Lab 01: Monomials and Polynomials

In this lab, you will play with object-oriented programming and inheritance. You will also got to see some of python's magic methods at work. These are methods that start and end with two underscores. You should not make up your own variable names that start and end with two underscores. These are special to python.

Recall that a **monomial** is an expression ax^b where a is a number and b is (usually) an integer. A **polynomial** is a sum of monomials. We also think of polynomials as functions. That is, if we specify a number for x, we can compute the value of the polynomial *at* x. This is called **evaluating** the polynomial. The polynomial $x^2 + 3x + 7$ evaluated at 2 is 17 because $2^2 + 3(2) + 7 = 17$.

Here are the descriptions of all the methods we want to be able to call on `Monomial` s and `Polynomial` s. The acronym **ADT** stands for **Abstract Data Type**. Usually, we will use this term to describe the **interface** to the classes we are writing, but often it will be more abstract in that it might not describe just a single class There are a lot of methods, but most of them are provided in the starter code.

The Monomial ADT

- **__init__(self, coeff, exp)** - Initializes a new `Monomial` with the given coefficient and the given exponent.
- **evaluate(self, x)** - Evaluate the monomial at `x` and return teh result.
- **__eq__(self, other)** Returns `True` if `self` and `other` have the same coefficient and same exponent. Returns `False` otherwise.
- **__lt__(self, other)** - Returns `True` if `self.exp < other.exp` . Also returns `True` if `self.exp == other.exp` and `self.coeff < other.coeff` . Returns `False` otherwise.
- **__mul__(self, other)** - Returns the product of the two monomials.
- **__neg__(self)** - Returns a new `Monomial` whose coefficient is `-self.coeff` and whose exponent is `self.exp` . This is the same as multiplying by the monomial by -1.

The Polynomial ADT

- **__init__(self, terms)** Initializes a new `Polynomial` with the given sequence of terms. It could be a list of Monomials or a list of (coefficient, exponent)-pairs.

- **evaluate(self, x)** - Evaluate the polynomial at `x` and return the result.
- **__eq__(self, other)** - Returns `True` if `self` and `other` have the same terms. Returns `False` otherwise.
- **__mul__(self, other)** - Return a new monomial that is the product of `self` and `other`.
- **__add__(self, other)** - Return a new polynomial that is the sum of `self` and `other`.
- **__neg__(self)** - Returns a new `Monomial` whose coefficient is `-self.coeff` and whose exponent is `self.exp`. This is the same as multiplying by the monomial by -1.
- **reduce** - Combine like terms. Remove zero terms. Sort the terms by the monomial ordering.

Evaluating Polynomials

Write a method called `evaluate` for the `Polynomial` class that takes a parameter `x` and evaluates the polynomial at `x`. You will have to add up the evaluations for each of the monomials. Don't forget to make the first parameter named `self` for the polynomial you are evaluating.

Reducing Polynomials

There is some code provided to reduce a polynomial by combining like terms and eliminating zeros. This will be handy for several different functions. It uses a dictionary (`dict`) object to keep track of terms. It maps exponents to coefficients. If the exponent already appeared, it updates the coefficient. If the exponent is new, it adds it to the `dict`. Then, it creates a list of terms from the dictionary and updates the list of terms. Finally, it sorts it.

Sorting the terms works because we defined `__lt__` on the `Monomial` class. It allows us to use the `<` symbol to compare terms, and this is enough to `sort` them.

Take a look at the code and make sure you understand what it is doing. Take a moment to think about how you would do this without a dictionary. I suspect it would be a little tricky.

Testing Equality

There is some code provided to test if two polynomials are equal. It uses the fact that python compares list by comparing the elements. Each time it compares terms, it uses the `__eq__` method from the `Monomial` class. It also reports `False` if the lengths are different. It calls the `reduce` function first to be sure the terms are aligned.

Take a moment to look over this code. Ask a question if you don't understand it.

A case for inheritance.

Once we have these two classes, it might make sense to try to add two monomials to produce a polynomial. Or, we might want to add a `Monomial` to a `Polynomial`. Or, we might want to multiply a `Monomial` times a `Polynomial`. Without care, these operations are all problematic. The reason these operations all make sense is really simple: a *monomial is a polynomial*. So, we might hope that a `Monomial` can be treated as a `Polynomial`. We have to be careful to make this work, because objects really do have types. This is accomplished using inheritance.

Here is an important rule of thumb: "Inheritance always means **is a**". That is, if we make our `Monomial` class *inherit from* `Polynomial`, then it better be the case that a monomial **is a** polynomial.

In the code, the **parent** class (AKA the **superclass**) is put in parentheses after the name of the child class (AKA the **subclass**).
We say that the subclass **extends** the superclass. If we want to call the initializer for the superclass, we have to do it explicitly. So, the code looks like the following:

```
class Monomial(Polynomial):
    def __init__(self, coeff, exp):
        self.coeff = coeff
        self.exp = exp
        Polynomial.__init__(self, [self])
```

We have to use the "full name" of the superclass's initializer. If we wrote `self.__init__` we would be calling this function again (recursion!) and we don't want that. In this explicit call, you also need both arguments, `self` (the object to operate on) and `[self]` (the list of monomial terms). It's okay if this blows your mind a little to begin with. It does mean that the monomial will have a list of terms and it is itself an element of that list. It's not the only way we could have done this, but it's fine for now.

Modify the `Monomial` class to extend `Polynomial`.

Addition

We want to be able to add two polynomials. Ideally, we want code like the following to work

seamlessly.

```
f = Polynomial([(1,2), (2,1)])
g = Polynomial([(3,2), (5,1), (3,0)])
h = f + g
```

In this case, we need to define our own function to be called when the `+` sign is used with `Polynomial` s. In python, the magic method `__add__` does exactly this.

```
def __add__(self, other):
    # Add self and other.
```

You have been provided with a method called reduce that combines like terms and removes zeros. So, it suffices to simply make a new polynomial whose list of terms contains all the terms from `self` and all the terms from `other` . Then reduce it and return it.

Multiplication

The magic method for multiplication is called `__mul__` .

To multiply two polynomials, you add up all the products that can formed by multiplying a term from the first with a term from the second. This will require two loops. One to loop over the terms in the first polynomial and one to loop over the terms in the second. Because multiplication has already been implemented for `Monomial` , it's easy to produce these terms.

Work out small examples on paper if you are stuck. Also, use the reduce method on the result before you return it.

There is one weird thing about inheritance and magic methods. Regardless of the order, python will try to call the magic method from the subclass. In our case, we want to use `Polynomial.__mul__` if we are multiplying a `Polynomial` times a `Monomial` . This is why we added an explicit check `if not isinstance(other, Monomial):` to make sure python gets it right. We had to do the same thing for the `__eq__` method. This is a rather rare case and I highlight it here in case you wanted to know why that code is there. Many things will still work without that line, but not all.

Delegating to the superclass

One of the most important reasons to use inheritance is to avoid code duplication. This is partly from the convenience of not repeating ourselves, but also because duplicated code is prone to bugs. If you copy the same method for two different classes and you fix a bug in one copy, you have to fix the same bug in the other copy. You think you will remember, but you need all your working memory to write code. You waste your mental resources when you expect yourself to remember unimportant things. Also, experience has shown that *many* errors are introduced by copy and paste programming.

Look closely at the method for negating a monomial. It works by creating a monomial -1 and multiplying it by self.

```
def __neg__(self):
    return self * Monomial(-1, 0)
```

It seems that this is also the way to negate polynomials. So, we can simply move this function from `Monomial` to `Polynomial`. We don't want to have two copies of the same code. We use inheritance so the method works for both classes.

In general, we always want to be on the lookout for functions that can be handled higher up the class hierarchy.

Implement `__neg__` for `Polynomial`s.

Note that we can also implement subtraction now quite easily:

```
def __sub__(self, other):
    return self + (-other)
```

Polynomials are also functions, let's treat them that way.

Often, in math, if we said p is a polynomial and $p(x)$ is p evaluated at x, we are treating p like a function. It would be nice to write our code accordingly. In python, to make an object *callable* this way, we just have to implement the magic method `__call__`.

The `__call__` method is not implemented on every class. For example, see the following code and its output.

```
s = "Hello"
s(3)
```

```
Traceback (most recent call last):
  File "/Users/don/Dropbox/work/teaching/new_ds/18_spring/01_OOP/lab/skeleton/xb4mogl08_code
    s(3)
TypeError: 'str' object is not callable
```

On the other hand, the following works.

```
class Foo:
    def __call__(self, x):
        print("Foo", x)

f = Foo()
f("exececuted f(...)")
# This is the same as
Foo.__call__(f, "executed Foo.__call__(f, ...)")
```

```
Foo exececuted f(...)
Foo executed Foo.__call__(f, ...)
```

In our case, we want `Polynomial.__call__(p, x)` to return `p.evaluate(x)`. To do this, in the `Polynomial` class, add the following method.

```
def __call__(self, x):
    return self.evaluate(x)
```

Now, try it out with code such as the following.

```
# p = x^2 + 3x + 7
p = Polynomial([(1,2), (3,1), (7,0)])
print(p(0))
print(p(1))
print(p(2))
```

Did it do what you expected?

Now, try to use the same syntax with a `Monomial`. That is, run `m(2)` for some `Monomial` called `m`. But first, predict what you think will happen. Discuss it with your partner.

It is very useful to think about what is happening in this case. If `m` is a `Monomial` then writing `m(2)` invokes the `__call__` method. This method has not been implemented in our `Monomial` class. As a result, python checks the superclass, `Polynomial`. The `__call__` method *is* implemented for `Polynomial`, so it gets called. Inside the `__call__` method, it calls `self.evaluate` where `self` in this case is `m`. Since `m` is a `Monomial`, it calls the `Monomial.evaluate` method rather than the `Polynomial.evaluate` method. Every time we look up a name, we restart the lookup process.

This is an important idea. Inheritance can (and often does) lead to an object having multiple methods with the same name. The method that actually gets called is the most specific class of that object that has the method defined. This is really just an ordered list of namespaces to search to find the attribute or method with the given name.

In this case, we only have to write the `__call__` method once and get the benefit for both classes.

Just for Fun: Monomials don't really need terms

Only go through this last part if you thoroughly understood everything above and want to learn about a cool feature of python.

If you felt uncomfortable with the idea that an instance of the `Monomial` class was required to have a list of terms that included just itself, there is another approach you could take. Notice that every time you need the terms of a `Polynomial`, you are iterating over all of them. Iteration works by yet another magic method, `__iter__`. You could make `Polynomial` **iterable** by implementing this method as follows.

```
# In class Polynomial
def __iter__(self):
    return iter(self.terms)
```

It just returns an iterator over the terms. Now, it's possible to write code like: `for t in mypolynomial:` instead of `for t in mypolynomial.terms:`. It's a little shorter and makes sense. To implement the iterator for `Monomial`, we do the following.

```
# In class Monomial
def __iter__(self):
    yield self
```

The `yield` keyword is a little like `return` but is slightly different. We'll learn more about it as the

course goes on and I encourage you to look it up if you are interested. In this case, iterating over a `Monomial`, will loop one time and will only have the one value, `self`. This is exactly what we wanted.

To make this work, you will want to remove most references to the `terms` in the `Polynomial` class and only iterate instead.