Lecture 18

Using Classes Effectively

Announcements for This Lecture

Assignments

- **A4** Due Thursday at midnight
  - Hopefully you are on Task 4
  - Extra consultants available
- Will post **A5** on Thursday
  - Written assignment like A2
  - Needs material from next Tues
- Will post **A6** on *Saturday*
  - Not due until November 15
  - Want to avoid exam crunch

Lab this Week

- Simple class exercise
  - Fill in predefined methods
  - Setting you up for A6…

Exams

- Still in the handback room
  - Located in Gates 216
  - Open 12-4:30 daily
- Regrades still open this week

Recall: The __init__ Method

two underscores

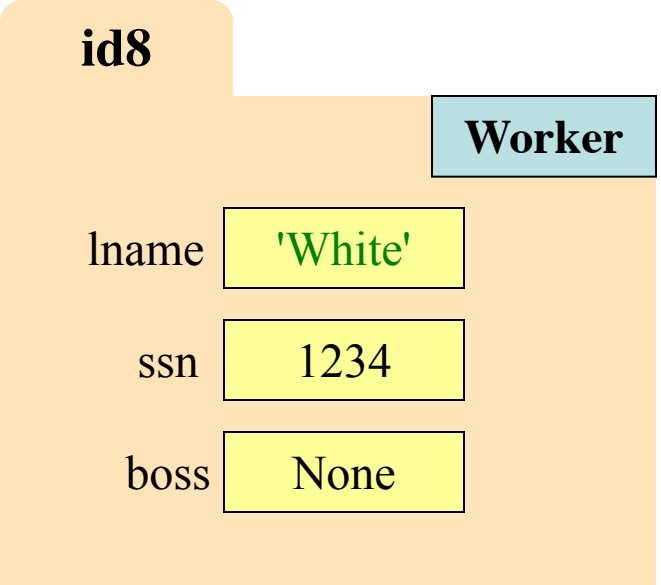
```
w = Worker('Obama', 1234, None)
```

```
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
```

Called by the constructor



Recall: The __init__ Method

two underscores

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

```
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
```

Are there other special methods that we can use?

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
- How does it convert?
  - repr(2) → '2'
  - repr(True) → 'True'
  - repr('True') → "'True'"
  - repr(Point3()) → "<class 'Point3'> (0.0,0.0,0.0)"

Example: Converting Values to Strings

str() Function

- **Usage**: str(<expression>)
  - Evaluates the expression
  - Converts it into a string
- How does it con

What type is this value?

  - str(2) → '2'
  - str(True) → 'True
  - str('True') → 'True'
  - str(Point3()) → '(0.0,0.0,0.0)'

repr() Function

-

repr() is for *unambigious* representation

How does it co

The value's type is clear

  - repr(2) → '2'
  - repr(True) →
  - repr('True') → "'True'"
  - 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
  - repr() 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 '('+self.x + ',' +
                  self.y + ',' +
                  self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
                str(self)
```

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
  - repr() 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 '('+self.x + ',' +
                   self.y + ',' +
                   self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
               str(self)
```

Gives the class name

__repr__ using __str__ as helper

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 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):
    """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.

Planning out a Class

```
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.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
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 <

    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

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

?

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 + mins
        self.hour = (self.hour + hours +
                     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

*** …who might well be you!**

in•ter•face |ˈintərˌfās| 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 precondi
  2. Assume class inv
  3. Ensure method sp
  4. Ensure class invariant 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):
    """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

```
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)
        ...
```

HIDDEN

Helper method

Enforcing Invariants

```
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(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
```

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.hour = 5
```

- **Good**:

```
>>> f.setHour(5)
```

Getter Method

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

```
>>> x = 3*t.hour
```

- **Good**:

```
>>> x = 3*t.getHour()
```

Data Encapsulation

```
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
```

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.

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 w/ asserts

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 differetiate 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
  - Enforce invar

Where?
Next time.

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 differetiate on the exam

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?