Announcements

- A3 due tonight at 11:59pm.
- Spring break next week:
  - No office hours
  - No consulting hours
  - Limited piazza

Designing Types

From first day of class!

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, /, …)
  - Time (**set**: times of day; **ops**: time span, before/after, …)
  - Worker (**set**: all possible workers; **ops**: hire,pay,promote,…)
  - Rectangle (**set**: all axis-aligned rectangles in 2D; **ops**: contains, intersect, …)
- To define a class, think of a *real type* you want to make
  - Python gives you the tools, but does not do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want

Case Study: Fractions

- Want to add a new ***type***
  - Values are fractions: ½, ¾
  - Operations are standard multiply, divide, etc.
  - **Example**: ½*¾ = ⅜
- Can do this with a class
  - Values are fraction objects
  - Operations are methods
- **Example**: simplefrac.py

```
class Fraction(object):
    """Instance is a fraction n/d

    Attributes:
      numerator:   top      [int]
      denominator: bottom [int > 0]
    """

    def __init__(self,n=0,d=1):
        """Init: makes a Fraction"""
        self.numerator = n
        self.denominator = d
```

Making a Class into a Type

1. Think about what values you want in the set
   - What are the attributes? What values can they have?
2. Think about what operations you want
   - This often influences the previous question
- To make (1) precise: write a *class invariant*
  - Statement we promise to keep true **after every method call**
- To make (2) precise: write *method specifications*
  - Statement of what method does/what it expects (preconditions)
- Write your code to make these statements true!

Planning out a Class

```
class Time(object):
    """Instances represent times of day.
    Instance Attributes:
        hour: hour of day [int in 0..23]
        min:  minute of hour [int in 0..59]"""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""

    def isPM(self):
        """Returns: this time is noon or later."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Time instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Rectangle(object):
    """Instances represent rectangular
    regions of the plane.
    Instance Attributes:
        t: y coordinate of top edge [float]
        l: x coordinate of left edge [float]
        b: y coordinate of bottom edge [float]
        r: x coordinate of right edge [float]
    For all Rectangles, l <= r and b <= t."""

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b]
        Pre: args are floats; l <= r; b <= t"""

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
           intersection of self with other."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Implementing an Initializer

```
def __init__(self, hour, min):
    """The time hour:min.
    Pre: hour in 0..23; min in 0..59"""
```
This is true to start

```
self.hour = hour
self.min = min
```
You put code here

Instance variables:
hour: hour of day [int in 0..23]
min: minute of hour [int in 0..59]

This should be true at the end

Implementing a Method

Instance variables:
hour: hour of day [int in 0..23]
min: minute of hour [int in 0..59]

This is true to start

```
def increment(self, hours, mins):
    """Move this time <hours> hours
    and <mins> minutes into the future.
    Pre: hours [int] >= 0; mins in 0..59"""
```
What we are supposed to accomplish

This is also true to start

```
self.min  = self.min + mins
self.hour = self.hour + hours
```
?

You put code here

Instance variables:
hour: hour of day [int in 0..23]
min: minute of hour [int in 0..59]

This should be true at the end

Hiding Methods From Access

- Put underscore in front of a method will make it **hidden**
  - Will not show up in help()
  - But it is still there…
- Hidden methods
  - Can be used as **helpers** inside of the same class
  - But it is bad style to use them outside of this class
- Can do same for attributes
  - Underscore makes it hidden
  - Do not use outside of class

```
class Fraction(object):
    """Instance attributes:
        numerator:   top      [int]
        denominator: bottom [int > 0]"""
    def _is_denominator(self,d):
        """Return: True if d valid denom"""
        return type(d) == int and d > 0
    def __init__(self,n=0,d=1):
        assert self._is_denominator(d)
        self.numerator = n
        self.denominator = d
```
HIDDEN

Helper method

Data Encapsulation

```
class Fraction(object):
    """Instance attributes:
        _numerator:   top      [int]
        _denominator: bottom [int > 0]"""
    def getDenominator(self):
        """Returns: numerator attribute"""
        return self._denominator
    def setDenominator(self, d):
        """Alters denomenator to be d
        Pre: d is an int > 0"""
        assert type(d) == int
        assert 0 < d
        self._denominator = d
```
Getter

Setter

Naming Convention
The underscore means "should not access the attribute directly."

Precondition is same as attribute invariant.

Example: Converting Values to Strings

str() Function

- **Usage**: str(<expression>)
  - Evaluates the expression
  - Converts it into a string
- How does it convert?
  - str(2) → '2'
  - str(True) → 'True'
  - str('True') → 'True'
  - str(Point3()) → '(0.0,0.0,0.0)'

Backquotes

- **Usage**: `<expression>`
  - Evaluates the expression
  - Converts it into a string
- How does it convert?
  - `2` → '2'
  - `True` → 'True'
  - `'True'` → '"True"'
  - `Point3()` → "<class 'Point3'> (0.0,0.0,0.0)"

What Does str() Do On Objects?

- Does **NOT** display contents

```
>>> p = Point3(1,2,3)
>>> str(p)
'<Point3 object at 0x1007a90>'
```

- Must add a special method
  - __str__ for str()
  - __repr__ for backquotes
- Could get away with just one
  - Backquotes require __repr__
  - str() can use __repr__ (if __str__ is not there)

```
class Point3(object):
    """Instances are points in 3d space"""
    …
    def __str__(self):
        """Returns: string with contents"""
        return '('+self.x + ',' +
                   self.y + ',' +
                   self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
               str(self)
```