Overview: In this lab, you are provided with three classes: `Number`, `Integer`, and `Float`. `Number` is an abstract base class, while `Integer` and `Float` are derived classes. You will modify and extend `Integer` and `Float`, and you will write an additional derived class representing a complex number type. You are required to use a test-driven development approach in completing this work, using pytest. You are also required to use git/GitHub for frequently tracking your updates.

You are **not allowed to import any libraries** (other than pytest-related libraries) for your solution. For example, use of the `numpy` or `scipy` library, etc., is prohibited. The goal is for you to think about inheritance and about algorithmic implementations here, rather than relying on external libraries to quickly solve your task.

IMPORTANT: In grading this assignment, I will be strictly enforcing the requirements below that (a) each team member contributes equally, and (b) that each team member follows the structure of (1) write a pytest test, then (2) git commit, then (3) write dev code, then (4) commit. In other words, commits on git are how I will be able to tell that each team member followed the TDD protocol. Follow the instructions carefully!

Setup: Start by cloning the GitHub repo corresponding to your team assignment. This repo will already have an appropriate directory structure for pytest, allowing you to separate your development code from your testing code. Below is the structure:

```
.
|-- code_base
|   |-- __init__.py
|   |-- Number.py
|   |-- Integer.py
|   |-- Float.py
|-- tests
    |-- __init__.py
    |-- test_Integer.py
    |-- test_Float.py
```

Note the empty `__init__.py` files in each directory to ensure that you have a package from Python's perspective.

Note: You will need to change any existing imports within the code to reflect the package structure, i.e., put `code_base.` as part of the import similar to:

```
from code_base.Number import Number
```

Intentionally, there is very little testing already for `Integer` — and none for `Float` — in the provided code. However, if you look at `tests/test_Integer.py`, you will see some starter pytest code that includes:

- appropriate imports
- use of pytest fixtures to set up data for later use
- examples of pytest tests using `assert`
- an example of a pytest test looking for an exception

To get you going in writing TDD code in pytest, I am asking you to write a complete set of tests for `Integer` and for `Float` using pytest. At this point, you will not be doing true TDD as the development (so far) in `Integer` and `Float` have already been done for you.

Guidelines:

- The number of tests required for each method in each class will vary. For example, `__add__` in `Integer` will require at least five different tests (inspect the code and determine why). Make sure that you have written enough pytest tests to thoroughly cover all parts of the code implementation.
- Each team member must write a roughly-equal proportion of tests using pytest, and commit them using git, and push to the GitHub repo.
- Tests for `Integer` must go in `test_Integer.py` and tests for `Float` must go in `test_Float.py`.
- Remember the git best practice of using a working branch (not the local main branch) to perform your work.
- For this first set of tests (for the existing `Integer` and `Float`, you do not need to git-commit after each test (that will change below).

⇒⇒ **As you are developing each method inside `Complex` below, but sure to use a TDD-based approach using pytest, and make abundant use of pytest fixtures so that you aren't hardcoding-and-repeating data and/or steps inside of test functions, but rather using the fixtures instead. Also use prints within your pytest tests that indicate what you're testing.**

Each team member must develop a set of tests, following this protocol:

1. **Write a pytest test, and demonstrate that the test fails.**
2. **git commit that test before writing the dev code, with a commit comment describing the test.**
3. **Write the dev code to pass the test, and demonstrate that the test passes.**
4. **git commit that dev code before, with a commit comment describing the dev code, before moving to the next test.**

⇒⇒

Complex class: A complex number is a number of the form $a + bi$, where a and b are real-valued numbers (either integer or floating-point), and the special number i represents the "imaginary unit" and satisfies the equation $i^2 = -1$. Complex numbers have many different applications — including in image processing, which allows much more efficient filtering of images and other digital signals. You need not worry about the details of complex numbers — other than that your class needs to store a numeric real number (the a part of the complex number will be inherited from `Number`, and should not be redefined) and a numeric coefficient for the imaginary part (the b). The i will just appear as part of the string returned by `__str__`. That is, from an implementation perspective, a `Complex` number will consist of an inherited real value and a new imaginary coefficient (number). Any "i" that appears in printed output will just be a result of what you do inside `__str__`.

- The name of your third derived class must be `Complex`, and it must also inherit from `Number`.
- Include an additional instance variable that will allow you to store the imaginary part of the complex number (the real part will be inherited from `Number`). Also include an additional instance variable that will allow you to do appropriate formatting of decimal places, similar to `Float`.
- In your `__init__`, make sure to call `super().__init__`. Allow your formatting variable to have a default value of 1 (decimal places) for any floating-point coefficients (subject to details below).

- Include a `setPrecision` method that will allow the user to change the number of decimal places used in `__str__` below (similar to what you did for `Float`).

- Include a `__str__(self)` method (which will override that inherited from `Number`) to appropriately format the numeric parts of the complex number appropriately:
  - If the real part is:
    - integer, format as an integer (no decimals);
    - floating point, format using the stored number of decimal places;
    - zero, do not include as part of the returned string, and do not include an operator (let the sign of the imaginary part do that work).
  - If the imaginary coefficient is:
    - integer, format as an integer (no decimals);
    - floating point, format using the stored number of decimal places;
    - 1, include only the i (no number);
    - zero, do not include as part of the returned string, and also omit the plus (or minus) operator and the i.
  - If the imaginary coefficient is:
    - positive, include a + operator in the string representation;
    - negative, include a – operator in the string representation, and do not (also) include a – with the imaginary coefficient.
  - If both the real and imaginary parts are 0, just return 0 (no i or operator).

  Below are some examples:

```
a = Complex(1,0)     # 1 + 0i  -- should print as "1"
b = Complex(-1,0)    # -1 + 0i -- should print as "-1"
c = Complex(0,1)     # 0 + 1i  -- should print as "i"
d = Complex(0,-1)    # 0 + -1i -- should print as "-i"
e = Complex(0,0)     # 0 + 0i  -- should print as "0"
f = Complex(0,-2)    # 0 + -2i -- should print as "-2i"

g = Complex(3,2)     # should print as "3 + 2i"
h = Complex(3,-2)    # should print as "3 - 2i"
i = Complex(3.1,2)   # should print as "3.1 + 2i"
j = Complex(3.1,-2)  # should print as "3.1 - 2i"
k = Complex(3,2.1)   # should print as "3 + 2.1i"
l = Complex(3,-2.1)  # should print as "3 - 2.1i"
m = Complex(-3,-2.1) # should print as "-3 - 2.1i"
complexes = [a,b,c,d,e,f,g,h,i,j,k,l,m]
for comp in complexes:
    print(comp)
```

- Override the `__add__` method to allow addition of two complex numbers. Remember this should return a `Complex` object and should not modify either `self` or `other`.

  Given the two complex numbers $a + bi$ and $c + di$, the sum of those will result in a new complex number

  $$r + si \qquad \text{where} \qquad r = a + c, \quad s = b + d$$

  See https://bit.ly/complex_number_math for more information on adding two complex numbers.

- Can you add an integer and a complex? Can you add a floating-point and a complex? In both cases, the answer is yes, because any real-valued number can be represented as a complex number having imaginary coefficient of zero. So, make sure that your `__add__` methods inside `Integer`, inside `Float`, and inside `Complex` handle all such additions appropriately.

Note: You will be able to test a case such as

```
c = Complex(2,3)
i = Integer(7)
c_new = i + c
```

because the `i + c` expression has `i` as its leading operand which therefore plays the role of `self`[1], and being of type `Integer` will call the `__add__` method inside `Integer` (not inside `Complex`!).

You will not be able to test a case such as

```
c = Complex(2,3)
i = 7
c_new = i + c
```

because here `i` plays the role of `self`, and being of type `int` will call the `__add__` method inside `int` to which we have no access. That is, the `int` class will not know how to add an argument that is some (unknown to it) `Complex` type.

You will, however, be able to test a case such as

```
c = Complex(2,3)
i = 7
c_new = c + i
```

because here `c` plays the role of `self`, and being of type `Complex` will call the `__add__` method inside `Complex`.

Make sure that you are testing all possible arrangements to which you have access, and by using pytest to do so. Again, make sure to git-commit **immediately after** writing the test (i.e., **before writing the dev code**), and then again git-commit **immediately after** writing the dev code and ensuring the test passes.

- Override the `__mul__` method to allow addition of two complex numbers, or the multiplication of a complex number by a scalar (integer or floating-point). You should use `instanceof` to determine the appropriate approach to take. Remember this should return a `Complex` object and should not modify either `self` or `other`.

  Given the real-number (scalar) c and the complex number $a+bi$, the product of those will result in the new complex number[2]

  $$r + si \qquad \text{where} \qquad r = ca, \quad s = cb$$

  Given the two complex numbers $a + bi$ and $c + di$, the product of those will result in the new complex number

  $$r + si \qquad \text{where} \qquad r = ac - bd, \quad s = ad + bc$$

  See https://bit.ly/complex_number_math for more information on multiplying two complex numbers or a complex by a scalar.

  Just as a stress test for your `__mul__` and `__str__`, recall from above that $i^2 = -1$ by definition. If things are working correctly, you should be able to have your code produce that as output:

  ```
  i = Complex(0,1)  # 0 + 1i  -- should print as "i"

  minus_one = i * i
  print(f"i * i = minus_one)  # should print as "i * i = -1"
  ```

[1] `i` plays the role of `self` in any of the equivalent invocations `i + c` or `i.__add__(c)` or `Integer.__add__(i,c)`.

[2] Alternatively, the scalar can be considered to be a complex having zero for its imaginary coefficient. You can then apply the same rules above for multiplying two complex numbers, understanding that the zero-imaginary-coefficient part will cause those affected terms to just fall away.

- Can you multiply an integer and a complex? Can you multiply a floating-point and a complex? Yes — see the scalar-involved definition above. Make sure that your `__add__` methods inside `Integer`, inside `Float`, and inside `Complex` handle all such additions — and in differing arrangements — appropriately.

Submitting:

- The entire final collection of fully-tested development code and all pytest tests should be pushed to the GitHub repo where I can access it. As you will all be contributing to the TDD and development, you will each be pushing to (and likely pulling from) the GitHub repo.
- **All members** of the team must submit a reflection and evaluation — see below.

Reflection and Team Evaluation: In the online text on Lyceum, provide a brief reflection and an evaluation of your team. Specifically:

- What did you find rewarding about using pytest? Challenging?
- What did you find rewarding about using git? Challenging?
- What might you likely use again in the future?
- Describe for each of you and your teammates:
  - What you did specifically in completing the project.
  - Whether you did (a) fair share of the work, (b) more than their fair share, or (c) less than their fair share, providing justification.
  - For each teammate, whether they did (a) fair share of the work, (b) more than their fair share, or (c) less than their fair share, providing justification.