Write your code to make these statements true!

To make (1) precise: write a

- **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)'

- **Usage**: `repr(expression)`
  - Evaluates the expression
  - Converts it into a string
- **How does it convert?**
  - `repr(2)` → '2'
  - `repr(True)` → 'True'
  - `repr('True')` → 'True'
  - `repr(Point3())` → 'class Point3>(0.0,0,0,0)'

---

### Example: Converting Values to Strings

<table>
<thead>
<tr>
<th><code>str()</code> Function</th>
<th><code>repr()</code> Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage: str(expression)</td>
<td>Usage: repr(expression)</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)' | - repr(Point3()) → 'class Point3>(0.0,0,0,0)'

---

### 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 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 Method

**Method Signature**
- **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

**Implementation**
- `def increment(self, hours, mins):
  ...`

**Method Body**
- ```python
  self.min = self.min + mins
  self.hour = 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 type(min) == int
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        self._hour = d
        self._min = m
    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours in int >= 0; mins in 0..59"
        assert 0 <= mins and mins < 60
        assert type(mins) == int
        assert 0 <= hours and hours < 24
        assert type(hours) == int
        new_hours = self._hour + hours
        new_mins = self._min + mins
        if new_mins >= 60:
            new_mins -= 60
            new_hours += 1
        self._hour = new_hours
        self._min = new_mins
    def is_minute(self, m):
        """Return: True if m valid minute"
        return 0 <= m and m < 60
    def setHour(self, h):
        """Sets hour to value"
        assert 0 <= h and h < 24
        assert type(h) == int
        self._hour = h
    def getHour(self):
        """Returns: hour attribute"
        return self._hour
    def setMin(self, m):
        """Sets minute to value"
        assert 0 <= m and m < 60
        assert type(m) == int
        self._min = m
    def getMin(self):
        """Returns: min attribute"
        return self._min
    def __str__(self):
        """Returns: string representation of the Time"
        return f'{self._hour}:{self._min:02}

>>> t = Time(2,30)
>>> t.setMin(45)
>>> t.hour
2
>>> t.minute
45

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

Pre: hours is an int in 0..23
and mins is an int in 0..59

Initializer creates/initializes all of the instance attributes.

Asserts in initializer guarantee the initial values satisfy the invariant.

Asserts in other methods enforce
the method preconditions.
```

Hiding Methods From Access

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

**Example:**

```python
class Time(object):
    """INSTANCE ATTRIBUTES:
    hour: the hour [int in 0..23]
    min: the minute [int in 0..59]
    """
    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 type(min) == int
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        self._hour = d
        self._min = m
    def is_minute(self, m):
        """Return: True if m valid minute"
        return 0 <= m and m < 60
    def setHour(self, h):
        """Sets hour to value"
        assert 0 <= h and h < 24
        assert type(h) == int
        self._hour = h
    def getHour(self):
        """Returns: hour attribute"
        return self._hour
    def setMin(self, m):
        """Sets minute to value"
        assert 0 <= m and m < 60
        assert type(m) == int
        self._min = m
    def getMin(self):
        """Returns: min attribute"
        return self._min
    def __str__(self):
        """Returns: string representation of the Time"
        return f'{self._hour}:{self._min:02}

>>> t = Time(2,30)
>>> t.setMin(45)
>>> t.hour
2
>>> t.minute
45
```

Enforcing Invariants

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

**Example:**

```python
class Time(object):
    """INSTANCE ATTRIBUTES:
    hour: the hour [int in 0..23]
    min: the minute [int in 0..59]
    """
    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 type(min) == int
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        self._hour = d
        self._min = m
    def is_minute(self, m):
        """Return: True if m valid minute"
        return 0 <= m and m < 60
    def setHour(self, h):
        """Sets hour to value"
        assert 0 <= h and h < 24
        assert type(h) == int
        self._hour = h
    def getHour(self):
        """Returns: hour attribute"
        return self._hour
    def setMin(self, m):
        """Sets minute to value"
        assert 0 <= m and m < 60
        assert type(m) == int
        self._min = m
    def getMin(self):
        """Returns: min attribute"
        return self._min
    def __str__(self):
        """Returns: string representation of the Time"
        return f'{self._hour}:{self._min:02}

>>> t = Time(2,30)
>>> t.setMin(45)
>>> t.hour
2
>>> t.minute
45
```

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; t.hour = 5</td>
<td>&gt;&gt;&gt; x = 5*t.hour</td>
</tr>
<tr>
<td><strong>Good:</strong></td>
<td><strong>Good:</strong></td>
</tr>
<tr>
<td>&gt;&gt;&gt; t.setHour(5)</td>
<td>&gt;&gt;&gt; x = 5*t.getHour()</td>
</tr>
</tbody>
</table>

<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>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>

<table>
<thead>
<tr>
<th>Naming Convention</th>
<th>Precondition is same as attribute invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should not access the attribute directly.</td>
<td>Do this for all of your attributes</td>
</tr>
</tbody>
</table>

Mutual vs. Immutable Attributes

<table>
<thead>
<tr>
<th>Data Encapsulation</th>
<th>Data Encapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do this for all of your attributes</td>
<td>Do this for all of your attributes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Getter</th>
<th>Setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>def getHour(self): &quot;&quot;&quot;Returns: hour attribute&quot;&quot; return self._hour</td>
<td>def setHour(self, h): &quot;&quot;&quot;Sets hour to h&quot;&quot; Pre: h is an int in 0..23&quot;&quot; assert type(h) == int assert 0 &lt;= h and h &lt; 24 self._hour = d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Do this for all of your attributes</th>
<th>Do this for all of your attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition is same as attribute invariant.</td>
<td>Precondition is same as attribute invariant.</td>
</tr>
</tbody>
</table>