Lecture 17

Using Classes Effectively

Important!

YES

```
class Point(object):
    """Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    …
```

3.0-Style Classes
Well-Designed

NO

```
class Point:
"""Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    …
```

"Old-Style" Classes
Very, Very Bad

Designing Types

From first day of class!

- **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 attributes? What values can they have?
2. Think about what operations you want
   - 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 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.

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, 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.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
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

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

This is true to start

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

You put code here

```
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

Implementing a Method

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

This is true to start

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

What we are supposed to accomplish

This is also true to start

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

You put code here

```
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

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
  - Will return to this idea in a week

*** …who might well be you!**

in•ter•face |ˈintərˌfās| noun

1. a 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 precondi…
  2. Assume class inv…
  3. Ensure method sp…
  4. Ensure class invariant … ue when done

Easy(ish) if we are the user.

But what if we aren't?

- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Recall: Enforce Preconditions with **assert**

```
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

(Optional) Error message when precondition violated

Enforce Method Preconditions with assert

```
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 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 and
        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.

What About Attributes?

- User can access instance attributes via assignment
- Example:

```
>>> t = Time(2,45)
>>> t.min = 70
```

Invariant violation!

- Nothing we can do
  - Wrote methods assuming invariant always true
  - Our enforcement code is all in method definitions

```
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"""
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60
```

Only protects inside initializer

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**:

```
>>> t.min = 55
```

- **Good**:

```
>>> t.setMin(55)
```

Getter Method

- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**:

```
>>> h = 60*t.min
```

- **Good**:

```
>>> h = 60*t.getMin()
```

Data Encapsulation

```
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 getMin(self):
        """Returns: min attribute"""
        return self._min

    def setMin(self, mins):
        """Alters min attribute to be mins
        Pre: mins is in 0..59"""
        assert type(mins) == int
        assert 0 <= mins and mins < 60
        self._min = mins
```

Getter

Setter

Do this for all of your attributes

Naming Convention

The underscore means "should not access the attribute directly."

Precondition is same as attribute invariant.