



CHAPTER 1

Python Classes

The built-in types, like strings have functions associated with them. So, for example, if you needed a string converted to uppercase, you would call its `upper()` function: -

```
my_string = "houston, we have a problem!"  
louder_string = my_string.upper()
```

This would set `louder_string` to "HOUSTON, WE HAVE A PROBLEM!" When a function is associated with a datatype like this, it called a *method*. A datatype with methods is known as a *class*. The data of that type is known as *instance* of that class. For example, in the example, we would say "my_string is an instance of the class `str`. `str` has a method called `upper`"

The function `type` will tell you the type of any data:

```
print(type(my_string))
```

This will output

```
<class 'str'>
```

A class can also define operators. `+`, for example, is redefined by `str` to concatenate strings together:

```
long_string = "I saw " + "15 people"
```

1.1 Making a Polynomial class

You have created a bunch of useful python functions for dealing with polynomials. Notice how each one has the word “polynomial” in the function name like `derivative_of_polynomial`. Wouldn’t it be more elegant if you had a `Polynomial` class with a `derivative` method? Then you could use your polynomial like this:

```
a = Polynomial([9.0, 0.0, 2.3])
b = Polynomial([-2.0, 4.5, 0.0, 2.1])
```

```
print(a, "plus", b , "is", a+b)
print(a, "times", b , "is", a*b)
print(a, "times", 3 , "is", a*3)
print(a, "minus", b , "is", a-b)
```

```
c = b.derivative()
```

```
print("Derivative of", b , "is", c)
```

And it would output:

```
2.30x^2 + 9.00 plus 2.10x^3 + 4.50x + -2.00 is 2.10x^3 + 2.30x^2 + 4.50x + 7.00
2.30x^2 + 9.00 times 2.10x^3 + 4.50x + -2.00 is 4.83x^5 + 29.25x^3 + -4.60x^2 + 40.50x + -18.00
2.30x^2 + 9.00 times 3 is 6.90x^2 + 27.00
2.30x^2 + 9.00 minus 2.10x^3 + 4.50x + -2.00 is -2.10x^3 + 2.30x^2 + -4.50x + 11.00
Derivative of 2.10x^3 + 4.50x + -2.00 is 6.30x^2 + 4.50
```

Create a file for your class definition called `Polynomial.py`. Enter the following:

```
class Polynomial:
    def __init__(self, coeffs):
        self.coefficients = coeffs.copy()

    def __repr__(self):
```

```
# Make a list of the monomial strings
monomial_strings = []

# For standard form we start at the largest degree
degree = len(self.coefficients) - 1

# Go through the list backwards
while degree >= 0:
    coefficient = self.coefficients[degree]

    if coefficient != 0.0:
        # Describe the monomial
        if degree == 0:
            monomial_string = "{:.2f}".format(coefficient)
        elif degree == 1:
            monomial_string = "{:.2f}x".format(coefficient)
        else:
            monomial_string = "{:.2f}x^{0}".format(coefficient, degree)

        # Add it to the list
        monomial_strings.append(monomial_string)

    # Move to the previous term
    degree = degree - 1

# Deal with the zero polynomial
if len(monomial_strings) == 0:
    monomial_strings.append("0.0")

# Separate the terms with a plus sign
return " + ".join(monomial_strings)

def __call__(self, x):
    sum = 0.0
    for degree, coefficient in enumerate(self.coefficients):
        sum = sum + coefficient * x ** degree
    return sum

def __add__(self, b):
    result_length = max(len(self.coefficients), len(b.coefficients))
    result = []
    for i in range(result_length):
        if i < len(self.coefficients):
            coefficient_a = self.coefficients[i]
        else:
            coefficient_a = 0.0
```

```
        if i < len(b.coefficients):
            coefficient_b = b.coefficients[i]
        else:
            coefficient_b = 0.0
        result.append(coefficient_a + coefficient_b)

    return Polynomial(result)

def __mul__(self, other):

    # Not a polynomial?
    if not isinstance(other, Polynomial):
        # Try to make it a constant polynomial
        other = Polynomial([other])

    # What is the degree of the resulting polynomial?
    result_degree = (len(self.coefficients) - 1) + (len(other.coefficients) - 1)

    # Make a list of zeros to hold the coefficients
    result = [0.0] * (result_degree + 1)

    # Iterate over the indices and values of a
    for a_degree, a_coefficient in enumerate(self.coefficients):

        # Iterate over the indices and values of b
        for b_degree, b_coefficient in enumerate(other.coefficients):

            # Calculate the resulting monomial
            coefficient = a_coefficient * b_coefficient
            degree = a_degree + b_degree

            # Add it to the right bucket
            result[degree] = result[degree] + coefficient

    return Polynomial(result)

__rmul__ = __mul__

def __sub__(self, other):
    return self + other * -1.0

def derivative(self):

    # What is the degree of the resulting polynomial?
    original_degree = len(self.coefficients) - 1
```

```
if original_degree > 0:
    degree_of_derivative = original_degree - 1
else:
    degree_of_derivative = 0

# We can ignore the constant term (skip the first coefficient)
current_degree = 1
result = []

# Differentiate each monomial
while current_degree < len(self.coefficients):
    coefficient = self.coefficients[current_degree]
    result.append(coefficient * current_degree)
    current_degree = current_degree + 1

# No terms? Make it the zero polynomial
if len(result) == 0:
    result.append(0.0)

return Polynomial(result)
```

Create a second file called `test_polynomial.py` to test it:

```
from Polynomial import Polynomial

a = Polynomial([9.0, 0.0, 2.3])
b = Polynomial([-2.0, 4.5, 0.0, 2.1])

print(a, "plus", b, "is", a+b)
print(a, "times", b, "is", a*b)
print(a, "times", 3, "is", a*3)
print(a, "minus", b, "is", a-b)

c = b.derivative()

print("Derivative of", b, "is", c)

slope = c(3)
print("Value of the derivative at 3 is", slope)
```

Run the test code:

```
python3 test_polynomial.py
```




APPENDIX A

Answers to Exercises



INDEX

class in python, [2](#)