61A Lecture 25

Friday, October 28
From Last Time: Adjoining to a Tree Set
From Last Time: Adjoining to a Tree Set

```
    5
   /|
  /  |
 3   9
 / \
1   7 11
```
From Last Time: Adjoining to a Tree Set

```
8

5

3 9

1 7 11
```
From Last Time: Adjoining to a Tree Set

Right!
From Last Time: Adjoining to a Tree Set

Right!
From Last Time: Adjoining to a Tree Set

Right!
From Last Time: Adjoining to a Tree Set

Right!  Left!
From Last Time: Adjoining to a Tree Set

```
  8
 / \   \\
 5   9   \
 |   |    \\
3   7   11
```

```
  8
 / \   \\
 9   7   11
```

Right!  Left!
From Last Time: Adjoining to a Tree Set

\[
\begin{array}{c}
8 \\
5 \\
3 \\
1 \\
7 \\
9 \\
7 \\
11 \\
8 \\
9 \\
11 \\
7 \\
None \\
None
\end{array}
\]

Right! Left!
From Last Time: Adjoining to a Tree Set

```
  8
 /|
/  |
5   9
 /  |
3   7
  1  11
```

```
  8
 /|
/  |
9   
 /  |
7   11
```

```
  8
     |
     \
     None
```

**Right!**  
**Left!**  
**Right!**
From Last Time: Adjoining to a Tree Set

Right!  Left!  Right!

Friday, October 28, 2011
From Last Time: Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

```
1 7 11
  \
  3 9
   \  \
    5

1 7 11
  \
  3 9
   \  \
    5

1 7 11
  \
  3 9
   \  \
    5

1 7 11
  \
  3 9
   \  \
    5
```
From Last Time: Adjoining to a Tree Set

```
8
  / \    / \    /   \   /
5   9  9   7  11/\  
3   1  7  11/   \
1     7     11
```

- Right!
- Left!
- Right!
- Stop!

Friday, October 28, 2011
From Last Time: Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

\[
\begin{array}{c}
8 \\
\downarrow \\
5 \\
\downarrow \\
3 & 9 \\
\downarrow & \downarrow \\
1 & 7 & 11
\end{array}
\begin{array}{c}
8 \\
\downarrow \\
9 \\
\downarrow \\
7 & 11
\end{array}
\begin{array}{c}
8 \\
\downarrow \\
7 \\
\downarrow \\
None & None
\end{array}
8
\]
From Last Time: Adjoining to a Tree Set

Right! Left! Right! Stop!

1 7 11

From Last Time: Adjoining to a Tree Set

Right! Left! Right! Stop!

1 7 11

None None
From Last Time: Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

Friday, October 28, 2011
From Last Time: Adjoining to a Tree Set

Right!  Left!  Right!  Stop!

Friday, October 28, 2011
From the Exam: Pruned Trees

a  
b  
c  
d
From the Exam: Pruned Trees

(a,b)  (a,c)  (a,d)

pruned | True | True | False
From the Exam: Pruned Trees

a

b

c

Friday, October 28, 2011
pruned(a, c)
pruned(a, c)
From the Exam: Pruned Trees

pruned(a, c)

implies
From the Exam: Pruned Trees

pruned(a, c)

implies

pruned(a.right, c.right)
pruned(a, c)

implies

pruned(a.right, c.right)
pruned(a, c) implies

pruned(a.right, c.right)
From the Exam: Pruned Trees

pruned(a, c) implies

pruned(a.right, c.right)
From the Exam: Pruned Trees

pruned(a, c)

implies

pruned(a.right, c.right)

what about c.left?
pruned(a, c) implies
pruned(a.right, c.right)

what about c.left?
From the Exam: Pruned Trees

pruned(a, c)

implies

pruned(a.right, c.right)

what about c.left?
From the Exam: Pruned Trees
From the Exam: Pruned Trees

pruned(a, d)
From the Exam: Pruned Trees

\[ \text{pruned}(a, d) \]
pruned(a, d)

would imply
From the Exam: Pruned Trees

pruned(a, d)

would imply

pruned(a.left, d.left)
From the Exam: Pruned Trees

pruned(a, d)

would imply

pruned(a.left, d.left)
pruned(a, d)

would imply

pruned(a.left, d.left)
pruned(a, d)

would imply

pruned(a.left, d.left)
From the Exam: Pruned Trees

pruned(a, d)

would imply

pruned(a.left, d.left)
pruned(a, d)

would imply

pruned(a.left, d.left)
From the Exam: Pruned Trees

\begin{figure}
\centering
\begin{tabular}{cccc}
\hspace{1cm} a & \hspace{1cm} b & \hspace{1cm} c & \hspace{1cm} d \\
\end{tabular}
\end{figure}
From the Exam: Pruned Trees

Recursive call: both branches are pruned as well
Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None
Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None

def pruned(t1, t2):

Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None

def pruned(t1, t2):
    if t2 is None:
def pruned(t1, t2):
    if t2 is None:
        return True

Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None

def pruned(t1, t2):
    if t2 is None:
        return True
From the Exam: Pruned Trees

Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None

```python
def pruned(t1, t2):
    if t2 is None:
        if t1 is None:
            return True
        return False
    return False
```
def pruned(t1, t2):
    if t2 is None:
        return True
    if t1 is None:
        return False
    Recursive call: both branches are pruned as well
    Base cases: one (or more) of the trees is None

    def pruned(t1, t2):
        if t2 is None:
            return True
        if t1 is None:
            return False
def pruned(t1, t2):
    if t2 is None:
        return True
    if t1 is None:
        return False
    return pruned(t1.left, t2.left) and pruned(t1.right, t2.right)

Recursive call: both branches are pruned as well

Base cases: one (or more) of the trees is None
Today's Topic: Handling Errors
Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect
Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) does not exist
Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) does not exist
- Network connections are lost
Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) does not exist
- Network connections are lost

Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer
Different Error Handling Policies
Different Error Handling Policies
Different Error Handling Policies
Different Error Handling Policies

Versus

Python 3.2 (r32:88452, Feb 20 2011, 11:12:31)  
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin  
Type "copyright", "credits" or "license()" for more information.  

>>> from math import sqrt  
>>> for value in map(sqrt, [4 - x for x in range(10)]):  
    print(value)

2.0
1.7320508075688772
1.4142135623730951
1.0
0.0

Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in <module>  
    for value in map(sqrt, [4 - x for x in range(10)]):  
  ValueError: math domain error
Exceptions
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing a crash
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

Mastering exceptions:
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

Mastering exceptions:

Exceptions are objects! They have classes with constructors.
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions.

Python *raises* an exception whenever an error occurs.

Exceptions can be *handled* by the program, preventing a crash.

Unhandled exceptions will cause Python to halt execution.

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

They enable *non-local* continuations of control.
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

They enable *non-local* continuations of control:

If \( f \) calls \( g \) and \( g \) calls \( h \), exceptions can shift control from \( h \) to \( f \) without waiting for \( g \) to return.
Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

Mastering exceptions:

Exceptions are objects! They have classes with constructors.

They enable non-local continuations of control:

If \( f \) calls \( g \) and \( g \) calls \( h \), exceptions can shift control from \( h \) to \( f \) without waiting for \( g \) to return.

However, exception handling tends to be slow.
Assert Statements

Assert statements raise an exception of type AssertionError
Assert Statements

Assert statements raise an exception of type AssertionError

assert <expression>, <string>
Assert Statements

Assert statements raise an exception of type AssertionError

```python
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.
**Assert Statements**

Assert statements raise an exception of type `AssertionError`

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```
Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```

Whether assertions are enabled is governed by a bool __debug__
Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```

Whether assertions are enabled is governed by a bool `__debug__`

Demo
Raise Statements
Raise Statements

Exceptions are raised with a raise statement.
Raise Statements

Exceptions are raised with a raise statement.

raise <expression>
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.
Raise Statements

Exceptions are raised with a raise statement.

\[
\text{raise } <\text{expression}>
\]

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.
Raise Statements

Exceptions are raised with a raise statement.

```python
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

**TypeError** — A function was passed the wrong number/type of argument
Raise Statements

Exceptions are raised with a raise statement.

\[ \texttt{raise <expression>} \]

\(<\texttt{expression}>\) must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

\texttt{TypeError} -- A function was passed the wrong number/type of argument

\texttt{NameError} -- A name wasn't found
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

**TypeError** — A function was passed the wrong number/type of argument

**NameError** — A name wasn't found

**KeyError** — A key wasn't found in a dictionary
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

- **TypeError** — A function was passed the wrong number/type of argument
- **NameError** — A name wasn't found
- **KeyError** — A key wasn't found in a dictionary
- **RuntimeError** — Catch-all for troubles during interpretation
Try Statements
Try Statements

Try statements handle exceptions
Try Statements

Try statements handle exceptions

```python
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...```

Execution rule:
Try Statements

Try statements handle exceptions

```python
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The `<try suite>` is executed first;
Try Statements

Try statements handle exceptions

```python
try:
    <try suite>
    except <exception class> as <name>:
        <except suite>
...
```

**Execution rule:**

The `<try suite>` is executed first;

If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and
Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The `<try suite>` is executed first;

If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and

If the class of the exception inherits from `<exception class>`, then
Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...```

Execution rule:

The `<try suite>` is executed first;

If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and

If the class of the exception inherits from `<exception class>`, then

The `<except suite>` is executed, with `<name>` bound to the exception
Handling Exceptions
Handling Exceptions

Exception handling can prevent a program from terminating
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
except ZeroDivisionError as e:
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0

handling a <class 'ZeroDivisionError'>
```
Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>

>>> x
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

**Multiple try statements:** Control jumps to the except suite of the most recent try statement that handles that type of exception.
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
    x = 0

 handling a <class 'ZeroDivisionError'>
>>> x
0
```

**Multiple try statements:** Control jumps to the except suite of the most recent try statement that handles that type of exception.

Demo
WWPD: What Would Python Do?

How will the Python interpreter respond?
WWPD: What Would Python Do?

How will the Python interpreter respond?
WWPD: What Would Python Do?

How will the Python interpreter respond?

```python
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

Friday, October 28, 2011
How will the Python interpreter respond?

```python
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)

>>> invert_safe(1/0)
```

---

Friday, October 28, 2011
WWPD: What Would Python Do?

How will the Python interpreter respond?

def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)

>>> invert_safe(1/0)

>>> try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')
How will the Python interpreter respond?

```python
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)

>>> invert_safe(1/0)
>>> try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')
>>> invert_safe(1/0)
```

**WWPD: What Would Python Do?**

Friday, October 28, 2011
Example: Safe Iterative Improvement
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function

• The **update** argument provides better guesses
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function

- The `update` argument provides better guesses
- The `done` argument indicates completion
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function

• The **update** argument provides better guesses
• The **done** argument indicates completion
• Used to implement Newton's method (find_root)
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function

• The `update` argument provides better guesses
• The `done` argument indicates completion
• Used to implement Newton's method (`find_root`)
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function
• The `update` argument provides better guesses
• The `done` argument indicates completion
• Used to implement Newton's method (`find_root`)

```python
def newton_update(f):
    """Return an update function for f using Newton's method.""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

```python
def cube_root_newton(x):
    """Return the cube root of x."
    return find_root(lambda y: y * y * y == x)
```

```python
def square_root_newton(x):
    """Return the square root of x."
    return find_root(lambda y: y * y == x)
```

```python
def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x.""
    df = f(x + delta)
    return df / delta
```

```python
def newton_update(f):
    """Return an update function for f using Newton's method.""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

```python
def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.""
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function
• The `update` argument provides better guesses
• The `done` argument indicates completion
• Used to implement Newton's method (find_root)

```python
def newton_update(f):
    """Return an update function for f using Newton's method.""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    ""
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

"""
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function
• The update argument provides better guesses
• The done argument indicates completion
• Used to implement Newton's method (find_root)

```python
def newton_update(f):
    """Return an update function for f using Newton's method.""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess."

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    ""
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```
Example: Safe Iterative Improvement

Iterative improvement is a higher-order function
• The **update** argument provides better guesses
• The **done** argument indicates completion
• Used to implement Newton's method (find_root)

```python
def newton_update(f):
    """Return an update function for f using Newton's method.""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess."

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```
Exception Chaining

The except suite of a try statement can raise another exception that adds additional information.

Demo