


```
>>> x, y = [1, 2]
>>> x
1
>>> y
2
```

We can represent rational numbers using lists:

```
def rational(n,d):
    return [n,d]

def numer(x):
    return x[0]

def denom(y):
    return x[1]
```

We can also define a function to print rationals:

```
def print_rational(x):
    print(numer(x),"/",denom(x))
```

Let's try:

```
>>> print_rational(add_rational(rational(2,5), rational(6,10)))
50 / 50
```

We can simplify this by adding a function:

```
from math import gcd


def rational(n,d):
    g = gcd(n,d)
    return [n//g,d//g]
```

Because we've already defined *all* of the other functions, we don't need to worry about the other functions and add these two lines to add the ability to simplify fractions.

Abstraction Barriers

When writing longer programs, it is important to isolate the parts of the program that represent data. This is so that if you want to change how you show your data later on, it doesn't break the program as a whole, or require you to rewrite the entire program.

By doing so, we make our program **modular**.

 9fd87126.png

The lines above represent the abstraction barriers that we must keep in order to isolate our program properly.

What's wrong with this data?

Take a look at the below code:

```
add_rational([1,2],[1,4])

def divide_rational(x,y):
    return [ x[0] * y[1], y[0] * x[1] ]
```

The above code **violates** abstraction barriers, and doing so only makes someone else do more work down the line.



What is Data?

- We need to guarantee the constructor and selector functions work together to specify the right behavior.
- Behavior condition: If we construct rational number `x` from numerator `n` and denominator `d`, then `numer(x)/denom(x)` must equal `n/d`.
- Data abstraction uses selectors and constructors to define behavior
- If behavior conditions are met, then the representation is valid

You can recognize an abstract data representation by its behavior

You can now change the definition of `rational` to return a function instead of a list, or whatever definition of `rational` you can think of. And as long as you change `numer` and `denom` too, the rest of your program **doesn't need to be changed**.

Dictionaries

Dictionaries are unordered collections of key-value pairs Dictionary keys do have two restrictions:

- A key of a dictionary cannot be a list or a dictionary (or any mutable type)
- Two keys cannot be equal; There can be at most one value for a given key

```
>>> {1:'I', 5:'V', 10: 'X'}
{1:'I', 5:'V', 10: 'X'}
>>> d = {1:'I', 5:'V', 10: 'X'}
>>> d[5]
'V'
```

This first restriction is tied to Python's underlying implementation of dictionaries, while the second restriction is part of the dictionary abstraction.

You can look up 'V' using its key, although it doesn't work the other way around.

You could build the reverse of the dictionary with:

```
>>> {v: k for k, v in d.items()}
{'I': 1, 'V': 5, 'X': 10}
```

If you want to associate multiple values with a key, store them all in a sequence value.

You can use the `.get()` function to specify a value to return if the index is not found in the dictionary, instead of returning a `KeyError`.

```
>>> d.get(1,0)
'I'
>>> d.get(3,0)
0
```

Every frame is basically a dictionary in Python: under the hood, it uses the same implementation, because each frame has a unique name and its values.