

CS 1110

Prelim 2 Review
Fall 2013

Exam Info

- Prelim 2: 7:30–9:00PM, Thursday, Nov. 14th
 - Last name **A – G** in Olin 155
 - Last name **H – K** in Olin 165
 - Last name **L – R** in Olin 255
 - Last name **S – Z** in Upson B17
- To help you study:
 - Study guides, review slides are online
 - Review solution to prelim 1 (esp. call stack!)
- Grades will be released before next class

What is on the Exam?

- Five questions from the following topics:
 - Recursion (Lab 7, A4)
 - Iteration and Lists (Lab 6, A4, A5)
 - Defining classes (Lab 8, Lab 9, A5)
 - Drawing folders (Lecture, Study Guide)
 - Exceptions (Lectures 10 and 20)
 - Short Answer (Terminology, Potpourri)
- +2 points for name, netid **AND SECTION**

If You Study the Past Prelims

- Past prelims are not a good example
- **Fall 2012** has all the right questions but...
 - We will not have properties on this exam
 - Folders are drawn **completely different**
 - The recursion is too easy (look at **Final** for 2012FA)
- **Spring 2013** has better recursion, for-loops but...
 - It includes loop invariants (those will be on final)
 - It is one question too short (no very easy questions)

What is on the Exam?

- Recursion (Lab 7, A4)
 - Will be given a function specification
 - Implement it using recursion
 - May have an associated call stack question
- Iteration and Lists (Lab 6, A4, A5)
- Defining classes (Lab 8, Lab 9, A5)
- Drawing folders (Lecture, Study Guide)
- Exceptions (Lectures 10 and 20)
- Short Answer (Terminology, Potpourri)

Recursive Function

def merge(s1,s2):

"""Returns: characters of s1 and s2, in alphabetical order.

Examples: merge('ab', '') = 'ab'

merge('abbce', 'cdg') = 'abbccdeg'

Precondition: s1 a string with characters in alphabetical order

s2 a string with characters in alphabetical order"""

Recursive Function

def merge(s1,s2):

"""Returns: characters of s1 and s2, in alphabetical order.

Examples: merge('ab', '') = 'ab'

merge('abbce', 'cdg') = 'abbccdeg'

Precondition: s1 a string with characters in alphabetical order
s2 a string with characters in alphabetical order"""

Hint:

- Make input “smaller” by pulling off first letter
- Only make **one** of two strings smaller each call
- Which one should you make smaller each call?

Recursive Function

```
def merge(s1,s2):
```

```
    """Returns: characters of s1 and s2, in alphabetical order. """
```

```
    if s1 == ":
```

```
        return s2
```

```
    if s2 == ":
```

```
        return s1
```

```
    if s1[0] < s2[0]:          # Pick first from s1 and merge the rest
```

```
        return s1[0]+merge(s1[1:],s2)
```

```
    else:                    # Pick first from s1 and merge the rest
```

```
        return s2[0]+merge(s1,s2[1:])
```


Call Stack Question

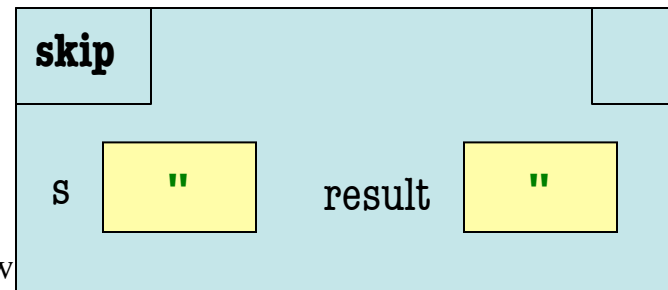
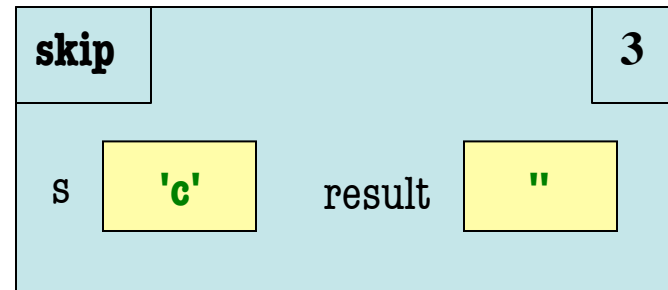
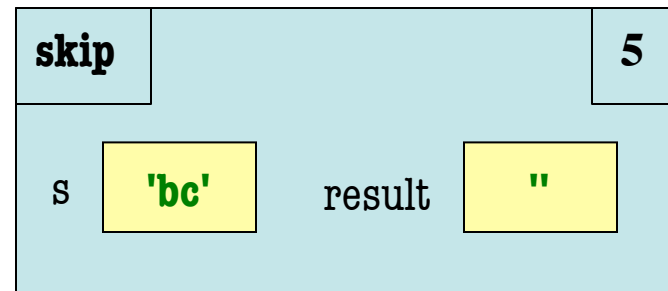
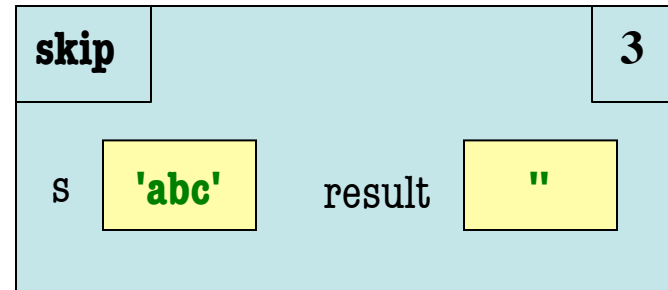
```
def skip(s):  
    """Returns: copy of s  
    Odd letters dropped"""  
1  result = ""  
2  if (len(s) % 2 == 1):  
3      result = skip(s[1:])  
4  elif len(s) > 0:  
5      result = s[0]+skip(s[1:])  
6  return result
```

- **Call:** skip('abc')
- Recursive call results in four frames (why?)
 - Consider when 4th frame completes line 6
 - Draw the entire call stack at that time
- Do not draw more than four frames!

Call Stack Question

- **Call:** skip('abc')

```
def skip(s):  
    """Returns: copy of s  
    Odd letters dropped"""  
1   result = ""  
2   if (len(s) % 2 = 1):  
3       result = skip(s[1:])  
4   elif len(s) > 0:  
5       result = s[0]+skip(s[1:])  
6   return result
```



Call Stack Question

- **Call:** skip('abc')

```
def skip(s):
```

```
    """Returns: copy of s
```

```
    Odd letters dropped"""
```

```
1 result = ""
```

```
2 if (len(s) % 2 = 1):
```

```
3     result = skip(s[1:])
```

```
4 elif len(s) > 0:
```

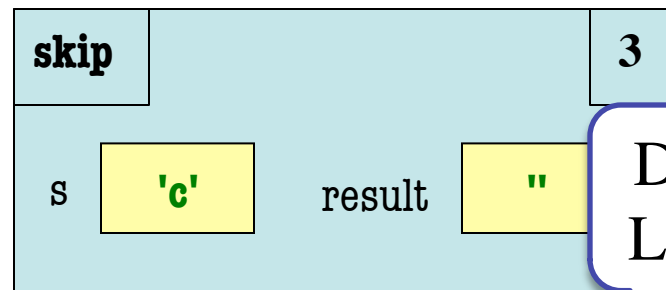
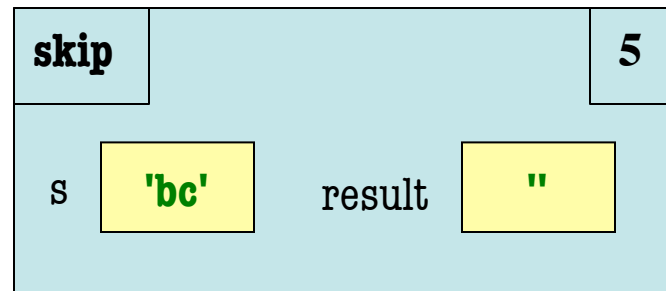
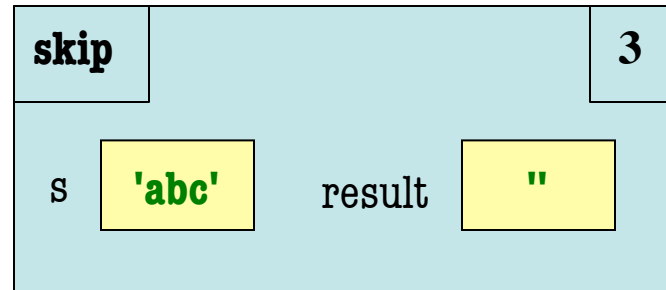
```
5     result = s[0]+skip(s[1:])
```

```
6 return result
```

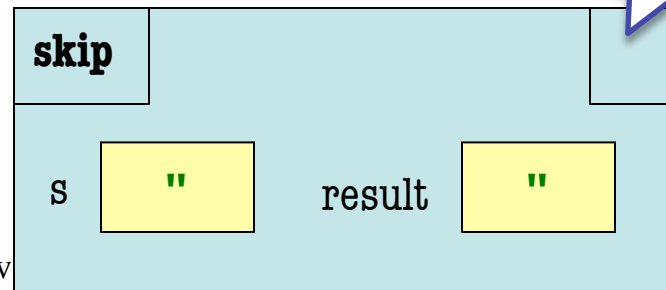
s = 'abc'
s = 'c'

s = 'bc'

s = ""



Done
Line 6



What is on the Exam?

- Recursion (Lab 7, A4)
- Iteration (Lab 6, A4, A5)
 - Again, given a function specification
 - Implement it using a for-loop
 - May involve 2-dimensional lists
- Defining classes (Lab 8, Lab 9, A5)
- Drawing folders (Lecture, Study Guide)
- Exceptions (Lectures 10 and 20)
- Short Answer (Terminology, Potpourri)

Implement Using Iteration

def evaluate(p, x):

"""Returns: The evaluated polynomial p(x)

We represent polynomials as a list of floats. In other words

[1.5, -2.2, 3.1, 0, -1.0] is $1.5 - 2.2x + 3.1x^2 + 0x^3 - x^4$

We evaluate by substituting in for the value x. For example

evaluate([1.5, -2.2, 3.1, 0, -1.0], 2) is $1.5 - 2.2(2) + 3.1(4) - 1(16) = -6.5$

evaluate([2], 4) is 2

Precondition: p is a list (len > 0) of floats, x is a float"""

Implement Using Iteration

```
def evaluate(p, x):
```

```
    """Returns: The evaluated polynomial p(x)
```

```
    Precondition: p is a list (len > 0) of floats, x is a float"""
```

```
    sum = 0
```

```
    xval = 1
```

```
    for c in p:
```

```
        sum = sum + c*xval    # coefficient * (x**n)
```

```
        xval = xval * x
```

```
    return sum
```

Example with 2D Lists (Like A5)

```
def max_cols(table):
```

```
    """Returns: Row with max value of each column
```

```
We assume that table is a 2D list of floats (so it is a list of rows and  
each row has the same number of columns. This function returns  
a new list that stores the maximum value of each column.
```

```
Examples:
```

```
    max_cols([ [1,2,3], [2,0,4], [0,5,2] ]) is [2,5,4]
```

```
    max_cols([ [1,2,3] ]) is [1,2,3]
```

```
Precondition: table is a NONEMPTY 2D list of floats"""
```

Example with 2D Lists (Like A5)

```
def max_cols(table):
```

```
    """Returns: Row with max value of each column
```

```
    Precondition: table is a NONEMPTY 2D list of floats"""
```

```
    # Use the fact that table is not empty
```

```
    result = table[0][:] # Make a copy, do not modify table.
```

```
    # Loop through rows, then loop through columns
```

```
    for row in table:
```

```
        for k in range(len(row))
```

```
            if row[k] > result[k]
```

```
                result[k] = row[k]
```

```
    return result
```


What is on the Exam?

- Recursion (Lab 7, A4)
- Iteration (Lab 6, A4, A5)
- Defining Classes (Lab 8, Lab 9, A5)
 - Given a specification for a class
 - Also given a specification for a subclass
 - Will “fill in blanks” for both
- Drawing folders (Lecture, Study Guide)
- Exceptions (Lectures 10 and 20)
- Short Answer (Terminology, Potpourri)

What is on the Exam?

- Recursion (Lab 7, A4)
 - Iteration (Lab 6, A4, A5)
 - Defining Classes (Lab 8, Lab 9, A5)
 - Given a specification of a class
 - Given a specification of a class
 - Given a specification of a class
 - Drawing folders (Lecture, Study Guide)
 - Exceptions (Lectures 10 and 20)
 - Short Answer (Terminology, Potpourri)
- There are **NO PROPERTIES** on exam

```
class Customer(object):
```

```
    """Instance is a customer for our company
```

```
    Mutable attributes:
```

```
        _name: last name [string or None if unknown]
```

```
        _email: e-mail address [string or None if unknown]
```

```
    Immutable attributes:
```

```
        _born: birth year [int > 1900; -1 if unknown]"""
```

Not the same
as properties

```
# DEFINE GETTERS/SETTERS HERE
```

```
# Enforce all invariants and enforce immutable/mutable restrictions
```

```
# DEFINE INITIALIZER HERE
```

```
# Initializer: Make a Customer with last name n, birth year y, e-mail address e.
```

```
# E-mail is None by default
```

```
# Precondition: parameters n, b, e satisfy the appropriate invariants
```

```
# OVERLOAD STR() OPERATOR HERE
```

```
# Return: String representation of customer
```

```
# If e-mail is a string, format is 'name (email)'
```

```
# If e-mail is not a string, just returns name
```

```
class Customer(object):
    """Instance is a customer for our company
    Mutable attributes:
        _name: last name [string or None if unknown]
        _email: e-mail address [string or None if unknown]
    Immutable attributes:
        _born: birth year [int > 1900; -1 if unknown]"""
```

```
# DEFINE GETTERS/SETTERS HERE
```

```
def getName(self):
    | return self._name
```

Getter

```
def setName(self,value):
    | assert value is None or type(value) == str
    | self._name = value
```

Setter

Actual Exam Question
will not be this long.
Just for this practice.

```
class Customer(object):
    """Instance is a customer for our company
    Mutable attributes:
        _name: last name [string or None if unknown]
        _email: e-mail address [string or None if unknown]
    Immutable attributes:
        _born: birth year [int > 1900; -1 if unknown]"""
```

```
# DEFINE GETTERS/SETTERS HERE
```

```
....
```

```
def getEmail(self):
    | return self._email
```

Getter

```
def setEmail(self,value):
    | assert value is None or type(value) == str
    | self._email = value
```

Setter

Actual Exam Question
will not be this long.
Just for this practice.

```
class Customer(object):
```

```
    """Instance is a customer for our company
```

```
    Mutable attributes:
```

```
        _name: last name [string or None if unknown]
```

```
        _email: e-mail address [string or None if unknown]
```

```
    Immutable attributes:
```

```
        _born: birth year [int > 1900; -1 if unknown]"""
```

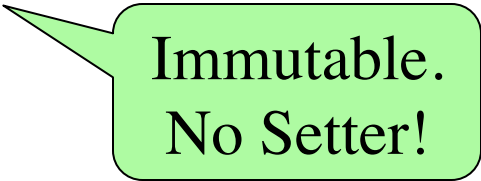
```
# DEFINE GETTERS/SETTERS HERE
```

```
....
```

```
def getBorn(self):  
    | return self._born
```



Getter



Immutable.
No Setter!

Actual Exam Question
will not be this long.
Just for this practice.

```

class Customer(object):
    """Instance is a customer for our company
    Mutable attributes:
        _name: last name [string or None if unknown]
        _email: e-mail address [string or None if unknown]
    Immutable attributes:
        _born: birth year [int > 1900; -1 if unknown]"""

    # DEFINE GETTERS/SETTERS HERE

    ...

    # DEFINE INITIALIZER HERE
    def __init__(self, n, y, e=None):
        assert type(y) == int and (y > 1900 or y == -1)
        self.setName(n) # Setter handles asserts
        self.setEmail(e) # Setter handles asserts
        self._born = y # No setter

```

Actual Exam Question
will not be this long.
Just for this practice.

```

class Customer(object):
    """Instance is a customer for our company
    Mutable attributes:
        _name: last name [string or None if unknown]
        _email: e-mail address [string or None if unknown]
    Immutable attributes:
        _born: birth year [int > 1900; -1 if unknown]"""

    # DEFINE GETTERS/SETTERS HERE

    ...

    # DEFINE INITIALIZER HERE

    ...

    # OVERLOAD STR() OPERATOR HERE
    def __str__(self):
        if self._email is None:
            | return = " if self._name is None else self._name
        else:
            | s = " if self._name is None else self._name
            | return s+'('+self._email+')'

```

Actual Exam Question
will not be this long.
Just for this practice.

None or str

If not None,
always a str


```

class PrefCustomer(Customer):
    """An instance is a 'preferred' customer
    Mutable attributes (in addition to Customer):
        _level: level of preference [One of 'bronze', 'silver', 'gold'] """

    # DEFINE GETTERS/SETTERS HERE
    # Enforce all invariants and enforce immutable/mutable restrictions

    # DEFINE INITIALIZER HERE
    # Initializer: Make a new Customer with last name n, birth year y,
    # e-mail address e, and level l
    # E-mail is None by default
    # Level is 'bronze' by default
    # Precondition: parameters n, b, e, l satisfy the appropriate invariants

    # OVERLOAD STR() OPERATOR HERE
    # Return: String representation of customer
    # Format is customer string (from parent class) +', level'
    # Use __str__ from Customer in your definition

```

```
class PrefCustomer(Customer):
```

```
    """An instance is a 'preferred' customer
```

```
    Mutable attributes (in addition to Customer):
```

```
        _level: level of preference [One of 'bronze', 'silver', 'gold'] """
```

```
# DEFINE GETTERS/SETTERS HERE
```

```
def getLevel(self):
```

```
    return self._level
```

Getter

```
def setLevel(self,value):
```

```
    assert type(value) == str
```

```
    assert (value == 'bronze' or value == 'silver' or value == 'gold')
```

```
    self._level = value
```

Setter

Actual Exam Question
will not be this long.
Just for this practice.

```

class PrefCustomer(Customer):
    """An instance is a 'preferred' customer
    Mutable attributes (in addition to Customer):
        _level: level of preference [One of 'bronze', 'silver', 'gold'] """

    # DEFINE GETTERS/SETTERS HERE

    ...

    # DEFINE INITIALIZER HERE
    def __init__(self, n, y, e=None, l='bronze'):
        Customer.__init__(self,n,y,e)
        self.setLevel(l) # Setter handles asserts

    # OVERLOAD STR() OPERATOR HERE
    def __str__(self):
        return Customer.__str__(self)+' '+self._level

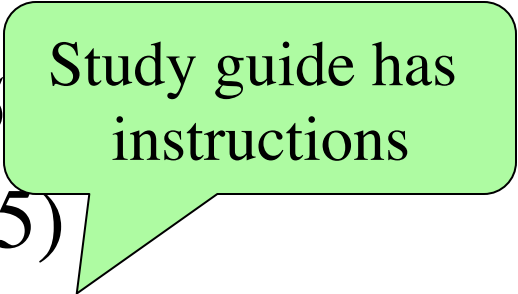
```

Actual Exam Question
will not be this long.
Just for this practice.

explicit calls uses method
in parent class as helper

What is on the Exam?

- Recursion (Lab 7, A4)
- Iteration and Lists (Lab 6, A4, A5)
- Defining classes (Lab 8, Lab 9, A5)
- **Drawing class folders (Lecture, Study Guide)**
 - Given a skeleton for a class
 - Also given several assignment statements
 - Draw all folders and variables created
- Exceptions (Lectures 10 and 20)
- Short Answer (Terminology, Potpourri)



Study guide has instructions

Two Example Classes

```
class Congressman(object):
    """Instance is legislator in congress
    Instance attributes:
        _name: Member's name [str]"""

    def getName(self):
        | return self._name

    def setName(self,value):
        | assert type(value) == str
        | self._name = value

    def __init__(self,n):
        | self.setName(n) # Use the setter

    def __str__(self):
        | return 'Honorable '+self.name
```

```
class Senator(CongressMember):
    """Instance is legislator in congress
    Instance attributes (plus inherited):
        _state: Senator's state [str]"""

    def getState(self):
        | return self._state

    def setName(self,value):
        | assert type(value) == str
        | self._name = 'Senator '+value

    def __init__(self,n,s):
        | assert type(s) == str and len(s) == 2
        | Congressman.__init__(self,n)
        | self._state = s

    def __str__(self):
        | return (Senator.__str__(self)+
        |         ' of '+self.state)
```

'Execute' the Following Code

```
>>> b = CongressMember('Jack')
>>> c = Senator('John', 'NY')
>>> d = c
>>> d.setName('Clint')
```

Remember:

Commands outside of
a function definition
happen in global space

- Draw two columns:
 - **Global space**
 - **Heap space**
- Draw both the
 - Variables created
 - Object folders created
 - Class folders created
- If an attribute changes
 - Mark out the old value
 - Write in the new value

Global Space

b

id1

c

id2

d

id2

Heap Space

id1

CongressMember

_name

'Jack'

id2

Senator

_name

~~'Senator John'~~

'Senator Clint'

_state

'NY'

CongressMember

__init__(n)

getName()

__str__()

setName(value)

Senator

__init__(n,s)

getState()

__str__()

setName(value)



Global Space

b

Instance attributes
in object folders

Methods and
class attributes
in class folders

Arrow to
superclass

CongressMember

```
__init__(n)    getName()  
__str__()     setName(value)
```

Heap Space

id1

CongressMember

_name 'Jack'

id2

Senator

_name ~~'Senator John'~~ 'Senator Clint'

state 'NY'

Senator

```
__init__(n,s)  getState()  
__str__()     setName(value)
```


Global Space

b

id1

c

id2

Methods parameters are **optional**

CongressMember

```
__init__(n)    getName()
__str__()     setName(value)
```

Heap Space

id1

CongressMember

_name

'Jack'

id2

Senator

_n

_s

Method **name** is enough for credit

Senator

```
__init__(n,s)  getState()
__str__()     setName(value)
```



Method Overriding

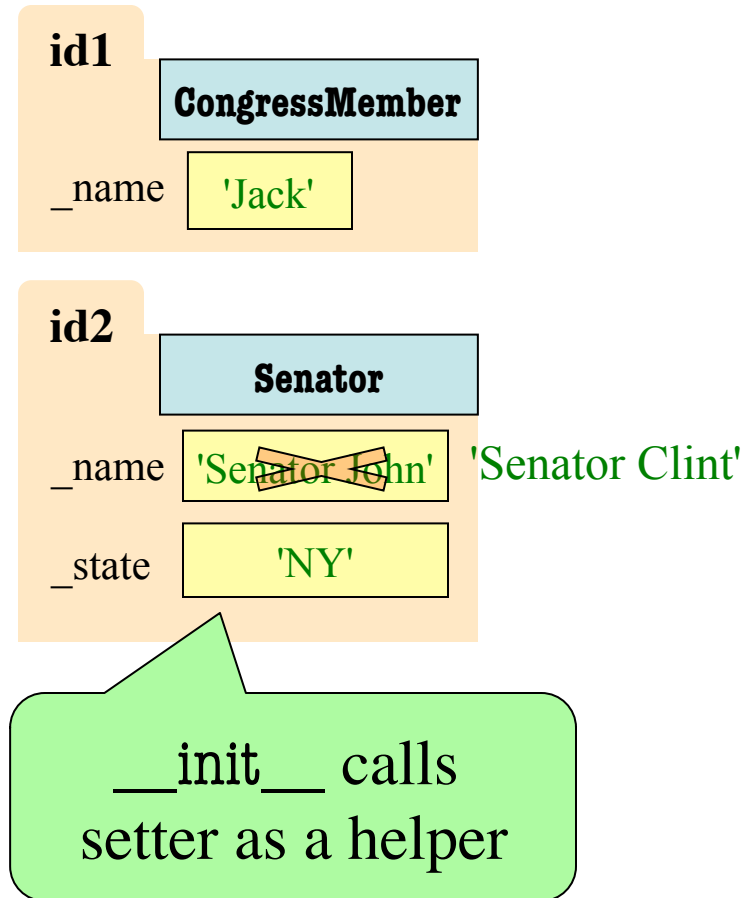
```
class Senator(CongressMember):
    """Instance is legislator in congress
    Instance attributes (plus inherited):
        _state: Senator's state [str]"""
    def getState(self):
        | return self._state

    def setName(self,value):
        | assert type(value) == str
        | self._name = 'Senator '+value

    def __init__(self,n,s):
        | assert type(s) == str and len(s) == 2
        | Senator.__init__(self,n)
        | self._state = s

    def __str__(self):
        | return (Senator.__str__(self)+
        |         ' of '+self.state)
```

Heap Space



What is on the Exam?

- Recursion (Lab 7, A4)
- Iteration and Lists (Lab 6, A4, A5)
- Defining classes (Lab 8, Lab 9, A5)
- Drawing class folders (Lecture, Study Guide)
- Exceptions (Lectures 10 and 20)
 - Try-except tracing (skipped on Prelim 1)
 - But now with dispatch on type
 - Will give you exception hierarchy
- Short Answer (Terminology, Potpourri)

Exceptions and Dispatch-On-Type

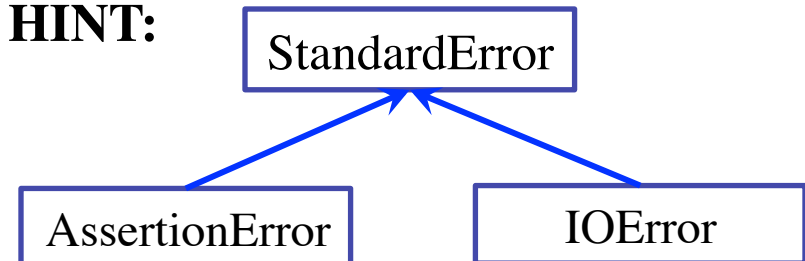
```
def first(x):  
    print 'Starting first.'  
    try:  
        second(x)  
    except IOError:  
        print 'Caught at first'  
    print 'Ending first'
```

```
def second(x):  
    print 'Starting second.'  
    try:  
        third(x)  
    except AssertionError:  
        print 'Caught at second'  
    print 'Ending second'
```

```
def third(x):  
    print 'Starting third.'  
    if x < 0:  
        raise IOError()  
    elif x > 0:  
        raise AssertionError()  
    print 'Ending third.'
```

What is the output of first(-1)?

HINT:



Exceptions and Dispatch-On-Type

```
def first(x):  
    print 'Starting first.'  
    try:  
        second(x)  
    except IOError:  
        print 'Caught at first'  
    print 'Ending first'
```

```
def second(x):  
    print 'Starting second.'  
    try:  
        third(x)  
    except AssertionError:  
        print 'Caught at second'  
    print 'Ending second'
```

```
def third(x):  
    print 'Starting third.'  
    if x < 0:  
        raise IOError()  
    elif x > 0:  
        raise AssertionError()  
    print 'Ending third.'
```

What is the output of first(-1)?

```
Starting first.  
Starting second.  
Starting third.  
Caught at first.  
Ending first.
```

Exceptions and Dispatch-On-Type

```
def first(x):  
    print 'Starting first.'  
    try:  
        second(x)  
    except IOError:  
        print 'Caught at first'  
    print 'Ending first'
```

```
def second(x):  
    print 'Starting second.'  
    try:  
        third(x)  
    except AssertionError:  
        print 'Caught at second'  
    print 'Ending second'
```

```
def third(x):  
    print 'Starting third.'  
    if x < 0:  
        raise IOError()  
    elif x > 0:  
        raise AssertionError()  
    print 'Ending third.'
```

What is the output of first(1)?

Exceptions and Dispatch-On-Type

```
def first(x):  
    print 'Starting first.'  
    try:  
        second(x)  
    except IOError:  
        print 'Caught at first'  
    print 'Ending first'
```

```
def second(x):  
    print 'Starting second.'  
    try:  
        third(x)  
    except AssertionError:  
        print 'Caught at second'  
    print 'Ending second'
```

```
def third(x):  
    print 'Starting third.'  
    if x < 0:  
        raise IOError()  
    elif x > 0:  
        raise AssertionError()  
    print 'Ending third.'
```

What is the output of first(1)?

```
Starting first.  
Starting second.  
Starting third.  
Caught at second.  
Ending second.  
Ending first.
```

What is on the Exam?

- Recursion (Lab 7, A4)
 - Iteration and Lists (Lab 6, A4, A5)
 - Defining classes (Lab 8, Lab 9, A5)
 - Drawing class folders (Lecture, Study Guide)
 - Exceptions (Lectures 10 and 20)
 - Short Answer (Terminology, Potpourri)
 - See the study guide
 - Look at the lecture slides
 - Read relevant book chapters
- In that order

Any More Questions?

