CS 1110:

Introduction to Computing Using Python

Lecture 18

Using Classes Effectively

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements

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

Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(1,2,3) # (1,2,3)
  - p = Point3(1,2) # (1,2,0)
  - p = Point3(y=3) # (0,3,0)
  - p = Point3(1,z=2) # (1,0,2)

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
        self.z = z
    …
```

Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(
  - p = Point3(1,2)
  - p = Point3(y=3)
  - p = Point3(1,z=2)

Assigns in order

Use parameter name when out of order

Can mix two approaches

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
        self.z = z
```

Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(
  - p = Point3(1,2)
  - p = Point3(y=3) # (0,3,0)
  - p = Point3(1,z=2)

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
```

Assigns in order

Use parameter name when out of order

Can mix two approaches

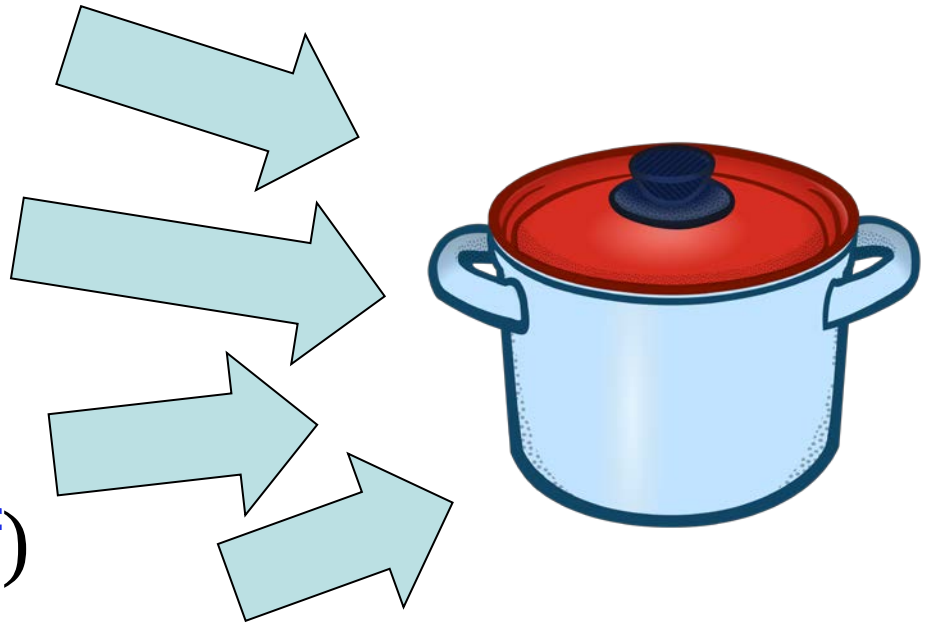
Not limited to methods. Can do with any function.

On Tuesday, we learned how to make:

- Class definitions
- Class specifications
- Class variables
- Methods
- Attributes (using self)
- A constructor with **__init__**

Today

- Class definitions
- Class specifications
- Class variables
- Methods
- Attributes (using self)
- A constructor with __init__



Designing Types

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, /, …)
  - Time (**set**: times of day; **ops**: time span, before/after, …)
  - Rectangle (**set**: all axis-aligned rectangles in 2D;
    ops: contains, intersect, …)
- To define a class, think of a *real type* you want to make

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: Time

- What *attributes*?
- What *invariants*?
- What *methods*?
- What *constructor*?

(24-hour clock)

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

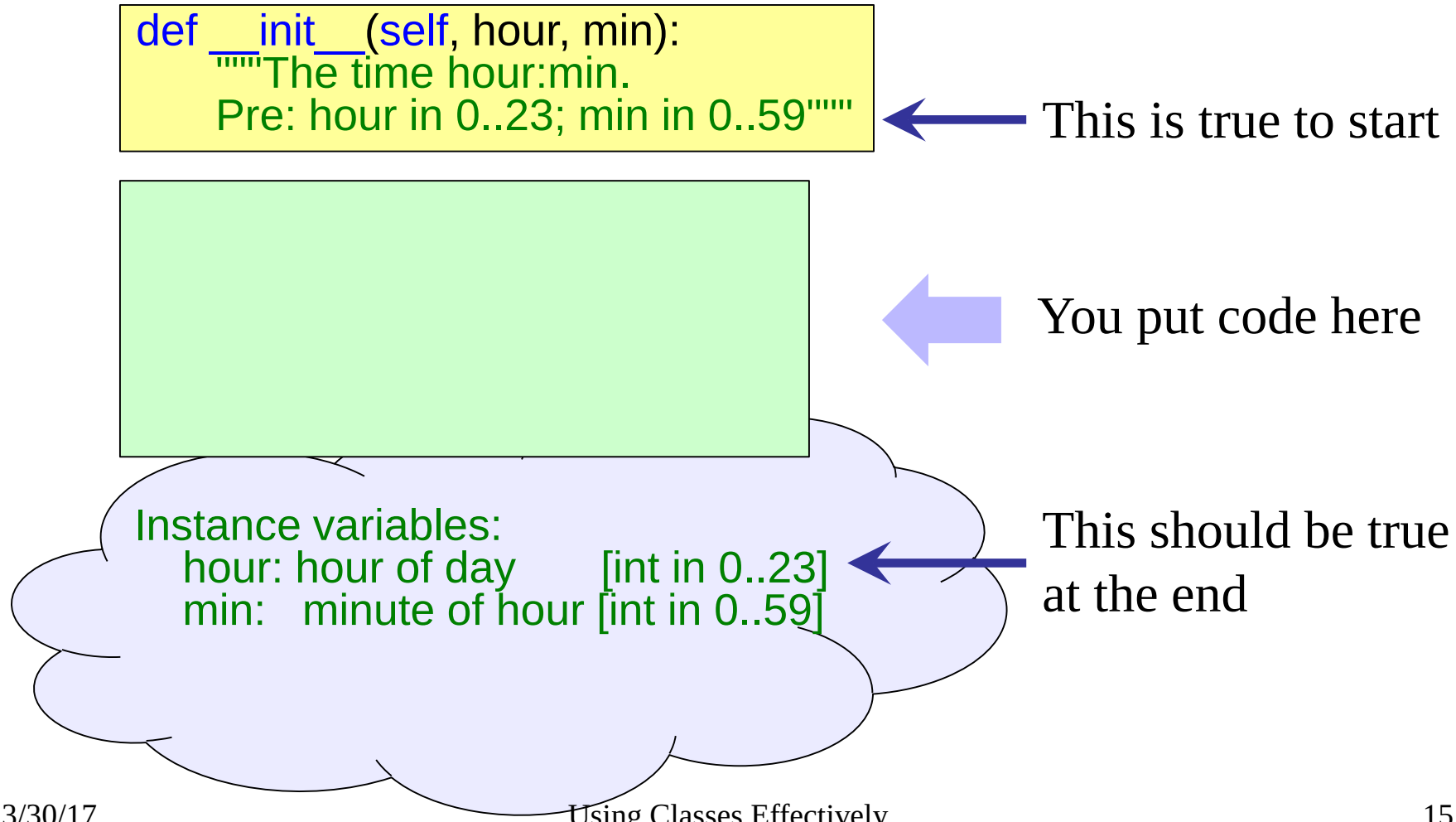
States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Implementing a Class

- All that remains is to fill in the methods. (All?!)
- When implementing methods:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Implementing an Initializer



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

A:

```
Time.hour = hour
Time.min = min
```

B:

```
hour = hour
min = min
```

C:

```
self.hour = hour
self.min = min
```

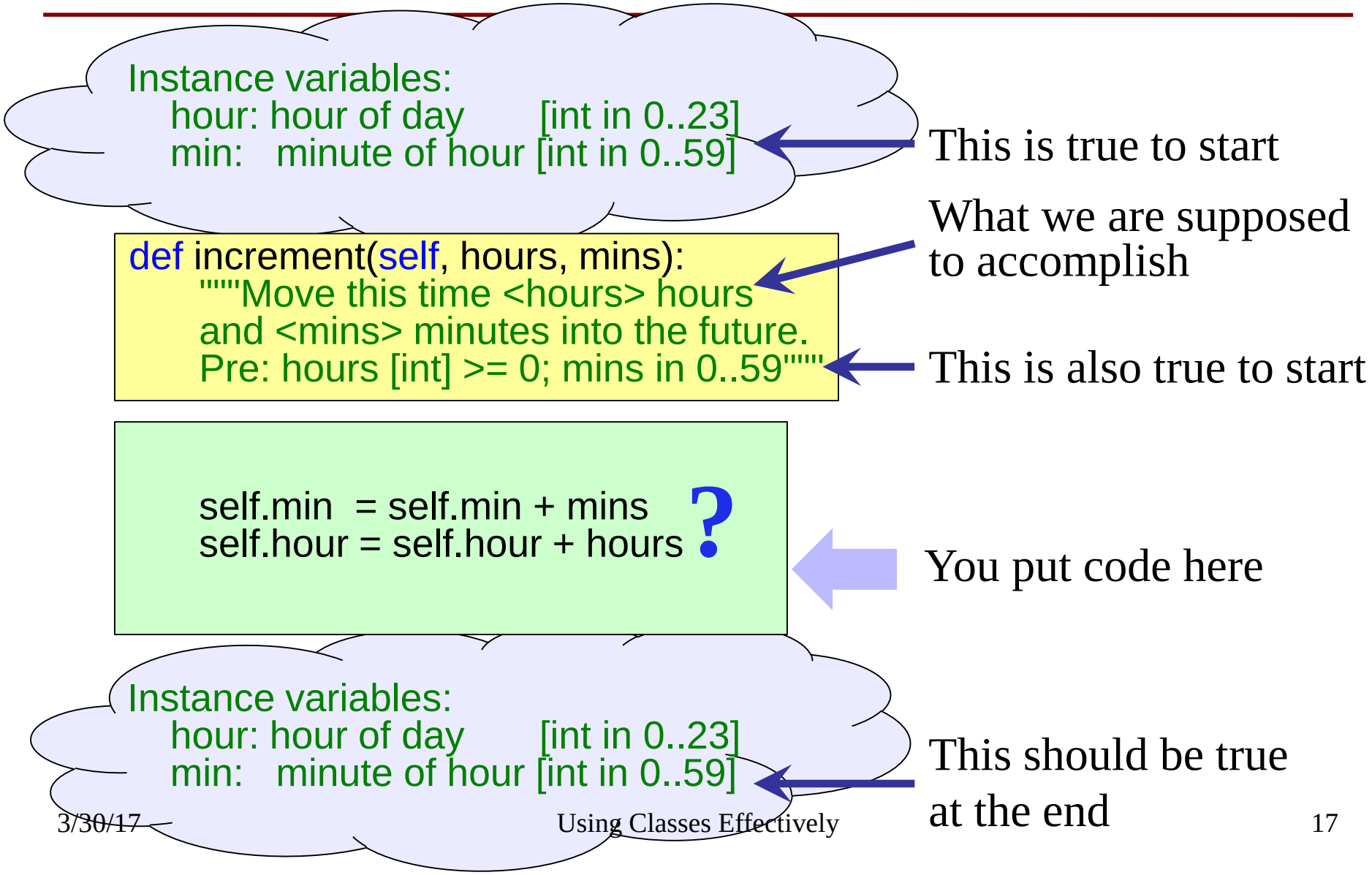
Instance variables:
hour: hour of day [int in 0..23]
min: minute of hour [int in 0..59]

This should be true at the end

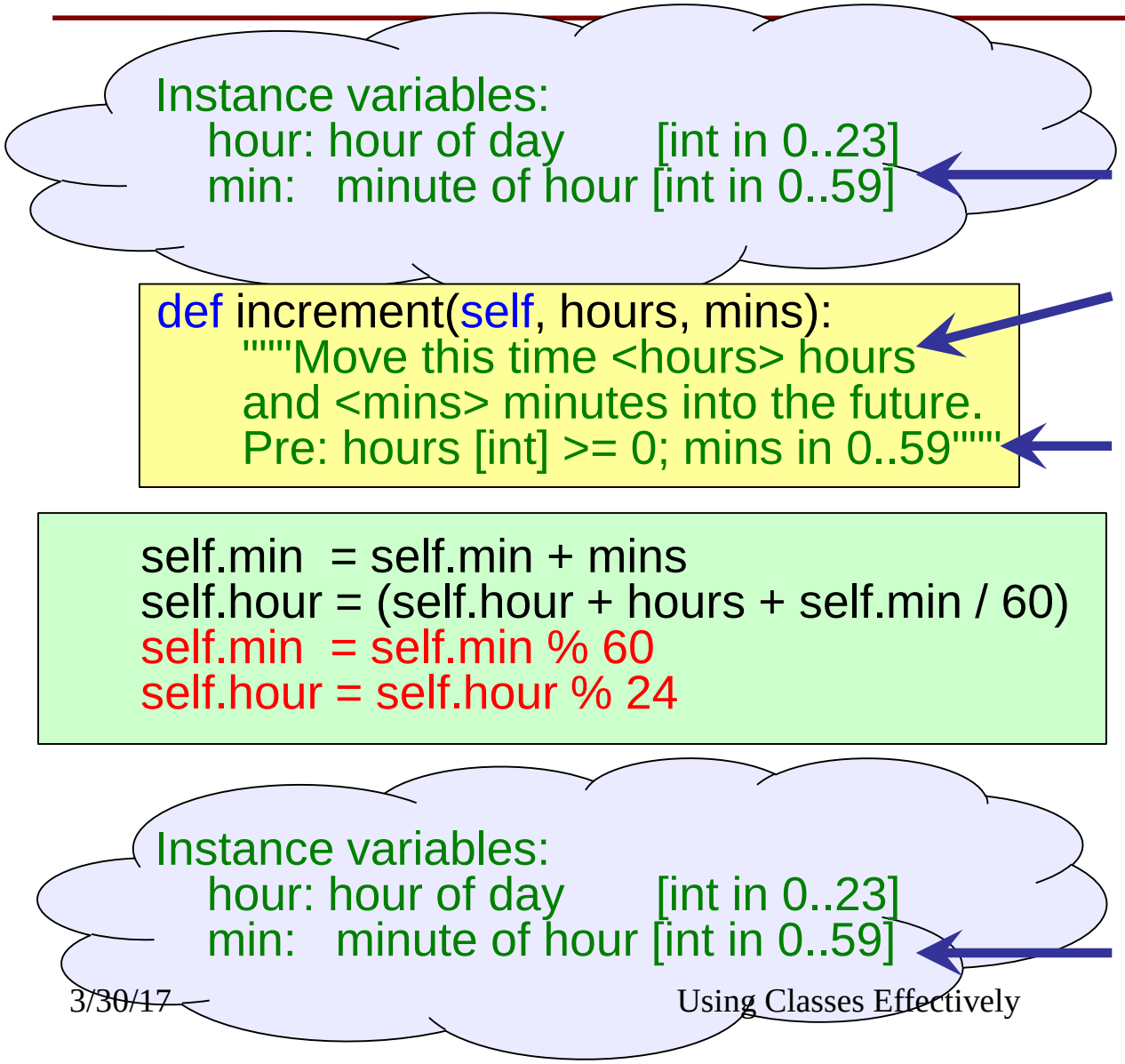
D:

```
self.hour = Time.hour
self.min = Time.min
```

Implementing a Method



Implementing a Method



Example: class Time

Special Methods in Python

- __init__ for initializer
- __str__ for str()
- __repr__ for backquotes
- Start/end with 2 underscores
  - This is standard in Python
  - Used in all special methods
  - Also for special attributes
- For a complete list, see

  http://docs.python.org/reference/datamodel.html

```
class Point3(object):
    """Instances are points in 3D space"""
    …

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes new Point3"""
        …

    def __str__(self,q):
        """Returns: string with contents"""
        …

    def __repr__(self,q):
        """Returns: unambiguous string"""
        …
```

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)"

Example: Converting Values to Strings

str() Function

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

What type is this value?

Backquotes

-

Backquotes are for *unambigious* representation

How does it co

- `2` → '2'
- `True` → 'Tru
- `'True'` → '"True"'
- `Point3()` → "<class 'Point3'> (0.0,0.0,0.0)"

The value's type is clear

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)
```

Gives the class name

__repr__ using __str__ as helper

Planning out a Class: Fraction

Planning out a Class: Fraction

- What *attributes*?
- What *invariants*?
- What *methods*?
- What *constructor*?

```
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
```

Example: class Fraction

Problem: Doing Math is Unwieldy

What We Want

$$\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\right)*\frac{5}{4}$$

Why not use the standard Python math operations?

What We Get

```
>>> p = Fraction(1,2)
>>> q = Fraction(1,3)
>>> r = Fraction(1,4)
>>> s = Fraction(5,4)
>>>
(p.add(q.add(r))).mult(s)
```

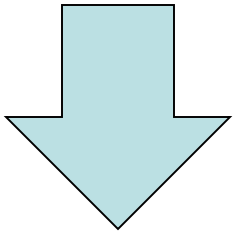
Pain!

Operator Overloading: Addition

```
class Fraction(object):
    """Instance attributes:
        numerator:    top       [int]
        denominator: bottom [int > 0]"""

    def __add__(self,q):
        """Returns: Sum of self, q
        Makes a new Fraction
        Precondition: q a Fraction"""
        assert type(q) == Fraction
        bot = self.denominator*q.denominator
        top = (self.numerator*q.denominator+
               self.denominator*q.numerator)
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p+q
```



Python converts to

```
>>> r = p.__add__(q)
```

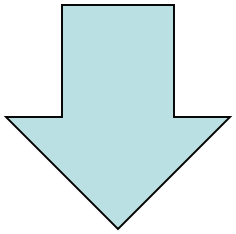
Operator overloading uses method in object on left.

Operator Overloading: Multiplication

```
class Fraction(object):
    """Instance attributes:
        numerator:    top       [int]
        denominator: bottom [int > 0]"""

    def __mul__(self,q):
        """Returns: Product of self, q
        Makes a new Fraction; does not
        modify contents of self or q
        Precondition: q a Fraction"""
        assert type(q) == Fraction
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p*q
```



Python converts to

```
>>> r = p.__mul__(q)
```

Operator overloading uses method in object on left.

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.

Enforcing Invariants

```
class Fraction(object):
    """Instance attributes:
      numerator:   top      [int]
      denominator: bottom [int > 0]
    """
```

Invariants: Properties that are always true.

- These are just comments!
- Do not enforce anything.

- **Idea**: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- Examples:

```
def getNumerator(self):
    """Returns: numerator"""
    return self.numerator

def setNumerator(self,value):
    """Sets numerator to value"""
    assert type(value) == int
    self.numerator = value
```