Finally, add code to see if the new position of the ball is less than 0 or greater than the screen dimension in both the x and y directions. If so, negate the corresponding speed value.

ball.py

```
...
  def move(self):
    # Increase __x and __y by some amount
    currentX, currentY = self.getLoc()
    newX = currentX + self.__xSpeed
    newY = currentY + self.__ySpeed
    self.setX(newX)
    self.setY(newY)

    surface = pygame.display.get_surface()
    width, height = surface.get_size()

    if newX <= 0 or newX >= width:
      self.__xSpeed *= -1

    if newY <= 0 or newY >= height:
      self.__ySpeed *= -1
...
```

Check Your Work

If you are on the right track, running hw4.py should now result in a circle that reverses direction when it exceeds the edge of the screen. Unfortunately, the circle only turns around when half of it is already off the edge of the screen. What we actually want is for the circle to turn around when its outside edge touches the display boundary. To achieve that, we need to offset our boundary checks by an amount equal to the Ball's radius.

Revise Ball's move() method once more.

ball.py

```
...
~~    if newX <= 0 or newX >= width:~~
~~      self.__xSpeed *= -1~~

    if newX <= self.__radius or newX + self.__radius >= width:
      self.__xSpeed *= -1

~~    if newY <= 0 or newY >= height:~~
~~      self.__ySpeed *= -1~~

    if newY <= self.__radius or newY + self.__radius >= height:
      self.__ySpeed *= -1
...
```

Finishing Up Ball

Since the Ball class is a Drawable object, it must implement all of the abstract methods in Drawable. In ball.py, implement get_rect() so it returns a pygame.Rect object that just covers the Ball's area. In hw4.py, test the get_rect() function by drawing the Ball's rectangle to confirm that the rectangle just covers the circle that represents the Ball. When you have confirmed that get_rect() works properly, you can remove the line from hw4.py that was added for testing.

ball.py

```
...
  def get_rect(self):
    location = self.getLoc()
    radius = self.__radius
    return pygame.Rect(location[0] - radius, location[1] - radius, \
      2 * radius, 2 * radius)
...
```

hw4.py

```
...
  surface.fill((255, 255, 255))
  pygame.draw.rect(surface, (123, 123, 123), myBall.get_rect()) # Testing
  myBall.draw(surface)
  myBall.move()
...
```

Currently, our Ball's draw() method does not honor the Drawable's __visible flag. Rectify this by adding some methods to support visibility at the Drawable level – isVisible() and setVisible(), for instance.

drawable.py

```
...
  def isVisible(self):
    return self.__visible

  def setVisible(self, visible):
    if visible == True:
      self.__visible = True
    else:
      self.__visible = False
...
```

In ball.py, modify the draw() method so it only draws the circle if the object is marked as visible. Test this by updating the game loop to check for mouse button clicks. If the user clicks the mouse, toggle the Ball's visibility. You can remove these testing instructions after you confirm that the visibility methods work as intended.

ball.py

```
...
  def draw(self, surface):
    if self.isVisible():
      pygame.draw.circle(surface, self.__color, \
                         self.getLoc(), self.__radius)
...
```

hw4.py

```
...
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
    elif event.type == pygame.MOUSEBUTTONDOWN:        # Testing
      myBall.setVisible(not myBall.isVisible())       # Testing
...
```

Add any additional getters, setters, and helpful methods that you might need to use your Ball object in a game. These may be added at the Ball or the Drawable level, as needed. Test each one that you add.

ball.py

```
import math
...
  def getColor(self):
    ...

  def setColor(self, color):
    ...

  def getRadius(self):
    ...

  def setRadius(self, radius):
    ...

  def isTouchingBall(self, other):
    ...

  def setXSpeed(self, speed):
    ...

  def getXSpeed(self):
    ...

  def setYSpeed(self, speed):
    ...

  def getYSpeed(self):
    ...
...
```

drawable.py

```
...
  def setLoc(self, newLoc):
    ...
...
```

Add A Paddle

To change this from a simple simulation into a game, let's add a paddle which can be used to catch the ball. A paddle can be drawn using a Pygame rectangle primitive, and its position can be synchronized with the position of the mouse.

In a file called paddle.py, create a Paddle class which inherits from Drawable with a constructor that takes the paddle's width, height, and color. For now, we will start our paddle to be centered horizontally on the display a distance of 20 pixels from the bottom. Whenever the paddle is drawn, use pygame.mouse.get_pos() to access the mouse cursor's current position and draw the paddle so it is centered horizontally at the location of the user's mouse cursor. To fully implement the Drawable abstract base class, you should also complete get_rect(), which returns a rectangle which covers the full shape. Since the paddle is a rectangle, you can also use this method to help draw the paddle.

paddle.py

```
from drawable import Drawable
import pygame

class Paddle(Drawable):

  def __init__(self, width, height, color):
    surface = pygame.display.get_surface()
    screenWidth, screenHeight = surface.get_size()
    super().__init__(screenWidth/2, screenHeight/2)
    self.__color = color
    self.__width = width
    self.__height = height

  def draw(self, surface):
    pygame.draw.rect(surface, self.__color, self.get_rect())

  def get_rect(self):
    surface = pygame.display.get_surface()
    screenWidth, screenHeight = surface.get_size()
    mouseX = pygame.mouse.get_pos()[0]
    return pygame.Rect(mouseX - self.__width/2,  \
            screenHeight - 20 - (self.__height),   \
            self.__width,             \
            self.__height)
```

In hw4.py, create an instance of the Paddle class and confirm that it works as expected. At this point, you have not programmed any interaction with the Ball class, so the Ball should pass right over/under the Paddle.

hw4.py

```
...
from paddle import Paddle
...
myBall = Ball(400, 300, 25, DREXEL_BLUE)
myPaddle = Paddle(200, 25, DREXEL_BLUE)
...
while running:
  surface.fill((255, 255, 255))
  myBall.draw(surface)
  myPaddle.draw(surface)
  myBall.move()
...
```

We now need to detect when one Drawable is near enough to another Drawable that they could be touching. All Drawable objects implement get_rect(), but we can add a method to the Drawable class that lets us check if one Drawable intersects some other Drawable.

drawable.py

```
...
  def intersects(self, other):
    rect1 = self.get_rect()
    rect2 = other.get_rect()
    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
...
```

Then in hw4.py, print a line to the console any time the ball and the paddle are close enough to touch.

hw4.py

```
...
  myBall.draw(surface)
  myPaddle.draw(surface)
  if myBall.intersects(myPaddle):
    print("Touching!")
  myBall.move()
...
```

Now that we can detect paddle/ball collisions, let's negate the ball's ySpeed value any time the ball comes into contact with the paddle.

hw4.py

```
...
  myBall.draw(surface)
  myPaddle.draw(surface)
  if myBall.intersects(myPaddle):
    ~~print("Touching!")~~
    myBall.setYSpeed(myBall.getYSpeed()*-1)
  myBall.move()
...
```

Check Your Work

You should now have a ball that moves around the screen, bouncing off the edges and now bouncing off the paddle when the two intersect. If you haven't done so already, you may need to implement getters and setters for the Ball's ySpeed value in order for the provided code to operate correctly.

Add A Counter

Let's begin my simply counting the number of times that the ball and the paddle come into contact with one another. Add a variable to keep track of this, then update it any time we reverse the ySpeed of the ball due to an intersection.

hw4.py

```
...
numHits = 0
...
  if myBall.intersects(myPaddle):
    myBall.setYSpeed(myBall.getYSpeed()*-1)
    numHits += 1
  myBall.move()
...
```

In order to see this number within our game window, we need to add another class which is descended from Drawable called Text.

text.py

```
from drawable import Drawable
import pygame

class Text(Drawable):

  def __init__(self, message="Pygame", x=0, y=0, \
               color=(0,0,0), size=24):
    super().__init__(x, y)
    self.__message = message
    self.__color = color
    self.__fontObj = pygame.font.Font("freesansbold.ttf", size)

  def draw(self, surface):
    self.__surface = self.__fontObj.render(self.__message, \
                                            True, self.__color)
    surface.blit(self.__surface, self.getLoc())

  def get_rect(self):
    return self.__surface.get_rect()

  def setMessage(self, message):
    self.__message = message
```

In hw4.py, add an instance of the Text class to show the score.

hw4.py

```
...
from text import Text
...
myPaddle = Paddle(200, 25, DREXEL_BLUE)
myScoreBoard = Text("Score: 0", 10, 10)
running  = True
fpsClock = pygame.time.Clock()
while running:
  surface.fill((255, 255, 255))
  myBall.draw(surface)
  myPaddle.draw(surface)
  myScoreBoard.draw(surface)
  if myBall.intersects(myPaddle):
    myBall.setYSpeed(myBall.getYSpeed()*-1)
    numHits += 1
    myScoreBoard.setMessage("Score: " + str(numHits))
  myBall.move()
...
```

Check Your Work

You should now have a score shown at the top left of your game window that increases by one point each time the paddle and the ball intersect. The logic is not perfect – if the paddle is moved into the middle of the ball, the score will increase as long as the ball and the paddle are touching, but it's a start.

Make It A Game

Now it's your turn to take this very basic platform and turn it into a game. At a minimum, your game must allow the player to

- Earn Points – Your game must allow the player to earn points somehow. Most simply, this is done by navigating the ball so it intersects with something of value

- Lose Points – Your game must allow the player to leose points somehow. Most simply, this is done by navigating the ball so it intersects with something of negative value

- Win – Your game must allow the player to win. This could be by accumulating a certain number of points, collecting all of a particular set of things, or surviving for a minimum amount of time.

- Lose – Your game must allow the player to lose. This could be by losing too many points, allowing the ball to hit the bottom of the screen, or anything else that makes sense.

- See Changes In Game State – Your game must allow the player to see that their actions are having some effect. Most simply, this is a score, life counter, health meter, etc. You could change the color the screen, change the color of Drawable objects, change the size of the paddle, or do lots of other creative things to keep the player in the loop.

Grading

Once you have followed the Step-By-Step guide, extended the starter code into a game unlike anyone else's, and provided clear documentation in a readme.txt file, check with the guidelines below to ensure that your submission will earn full credit.

Code Quality

Each of your files must have a header comment listing your full name, Drexel user id, and the purpose of the file – at the very least. Repetitive code (code that appears in multiple places in the main script) should be written as a function. Your program must use good style, including proper identifier names, useful comments, and proper use of indentation and whitespace. Could someone in the future easily open your project and add extra functionality to your project?

Required Elements

Your game should include, at a minimum, each of the required elements:

- Drawable abstract class
- Three Drawable object types which inherit from the Drawable abstract class
    - For example, Ball, Paddle, and Text
- Main Script
- Scoring
    - A way to earn points
    - A way to lose points
- Playability
    - A way to win
    - A way to lose
    - Game state visible to player
- Documentation
    - A readme.txt which tells the user how the game works

Rubric

Aspect	Low	Average	High
Drawable Abstract Base Class (5 points)	Not present, does not inherit from ABC, etc.	The Drawable class inherits from ABC but includes some methods that do not make sense for an abstract class	The Drawable class inherits from ABC and includes draw(), get_rect(), and other reasonable methods
Three or more derived classes (30 points)	Not present, does not inherit from Drawable, etc.	The project includes at least two classes which inherit from the Drawable class and which implement draw(), get_rect(), and other reasonable methods; Most methods in the derived classes make sense in the context of the project they were developed for	The project includes at least three classes which inherit from the Drawable class and which implement draw(), get_rect(), and other reasonable methods
Main Script (10 points)	Not present, does not properly initialize a Pygame instance, etc.	The project includes a main script which initializes the Pygame instance The main script would benefit from reorganization, such as moving some aspects into functions	The project includes a main script which initializes the Pygame instance and coordinates interaction among the game elements. The main script is well-organized and easy to read
Documentation (15 Points)	Documentation is not present in a file called readme.txt or does not describe at least three of the required elements.	A readme.txt file is present which describes at least three of the following: 1) the player's objective, 2) how to earn points, 3) how to lose points, 4) how to win the game, and 4) how to lose the game	A readme.txt file is present which describes how to play the game, including the player's objective, how to earn points, how to lose points, how to win the game, and how to lose the game
Code Quality (10 Points)	Little or no code exhibits the elements of good style.	Some code exhibits elements of good style, though style could be improved.	All code submitted exhibits elements of good style, including meaningful variable

			names, appropriate comments, proper use of white space, and bundling repetitive code into functions
Playability (15 points)	The grader was able to experience few or no elements of gameplay described in the readme.txt file	The grader was able to experience some elements of gameplay described in the readme.txt file	The grader was able to experience all elements of gameplay described in the readme.txt file
Originality (15 points)	The submission was substantially similar to others received for this assignment	The submission included at least one aspect that set it apart from others submitted for this assignment	The submission included unique elements which set the game apart from others submitted for this assignment

How to Submit

Once your project is completed, compress all the files into one .zip or .tar.gz archive with a name based on your Drexel UserID. A student with UserID abc123 should submit a file named hw4_abc123.zip or hw4_abc123.tar.gz. The file should include all code as well as the required readme.txt file.

The compressed file must be submitted to Blackboard Learn by the assignment's due date. Late assignments will not be accepted. If you accidentally submit the wrong file, you must reach out to a TA to have the submission cleared before the assignment's due date.

Academic Honesty

You must be the sole original author of the entire solution you submit. You must compose all program and written material yourself. All material taken from provided sources (e.g. this tutorial, textbooks, in-class examples, labs, etc.) must be appropriately cited.