Lab 2

Object Oriented Programming

Lab Objective: *Teach how to use OOP in Python and illustrate the uses of OOP in programming graphical user interfaces.*

Programming Paradigms

Writing readable code is one of the most important skills for a programmer to learn. Good code should not only perform well, but should be understandable to others. The main key to understandability is good organization; common ways of organizing code are called *programming paradigms*.

Object-oriented programming (OOP) is one of these paradigms. It allows us to model and replicate organization and structure from real life. OOP also allows us to create a "black box" that can be used without understanding the inner workings of the class and its methods.

Object-oriented programming has become a prominent and important programming paradigm used to simplify, organize, and clarify code. In this lab we will first learn how to use OOP with Python and implement some simple examples. Then we will learn about *graphical user interfaces*, or GUI's (pronounced "gooeys") and OOP's uses in implementing them.

Fundamental Concepts

At its core, object-oriented programming relies on the manipulation and coordination of *objects*. A programming object represents a piece of code that tracks the *state* of an object's various attributes and provides methods for accessing or altering these attributes. Python, like most modern programming languages, supports object oriented programming concepts; in fact, *everything* in Python is an object.

The attributes of an object are referred to as variables; the grouping of an object's variables and the various methods for accessing or altering these variables is called *encapsulation*, which is another important concept in OOP. Strings, lists, and integers are examples of objects in Python, which is why they have *methods*,

functions that belong specifically to those objects. We can create our own objects in Python by defining a *class*, or a blueprint that describes how to create an object.

```
class Backpack(object):
    # Use pass whenever you need to declare a function or class but are not yet ↩
        ready to implement it.
    pass
```

We now have a class `Backpack` which contains nothing. We put `object` in parenthesis to make our `Backpack` a subclass of Python's `object` class. This is an example of *inheritance*, the third important concept in OOP. Because our backpack class is a subclass of Python's `object` class, it inherits the properties and methods implemented in the `object` class.

To create a `Backpack` object, we can run something like `b = Backpack()`, which defines the variable `b` to be an *instance* of the class `Backpack`. However, since the class contains nothing, we can't really do anything with `Backpack` objects after we create them. Let's define an initial state for our object.

```
class Backpack(object):
    def __init__(self):
        self.color = 'black'
```

What's going on here? First, we defined a method, `self.color`. It can be called on instances of our class (`Backpack` objects) but on nothing else. The `__init__` method is defined for all objects and is executed immediately after the object is created. Its purpose is to set the initial state of the object. The `self` is a reference to the current instance of the class. All class methods take `self` as their first argument. In our `__init__` method, we are telling it to set the backpack's color to black. Now let's allow the user to define a different color, but keep black as the default.

```
class Backpack(object):

    # __init__ takes color as a keyword argument.
    # color = 'black' sets black as the default color if no alternate argument is↩
        passed in.
    def __init__(self, color='black'):
        self.color = color
```

The variable `self.color` is an *instance variable*. It is defined for a particular instance of a backpack, and can be different for different instances of the class. We can also change it outside of the class definition. Let's instantiate a `Backpack` object of the color purple. If we do not specify a color then it will default to black.

```
b = Backpack(color='purple')
a = Backpack()
print b.color # print 'purple'
print a.color # print 'black'
a.color = 'yellow'
print a.color # print 'yellow'
```

Finally, let's add some methods and additional state variables.

```
class Backpack(object):
```

```
    def __init__(self, color='black'):
        self.color = color
        # This creates an empty list when a backpack is created; it will store ↩
            the backpack's contents.
        self.contents = []

    def put(self, item):
        # This adds "item" into the contents list.
        # self.contents accesses the instance variable.
        # Then, .append() uses the list object's append function to add "item" to↩
             the end of the contents list.
        self.contents.append(item)


    def take(self, item):
        # The index() method returns the index of "item."
        # Then, the pop() method removes the item at the given index.
        return self.contents.pop(self.contents.index(item))
```

Now that we have defined methods for our `Backpack` class, we can interact with it.

```
b = Backpack(color='green')
print b.color # print 'green'
print b.contents # print '[]'
b.put(5)
b.put(3)
b.put(9)
b.put(1)
print b.contents # print '[5, 3, 9, 1]'
b.take(3)
print b.contents # print '[5, 9, 1]'
```

Our `Backpack` class is an example of *encapsulation*. Encapsulation refers to bundling together data and the methods that are used on that data. In `Backpack`, the color and list of contents are our data while `put()` and `take()` are our methods.

Problem 1. Create a class called `People` that allows you to keep track of professors and students. Each instance of `People` should have a variable that can be set to 'Student' or 'Professor' and a list to keep track of their courses. Include methods that allow you to add and take away a course.

One ability that our classes are missing is the ability to interact with common operations and functions. Built-in Python objects such as strings, lists, and integers can interact with `print`, $=, -, >,$ and $<$. However, if we were to use `print` on an instance of `Backpack`, Python would not know how to interpret it as a string and not know what to print. Similarly, Python would not know how to compare one backpack to another using $=, -, >,$ or $<$. We can fix this by using *magic methods*. Magic methods are special methods that allow classes to interact with these operations, so that they behave like built-in Python objects. This way, we can avoid writing ugly, counterintuitive functions like `self.equals(other)`, as we might do in other programming languages, and instead use magic methods to define what `self == other` means

for a particular class. Note that if not overwritten, non-overridden magic functions tend to base their output on memory location; be aware of this when testing your code. Magic methods can be recognized by the double underscores in their names.

Let's add two magic methods to `Backpack`. The `__repr__` method tells Python how to represent our class as a string. Then we will be able to call `print b` and it will print our backpack, b. The `__lt__` method will implement the $<$ operator, letting us compare two backpacks.

```
class Backpack(object):
    def __init__(self, color='black'):
        self.color = color
        self.contents = []

    def put(self, item):
        self.contents.append(item)

    def take(self, item):
        return self.contents.pop(self.contents.index(item))

    def __repr__(self):
    '''Tell the class how to represent the Backpack object as a string.'''
        # Return a string version of the contents list
        return str(self.contents)


   def __lt__(self, other):
   '''Compare the length of the two backpacks' contents' lists.'''
         return len(self.contents) < len(other.contents)
```

The other comparison operators are implemented in a similar way. Now, let's test our magic methods.

```
b = Backpack(color='green')
c = Backpack(color='yellow')
b.put(2)
b.put(5)
c.put(7)
c.put(1)
c.put(8)
print b # print '[2, 5]'
print c # print '[7, 1, 8]'
# Should return True, sending you into the if branch.
if b < c:
    print 'b is less than c'
```

For more information on creating methods and using magic methods, read the official Python documentation at `http://docs.python.org/2/reference/datamodel.html` or another useful reference at `http://www.rafekettler.com/magicmethods.html`. This information may be helpful in solving Problem 2.

Problem 2. Create a `ComplexNumber` object that supports the basic operations of a complex number. You must implement methods that will compare, add, subtract, multiply, divide, and conjugate complex numbers. Also, im-

plement a `norm()` method that will calculate the Euclidean distance between two points on the complex plane.

Graphical User Interfaces

GUI's are powerful tools for creating applications that users can interact with. PySide is a helpful library to build GUI's. From PySide, we will use two modules, QtGui and QtCore. QtCore has functions that will help us implement the inner workings of our GUI while QtGui will allows us to implement the graphics as well as the interface itself. To demonstrate the main ideas behind building a GUI, we will create a simple program that displays inputted text. As you read, follow along by typing the code into a text editor and running it from your command line (type in python filename.py). Hands on experience is the best way for GUI concepts to solidify.

```
from PySide import QtGui, QtCore

class Printer(QtGui.QWidget):
    # This class inherits from the QWidget class found in the QtGui module.
    def __init__(self):
        # Call the __init__ method from QWidget, the class Printer is sub-classed↩
             from (its "super" class).
        super(Printer, self).__init__()
```

Widgets are what make the magic happen in GUI's. In Qt, widgets are objects that represent various elements of a GUI. They keep track of drawing and refreshing the graphical display of the elements, abstracting the behavior of the elements, and defining ways to interact with other widgets. When you push a button or enter text, widgets are what notice and respond accordingly. In our `Printer` we will use `QLineEdit` and `QLabel`.

```
from PySide import QtGui, QtCore

class Printer(QtGui.QWidget):
    def __init__(self):
        super(Printer, self).__init__()
        # Call the _initUI function.
        self._initUI()

    def _initUI(self):
        '''Creates the widgets and tells them how to interact.'''

        # Create a class variable called textBar that is a QLineEdit widget.
        self.textBar = QtGui.QLineEdit()
        # Create a class variable called label that is a QLabel widget.
        self.label = QtGui.QLabel()
```

Now that we have our widgets, we need to tell them how to communicate. Qt uses a system of *signals* and *slots*. When a button is pushed or when text is entered, a widget throws a *signal*. We can specify which *slot* catches the signal. In this case, the signal being thrown is the text being entered in the text bar. We want a function that catches the signal, updates the displayed text, and clears the text bar.

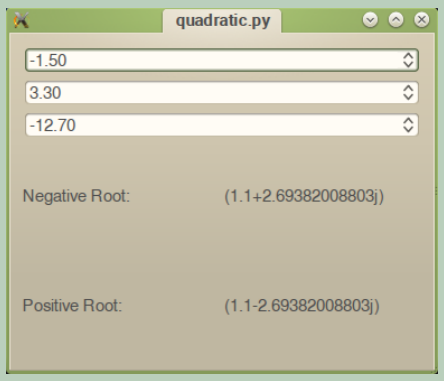
```
from PySide import QtGui, QtCore

class Printer(QtGui.QWidget):
    def __init__(self):
        super(Printer, self).__init__()
        self._initUI()

    def _initUI(self):
        self.textBar = QtGui.QLineEdit()
        self.label = QtGui.QLabel()

        # When return is pressed in the textBar, it sends a signal and goes into ↩
            the function updateText.
        # self.textBar accesses the textBar object.
        # returnPressed defines the inputted signal.
        # connect links the signal to the method later defined in this class: ↩
            updateText.
        self.textBar.returnPressed.connect(self.updateText)


    def updateText(self):
    '''Updates what text is displayed and clears the textBar'''
        self.label.setText(self.textBar.displayText())
        self.textBar.clear()
```

Next, we need to set the layout and create a function that can be called from the command line.

```
from PySide import QtGui, QtCore
import sys

class Printer(QtGui.QWidget):
    def __init__(self):
        super(Printer, self).__init__()
        self._initUI()

    def _initUI(self):
        self.textBar = QtGui.QLineEdit()
        self.label = QtGui.QLabel()

        self.textBar.returnPressed.connect(self.updateText)

        # Create a vertical box layout.
        # This will stack all widgets added to it vertically.
        vbox = QtGui.QVBoxLayout()
        # Add textBar as a widget and display it on the first row (0) in the ↩
            first column (0).
        vbox.addWidget(self.textBar, 0, 0)
        vbox.addWidget(self.label, 1, 0)

        # Assemble the layout.
        self.setLayout(vbox)
        # Tell the dimensions of the vbox.
        # The first two numbers indicate placement on the screen while the second↩
             two represent the dimensions.
        self.setGeometry(50, 50, 200, 200)
        self.setWindowTitle("Simple Printer")
        self.show()
```

```
    def updateText(self):
        self.label.setText(self.textBar.displayText())
        self.textBar.clear()

def main():
    # Create a QApplication.
    # Note that if you are working in IPython Notebook, you need to restart your ↩
        kernel before running the program, or a RuntimeError will occur.
    app = QtGui.QApplication(sys.argv)
    # Create a Printer object.
    # Since _initUI is called in the constructor, the GUI will appear and run.
    p = Printer()
    sys.exit(app.exec_())
if __name__ == "__main__":
    main()
```

Problem 3. Create a simple graphical user interface that will solve the quadratic formula given the necessary parameters. Make the GUI look similar to the one below.



The widgets that you will need are: `QDoubleSpinBox`, `QLabel`, `QGridLayout`, and `QVBoxLayout`. You may also want to import the cmath module in order to calculate complex solutions. You can view the documentation for these classes, including all their methods and signals, at `http://qt-project.org/doc/qt-4.8/classes.html`

Specifications

The following is a guideline for your solutions.

```
import sys
from PySide import QtGui, QtCore

class People(object):
    pass
```

```
class ComplexNumber(object):
    pass

class QuadraticCalculator(QtGui.QWidget):
    pass
```