Lecture 18

Using Classes Effectively
Announcements for Today

Exam Time

- Prelim, Nov 9th 5:15 or 7:30
  - Same break-up as last time
  - But will swap times assigned
- Material up to November 1
  - Review posted this weekend
  - Recursion + Loops + Classes
- Conflict with Prelim time?
  - Prelim 2 Conflict on CMS
  - Submit by next Thursday
  - SDS students must submit!

Assignments

- A4 is due tonight!
  - Survey is still open
- A5 was posted yesterday
  - Shorter written assignment
  - Due Wednesday at Midnight
- A6 to be posted tomorrow
  - Due a week after prelim
  - Designed to take two weeks
  - Finish Task 3 before exam
Recall: The __init__ Method

def __init__(self, n, s, b):
    """Initializer: creates a Worker

Has last name n, SSN s, and boss b

Precondition: n a string, s an int in range 0..999999999, and b either a Worker or None.

self.lname = n
self.ssn = s
self.boss = b"""
Recall: The `__init__` Method

```python
def __init__(self, n, s, b):
    """Initializer: creates a Worker

    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in range 0..999999999, and b either a Worker or None.
    self.lname = n
    self.ssn = s
    self.boss = b"

w = Worker('Obama', 1234, None)
```

Are there other special methods that we can use?
### Example: Converting Values to Strings

<table>
<thead>
<tr>
<th><strong>str() Function</strong></th>
<th><strong>repr() Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong> <code>str(&lt;expression&gt;)</code></td>
<td><strong>Usage:</strong> <code>repr(&lt;expression&gt;)</code></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><strong>How does it convert?</strong></td>
<td><strong>How does it convert?</strong></td>
</tr>
<tr>
<td>- <code>str(2) → '2'</code></td>
<td>- <code>repr(2) → '2'</code></td>
</tr>
<tr>
<td>- <code>str(True) → 'True'</code></td>
<td>- <code>repr(True) → 'True'</code></td>
</tr>
<tr>
<td>- <code>str('True') → 'True'</code></td>
<td>- <code>repr('True') → '&quot;True&quot;'</code></td>
</tr>
<tr>
<td>- <code>str(Point3()) → '(0.0,0.0,0.0)'</code></td>
<td>- <code>repr(Point3()) → &quot;&lt;class 'Point3'&gt; (0.0,0.0,0.0)&quot;</code></td>
</tr>
</tbody>
</table>
### 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)'`

**repr() Function**

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

What type is this value? The value’s type is clear
What Does \texttt{str()} Do On Objects?

- **Does NOT** display contents
  
  ```python
  >>> p = Point3(1,2,3)
  >>> str(p)
  '<Point3 object at 0x1007a90>'
  ```

- **Must add a special method**
  - \_\_\_\texttt{str}\_\_ for \texttt{str()}  
  - \_\_\_\texttt{repr}\_\_ for \texttt{repr()}  

- **Could get away with just one**
  - \texttt{repr()} requires \_\_\_\texttt{repr}\_\_
  - \texttt{str()} can use \_\_\_\texttt{repr}\_\_
    (if \_\_\_\texttt{str}\_\_ is not there)

```python
class Point3(object):
    """Class for points in 3d space""
    ...
    def \_\_\_str\_\_(self):
        """Returns: string with contents""
        return '('+str(self.x)+','+
                 str(self.y)+','+
                 str(self.z)+')'

    def \_\_\_repr\_\_(self):
        """Returns: unambiguous string""
        return str(self.__class__)+str(self)
```
What Does `str()` Do On Objects?

- **Does NOT** display contents
  ```python
g>> p = Point3(1,2,3)
g>> str(p)
'<Point3 object at 0x1007a90>'
```
- **Must add a special method**
  - `__str__` for `str()`
  - `__repr__` for `repr()`
- **Could get away with just one**
  - `repr()` requires `__repr__`
  - `str()` can use `__repr__` (if `__str__` is not there)

```python
class Point3(object):
    """Class for 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)

  __repr__ using __str__ as helper
```
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, …)
  - 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
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

```python
class Time(object):
    """Class to 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.
class Rectangle(object):
    """Class to represent rectangular region
    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."""
Planning out a Class

```python
class Rectangle(object):
    
    """Class to represent rectangular region

    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.

Special invariant relating attributes to each other

Method Specification

States what the method does.
Gives preconditions stating what is assumed true of the arguments.
**Planning out a Class**

```python
class Hand(object):
    """Instances represent a hand in cards.
    INSTANCE ATTRIBUTES:
        cards: cards in the hand [list of card]
    This list is sorted according to the ordering defined by the Card class."

    def __init__(self, deck, n):
        """Draw a hand of n cards.
        Pre: deck is a list of >= n cards"

    def isFullHouse(self):
        """Return: True if this hand is a full house; False otherwise"

    def discard(self, k):
        """Discard the k-th card."
```

**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 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

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

You put code here

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]

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

This is true to start
What we are supposed to accomplish
This is also true to start
You put code here

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]

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 = self.min + mins
    self.hour = (self.hour + hours +
                 self.min // 60)
    self.min = self.min % 60
    self.hour = self.hour % 24

This is true to start
What we are supposed to accomplish
This is also true to start
You put code here
This should be true at the end
Role of Invariants and Preconditions

- They both serve two purposes
  - Help you think through your plans in a disciplined way
  - Communicate to the user* how they are allowed to use the class
- Provide the *interface* of the class
  - interface btw two programmers
  - interface btw parts of an app
- Important concept for making large software systems

* ...who might well be you!

inter•face ˈɪntərˌfæs noun
1. point where two systems, subjects, organizations, etc., meet and interact: the interface between accountancy and the law.
   - chiefly Physics a surface forming a common boundary between two portions of matter or space, e.g., between two immiscible liquids: the surface tension of a liquid at its air/liquid interface.
2. Computing a device or program enabling a user to communicate with a computer.
   - a device or program for connecting two items of hardware or software so that they can be operated jointly or communicate with each other.

—The Oxford American Dictionary
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

Easy(ish) if we are the user. But what if we aren’t?
Recall: Enforce Preconditions with `assert`

```python
def anglicize(n):
    
    """Returns: the anglicization of int n.
    Precondition: n an int, 0 < n < 1,000,000"""
    assert type(n) == int, str(n)+' is not an int'
    assert 0 < n and n < 1000000, str(n)+' is out of range'

    # Implement method here...
```

Check (part of) the precondition

(Optionalal) Error message when precondition violated
Enforce Method Preconditions with `assert`

```python
class Time(object):
    """Class to represent times of day."""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60

    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"""
        assert type(hour) == int
        assert type(min) == int
        assert hour >= 0
        assert 0 <= min and min < 60

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

- 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

```python
class Time(object):
    """INSTANCE ATTRIBUTES:
    hour: the hour [int in 0..23]
    min: the minute [int in 0..59]""

    def __is_minute__(self, m):
        """Return: True if m valid minute""
        return (type(m) == int and m >= 0 and m < 60)

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

10/25/18 Using Classes Effectively 23
Enforcing Invariants

```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

    def setHour(self, value):
        """Sets hour to value"""
        assert type(value) == int
        assert value >= 0 and value < 24
        self.numerator = value

Invariants:
Properties that are always true.

• These are just comments!
  >>> t = Time(2,30)
  >>> t.hour = 'Hello'

• How do we prevent this?

• Idea: Restrict direct access
  ▪ Only access via methods
  ▪ Use asserts to enforce them

• Example:
  def getHour(self):
      """Returns: the hour"""
      return self.hour

  def setHour(self, value):
      """Sets hour to value"""
      assert type(value) == int
      assert value >= 0 and value < 24
      self.numerator = value
```

10/25/18 Using Classes Effectively 24
### Data Encapsulation

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

#### Setter Method
- Used to change an attribute
- Replaces all assignment statements to the attribute
- **Bad**: 
  
  ```python
  >>> t.hour = 5
  ```

- **Good**: 
  
  ```python
  >>> f.setHour(5)
  ```

#### Getter Method
- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**: 
  
  ```python
  >>> x = 3*t.hour
  ```

- **Good**: 
  
  ```python
  >>> x = 3*t.getHour()
  ```
class Time(object):

    """""""INSTANCE ATTRIBUTES:
    _hour: the hour [int in 0..23]
    _min: the minute [int in 0..59]"

    def getHour(self):
        """"Returns: hour attribute"
        return self._hour

    def setHour(self, h):
        """"Sets hour to h
        Pre: h is an int in 0..23"
        assert type(h) == int
        assert 0 <= h and h < 24
        self._hour = d

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

Precondition is same as attribute invariant.
## 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>▪ 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>

May ask you to differentiate on the exam
mutable vs. immutable attributes

mutable

• can change value directly
  ▪ if class invariant met
  ▪ example: turtle.color
• has both getters and setters
  ▪ setters allow you to change
  ▪ enforce invariants

immutable

• can’t change value directly
  ▪ may change “behind scenes”
  ▪ example: turtle.x
• has only a getter
  ▪ no setter means no change
  ▪ getter allows limited access

may ask you to differentiate on the exam

where?
next time.
Exercise: Design a (2D) Circle

• What are the attributes?
  ▪ What is the bare minimum we need?
  ▪ What are some extras we might want?
  ▪ What are the invariants?

• What are the methods?
  ▪ With just the one circle?
  ▪ With more than one circle?