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
        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, mins is int 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, t <= b and l <= r"""
    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.

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
        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 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
        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, mins is int 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 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
        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, mins is int 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 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
        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 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
        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, mins is int 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 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
        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, mins is int 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 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
        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, mins is int 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.
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

Data Encapsulation

- **Idea**: Force the user to only use methods
- Do not allow direct access of attributes

<table>
<thead>
<tr>
<th>Setter Method</th>
<th>Getter Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Used to change an attribute</td>
<td>• Used to access an attribute</td>
</tr>
<tr>
<td>• Replaces all assignment statements to the attribute</td>
<td>• Replaces all usage of attribute in an expression</td>
</tr>
<tr>
<td><strong>Bad</strong>:</td>
<td><strong>Bad</strong>:</td>
</tr>
<tr>
<td>&gt;&gt;&gt; f.numerator = 5</td>
<td>&gt;&gt;&gt; x = f.numerator</td>
</tr>
<tr>
<td><strong>Good</strong>:</td>
<td><strong>Good</strong>:</td>
</tr>
<tr>
<td>&gt;&gt;&gt; f.setNumerator(3)</td>
<td>&gt;&gt;&gt; x = f.getNumerator()</td>
</tr>
</tbody>
</table>

Mutable vs. Immutable Attributes

<table>
<thead>
<tr>
<th>Mutable</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Can change value directly</td>
<td>• Can’t change value directly</td>
</tr>
<tr>
<td>* If class invariant met</td>
<td>* May change “behind scenes”</td>
</tr>
<tr>
<td>* Example: t.color</td>
<td>* Example: t.x</td>
</tr>
<tr>
<td>• Has both getters and setters</td>
<td>• Has only a getter</td>
</tr>
<tr>
<td>* Setters allow you to change</td>
<td>* No setter means no change</td>
</tr>
<tr>
<td>* Enforce invariants w/ asserts</td>
<td>* Getter allows limited access</td>
</tr>
</tbody>
</table>

Enforcing Invariants

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

```
class Fraction(object):
    """Returns: numerator attribute""
    def getNumerator(self):
        return self._numerator

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

Mutable: Invisible Setters and Getters

```
>>> p = Fraction(1,2)
>>> x = p.numerator
```
```
Python converts to
```
>>> x = p.numerator()
```

```
>>> p.numerator = 2
```
```
Python converts to
```
>>> p.numerator = 2
```