OOP in Python - Introduction

Here is a C++ class implementing rational numbers:

```
#include <iostream>
using namespace std;
class Rat {
private:
   int n, d;
public:
   // Constructors
   Rat() : n(0), d(1) {}
   Rat(int i, int j) : n(i), d(j) {}
   Rat(int i) : n(i), d(1) {}

   // Accessor functions
   int getN() { return n; }
   int getD() { return d; }
   void setN(int i) { n = i; }
   void setD(int i) { d = i; }

   // Arithmetic operators
   Rat operator+(const Rat& r) {
      return Rat(n * r.d + d * r.n, d * r.d);
   }
   Rat operator-(const Rat& r) {
      return Rat(n * r.d - d * r.n, d * r.d);
   }
   Rat operator*(const Rat& r) {
      return Rat(n * r.n, d * r.d);
   }

   Rat operator/(const Rat& r) {
        return Rat(n * r.d, d * r.n);

 // Overloading operators for Rat and int combinations

friend Rat operator+(int lhs, const Rat& rhs) {
        return Rat(lhs) + rhs;
    }
    friend Rat operator-(int lhs, const Rat& rhs) {
        return Rat(lhs) - rhs;
    }
    friend Rat operator*(int lhs, const Rat& rhs) {
        return Rat(lhs) * rhs;
    }
    friend Rat operator/(int lhs, const Rat& rhs) {
        return Rat(lhs) / rhs;
    }
    // Other overloaded operators...
};
 }
    // Overloaded I/O operators
    friend ostream& operator<<(ostream& os, const Rat& r);
    friend istream& operator>>(istream& is, Rat& r);
};

// Implement the I/O operators if needed
ostream& operator<<(ostream& os, const Rat& r) {
    os << r.n << "/" << r.d;
    return os;
}

istream& operator>>(istream& is, Rat& r) {
    is >> r.n >> r.d;
    return is;
}
int main() {
    // Test the Rat class
    Rat r1(3, 4), r2(2, 5);
    Rat sum = r1 + r2;
    Rat diff = r1 - r2;
    Rat prod = r1 * r2;
    Rat quot = r1 / r2;

    cout << "Sum: " << sum << endl;
    cout << "Difference: " << diff << endl;
    cout << "Product: " << prod << endl;
    cout << "Quotient: " << quot << endl;

    return 0;
}
```

Notice:

Data hiding and encapsulation

Public and private members

Multiple constructors \operator overloading

Here is the python version:

To create an equivalent Python class for the C++ Rat class, we need to translate the functionality into Python's class structure. This involves using Python's special methods for operator overloading and handling the initialization and representation of the class.

```
class Rat:
    def __init__(self, n=0, d=1):
        self.n = n
        self.d = d

    # Accessor and mutator methods
    @property
    def n(self):
        return self._n

    @n.setter
    def n(self, value):
        self._n = value

    @property
    def d(self):
        return self._d

    @d.setter
    def d(self, value):
        if value == 0:
            raise ValueError("Denominator cannot be zero")
        self._d = value




# Arithmetic operators
```

```
    def __add__(self, other):
        if isinstance(other, Rat):
            return Rat(self.n * other.d + self.d * other.n, self.d * other.d)
        elif isinstance(other, int):
            return self + Rat(other)

    def __radd__(self, other):
        return self + other

    def __sub__(self, other):
        if isinstance(other, Rat):
            return Rat(self.n * other.d - self.d * other.n, self.d * other.d)
        elif isinstance(other, int):
            return self - Rat(other)

    def __rsub__(self, other):
        return -self + other

    def __mul__(self, other):
        if isinstance(other, Rat):
            return Rat(self.n * other.n, self.d * other.d)
        elif isinstance(other, int):
            return self * Rat(other)

    def __rmul__(self, other):
        return self * other




  def __truediv__(self, other):
        if isinstance(other, Rat):
            return Rat(self.n * other.d, self.d * other.n)
        elif isinstance(other, int):
            return self / Rat(other)

    def __rtruediv__(self, other):
        return Rat(other) / self

    # String representation
    def __str__(self):
        return f"{self.n}/{self.d}"
```

Using it:

```
r1 = Rat(3, 4)
r2 = Rat(2, 5)
sum_ = r1 + r2
diff = r1 - r2
prod = r1 * r2
quot = r1 / r2

print("Sum:", sum_)
print("Difference:", diff)
print("Product:", prod)
print("Quotient:", quot)
```

This Python Rat class supports:

Arithmetic operations (+, -, *, /) between Rat objects and between a Rat object and an integer.

Overridden __str__ method for string representation, similar to the << operator overload in C++.

The __radd__, __rsub__, __rmul__, and __rtruediv__ methods are the "right" versions of the arithmetic methods and are used when the Rat instance is on the right side of the operator. Python's special method names (__methodname__) are used for operator overloading, providing a Pythonic way to implement the behavior similar to C++ operator overloading.

The __add__ method overloads the + operator, and __str__ provides a string representation of the object (similar to overloading << in C++). The from_input static method is an equivalent for the >> operator overload, facilitating input from the user.

Constructors

In Python, the concept of "multiple constructors" as found in languages like C++ or Java isn't directly supported because Python only allows one **__init__** method per class. However, there are several ways to achieve functionality similar to having multiple constructors in Python:

1. The __init__ method in Python is equivalent to the constructor in C++, and we've combined all three C++ constructors into one Python constructor with default arguments.

Default Arguments in __init__ Method: Using default arguments in the **__init__** method is the most straightforward way to simulate multiple constructors. This approach allows a single constructor to handle different scenarios based on the arguments passed to it. This illustrated in the Rat class above.

```
def __init__(self, n=0, d=1):
    self.n = n
    self.d = d
```

“self” is “this”, but must be explicitly passed. Any variable name is fine (not necessarily self).

2, Class methods also allow for multiple constructors.

```
@classmethod
def from_string(cls, string):
    parts = string.split('/')
    if len(parts) != 2:
        raise ValueError("String must be in the format 'numerator/denominator'")
    n, d = int(parts[0]), int(parts[1])
    return cls(n, d)
```

When you use **cls** in a class method in Python, it automatically refers to the class on which the method is called, not necessarily the class where the method is defined. This distinction is key, especially when dealing with inheritance.

In the context of the **Rat** class:

- When you define a class method in the **Rat** class and use **cls** within that method, **cls** refers to **Rat**.
- If you later create a subclass of **Rat**, say **SubRat**, and call that class method on **SubRat**, then within that method, **cls** will refer to **SubRat**, not **Rat**.

This behavior is what makes class methods particularly powerful for inheritance, as it allows the method to correctly instantiate objects of the subclass, rather than always creating instances of the base class.

Data hiding

Data hiding in C++ is achieved by bundling the data and the methods that operate on the data within a class and restricting access to them by using access specifiers. Private members of a class are hidden from outside the class.

Python does not have explicit private members like C++. However, by convention, a member prefixed with an underscore is treated as non-public.

```
class MyClass:
    def __init__(self):
        self._hidden_data = 0

    def set_data(self, value):
        self._hidden_data = value

    def get_data(self):
        return self._hidden_data
```

This convention serves as a hint to other programmers that a variable or method starting with an underscore is intended for internal use within the class or module, and it should not be accessed directly from outside.

You're correct in observing that in Python, prefixing a class member with an underscore is more about convention than enforcement. This convention serves as a hint to other programmers that a variable or method starting with an underscore is intended for internal use within the class or module, and it should not be accessed directly from outside.

Here's how this works in practice:

1. **Convention Over Enforcement:** Python's philosophy emphasizes the importance of readable and understandable code over strict access control. By convention, a single underscore (**_**) indicates that a member is intended for internal use. This is a "gentleman's agreement" among Python developers to avoid accessing these members directly from outside the class.
2. **Name Mangling for Stronger Encapsulation:** For a stronger approach, Python uses name mangling for any member with double underscores at the beginning (but not at the end). This changes the way the member name is stored internally. For example, **__myVar** in a class **MyClass** is mangled to **_MyClass__myVar**, making it harder (but not impossible) to access from outside the class.

3. **No Absolute Privacy:** Unlike languages like Java or C++, Python does not enforce strict access control. Even with name mangling, it's still possible to access and modify these members from outside the class. Python's philosophy is based on the idea of "we are all consenting adults here," meaning that it trusts programmers to respect these conventions rather than enforcing them through the language.

This approach aligns with Python's overall design principles, which favor simplicity and readability. It relies on the discipline of the developer to adhere to these conventions rather than imposing strict access control. While this might seem like a drawback in terms of encapsulation, it provides greater flexibility.

Properties for access control

In Python, properties can be used to control access to instance variables, providing a level of encapsulation. A property in Python is a special kind of attribute whose access is managed by getter, setter, and deleter methods. This mechanism allows you to add logic around getting and setting a value, which can be used to implement validations, calculations, or other controls.

Here's how properties can be used for encapsulation:

1. Getter Method: This method is used to access the value of an attribute without exposing the attribute itself. It allows for computation or processing of data before returning it.

2. Setter Method: This method is invoked when the value of an attribute is modified. It can be used to validate or modify the data before it's actually changed.

3. Deleter Method: This method is called when the attribute is deleted. It allows for cleanup actions or other processing.

For example:

Consider a class `Person` with a private attribute `_age`:

```
class Person:
    def __init__(self, age):
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value < 0:
            raise ValueError("Age cannot be negative")
        self._age = value

    @age.deleter
    def age(self):
        del self._age
```

In this example:

- The `@property` decorator defines a getter for `age`. When you access `p.age`, it returns the value of `_age`.
- The `@age.setter` decorator defines a setter for `age`. When you set `p.age = value`, it validates the value before setting `_age`.
- The `@age.deleter` decorator defines a deleter for `age`. When you delete `age` with `del p.age`, it performs the necessary cleanup.

Encapsulation through Properties

Properties provide a way to control access to an attribute:

Data Validation: The setter method can include validation logic to ensure that only valid data is assigned to the attribute.
Data Hiding: By using properties, the direct access to underlying data (like `_age`) is restricted, enforcing the use of getter and setter methods.
Additional Processing: Getter and setter methods can include additional logic like logging, error handling, or data transformation.

Properties offer a Pythonic way to achieve a level of encapsulation similar to private and public access modifiers in other languages. While they don't completely prevent access to the underlying data (as Python's philosophy allows for more flexibility), they provide a structured way to manage how data is accessed and modified within a class.

Instance variables and class variables

Instance variables and class variables serve different purposes and have different behaviors. Instance variables are like data members in a C++ class, Python class variables are like C++ static data members.

Instance variables are variables that are specific to each instance of a class. Every object (instance) of the class has its own copy of these variables. They are usually defined and initialized within the `__init__` method.

In the `Rat` class, `n` (numerator) and `d` (denominator) are instance variables:

```
class Rat:

  def __init__(self, n=0, d=1):
    self.n = n  # Instance variable
    self.d = d  # Instance variable
```

Each `Rat` object will have its own `n` and `d` values, independent of other `Rat` objects.

Class Variables

Class variables, on the other hand, are variables that are shared across all instances of the class. They are not specific to any one instance. Instead, if you change the value of a class variable, it affects all instances of the class.

For example, let's add a class variable to the `Rat` class:

```
class Rat:
    count = 0  # Class variable
    def __init__(self, n=0, d=1):
        self.n = n
        self.d = d
        Rat.count += 1
```

So,

```
r1 = Rat(1, 2)
r2 = Rat(3, 4)
print(Rat.count)  # Output will be 2, shared across all instances. Notice the syntax.
```

Here, `count` is a class variable that keeps track of how many `Rat` instances have been created.

Static Methods

A static method in Python is a method that belongs to a class but does not require an instance of the class to be used. It's not bound to an instance or class variable. Static methods are defined with the `@staticmethod` decorator.

Static methods are used when some functionality is related to the class but does not need to access any instance-specific data or class variables.

Class variables and static methods in Python are related in the sense that they both exist at the class level, rather than the instance level, but they serve different purposes:

Class Variables

1. Shared Across Instances: Class variables are shared among all instances of the class. A change to a class variable affects all instances of the class.

2. Defined in Class Body: They are defined within the class body, outside of any instance methods (including `__init__`).

3. Usage: Class variables are useful for storing data that is common to all instances of a class. This might include constants, default values, or counters that keep track of class-wide information.

Static Methods

1. Not Bound to Instances or Class: Static methods are methods that belong to a class but do not require an instance of the class to be used. They don't have access to the instance (`self`) or the class (`cls`) unless explicitly passed.

2. Defined with @staticmethod Decorator: They are marked with the `@staticmethod` decorator to indicate that they are static methods.

3. Usage: Static methods are used for functionality that is related to a class but is self-contained and does not need to access or modify class or instance state.

<u>Relationship</u>

<u>Class Level</u>: Both class variables and static methods are defined at the class level.

<u>Independence</u>: Static methods can be used independently of class or instance state. They can be called without creating an instance of the class. While they can access class variables if necessary, they do not rely on instance variables.

<u>Utility Functions</u>: Static methods often act as utility functions related to the class's purpose but don't need to interact with class or instance variables.

For example:

```
class Rat:
    count = 0  # Class variable

    def __init__(self, n=0, d=1):
        self.n = n
        self.d = d
        Rat.count += 1

    @staticmethod
    def get_total_rats_created():
        return Rat.count  # Accessing class variable

# Usage
r1 = Rat(1, 2)
r2 = Rat(3, 4)
print(Rat.get_total_rats_created())  # Static method accessing class variable
```

In this example, the static method `get_total_rats_created` accesses the class variable `count` to provide class-level information, demonstrating how static methods can interact with class variables if needed. However, it's important to note that static methods are not inherently tied to class variables and can be used independently.

Let's add a static method to the `Rat` class:

```
class Rat:

    @staticmethod
    def info():
        return "This is a class for representing rational numbers."
```

You can call this method without creating an instance of `Rat`:

```
print(Rat.info())  # "This is a class for representing rational numbers."
```

To summarize, instance variables are for data unique to each instance, class variables are shared across all instances, and static methods provide functionality related to the class but are independent of class or instance variables.

Deleters

Deleter Method: This method is called when the attribute is deleted. It allows for cleanup actions or other processing.

We saw thie above:

```
    @age.deleter
    def age(self):
        del self._age
```

To define actions that happen upon deletion of a property, you use the deleter. This is a method with the same name as the property, decorated with `@property_name.deleter`.

```
class MyClass:
    def __init__(self, value):
        self._my_property = value

    @property
    def my_property(self):
        return self._my_property

    @my_property.setter
    def my_property(self, value):
        if value < 0:
            raise ValueError("Value cannot be negative")
        self._my_property = value

    @my_property.deleter
    def my_property(self):
        print("Property is being deleted")
        del self._my_property
```

Example:

```
del obj.my_property     # Deletes the property

# Output: Property is being deleted
```

Does a deleter change the definition of the class

Python does not change the definition of the class itself. Instead, it provides a specific mechanism for handling the deletion of a property within the class. The class definition remains the same, but the deleter adds functionality that is executed when the property is deleted.

1. **Class Definition:** When you define a class in Python, you typically include attributes (variables) and methods (functions). The class definition outlines the structure and behavior of objects that will be created from the class.
2. **Adding a Deleter:** When you add a deleter to a class, you are essentially defining how a specific property should be handled when it is deleted. This does not alter the overall structure of the class; rather, it extends the functionality of the property within the class.
3. **Deleter Functionality:** The deleter is a method that gets called when you delete a property. It can be used to perform cleanup actions, logging, or any other necessary operations before the property is removed. The syntax for a deleter uses the **@property_name.deleter** decorator:

In this example, when **del obj.my_property** is called, the deleter method is invoked, printing a message and deleting the attribute.

4. **Effect on Instances:** Deleting a property using its deleter affects instances of the class, not the class definition itself. When you delete a property from an instance, the property is removed from that specific instance. Other instances of the class are unaffected, and the class definition remains the same.

In summary, a deleter modifies the behavior of a property in terms of how it is deleted from instances of the class, but it does not change the underlying class definition. It's part of the broader concept of properties in Python, which are used to control access to data in an object-oriented manner.

Using a deleter in Python is a way to manage the deletion of a property in a controlled and safe manner. It's part of the property mechanism that allows you to define custom behavior for getting, setting, and deleting an attribute of a class. Here are several reasons why you might use a deleter:

1. **Cleanup Operations:** If the property being deleted is linked to external resources (like file handles, network connections, or memory allocations), the deleter can be used to ensure that these resources are properly released or closed.
2. **Preventing Memory Leaks:** In some cases, especially when dealing with large data structures or external resources, you may want to explicitly manage memory usage. A deleter can help ensure that memory is freed when a property is no longer needed.
3. **Maintaining State Consistency:** If your class maintains an internal state that depends on its properties, deleting a property might require recalculating or adjusting this state. A deleter can handle these adjustments to keep the object's state consistent.

4. **Enforcing Business Logic:** Sometimes, the deletion of a property might need to adhere to certain business rules or logic. For example, you might want to check if certain conditions are met before allowing a property to be deleted.

5. **Logging and Auditing:** If you need to keep track of when properties are deleted (for debugging, logging, or auditing purposes), a deleter can be used to log these deletions.

6. **Control and Validation:** Similar to getters and setters, deleters give you more control over how properties are managed. You can validate if a property should be deletable at a given point in time or under certain conditions.

While there are valid reasons to delete an attribute, it's essential to do so judiciously because:

- It can lead to inconsistent object states within the same class, potentially causing errors if other parts of the codebase expect the attribute to be present.
- It can make the code harder to understand and maintain, as the dynamic structure of objects is less predictable.

In many cases, it might be better to set the attribute to `None` or some other sentinel value to indicate that it's not currently valid or in use, rather than deleting it outright. This approach maintains the consistency of the object's structure while still conveying that the attribute is in a kind of "not set" or "not applicable" state.

Another example: a safearray class

```
#include<iostream>
#include <cstdlib>
#include<cassert> using namespace std; class SA{
private:
int low, high; int* p;
public:

// default constructor
// allows for writing things like SA a; SA(){low=0; high=-1;p=NULL;}

// 2 parameter constructor lets us write
// SA x(10,20);

SA(int l, int h){
if((h-l+1)<=0)
{cout<< "constructor error in bounds definition"<<endl; exit(1);}
low=l; high=h;
p=new int[h-l+1];
}


// single parameter constructor lets us
// create a SA almost like a "standard" one by writing
// SA x(10); and getting an array x indexed from 0 to 9

SA(int i){low=0; high=i-1; p=new int[i];
}

// copy constructor for pass by value and
// initialization

SA(const SA & s){
int size=s.high-s.low+1; p=new int[size];
for(int i=0; i<size; i++)
p[i]=s.p[i];
```

```
low=s.low; high=s.high;
}

// destructor

~SA(){
                    delete [] p;
}

//overloaded []
lets us write
//SA x(10,20);
x[15]= 100;

int&
operator[](int
i){
if(i<low ||
i>high)
{cout<< "index
"<<i<<" out of
range"<<endl;
exit(1);}
return p[i-low];
}

// overloaded
assignment lets
us assign
// one SA to
another

SA &
operator=(const
SA & s){
if(this==&s)ret
urn *this;
delete [] p;
int size=s.high-
s.low+1;
p=new
int[size];
for(int i=0;
i<size; i++)
p[i]=s.p[i];
low=s.low;
high=s.high;
```

```
return *this;
}

// overloads << so we can directly print SAs friend ostream&
operator<<(ostream& os, SA s);
};

delete [] p;
```

```
        //overloaded [] lets us write
        //SA x(10,20); x[15]= 100;

        int& operator[](int i){
                if(i<low || i>high)
                {cout<< "index "<<i<<" out of range"<<endl;
                exit(1);}
                return p[i-low];
        }

        // overloaded assignment lets us assign
        // one SA to another

        SA & operator=(const SA & s){
        if(this==&s)return *this;
        delete [] p;
        int size=s.high-s.low+1;
        p=new int[size];
        for(int i=0; i<size; i++)
                p[i]=s.p[i];
        low=s.low;
        high=s.high; return
        *this;
        }

        // overloads << so we can directly print SAs friend
        ostream& operator<<(ostream& os, SA s);

};
```

The Python version:

To translate the C++ `SA` (Simple Array) class into Python, we need to adapt the code to fit Python's object-oriented programming style. Python's built-in list type already provides much of the functionality of a dynamic array, so the `SA` class can be simplified significantly. Here's a Python version of the `SA` class:

```
class SA:
    def __init__(self, *args):
        if len(args) == 0:
            self.low = 0
            self.high = -1
            self.data = []
        elif len(args) == 1:
            self.low = 0
            self.high = args[0] - 1
            self.data = [0] * args[0]
        elif len(args) == 2:
            l, h = args
            if h - l + 1 <= 0:
                raise ValueError("constructor error
 in bounds definition")
            self.low = l
            self.high = h
            self.data = [0] * (h - l + 1)
        else:
            raise ValueError("Invalid number of
 arguments")

    def __copy__(self):
        copy_obj = SA(self.high - self.low + 1)
        copy_obj.data = self.data.copy()
        return copy_obj

    def __getitem__(self, index):
        if index < self.low or index > self.high:
            raise IndexError(f"index {index} out
 of range")
        return self.data[index - self.low]

    def __setitem__(self, index, value):
        if index < self.low or index > self.high:
            raise IndexError(f"index {index} out
 of range")
        self.data[index - self.low] = value
    def __str__(self):
        return str(self.data)
```

Using it:

```
sa = SA(10, 20)
sa[15] = 100
print(sa)  # [0, 0, 0, 0, 0, 100, 0, 0, 0, 0, 0]

sa2 = SA(10)

print(sa2)  # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Key points in the translation:

1. Constructors: Python doesn't support multiple constructors, so the `__init__` method uses variable arguments (`*args`) to handle different construction scenarios.

2. Array Representation: Python lists are used to represent the array. Unlike C++, Python handles memory management for lists automatically.

3. Indexing: The `__getitem__` and `__setitem__` methods are used to implement array indexing (`[]` operator).

4. Copy Constructor: Python doesn't have a copy constructor in the same way as C++. Instead, you can define a method to create a copy (using `__copy__` here).

5. Destructor: Python's garbage collector handles memory management, so an explicit destructor like C++'s `~SA()` is not needed.

6. Printing: The `__str__` method is used for string representation of the object, which allows printing the object directly.

This Python class maintains the core functionalities of the C++ `SA` class, adapting them to Python's style and capabilities.