Lecture 19: Object Oriented Programming
Data Abstraction

Comp 102

Forman Christian University

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**
- **OOP** but using pure C:
  - Linux Kernel
  - GTK+
  - Doom

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**
- **OOP** but using pure C:
  - Linux Kernel
  - GTK+
  - Doom
- Set of programming practices *(language, syntax independent)*

Recap

Abstraction

Abstraction

- Hiding the details, showing only the necessary

Abstraction

- Hiding the details, showing only the necessary
- **Expressions**:

```
pi = 355/113
area = pi * (R**2)
circ = 2 * pi * R
```

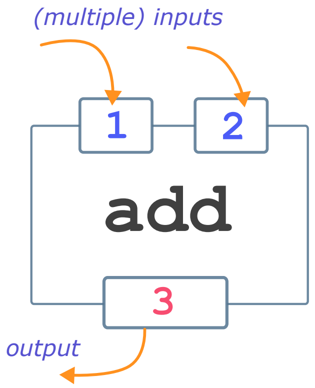
Abstraction

- Hiding the details, showing only the necessary
- **Expressions**:

```
pi = 355/113
area = pi * (R**2)
circ = 2 * pi * R
```

- **Functions**:

```
def add(x,y):
    return x+y
add(1,2)
add(4,5)
```



Compound Data

- Many things in reality are made of compund data *(non-scalar)*
- Try to Identify components of each:
  - `Vector` →
  - `Student` →
  - `Rational` →
  - `Book` →

Compound Data

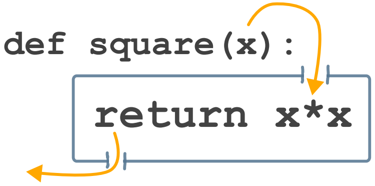
- Many things in reality are made of compund data *(non-scalar)*
- Try to Identify components of each:
  - `Vector` → `x,y,z`
  - `Student` → `name, rollno, gpa`
  - `Rational` → `num, denom`
  - `Book` → `title, author, price`

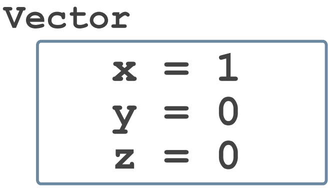
Data Abstraction

- We want to do the same with **data** that we did with **code**
- Put individual components *(data)* in a **black box**

Data Abstraction

- We want to do the same with **data** that we did with **code**
- Put individual components *(data)* in a **black box**

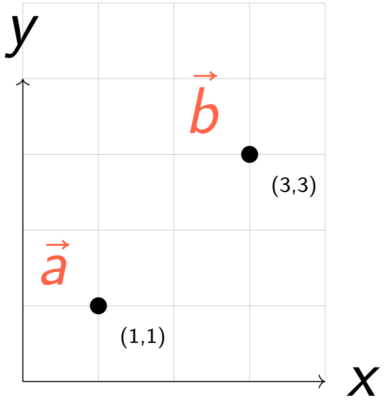




Code Abstraction

Data Abstraction

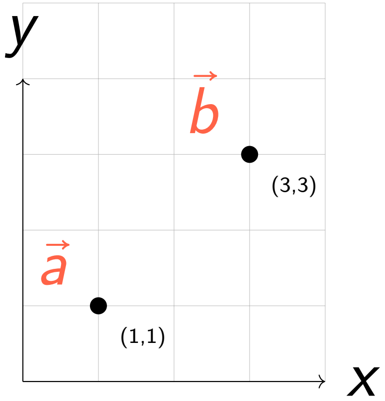
Example:



From high school math:

- $\vec{c} = \vec{a} + \vec{b}$
- $\vec{d} = \vec{a} - \vec{b}$

Example:



From high school math:

- $\vec{c} = \vec{a} + \vec{b}$
- $\vec{d} = \vec{a} - \vec{b}$

We're treating $\vec{a}$ and $\vec{b}$ as black boxes

Not concerned with their internal details

Big Idea

Data Abstraction:

Seperating how data is **represented** from how it is **manipulated**

Example: Rational Numbers

- Have the form: $\frac{numerator}{denominator}$

Example: Rational Numbers

- Have the form: $\frac{numerator}{denominator}$
- A rational 1/3 **can't be represented exactly** on a computer:

```
>>> 1/3
0.3333333333333333
>>> 1/3 == 0.33333333333333333300000
True
```

Rational Numbers

- But can represent exactly as compound data *(non-scalar)*
- Imagine the following functions *(again, black boxes)*:

```
rational(n,d) # constructor, returns rational
numer(x) # accessor function
denom(x) # accessor function
```

Rational Numbers

So far we have three functions:

```
rational(n,d)        numer(x)        denom(x)
```

Rational Numbers

So far we have three functions:

```
rational(n,d)     numer(x)     denom(x)
```

- No idea how these functions are implemented
- No idea how a rational number stores numerator and denominator

Rational Numbers

So far we have three functions:

```
rational(n,d)      numer(x)      denom(x)
```

- No idea how these functions are implemented
- No idea how a rational number stores numerator and denominator
- But we can still use these as black boxes to build more complex functions

Rational Numbers

Adding two rational numbers:

$$\frac{1}{2} + \frac{2}{3} = \frac{3 \times 1 + 2 \times 2}{2 \times 3} = \frac{7}{6}$$

Rational Numbers

Adding two rational numbers:

$$\frac{1}{2} + \frac{2}{3} = \frac{3 \times 1 + 2 \times 2}{2 \times 3} = \frac{7}{6}$$

Complete the following function:

```
def add_rationals(x, y):
    ''' Adds two rational numbers x,y
    Returns: A rational number '''
```

Rational Numbers

Solution:

```
def add_rationals(x, y):
    ''' Adds two rational numbers x,y
    Returns: A rational number '''

    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx*dy)
```

Rational Numbers: You Try!

```
def mul_rationals(x, y):
    ''' Multiplies two rational numbers x,y
    Returns: A rational number '''

def print_rational(x):
    ''' Prints a rational number as n/d '''

def rationals_are_equal(x, y):
    ''' Checks if two rational numbers are
       equal '''
```

Rational Numbers

```python
def add_rationals(x, y):
    ...

def mul_rationals(x, y):
    return rational(numer(x)*numer(y), denom(x)*denom(y))

def print_rational(x):
    print(numer(x), '/', denom(x))

def rationals_are_equal(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Rational Numbers

Data Abstraction: separation of

1. how data is manipulated ✓
   - `rational`, `numer`, `denom`
   - `add_rationals`, `mul_rationals`, `print_rational`, `rationals_are_equal`

Rational Numbers

Data Abstraction: separation of

1. how data is manipulated ✓
   - `rational`, `numer`, `denom`
   - `add_rationals`, `mul_rationals`, `print_rational`, `rationals_are_equal`
2. how data is represented ←

Representing Rational Numbers

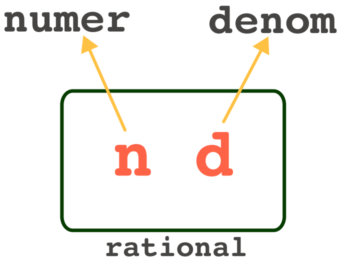
You can use a tuple or a list to store n,d:

```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```

Representing Rational Numbers

You can use a tuple or a list to store n,d:

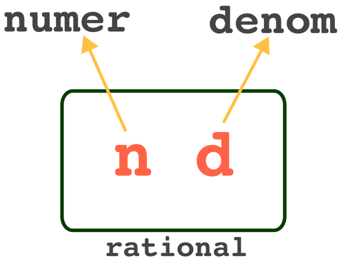
```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```



Representing Rational Numbers

You can use a tuple or a list to store n,d:

```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```



Only way to access the data is through accessors

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6

>>> print_rational(add_rationals(third, third))
6 / 9
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6

>>> print_rational(add_rationals(third, third))
6 / 9
```

Some rationals are not in **simplest form**

Rational Numbers

Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

Rational Numbers

Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

```
>>> print_rational(add_rationals(third, third))
2 / 3
```

Rational Numbers

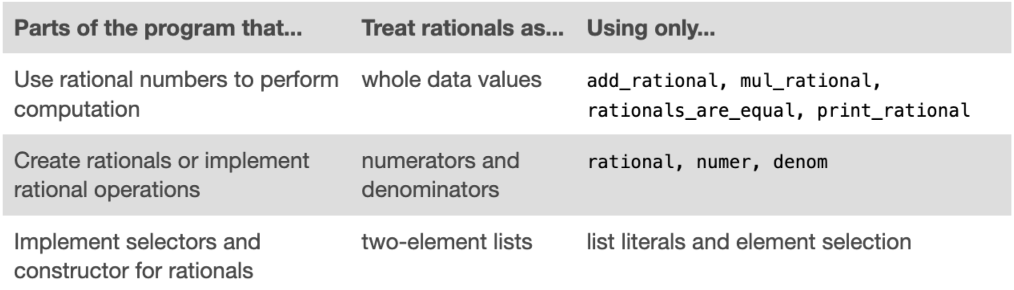
Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

```
>>> print_rational(add_rationals(third, third))
2 / 3
```

ZERO change required elsewhere

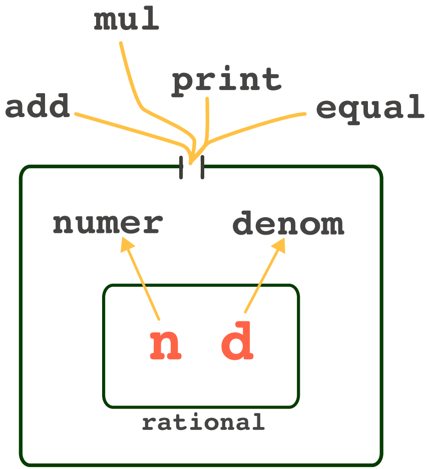
Abstraction Barriers



Parts of the program that...	Treat rationals as...	Using only...
Use rational numbers to perform computation	whole data values	`add_rational`, `mul_rational`, `rationals_are_equal`, `print_rational`
Create rationals or implement rational operations	numerators and denominators	`rational`, `numer`, `denom`
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Each function in last column enforce an **abstraction barrier**

Abstraction Barriers



Big Idea

Abstraction Barrier Violation happens when a higher-level function is **bypassed** to use lower-level implementation details.

Abstraction Barrier Violation:

```
def square_rational(x):
    return mul_rational(x, x)
```

Abstraction Barrier Violation:

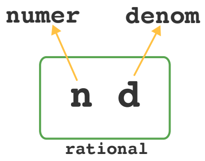
```
def square_rational(x):
    return mul_rational(x, x)

def square_rational_violating_once(x):
    return rational(numer(x)*numer(x), denom(x)*denom(x))
```

Abstraction Barrier Violation:

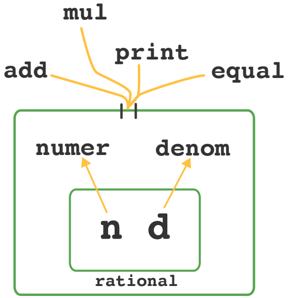
```
def square_rational(x):
    return mul_rational(x, x)

def square_rational_violating_once(x):
    return rational(numer(x)*numer(x), denom(x)*denom(x))

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

Abstraction Barrier Violation

- **Only** accessors and constructors should access **n,d** directly

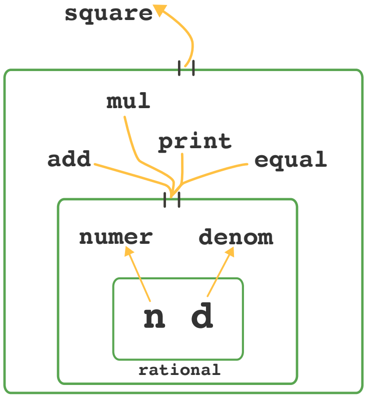


Abstraction Barrier Violation



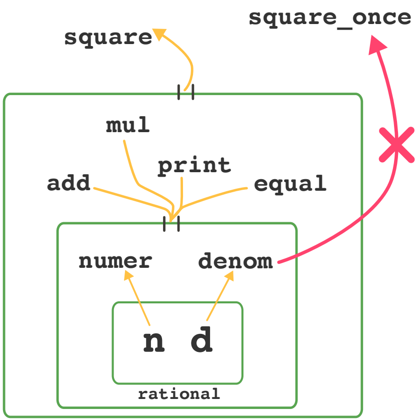
- **Only** accessors and constructors should access **n,d** directly
- These operations **should not** access **n,d** directly

Abstraction Barrier Violation



- **Only** accessors and constructors should access **n,d** directly
- These operations **should not** access **n,d** directly
- **square** should only access rational numbers through `add`, `mul`, `equal`, `print`

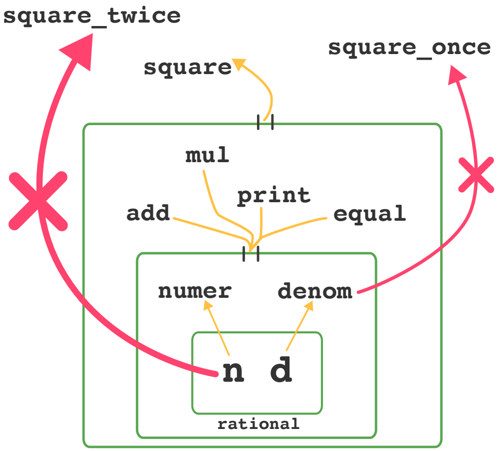
Abstraction Barrier Violation



- **Abstraction barrier** violated Once (Higher-level functions bypassed)

```
def sq_violate_once(x):
    return rational(
        numer(x)*numer(x),
        denom(x)*denom(x) )
```

Abstraction Barrier Violation



- **Abstraction barrier** violated Once (Higher-level functions bypassed)

```
    def sq_violate_once(x):
        return rational(
            numer(x)*numer(x),
            denom(x)*denom(x) )
```

- **Abstraction barrier** violated Twice

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

You Try!

Right now our rational numbers are internally represented as lists.

1. Change the internal representation to a dictionary.
2. What portion of the entire program really needed to change?

The Power of Abstraction

Changing the representation of rational numbers **requires no changes** to any other parts of code:

```
def rational(n, d):
    return {"numerator":n, "denominator":d}
def numer(x):
    return x["numerator"]
def denom(x):
    return x["denominator"]
```

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]`

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` **# use a Constructor!** `make_point(x,y)`

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` **# use a Constructor!** `make_point(x,y)`

2.
   ```
   p = make_point(3,4)
   y = p[1]
   ```

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` # **use a Constructor!** `make_point(x,y)`

2.
```
p = make_point(3,4)
y = p[1]
```
1st **level violated. Use accessors**
```
y = get_y(p)
```

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

```
def distance(p1, p2):
    return ((p1[0]-p2[0])**2 +
            (p1[1]-p2[1])**2)**0.5
```

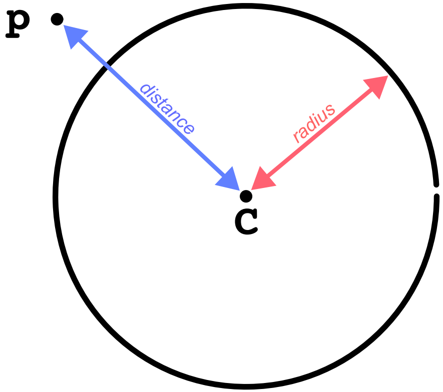
You Try! *(solution)*

1. **One level violated**
   - Bypassing the higher-level functions *(accessors)*
   - Directly accessing internal data *(very bad)*

```
def distance(p1, p2):
    return ((get_x(p1) - get_x(p2))**2 +
             (get_y(p1) - get_y(p2))**2)**0.5
```

You Try!

Check if a point is within a circle:

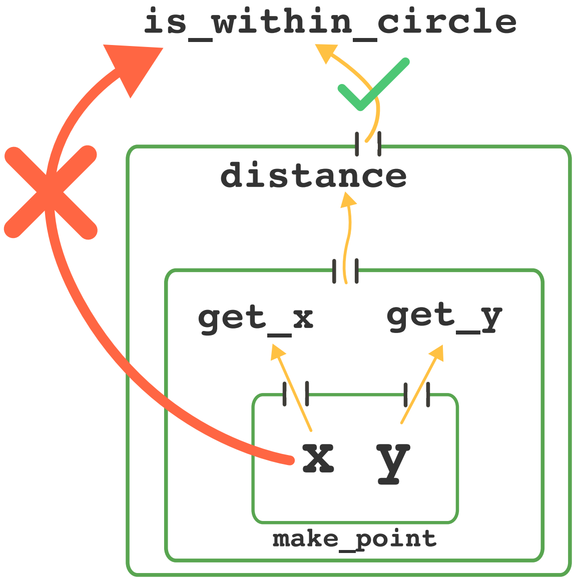


You Try!

Check if a point is within a circle. Identify **Abstraction Barrier Violations**:

```
def is_within_circle(point,center,radius):
    dist = ((center[0] - point[0])**2 +
             (center[1] - point[1])**2)**0.5

    return dist <= radius
```

You Try!



You Try!

Solution:

```
def is_within_circle(point,center,radius):
    return distance(point,center) <= radius
```

Summary

- **Data Abstraction**: Seperating how data is **represented** from how it is **manipulated**

Summary

- **Data Abstraction**: Seperating how data is **represented** from how it is **manipulated**
  Example: **Rational Numbers**
  - **How data is represented Data**: `[1,2]`
  - **How data is manipulated Operations:** `rational`, `numer`, `denom`, `add_rational`, `mul_rational`, ...

Summary

- **Data Abstraction**: Seperating how data is **represented** from how it is **manipulated**
  Example: **Rational Numbers**
  - **How data is represented Data**: `[1,2]`
  - **How data is manipulated Operations:** `rational`, `numer`, `denom`, `add_rational`, `mul_rational`, `...`
- **Abstraction Barrier Violation**: When a higher-level function is bypassed to use lower-level implementation details

Summary

- **Data Abstraction**: Seperating how data is **represented** from how it is **manipulated**
  Example: **Rational Numbers**
  - **How data is represented Data**: `[1,2]`
  - **How data is manipulated Operations:** `rational`, `numer`, `denom`, `add_rational`, `mul_rational`, `...`
- **Abstraction Barrier Violation**: When a higher-level function is bypassed to use lower-level implementation details
- **The Power of Abstraction**: Makes code **modular**, **easy to maintain**, **less buggy**, **very flexible**

Questions?