CS1010S Programming Methodology

Lecture 5
Data Abstraction & Debugging

12 Sep 2018

Collaboration Policy

By all means discuss

But write the solution <u>yourself</u>

Group Discussion

- Discard all code/solutions from group discussion
- Every one goes home and rewrite their own solution.
- No emailing of code allowed

Recap: Higher Order Functions

All three functions are very similar.

```
def sum_integers(a, b):
  if a > b:
    return 0
  else:
    return a +
           sum_integers(a + 1, b)
```

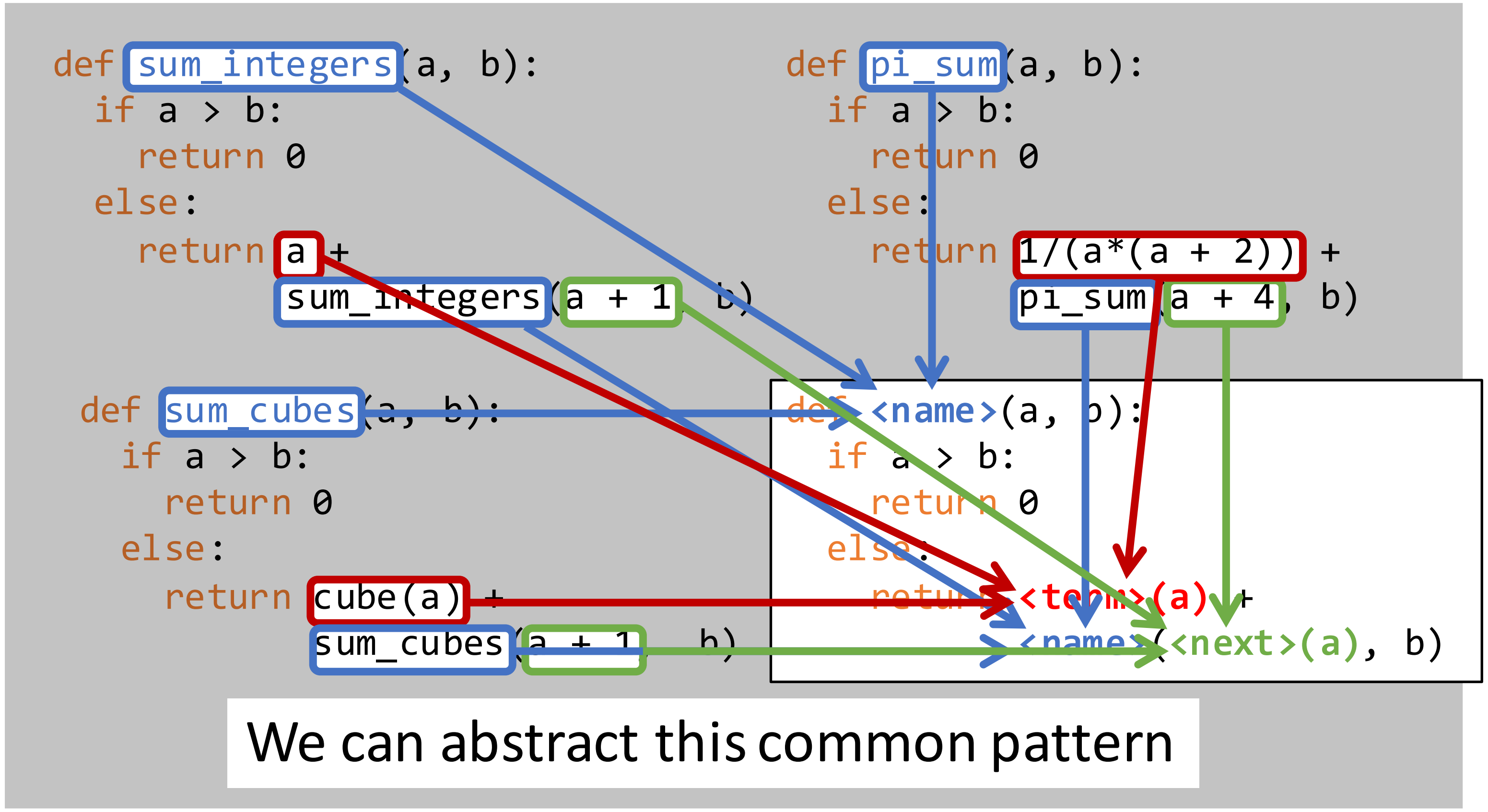
```
def pi_sum(a, b):
  if a > b:
    return 0
  else:
    return 1/(a*(a + 2)) +
           pi_sum(a + 4, b)
```

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) +
           sum_cubes(a + 1, b)
```

```
def <name>(a, b):
  if a > b:
    return 0
  else:
    return <term>(a) +
           <name>(<next>(a), b)
```

Recap: Higher Order Functions

All three functions are very similar.



```
def sum_integers(a, b):
  if a > b:
    return 0
  else:
    return a +
        sum_integers(a + 1, b)
```

```
def pi_sum(a, b):
  if a > b:
    return 0
  else:
    return 1/(a*(a + 2)) +
        pi_sum(a + 4, b)
```

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) +
        sum_cubes(a + 1, b)
```

```
def <name>(a, b):
  if a > b:
    return 0
  else:
    return <term>(a) +
        <name>(<next>(a), b)
```

We can abstract this common pattern

Common Abstraction

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) +
           sum(term, next(a), next, b)
```

functions

Re-defining

```
def sum_integers(a, b):
    return sum(inden, a, inc, b)

def sum_cubes(a, b):
    return sum(cube, a, inc, b)

def pi_sum(a, b):
    return sum(lambda x: 1/(x*(x+2)), a
               lambda x: x+4, b)
```

Isn't `sum` just

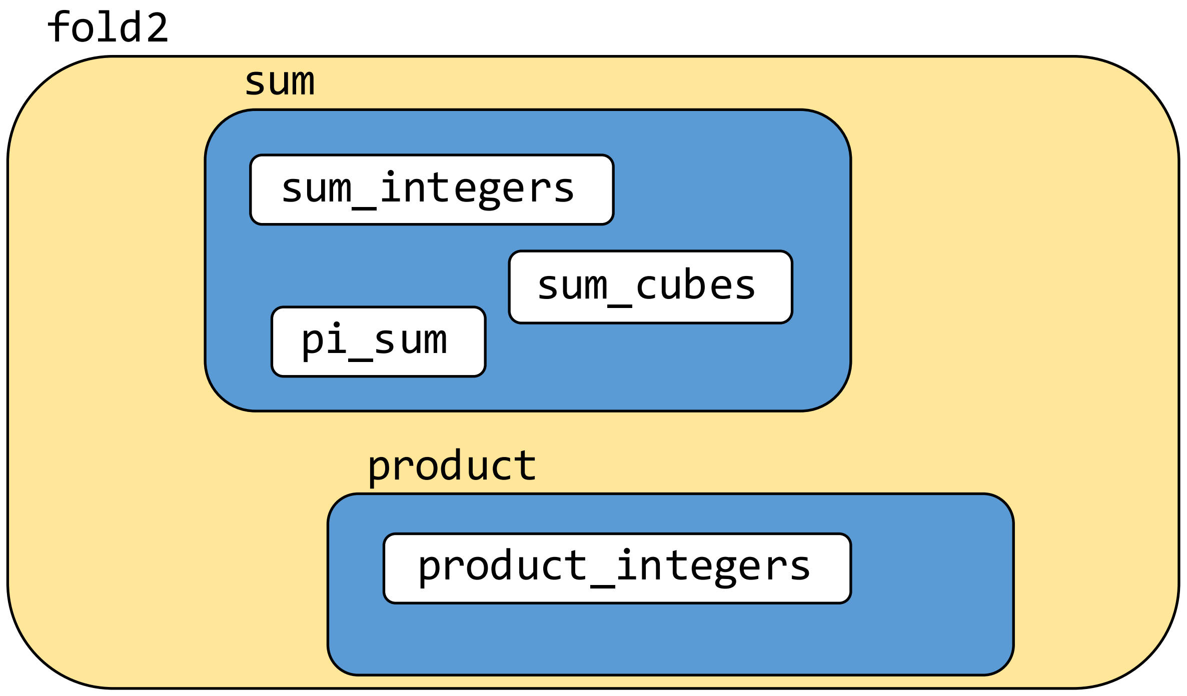
$$\sum_{a}^{b} t(x)$$

Taylor's Expansion

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!}$$

We can write a more general function to define other functions



Higher Order Functions

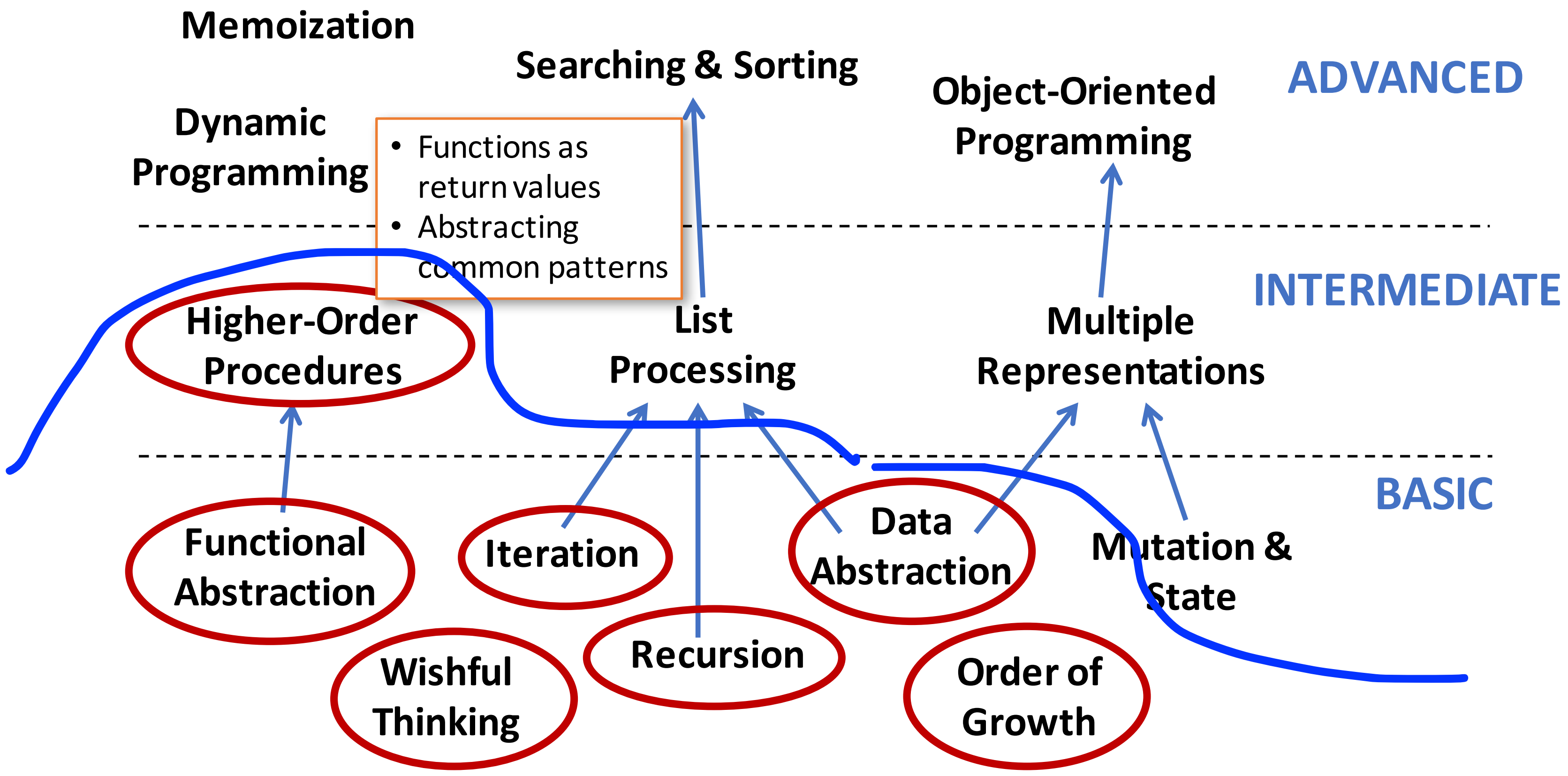
- Functions as input
  - Abstract common patterns

    e.g. `sum`, `fold`, `fold2`
- Function as output
  - Manipulate functions

    e.g. `deriv`, `composite`
  - Capture variables

    e.g. `adder`

CS1010S Road Map



Fundamental **concepts** of computer programming

So far we have only dealt with very simple data:

Numbers

(and some pictures)

However, life is complicated

AAAAHHH!!!
Why is life so Complicated??

To do anything useful, we need to model

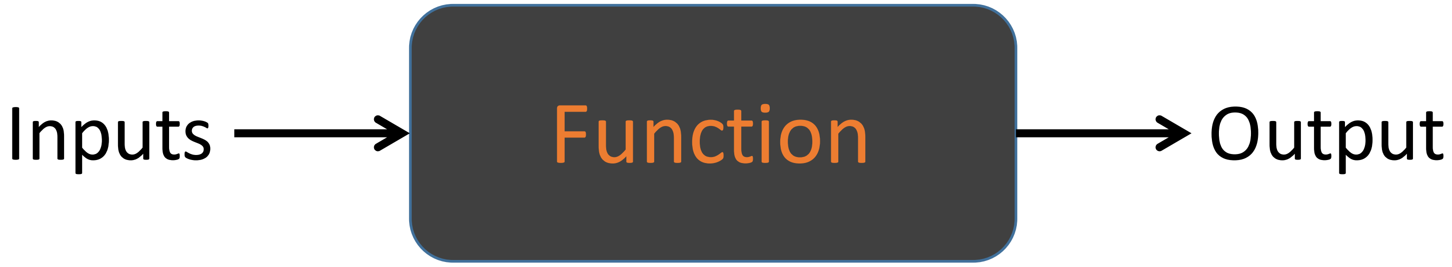
REAL objects

Example

NUS Registrar has a record of every student

- Personal info, modules taken, grades, etc.
- Record may be a paper folder or electronic document
- Record is a compound data

Recall: Functional Abstraction



- Only need to know how a function transforms inputs to an output
- Don't need to know how it is implemented

Recall: Functional Abstraction

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation
- Captures common programming patterns
- Serves as a building block for more complex functions

Key Idea

We can organize and reason about data the same way!

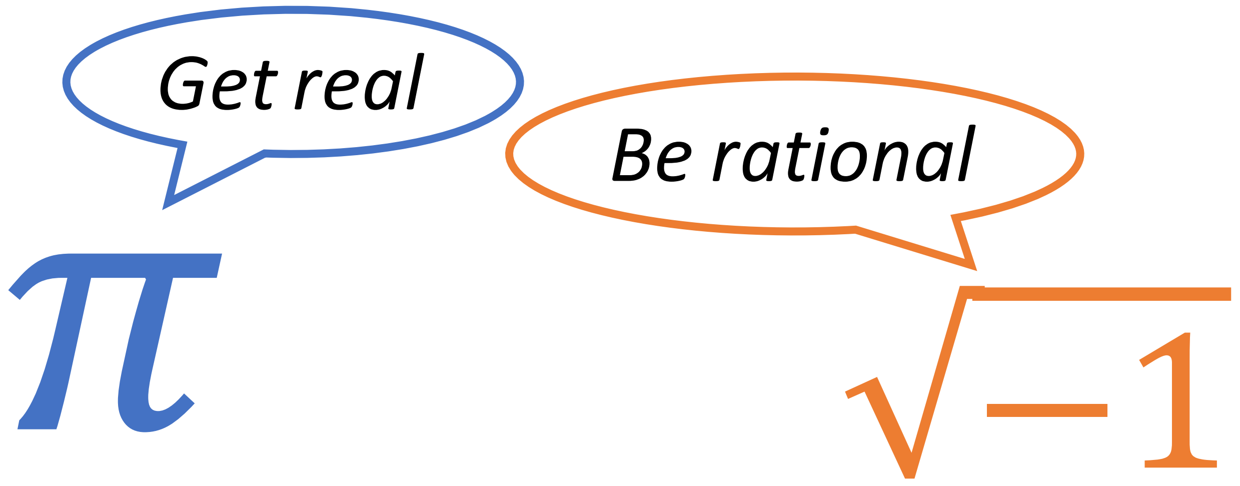
Data Abstraction

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation
- Captures common programming patterns
- Serves as a building block for other compound data

Case Study

`float` is imprecise
better to work with
fractions

Rational Numbers



Rational Number Package

- Rational number: $\frac{n}{d}$
  - ${}^{3}/_{5}$, ${}^{-1}/_{2}$
  - n: numerator
  - d: denominator
- Provide arithmetic operations
  - Addition
  - Subtraction
  - Multiplication, etc.

Guidelines for Creating Compound Data

- Constructors
  - To create compound data from primitive data
- Selector (Accessors)
  - To access individual components of compound data
- Predicates
  - To ask (true/false) questions about compound data
- Printers
  - To display compound data in human-readable form

Wishful Thinking

Let's wish for the following:

- `def make_rat(n, d): # constructor`
  - Returns a rational number with numerator *n*, denominator *d*
- `def numer(rat_number): # selector`
  - Returns the numerator of rat-number
- `def denom(rat_number): # selector`
  - Returns the denominator of rat-number

Arithmetic Operations

Addition:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

```
def add_rat(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return make_rat(nx * dy + ny * dx, dx * dy)
```

Arithmetic Operations

Subtraction:

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
def sub_rat(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return make_rat(nx * dy - ny * dx, dx * dy)
```

Arithmetic Operations

Multiplication:

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

```
def mul_rat(x, y):
  return make_rat(numer(x) * numer(y),
                  denom(x) * denom(y))
```

Arithmetic Operations

Division:

$$\frac{n_1}{d_1} \div \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

```
def div_rat(x, y):
    return make_rat(numer(x) * denom(y),
                    denom(x) * numer(y))
```

Predicates

Equality:

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \leftrightarrow n_1 d_2 = n_2 d_1$$

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

Printers

Displaying:

```
def print_rat(rat):
    print(f'{numer(rat)}/{denom(rat)})

print_rat(make_rat(1, 2)) → 1/2
```

Recall

- We assumed the existence of
  - `make_rat(n, d)`
  - `numer(rat_number)`
  - `denom(rat_number)`
- From which we defined new operations
  - `add_rat`, `sub_rat`, `mul_rat`, `div_rat`, `equal_rat`, `print_rat`
- Now what about our assumptions?
  - `make_rat`, `numer`, `denom`

Implementing rats

We can use a Python primitive called a tuple to “bind” data together

```
foo = (x, y) # creates a tuple out of
             # x and y
foo[0]    # return 1st component of foo
foo[1]    # return 2nd component of foo
```

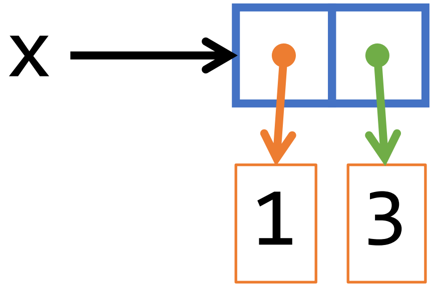
Tuple

```
x = (1, 2)
x → (1, 2)

x[0] → 1
x[1] → 2

y = (3, 4)
z = (x, y) # A tuple of tuples

z[0][0] → 1
z[1][1] → 4
```

Box-and-pointer notation

A way to visualize tuples

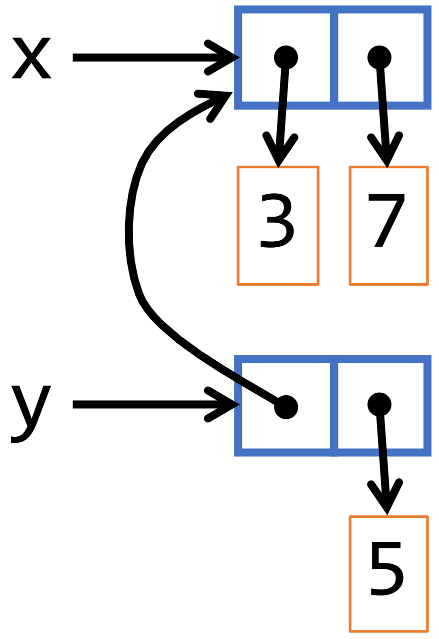
```
x = (1, 3)
```



- Variables x points to tuple
- Left arrow is `[0]`
- Right arrow is `[1]`
- Numbers are outside the tuple, not inside

Box-and-pointer notation

```
x = (3, 7)
x → (3, 7)

y = (x, 5)
y → ((3, 7), 5)

y[0][0] → 3
y[0][1] → 7
```



More on Tuples

A tuple is a common structure:

```
bar = (1, 2)
bat = (3, 4)
foo = bar + bat  # creates a new tuple
foo → (1, 2, 3, 4)

foo = (bar, bat)
foo → ((1, 2), (3, 4))
```

`()` is the empty tuple.

Recall String Slicing?

```
s[start:stop:step]
```

$\uparrow$ non-inclusive

```
>>> s = 'abcdef'
>>> s[0:2]
'ab'
>>> s[1:2]
'b'
>>> s[:2]
'ab'
```

```
>>> s[1:5:3]
'be'
>>> s[::2]
'ace'
```

Slicing returns a new string

Tuple Selectors

`foo`	returns the tuple foo
`foo[0]`	returns 1st element of foo
`foo[1:]`	returns tail of foo (rest of foo without 1st element)
`foo[a:b]`	returns tuple consisting of a+1th to bth element of foo
`foo[a:b:c]`	returns tuple consisting of a+1th to bth element of foo, in steps of c
`foo[-1]`	returns the last element of foo
`len(foo)`	returns the number of elements in foo

Examples

```
x = (1, 2, 3, 4)
x[0]      → 1
x[1:]     → (2, 3, 4)
x[0:]     → (1, 2, 3, 4)
x[1:3]    → (2, 3)
x[1:2]    → (2,) # not the same as (2)
x[1]      → 2
x[-1]     → 4
x[:3:2]   → (1, 3)
len(x)    → 4 # length of tuple
```

Iterating over tuples

```
x = (4, 2, 1, 3)

count = 0
for i in x:
    print(i)
    count = count + i
print(count)
```

```
4
2
1
3
10
```

Rational Number

We can complete our rational number package by defining:

```
def make-rat(n, d):
   return (n, d)

def numer(rat):
   return rat[0]

def denom(rat):
   return rat[1]
```

Using Rational Number

```
>>> one_half = make_rat(1, 2)
>>> print_rat(one_half)
1/2

>>> one_third = make_rat(1, 3)
>>> print_rat(one_third)
1/3

>>> print_rat(add_rat(one_half, one_third))
5/6

>>> print-rat(mul_rat(one_half, one_third))
1/6

>>> print-rat(add_rat(one_third, one_third))
6/9
```

Yikes! Why not 2/3?

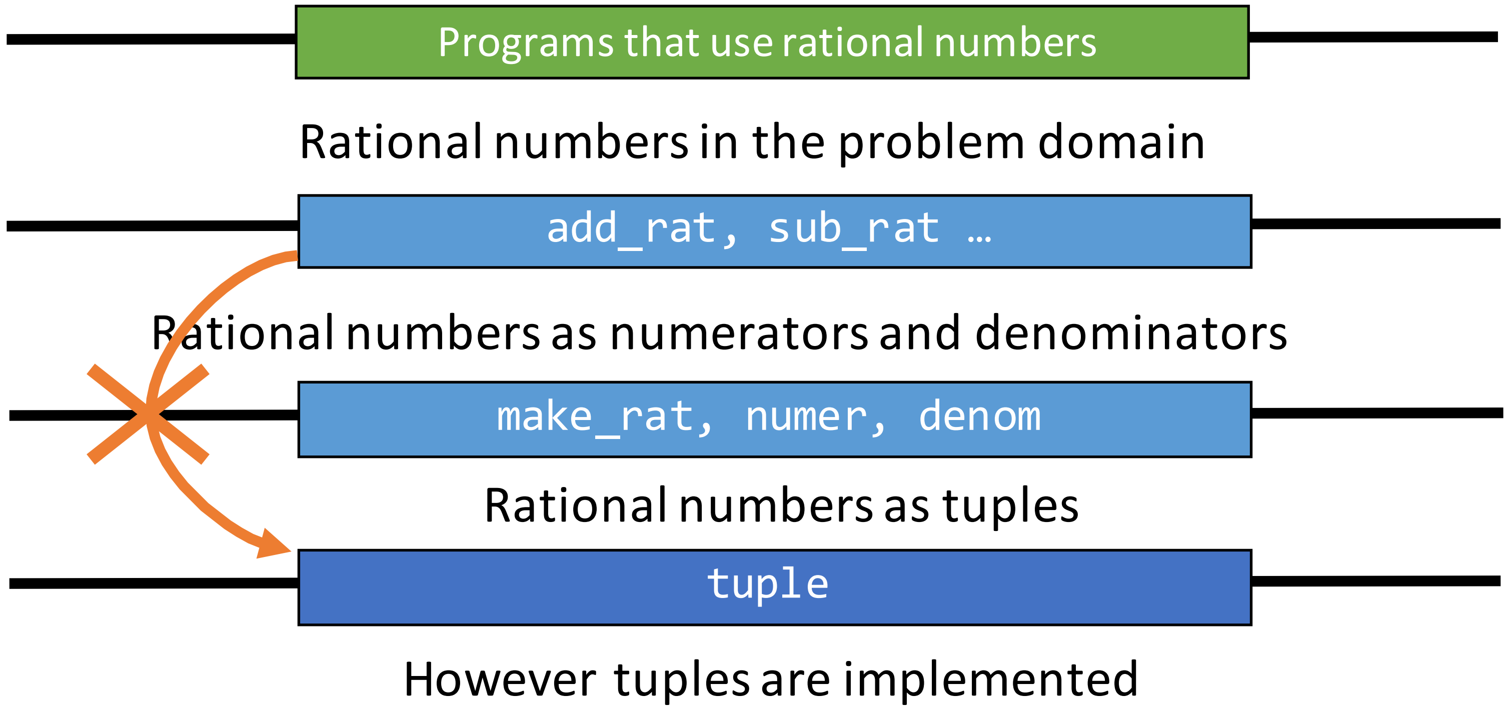
Improvement

We can “reduce to lowest terms” by modifying `make_rat`

$$\frac{kp}{kq} = \frac{p}{q} \text{ where } k \text{ is } \mathrm{gcd}(p, q)$$

```
from fractions import gcd

def make_rat(n, d):  # version 2
    g = gcd(n, d)
    return (n//g, d//g)
```

Abstraction Barrier


Programs that use rational numbers
Rational numbers in the problem domain
add_rat, sub_rat …
Rational numbers as numerators and denominators
make_rat, numer, denom
Rational numbers as tuples
tuple
However tuples are implemented

At each level, use only functions available at that interface, not below it.

Equality

What does equality mean?

Two possibilities (usually)

1. Identity
   - This means the SAME object (reference in memory)
   - In Python, we use `is` to test this.

Two possibilities (usually)

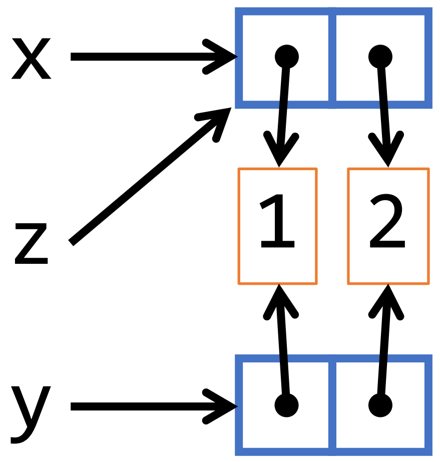
2. Equivalence
    - This means two objects are equivalence (of the same value) even if they are not the same object
    - In Python, we use == to test this.

Identity != Equivalence

Equality

`is` returns `True` if the two objects are the same object
`==` returns `True` if the two objects are equivalent

```
x = (1, 2)
y = (1, 2)
x is y → False
x == y → True
z = x
z is x → True
z is y → False
z == y → True
```



Caution with is

`is` cannot be used to compare numbers reliably

```
>>> 3 is 3
True

>>> 3.000 is 3
False
```

Equality

The predicate == returns True if the two object have the same contents

- works for numbers, strings and tuples

```
>>> ('apple', 1, 2, 3) == ('apple', 1, 2, 3)
True
>>> ('apple', 1, 2, 3) is ('apple', 1, 2, 3)
False
>>> ('apple', 1, 2, 3) == ('apple', (1, 2), 3)
False
```

To add to confusion: ==

```
>>> ('apple', 1, 2, 3) == ('apple', (1), 2, 3)
True
>>> ('apple', 1, 2, 3) == ('apple', (1,), 2, 3)
False

>>> t = (1)
>>> t
1

>>> s = (1,)
>>> s
(1,)
```

Moral of the story

Use `==` and `is` carefully, to save yourself grief.

Debugging

Humans make mistakes

You are only human

Therefore, you will make mistakes

Debugging

- Means to remove errors (“bugs”) from a program.
- After debugging, the program is not necessarily error-free.
  - It just means that whatever errors remain are harder to find.
  - This is especially true for large applications.

Common Types of Errors

- Omitting return statement

```
def square(x):
    x * x        # no error msg!
```

- Incompatible types

```
x = 5
def square(x):
    return x * x
x + square
```

- Incorrect # args

```
square(3,5)
```

Common Types of Errors

- Syntax

```
def proc(100)
    do_stuff()
        more()
```

- Arithmetic error

```
x = 3
y = 0
x/y
```

- Undeclared variables

```
x = 2
x + k
```

Common Types of Errors

- Infinite loop (from bad inputs)

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)

fact(2.1)
fact(-1)
```

Common Types of Errors

- Infinite loop (from not decrementing)

```
def fact_iter(n):
    counter, result = n, 1
    while counter!= 0:
        result *= counter
    return result
```

Common Types of Errors

- Numerical imprecision

```
def foo(n):
  counter, result = 0,0
  while counter != n:
    result += counter
    counter += 0.1
  return result

foo(5)
```

counter never exactly equals *n*

Common Types of Errors

- Logic

```
def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-1)
```

How to debug?

- Think like a detective
  - Look at the clues: error messages, variable values.
  - Eliminate the impossible.
  - Run the program again with different inputs.
    - Does the same error occur again?

How to debug?

- Work backwards
  - From current sub-problem backwards in time
- Use a debugger
  - IDLE has a simple debugger
  - Overkill for our class
- Trace a function
- Display variable values

Displaying variables

```
debug_printing = True
def debug_print(msg):
  if debug_printing:
    print(msg)

def foo(n):
  counter, result = 0,0
  while(counter != n):
    debug_print(f'{counter}, {n}, {result}')
    counter, result = counter + 0.1, result + counter
  return result
```

Example

```
def fib(n):
  debug_print(f'n:{n}')
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-1)
```

Other tips

- State assumptions clearly.

```
def factorial(n): # n integer >= 0
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

- Test each function before you proceed to the next.
  - Remember to test boundary cases

Summary

- Compound data helps us to reason at a higher conceptual level.
- Abstraction barriers separate usage of a compound data from its implementation.
- Only functions at the interface should be used.
- We can choose between different implementations as long as contract is fulfilled.

Summary

- Debugging often takes up more time than coding
- More an art than a science
- Play detective!
- Do it systematically
- Avoid debugging with good programming practices

Question of the Day

Implement a new Abstract Data Type (ADT) set with the following associated functions:

- `make_set()` – creates a new empty set object
- `add_set(set, object)` – adds an object to a set
- `remove_set(set, object)` – removes an object from a set
- `contains_set(set, object)` – returns `True` if the set contains the specified object