Name: Evan Leglar	**Lab Time** Tuesday 12:00 PM

Names of people you worked with:

- N/A

Websites you used:

- http://www.marinamele.com/2014/04/modifying-add-method-of-python-class.html
- https://www.journaldev.com/22460/python-str-repr-functions
- http://hplgit.github.io/primer.html/doc/pub/class/._class-solarized005.html
- https://www.geeksforgeeks.org/operator-overloading-in-python/
-

Approximately how many hours did it take you to complete this assignment (to nearest whole number)?	3

The Rules: Everything you do for this lab should be your own work. Don't look up the answers on the web, or copy them from any other source. You can look up general information about Python on the web, but no copying code you find there. Read the code, close the browser, then write your own code.

By writing or typing your name below you affirm that all of the work contained herein is your own, and was not copied or copied and altered.

Evan Leglar

Note: Failure to sign this page will result in a 50 percent penalty. Failure to list people you worked with may result in no grade for this lab. Failure to fill out hours approximation will result in a 10-percent penalty.

Turn .zip files of Python code to Canvas or your assignment will not be graded

BEFORE YOU BEGIN

1. Make a new project for this lab
    a. Create a complex.py file for your code
2. Read up on complex numbers if you've forgotten how they work
3. Read chapters 15 and 16 in the text book

Learning Objectives:

You should be able to do the following:

- Make a class
- Add member values to the class
- Override the base members such as repr, +, str

Thoughts (come back and read these after you've read the problems):

1. Python already has support for complex numbers. We're going to ignore that for the purposes of this assignment. **Your code should not use any built-in Python functionality for complex numbers** (except possibly for testing).

2. `__repr__` and `__str__` do similar things. Python internally uses `__repr__` sometimes and `__str__` at other times. You should implement `__repr__` to return the string representation of the complex number class. Then have `__str__` have a single line: `return self.__repr__()`

Grading Checkpoints

Use of NumPy is not allowed for this assignment. You can implement everything here using only the standard math library.

1. Working constructor: 1 for each case [3 points total] *
2. `__repr__` and `__str__` work as described [1 points total] *
3. Addition works: two complex numbers (1 point); one complex and one integer or float (1 point); one integer or float and one complex number (1 point). [3 points total] **
4. Subtraction works for all three cases (like addition). [1 point total] **
5. Multiplication works for all three cases (like addition). [1 point total] **
6. Division works for all three cases (like addition). [1 point total] **
7. Negation and complex conjugate work. [1 point total] **
8. Reasonable tests for your code, which do not run on import. [1 point total]
9. Your code passes PEP checks. [2 points total] *
10. Extra Credit: 1 point each for polar form, exponentiation, and roots. [3 points total) ^

* Autograder will inform you of correctness.
** Autograder will check to make sure that the operators are functional, but will not tell you if the number returned is right.
^ Autograder will grade but hide output until after grades are released.
Blank indicates the problem will be manually graded.

Turn in your code and this PDF to Gradescope.

Problem 1 Create a complex class

1. We're going to write a class for complex numbers. Start by writing a class called `Complex` that takes two arguments, representing the real and imaginary parts of a complex number. Put the class definition in a file called `complex.py`, and call the member variables `re` and `im`. You should make these arguments optional, and default to something sensible if they are missing. Check that your code does the right thing in response to the following:

```
a = Complex(1.0, 2.3)     # 1 + 2.3i
b = Complex(2)            # 2 + 0i
c = Complex()             # 0 + 0i
```

2. Implement the `__repr__` and `__str__()` functions, so that

```
a = Complex(1, 0)
b = Complex(0, -2)
c = Complex(1, 3)
print(a)
print(b)
print(c)
```

works and prints out (exactly, including parentheses and spaces)

```
(1 + 0i)
(0 - 2i)
(1 + 3i)
```

3. Ensure we can get at the real and imaginary components of your Complex class like this:

```
c = Complex(1.2, 3.4)
print(c.im)
print(c.re)
```

Comments for grader/additional information (if any)

Code (test)

`#!/bin/usr/env python3`
 	
`class Complex:`

```
    def __init__(self, re=0, im=0):
        self.re = float(re)
        self.im = float(im)

    def __str__(self):
        return "({0} + {1}i)".format(self.re, self.im)

    def __repr__(self):
        return '(' + str(self.re) + ' + ' + str(self.im) + 'i)'

    # def __add__(self,other):


if __name__ == '__main__':

    a = Complex(1.0, 2.3)     # 1 + 2.3i
    b = Complex(2, -3)        # 2 + 0i
    c = Complex()             # 0 + 0i
    print(a)
    print(b)
    print(c)
    print(b.re)
    print(repr(a))
```

Problem 2 addition and subtraction

Implement addition, such that the following works:

```
a = Complex(1, 2)
b = Complex(3, 4)
print(a + b)
print(a + 1)
print(1 + a)
```

and prints out

```
(4 + 6i)
(2 + 2i)
(2 + 2i)
```

Once you have addition working, **implement subtraction**. This should look a lot like addition.

Also, **write some test code** to verify that your code is doing the right thing. This test code should not run on import. You should also write test code for all other basic operations. It's up to you what your test code looks like, but it should be enough to convince yourself (and anyone looking at your code) that it's sufficient.

Comments for grader/additional information (if any)

Code (test)

```
#!/bin/usr/env python3


class Complex:

    def __init__(self, re=0, im=0):
        self.re = float(re)
        self.im = float(im)

    def __repr__(self):
        return '(' + str(self.re) + ' + ' + str(self.im) + 'i)'

    def __str__(self):
        return self.__repr__()

    def __add__(self, other):
        add_re = self.re + other.re
        add_im = self.im + other.im
        return Complex(add_re, add_im)

    def __sub__(self, other):
        subtract_re = self.re - other.re
        subtract_im = self.im - other.im
        return Complex(subtract_re, subtract_im)

    # def __mul__(self, other):
    #     multiply_re = self.re *
    #     multiply_im =
    #     return Complex(multiply_re, multiply_im)


if __name__ == '__main__':

    a = Complex(1.0, 2.3)    # 1 + 2.3i
    b = Complex(2, 0)       # 2 + 0i
    c = Complex(0, 4)            # 0 + 0i
    print(b-c)
    print(a+c)
    print(a-b)
```

Problem 3 Multiplication and division

Now, **implement multiplication and division.** Update your test code and verify that everything works as expected.

Comments for grader/additional information (if any)

I did the original multiplication math by hand to determine the algebra needed to handle the real and imaginary parts of operation. I then did some research and found a compact version of __mul__() that worked for my code.

Code (test)

```
#!/bin/usr/env python3


class Complex:

    def __init__(self, re=0, im=0):
        self.re = float(re)
        self.im = float(im)

    def __repr__(self):
        return '(' + str(self.re) + ' + ' + str(self.im) + 'i)'

    def __str__(self):
        return self.__repr__()

    def __add__(self, other):
        add_re = self.re + other.re
        add_im = self.im + other.im
        return Complex(add_re, add_im)

    def __sub__(self, other):
        subtract_re = self.re - other.re
        subtract_im = self.im - other.im
        return Complex(subtract_re, subtract_im)

    def __mul__(self, other):
        return Complex(self.re*other.re - self.im*other.im,
self.im*other.re + self.re*other.im)

    def __truediv__(self, other):
        selfr, selfi, otherr, otheri = self.re, self.im, \
                         other.re, other.im  # short forms
        r = float(otherr ** 2 + otheri ** 2)
        return Complex((selfr * otherr + selfi * otheri) / r, (selfi
* otherr - selfr * otheri) / r)


if __name__ == '__main__':

    a = Complex(4, 3)
    b = Complex(2, 6)
    c = Complex(0, 4)
    print(a*b)
    print(a/b)
```

Problem 4 Negation and complex conjugate

1. Finally, implement negation and the complex conjugate:

```
a = Complex(1, 2)
print(-a)
print(~a)
```

should print out

```
(-1 - 2i)
(1 - 2i)
```

Hint: The ~ is called a “tilde”.

Comments for grader/additional information (if any)

https://www.geeksforgeeks.org/operator-overloading-in-python/

Code (test)

```
#!/bin/usr/env python3


class Complex:

    def __init__(self, re=0, im=0):
        self.re = float(re)
        self.im = float(im)

    def __repr__(self):
        return '(' + str(self.re) + ' + ' + str(self.im) + 'i)'

    def __str__(self):
        return self.__repr__()

    def __add__(self, other):
        add_re = self.re + other.re
        add_im = self.im + other.im
        return Complex(add_re, add_im)

    def __sub__(self, other):
        subtract_re = self.re - other.re
        subtract_im = self.im - other.im
        return Complex(subtract_re, subtract_im)
    def __mul__(self, other):
        return Complex(self.re*other.re - self.im*other.im,
self.im*other.re + self.re*other.im)

    def __truediv__(self, other):
        selfr, selfi, otherr, otheri = self.re, self.im, \
                                 other.re, other.im  # short forms
        r = float(otherr ** 2 + otheri ** 2)
        return Complex((selfr * otherr + selfi * otheri) / r, (selfi
* otherr - selfr * otheri) / r)

    def __neg__(self):
        return Complex(-1*self.re, -1*self.im)

    def __invert__(self):
        return Complex(self.re, -1*self.im)


if __name__ == '__main__':

    a = Complex(4, 3)
    print(a)
    print(-a)
    print(~a)
```

Problem 5 Code quality

Clean up your code and make sure it follows PEP8 standards.

Comments for grader/additional information (if any)

Extra credit: Exponentiation and roots

1. Write an instance method `as_polar`, which takes in no additional arguments and returns two values, r and theta (in this order) representing the complex number in its polar form, written in the form re^(iθ) (where i is the imaginary unit). θ should range from –pi (non-inclusive) to pi (inclusive).

   For example: `Complex(1, 1).as_polar()  # Outputs (1.4142, 0.7854)`

2. Implement exponentiation with real numbers, for example:

   ```
   Complex(1, 1)**(-2.5)  # Outputs Complex(-0.1609, -0.3884)
   ```

3. Write a method `roots` which takes in one additional argument `n`, an integer (positive, negative, or 0), and returns a list of all numbers `r_i` (as Complex objects, in any order) such that `r_i**n = c` (where `c` is the value expressed by your Complex object).

   For example:

   ```
   Complex(0, 1).roots(2)
   # Returns [Complex(0.7071, 0.7071), Complex(-0.7071, -0.7071)]
   ```

Comments for grader/additional information (if any)