Part 1:

Object-oriented programming

Fractions

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

Fraction Abstract Data Type (ADT)

Fraction Abstract Data Type (ADT)

- Make new fraction

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator
- Add, other math operations

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator
- Add, other math operations
- convert to string

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator
- Add, other math operations
- Convert to string
- Equality / comparisons [$\frac{2}{3} \leq \frac{18}{26}$?]

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator
- Add, other math operations
- Convert to string
- Equality / comparisons [$\frac{2}{3} \leq \frac{18}{26}$?]
- No changing (like numbers + strings)

Fraction Abstract Data Type (ADT)

- Make new fraction
- Get Numerator / Denominator
- Add, other math operations
- convert to string
- Equality / comparisons [$\frac{2}{3} \leq \frac{18}{26}$?]
- No changing (like numbers + strings)
- Always want fraction to be stored in lowest terms, positive denominator

$$\frac{5}{-15} \Rightarrow \frac{-1}{3}$$

What data needs to be stored in a Fraction?

What data needs to be stored in a Fraction?

- Numerator
- Denominator

What data needs to be stored in a Fraction?

- Numerator
- Denominator

In Python, two simple ways to glue two numbers together

$\frac{2}{3}$ (2, 3) [2, 3]

tuple list

What data needs to be stored in a Fraction?

- Numerator
- Denominator

In Python, two simple ways to glue two numbers together

$\frac{2}{3}$ (2, 3) [2, 3]

tuple

list

↑ Choose this

Function Headers

Function Headers

```
f = fMake(n, d)
```

n, d: integers
f : fraction (really a pair)
s : string
b : boolean (T/F)

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
```

n, d: integers
f : fraction (really a pair)
s : String
b : boolean (T/F)

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
```

n, d: integers
f : fraction (really a pair)
s : String
b : boolean (T/F)

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
```

```
n, d: integers
f   : fraction
      (really a pair)
s   : String
b   : boolean (T/F)
```

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
```

n, d : integers
f : fraction (really a pair)
s : String
b : boolean (T/F)

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
```

n, d: integers
f : fraction (really a pair)
s : String
b : boolean (T/F)

Function Headers

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

n, d : integers
f : fraction (really a pair)
s : String
b : boolean (T/F)

From The User's Perspective

Non-OO	OO
f = fMake(n, d)	
n = fGetN(f)	
d = fGetD(f)	
s = fToString(f)	
f3 = fAdd(f1, f2)	
b = fEquals(f1, f2)	
b = fLessThan(f1, f2)	

From The User's Perspective

Non-OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

```
f = Fraction(n, d)
```

From The User's Perspective

Non-OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
```

From The User's Perspective

Non-OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
f3 = f1.add(f2)
b = f1.equals(f2)
b = f1.lessThan(f2)
```

Non OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

User view ← OO → Coding view

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
f3 = f1.add(f2)
b = f1.equals(f2)
b = f1.lessThan(f2)
```

Non OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

User view

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
f3 = f1.add(f2)
b = f1.equals(f2)
b = f1.lessThan(f2)
```

Coding view

```
Class Fraction
    def __init__(self, n, d)
    def getN(self)
    def getD(self)
    def toString(self)
    def add(self, other)
    def equals(self, other)
    def lessThan(self, other
```

Non OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

A fraction is a tuple
- f[0]: numerator
- f[1]: denominator

OO

User view ←

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
f3 = f1.add(f2)
b = f1.equals(f2)
b = f1.lessThan(f2)
```

→ Coding view

```
Class Fraction
    def __init__(self, n, d)
    def getN(self)
    def getD(self)
    def toString(self)
    def add(self, other)
    def equals(self, other)
    def lessThan(self, other
```

Non OO

```
f = fMake(n, d)
n = fGetN(f)
d = fGetD(f)
s = fToString(f)
f3 = fAdd(f1, f2)
b = fEquals(f1, f2)
b = fLessThan(f1, f2)
```

OO

User view → OO ← Coding view

```
f = Fraction(n, d)
n = f.setN()
d = f.setD()
s = f.toString()
f3 = f1.add(f2)
b = f1.equals(f2)
b = f1.lessThan(f2)
```

```
Class Fraction
    def __init__(self, n, d)
    def getN(self)
    def getD(self)
    def toString(self)
    def add(self, other)
    def equals(self, other)
    def lessThan(self, other
```

A fraction is a tuple
f[0]: numerator
f[1]: denominator

f is an object of type Fraction
Any data can be stored using .
eg f.numerator f.denominator
self is "me"

Magic Methods

Magic Methods

str(x) calls x.__str__()

Magic Methods

`str(x)`	calls	`x.__str__()`
`x + y`		`x.__add__(y)`

Magic Methods

str(x)	calls	x.__str__()
x + y		x.__add__(y)
x == y		x.__eq__(y)

Magic Methods

`str(x)`	calls	`x.__str__()`
`x + y`		`x.__add__(y)`
`x == y`		`x.__eq__(y)`
`hash(x)`		`x.__hash__()`

Magic Methods

`str(x)`	calls	`x.__str__()`
`x + y`		`x.__add__(y)`
`x == y`		`x.__eq__(y)`
`hash(x)`		`x.__hash__()`
	⋮	
etc		

Many more

Encapsulation + Privacy

Encapsulation + Privacy

- We don't want the user to be able to change the numerator

Encapsulation + Privacy

- We don't want the user to be able to change the numerator

```
f = Fracton(13/25)
f.numerator = 10
```

Encapsulation + Privacy

- We don't want the user to be able to change the numerator

```
f = Fracton(13/25)
f.numerator = 10
```

Bad! (1) Violates lowest terms

Encapsulation + Privacy

- We don't want the user to be able to change the numerator

```
f = Fracton(13/25)
f.numerator = 10
```

Bad! (1) Violates lowest terms
(2) Violates read-only

Encapsulation + Privacy

- We don't want the user to be able to change the numerator

```
f = Fracton(13/25)
f.numerator = 10
```

Bad! (1) Violates lowest terms
(2) Violates read-only

- Solution:
  Change this.denaminator → this._denominator

OO in Python : Summary

Fraction → class

f = Fraction(1,2) → f is an instance of class f

OO in Python : Summary

Fraction → class

f = Fraction(1,2) → f is an instance of class f

self is used to access the current instance

OO in Python : Summary

Fraction → class

f = Fraction(1,2) → f is an instance of class f

---

self is used to access the current instance

Data that lasts as long as the instance is accessed by methods using this.

OO in Python : Summary

Fraction → class

f = Fraction(1,2) → f is an instance of class f

---

self is used to access the current instance

Data that lasts as long as the instance is accessed by methods using this.

To create a new instance of a class, use the class name → This calls __init__ "constructor"

The

End

(part 1)