Example: Converting Values to Strings

<table>
<thead>
<tr>
<th>str() Function</th>
<th>repr() Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Usage: str(&lt;expression&gt;)</td>
<td>* Usage: repr(&lt;expression&gt;)</td>
</tr>
<tr>
<td>* Evaluates the expression</td>
<td>* Evaluates the expression</td>
</tr>
<tr>
<td>* Converts it into a string</td>
<td>* Converts it into a string</td>
</tr>
<tr>
<td>* How does it convert?</td>
<td>* How does it convert?</td>
</tr>
<tr>
<td>str(2) → '2'</td>
<td>repr(2) → '2'</td>
</tr>
<tr>
<td>str(True) → 'True'</td>
<td>repr(True) → 'True'</td>
</tr>
<tr>
<td>str('True') → 'True'</td>
<td>repr('True') → 'True'</td>
</tr>
</tbody>
</table>
| str(Point3()) → '(0.0,0.0,0.0)' | repr(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 repr()
- Could get away with just one
  * __str__ requires __repr__
  * __str__ can use __repr__
    (if __str__ is not there)
- class Point3(object):
  """Class for points in 3d space"
  --
  def __str__(self):
    """Returns string with contents"
    return "<Point3 x=%.0f, y=%.0f, z=%.0f>" % (self.x, self.y, self.z)
  def __repr__(self):
    """Returns unambiguous string"
    return str(self).class_name() + str(self)

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

<table>
<thead>
<tr>
<th>Class Time(object):</th>
</tr>
</thead>
</table>
| """Class to represent times of day."
DISTANCE ATTRIBUTES: |
| * hour: hour of day [int in 0..23] |
| * minute of hour [int in 0..59]""" |
| def __init__(self, hour, min):
  """The time hours: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 int >= 0; mins in 0..59"
| def __repr__(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 an Initializer

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

Implementing a Method

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

Enforce Method Preconditions with `assert`

```python
class Time(object):
    '''Instances represent times of day.''
    def __init__(self, hour, min):
        # The time hour:min.
        # Pre: hour in 0..23; min in 0..59
        assert 0 <= min and min < 60
        assert hour >= 0 and
        assert type(hour) == int
        assert type(min) == int
        hour += self.__init__
        self._hour += hours,
        self._min += mins
        assert 0 <= min and min < 60
        assert 0 <= hour and hour < 24
        assert type(hour) == int
        assert type(min) == int
        self._hour = h
        self._min = m
        assert value >= 0 and value < 24
        self._hour = h
        self._min = m
        assert self._hour < 24
        self._hour = h
        self._min = m
```

Hiding Methods From Access

- Put underscore in front of a method to make it `hidden`
  - Will not show up in `help`
  - But it is still there
- Hidden methods
  - Can be used as `helpers`
  - 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

Enforcing Invariants

```python
class Time(object):
    '''INSTANCE ATTRIBUTES:
    hour: the hour  [int in 0..23]
    min: the minute  [int in 0..59]
    '''
    # Invariants:
    # Properties that are always true:
    # These are just comments!
    >>> t = Time(3,30)
    >>> t.hour = 5
    >>> How do we prevent this?
    # Idea: Restrict direct access
    # Only access via methods
    # Use asserts to enforce them
    # Example:
    def setHour(self, h):
        '''Sets hour to h
        Pre: h is an int in 0..23
        assert 0 <= h and h < 24
        self._hour = d
        '''
        assert 0 <= h and h < 24
        self._hour = h
```

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; x = 3*t.hour</td>
<td>&gt;&gt;&gt; x = 3*t.getHour()</td>
</tr>
<tr>
<td><strong>Good:</strong></td>
<td><strong>Good:</strong></td>
</tr>
</tbody>
</table>

Data Encapsulation

```python
class Time(object):
    '''INSTANCE ATTRIBUTES:
    hour: the hour  [int in 0..23]
    min: the minute  [int in 0..59]
    '''
    def getHour(self):
        '''Returns: the hour'''
        return self._hour
```

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><strong>Example:</strong> turtle.color</td>
<td><strong>Example:</strong> turtle.x</td>
</tr>
<tr>
<td>Has both getters and setters</td>
<td>Has only a getter</td>
</tr>
<tr>
<td><em>Sets allow you to change</em></td>
<td><em>No setter means no change</em></td>
</tr>
<tr>
<td><em>Enforce invariants w/ asserts</em></td>
<td><em>Getter allows limited access</em></td>
</tr>
</tbody>
</table>

May ask you to differentiate on the exam