Federal Urdu University Islamabad

Week 7

Intro to Classes in Python

The most simple way to understand classes is to treat them as custom/user-defined datatype. Just as there are default datatypes such as `int`, `float`, `str` etc. In this lecture, a custom datatype named as `Fraction` will be created in python as there is no built-in datatype as `Fraction` in `python`.

Datatype Logic

We are going to create class that represents `Fraction` in python. There is no built-in datatype in python for such purpose. The class will be responsible for

> - Storing two numbers (one representing numerator, other presesting denominator).
> - Performing standard operations like addition, subtraction, multiplication, division, power on one of more instances of `Fraction` class.

Class Architecture

A `Class` has some special functions and some general functions.

> - By Special functions, I refer to functions that are special to python.
> - By General functions, I refer to functions that are meant to define the specific behaviors or operations of the class, created by the programmer to implement custom logic.

Special Functions

The Constructor

The most important function in a class is it's constructor (i.e. the function that automatically gets called when an instance of a class is created). In python the constructor function is always named `__init__`. Python sees `__init__` function and automatically calls it ones an instance of that class if created.

```
class Fraction:
    def __init__():
```

The set of python statements defined in `__init__` function are executed when an instance of class is created. Constructor may or may not be useful depending on the nature of class. Some

> - Classes that must be passed in with some data when an instance of that class is created. Forexample `int`, `float`, `string`, `Fraction` etc. An instance of `int` class cannot be created without a passing number. Similarly, a `Fraction` cannot be defined without a number acting as numerator and another number acting as denominator.
> - Other class do not need data when their instance is being created. Forexample an instance of a `list` class. It is possible to define a `list` passing no data to it (an empty `list`).

In later type of classes, one may not define a constructor. In case of `Fraction`, a constructor must be defined to handle numerator and denominator.

```

```

```
class Fraction:
    def __init__(self, numerator, denominator):
```

In the constructor of `Fraction` class, there are 3 arguments `self`, `numerator` and `denominator`. `self` is out of scope of this lecture. Just know that `self` is meant for python internal use to handle a class. Python will automatically pass self to the `__init__` function and we can treat the `__init__` function as if the 1st argument is `numerator` and not `self` (def __init__(numerator, denominator)). All of the class functions will have `self` as their 1st arguments but we don't need to worry about that. Python automatically passes it and we pretend if it is not even their. Coming to the next arguments i.e. `numerator` and `denominator`. As the name suggests, `numerator` is supposed to receive the number for numerator in Fraction and `denominator` is supposed to receive number for denominator of Fraction. After that we start to define function body. The 1st thing to do is to check if `numerator` and `denominator` passed both are integers since python function arguments have no strict datatype. This step is very important to make sure we don't end up declaring instance of that class with wrong data in it. ForExample, a boolean value in the numerator and a string in denominator.

```
class Fraction:
    def __init__(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
```

This checkpoint makes sure, a class is always declared with valid data.

```
class Fraction:
    def __init__(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

        print(numerator, denominator)


Fraction(2, 4)
```

The above set of code will output "`2 4`" to standard output.

```
class Fraction:
    def __init__(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def test_output(self):
        print(numerator, denominator)
```

```
Fraction(2, 4).test_output()
```

`test_output` is a class Function/Member Function of Fraction class. But the above set of code causes the following error

```
Traceback (most recent call last):
  File "/home/muhammad/IntroToPython/week7/test.py", line 12, in <module>
    Fraction(2, 4).test_output()
  File "/home/muhammad/IntroToPython/week7/test.py", line 10, in test_output
    print(numerator, denominator)
          ^^^^^^^^^
NameError: name 'numerator' is not defined.
```

- Can you think why we were able to print "`2 4`" in the `__init__` function and not in the `test_output` function?

The reason is the scope of `numerator` and `denominator` variables. Their scope is limited to the `__init__` function. They simply do not exist in the `test_output` function. Therefore, we somehow need `numerator` and `denominator` to exist in the `__init__` function. One way may be to take `numerator` and `denominator` as arguments in the `test_output` function. But that is a completely wrong thing to do since this function is supposed to be called on the instance/object of the class and one can pass different data from what we already received in `__init__` function. This technique may be applied to the functions which are called from inside of the class (I will show a function like that in a bit). For now we change the scope of `numerator` and `denominator` from function scope to class scope.

```
class Fraction:
    def __init__(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

        self.numerator   = numerator
        self.denominator = denominator
```

Note: It is absolutely not compulsory to name the other variables same as `numerator` and `denominator`. one may write.

```
self.n = numerator
self.d = denominator
```

The syntax `self.numerator` looks a bit absurd since `"."` is usually used to call class functions. The internal working of class is out of the scope of this lecture but you are encouraged to read it from web. Now one can `self.numerator` and `self.denominator` anywhere, where there is `self` being passed.

```
class Fraction:
    def __init__(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
```

```
integer.")

        self.numerator   = numerator
        self.denominator = denominator

    def test_output(self):
        print(self.numerator, self.denominator)

Fraction(2, 4).test_output()
```

Now we get "`2 4`" in the output without any issues. Even though the above code is valid there are two non-python issues, I want to clear out:

> - `__init__` function is getting too crowded.
> - Too many variables (i.e. `self.numerator`, `self.denominator`) It is a common practice to keep the constructor function clean and not to include too many functional statments in it. Therefore, we can shift the checkpoint code to another function and call that function inside of `__init__`.

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)
        self.numerator   = numerator
        self.denominator = denominator

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

Fraction(2, 4)
```

Now we cleaned out the `__init__` function. It is easy for anyone else to read. Also notice we passed the numbers `numerator` and `denominator` to `validate_data` function from inside the class. This is completely fine to do since the same data received in `__init__` is being passed to `validate_data` function. Defining two separate variables `self.numerator` and `self.denominator` for two values is not only computationaly costly but also programmatically, it is difficult to handle. In such cases a data structure like a `list` is well suited. Therefore:

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
```

```
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

Fraction(2, 4)
```

Now we know that a list named as `self.fraction` is holding two values, numerator at index 0 and denominator as index 1. Anywhere in the classes `self.fraction[0]` can be used for numerator and `self.fraction[1]` can be used to for denominator. That's it for the constructor.

General Functions

Till now we defined a class named as `Fraction` accepting two numbers as arguments, saving them in `self.fraction` list.

```
Fraction(32, 24)
```

Above code stores the fraction $\frac{32}{24}$ as a list in python. Now let's define a general function which will simplify the fraction. Taking it from $\frac{32}{24}$ to $\frac{4}{3}$.

Simplify Function

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def simplify(self, update = False):
        """
        This function simplifies the fraction.
        return:
          returns simplified Fraction if update is False
          updates self.fraction if update is True
        """
        temp_fraction = self.fraction
        # Apply all of the simplification algorithm to temp_fraction
        #
        #
        #

        if update:
            self.fraction = temp_fraction
            return
        return temp_fraction
```

```
Fraction(32, 24)
```

`simplify` is a class function and can be called on instance of `Fraction` class. `simplify` also takes an argument `update` which must be a boolean value. If `update` is `True`, the function updates the original fraction i.e. the self.fraction variable which contains the main data for this instance. Otherwise, `simplify` returns the simplified fraction. I am leaving the algorithm implementation up to you. Please enjoy.

> Enjoyable Task: Transform the simplify method into a generator function that yields intermediate simplified forms of the fraction, one step at a time, by dividing both the numerator and denominator by their next common factor. Each call to the generator yields a further simplified version until the fraction is fully reduced.
>
> - 1st call $\rightarrow \frac{16}{12} $
> - 2nd call $\rightarrow \frac{8}{6} $
> - 3rd call $\rightarrow \frac{4}{3} $

+,-,*,/

Operations like addition, subtraction, multiplication, division cannot be applied on a single instance of Fraction class. Atleast, two instances are required.

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def add(self, other_fraction):
        pass
```

The `add` function needs another fraction to add it to the current fraction. I am taking the `other_fraction` as argument of `add` function. The 1st step is data validity. That is to make sure that we are receiving a `Fraction` object to add and not an `int`, `float` etc.

```
def add(self, other_fraction):
        if isinstance(other_fraction, Fraction)
            pass
        else:
            raise Exception(f"Only Fraction can be added to Fraction. Not
{type(other_fraction)}")
```

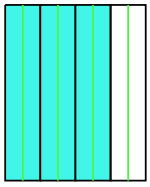
We now have two `fraction` lists one from `self` and other from `other_fraction`.

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def add(self, other_fraction):
        if isinstance(other_fraction, Fraction):
            pass
        else:
            raise Exception(f"Only Fraction can be added to Fraction. Not
{type(other_fraction)}")

        print(self.fraction, other_fraction.fraction)

Fraction(3, 4).add(Fraction(7, 8))
```

Execute this code and think about it for some time.

Math

Adding two fraction is not as simple as adding two integers. A number of Mathematical steps must be performed. Before we code let's see those steps $$ \frac{3}{4} + \frac{7}{8} $$ Take the denominators i.e. 4 and 8. And find their LCM (Least Common Multiple). Least multiple of 4 is 4 and 8 is 8. 8 is the number which is both multiple of 4 and 8 and is common among 4 and 8. That's the LCM of 4 and 8. $$ \frac{? + ?}{8} $$ Now we need to `Transform` numerators i.e. 3 from fraction $\frac{3}{4}$ to new Fraction i.e. $\frac{}{8}$. Visually:



6 parts out of 8 in a rectangle cover the same area as 3 parts out of 4 in the same rectangle. Therefore the new numerator is 6. Numerator from $\frac{7}{8}$ remains the same since the prior and the further denominator is the same. Therefore, $$\frac{6+7}{8} = \frac{13}{8}$$ $\frac{13}{8}$ means 13 parts from a rectangle (any entity) divided into 8 parts. 4 Mathematical Steps are performed

> - Determining the Least Common Multiple.
> - Determining Numerator from 1st Fraction.
> - Determining Numerator from 2nd Fraction.
> - Adding up numerators.

Coming Back to Code

Among 4 Math Operations, 3 can be abstracted out since they are going to be same in Addition and Subtraction.

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def lcm(self, dem1, dem2):
        """
        Function accepts two numbers to find their Least Common Multiple
        """
        # Write code to find the least common multiple of two numbers
        # return the result

    def new_numerator(self, old_fraction, new_denominator):
        """
        Function accepts the old_numerator and new_denominator.
        Calculates the new numerator
        """
        # Write code to Transform numerator from one fraction to new fraction
        # return the new numerator

    def add(self, other_fraction):
        if isinstance(other_fraction, Fraction):
            pass
        else:
            raise Exception(f"Only Fraction can be added to Fraction. Not
{type(other_fraction)}")

        lcm = self.lcm(self.fraction[1], other_fraction.fraction[1])
        return Fraction(self.new_numerator(self.fraction, lcm) +
self.new_numerator(other_fraction.fraction, lcm), lcm)

Fraction(3, 4).add(Fraction(7, 8))
```

That's it's for Addition. Code for substraction is exactly same except there's a minus instead of plus.

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.validate_data(numerator, denominator)

        self.fraction = [numerator, denominator]

    def validate_data(self, numerator, denominator):
        if denominator == 0:
            raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
```

```
integer.")
        for num in [numerator, denominator]:
            if isinstance(num, int):
                pass
            else:
                raise Exception(f"{num} is of type {type(num)}. Whereas it must be an
integer.")

    def lcm(self, dem1, dem2):
        """
        Function accepts two numbers to find their Least Common Multiple
        """
        # Write code to find the least common multiple of two numbers
        # return the result

    def new_numerator(self, old_fraction, new_denominator):
        """
        Function accepts the old_numerator and new_denominator.
        Calculates the new numerator
        """
        # Write code to Transform numerator from one fraction to new fraction
        # return the new numerator

    def add(self, other_fraction):
        if isinstance(other_fraction, Fraction):
            pass
        else:
            raise Exception(f"Only Fraction can be added to Fraction. Not
{type(other_fraction)}")

        lcm = self.lcm(self.fraction[1], other_fraction.fraction[1])
        return Fraction(self.new_numerator(self.fraction, lcm) +
self.new_numerator(other_fraction.fraction, lcm), lcm)

    def sub(self, other_fraction):
        if isinstance(other_fraction, Fraction):
            pass
        else:
            raise Exception(f"Only Fraction can be subtracted to Fraction. Not
{type(other_fraction)}")

        lcm = self.lcm(self.fraction[1], other_fraction.fraction[1])
        return Fraction(self.new_numerator(self.fraction, lcm) -
self.new_numerator(other_fraction.fraction, lcm), lcm)

Fraction(3, 4).add(Fraction(7, 8))
```

Multiplication, Division, Power is relatively simple. I expect you to do them by yourself.

Method OverRiding

Instead of doing

```
Fraction(3, 4).add(Fraction(7, 9))
```

to add. Syntax like

```
Fraction(3, 4) + Fraction(7, 9)
```

seems better. This can be achieved by overriding the special methods defined for addition, subtraction, multiplication, and division. You only need to change the function names to the corresponding magic methods like `__add__`, `__sub__`, `__mul__`, and `__truediv__`.