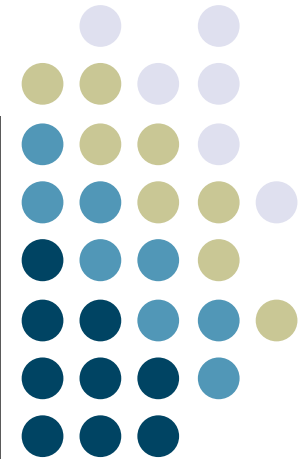


# Week 3: Quick and Dirty Jython Refresher

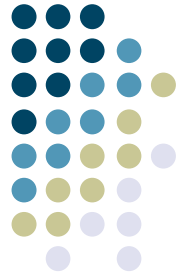
(Prelude to Asynchronous Programming)



**Georgia  
Tech**

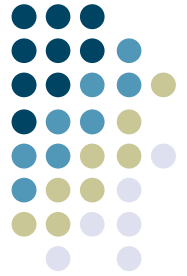


CS6452



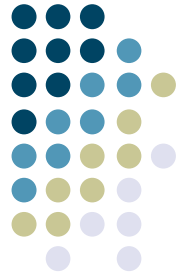
# Recap of Homework #1

- In general: great job
  - Where there were problems, most centered around insufficient detail
- A few points about the IM assignment
  - The IM protocol we'll be using doesn't support
    - Authentication/login
    - Sending messages to a user *before* that user joins the chat
    - Named, persistent chat rooms
    - Status changes ("available", "away")
    - Buddies
    - Adding people to existing chats
  - Some of these you can implement in your own client, even without server support
    - E.g., buffer messages sent to a user before he/she joins



## A Bit More Administrivia...

- Late policy for assignments:
  - Clear with me *first* if you have a valid excuse for missing a due date
    - Examples: medical or family emergency
  - My policy is -10% per late day, maximum 3 days late
- Grading criteria will be posted on the web for each assignment
  - Make sure you get the right one! IM versus Social Networking criteria
- Readings will be posted ~1 week in advance
  - So, next week's readings are already up
- In-class presentations
  - For each module we'll do a set of short in-class presentations
  - Sign-up is on the swiki for the GUI presentations (2/6/06)
- Please add yourself to the Swiki!

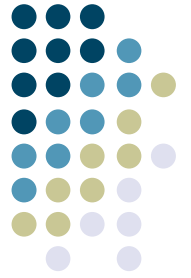


# Today's Menu

- Jython Refresher
  - Collection Classes
  - Scoping and Modules
  - Classes and Objects
  - GUI Programming
- Useful odds-and-ends that may come in handy for the next assignment

# Jython Collection Classes

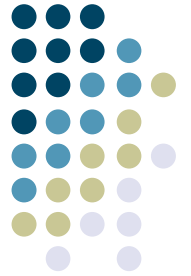
Georgia  
Tech





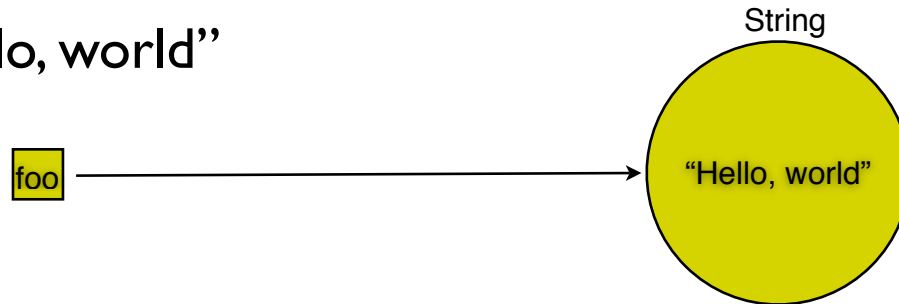
# Collections

- One of the strongest features of Python: powerful built-in data structures
- Let you organize and retrieve collections of data in ways that would be impractical if you had to stash every item in a variable
- Sequences
  - Lists
  - Tuples
- Dictionaries



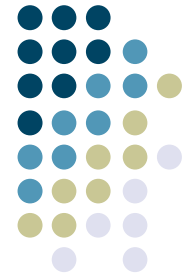
# Variables and References

- A variable is simply a name that contains a reference to some information
- `foo = "Hello, world"`

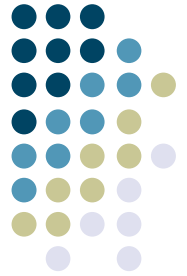


- Variables can be reassigned, and multiple variables can refer to the same thing.
- Stashing a reference in a variable gives you a way to name it, and get at it later.

# The Need for More Complex Data Structures

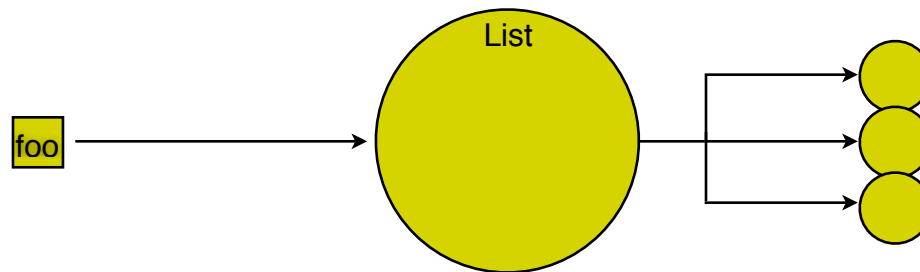


- Some more complex structures are hard to represent by just a named variable though.
- Example: you want to keep track of all of the users in a chat.
  - user1 = "Steven"
  - user2 = "Amy"
  - ...
- This is too *static*. Would you just create 1000 variables in case you ever had that many users? How would you do something to each one (can't easily iterate)

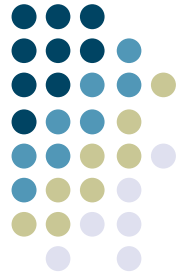


# Lists to the Rescue

- Fortunately, Python has a built-in way to do this: *lists*
- `foo = [ "one", "two", "three" ]`



- Lists collect multiple references to data items into a single data structure
- These references are *ordered*
- The contents of the list can be altered (it is *mutable*)
- `currentChatUsers = [ "Amy", "Steven", ... ]`



# A Quick Python List Refresher

- Lists are *ordered* collections of items

```
>>> L=[0,'zero','one', 1]
```

- Selecting items from a list (note indices start at 0!)

```
>>> print L[1]
```

```
'zero'
```

- Getting the length of a list

```
>>> len(L)
```

```
4
```

- Modifying lists

```
>>> L.append('two')
```

```
>>> L.remove('zero')
```

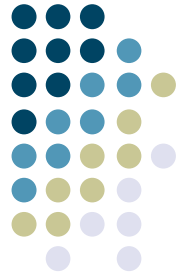
```
>>> print L
```

```
[0,'one', 1,'two']
```

- Iteration

```
for item in L:
```

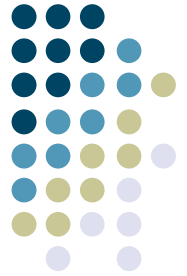
```
    print item
```



# Tuples: Fixed Sequences

- Like lists, only *immutable*
  - The set of references in a tuple is **fixed**
- Generally used either when:
  - You need a constant list
    - `daysOfWeek = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")`
  - You need to group together a set of data whose *structure* is fixed:
    - E.g., using tuples as quick-and-dirty records, such as address book entries:
      - `myContactInfo = ("Keith Edwards", "TSRB348", "keith@cc")`
- All list operations work on tuples, except ones that modify the set of references within the tuple
  - So, no `append()`, `remove()`, etc.

# Associating Data Items With Each Other

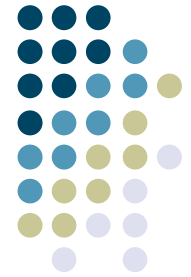


- Sometimes, you need to associate one item with another one
  - Example: hours worked on each day of the week:

"Sunday"	4.5
"Monday"	8
...	...

- You could do this with variables, as long as there's a fixed set of them:
  - `sunday=4.5`
  - `monday=8`

# Associating Data Items With Each Other (cont'd)

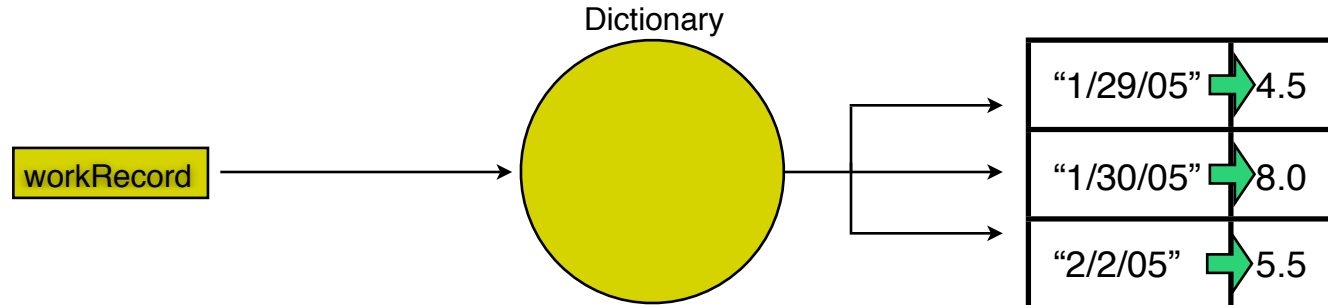


- If you don't know the associations you might have up front, you could use parallel lists:
  - `workDates = [ "1/29/05", "1/30/05", "2/1/05", ... ]`
  - `workHours = [ 4.5, 8, 5.5, ... ]`
- Then, iterate through the first list to find the date you're looking for, then look for the item with the corresponding index in the second list
- Too much work! Too error prone!
- Fortunately, Python has a built-in data structure for creating associations: the *dictionary*



# The Dictionary Data Structure

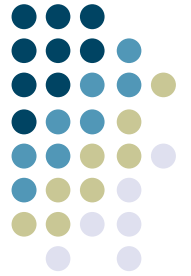
- Dictionaries associate *values* with *keys* (you *lookup* a value given its key)
- Both are references to data items
- `workRecord = { "1/29/05": 4.5, "1/30/05": 8, "2/2/05": 5.5 }`



- *Dictionaries are the most commonly used Jython data type*
- *Virtually any Jython data type can be used as a key or as a value*

# A Quick Python Dictionary Refresher

Georgia  
Tech



- Initializing a dictionary:  

```
>>> dict = {'one': 1, 'two': 2, 'three': 3}
```
- Looking up values:  

```
>>> dict["two"]
```

```
2
```
- Inserting and changing values:  

```
>>> dict["four"] = 4
```

```
>>> dict["two"] = 2222
```

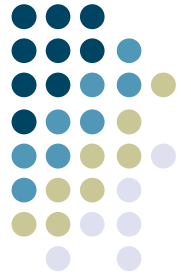
```
>>> dict
```

```
{'one': 1, 'two': 2222, 'three': 3, 'four': 4}
```
- Other operations:  

```
>>> del dict["one"]
```

```
>>> len(dict)
```

```
3
```



# More Dictionary Goodness

- Testing whether or not a dictionary has a given key

```
>> dict.has_key("two")
```

```
1
```

```
>> dict.has_key("five")
```

```
0
```

- Getting keys, values, and entire items

```
>> dict.keys()
```

```
["two", "three", "four"]
```

```
>> dict.values()
```

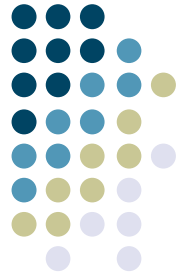
```
[2222, 3, 4]
```

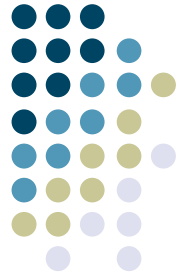
```
>> dict.items()
```

```
[("two", 2222), ("three", 3), ("four", 4)]
```

# Scoping and Modules

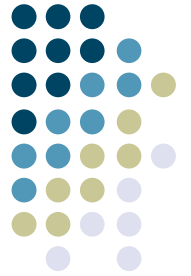
Georgia  
Tech





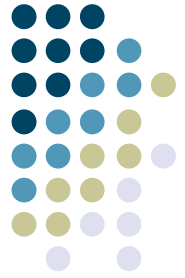
# Scoping

- What is scoping?
- Scoping is a fancy word that just means “the rules about what you can see from where” in a program
- The *namespace* is the collection of stuff that you can see from any given point in a program



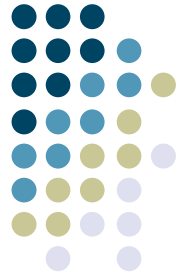
## An Example: Scoping Error

- `welcomeMsg = "Hello!"`
- `def changeWelcomeMsg():`
  - `welcomeMsg = "Bonjour!"`
  - `print "New welcome message is", welcomeMsg`
- `changeWelcomeMsg()`
- `>>> New welcome message is Bonjour!`
- `print welcomeMsg`
- `"Hello!"`



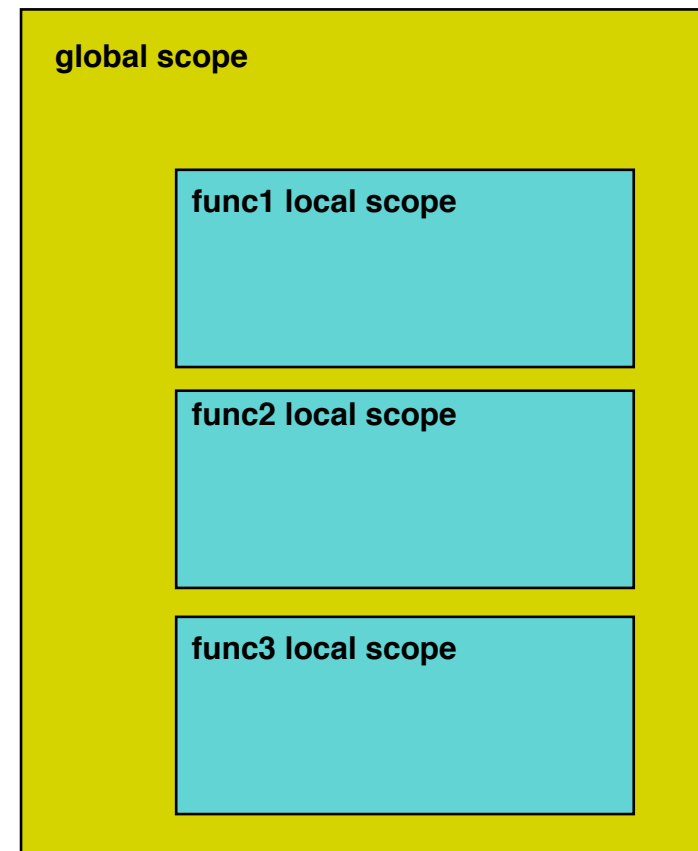
# An Example: Scoping Error

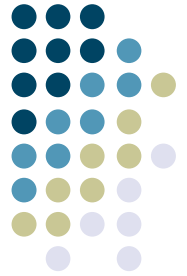
- `welcomeMsg = "Hello!"` ← `welcomeMsg` is defined in the **global scope**
- `def changeWelcomeMsg():`
  - `welcomeMsg = "Bonjour!"` ← This line defines a *new* variable with the same name, in the **local scope!**
  - `print "New welcome message is", welcomeMsg`
- `changeWelcomeMsg()`
- `>>> New welcome message is Bonjour!`
- `print welcomeMsg` ← Since this call to *print* is outside the function `changeWelcomeMsg()`, it refers to the `welcomeMsg` variable in the global scope.
- `"Hello!"`



# Thinking About Scopes

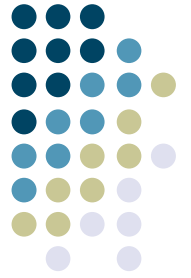
- Variables named in **the** global scope are available to statements in **any** scope
  - Unless they have been “hidden” by a local variable with the same name, as in the error example
- Variables named in **a** local scope are only available to statements in that scope
- The first **assignment** to a variable determines the scope it is in





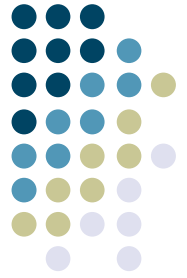
## More on Scopes

- “Global” really means the file the variable is in
  - When you start developing with multiple files, each file defines its own scope that is “global” for that file
- Each call to a function creates a new local scope
  - Thus if a variable `foo` is defined in function `func()`, each call to `func()` has its own new “namespace” and its own separate `foo`
- By default, *all* assignments that you make in a function create names in the local scope
  - Advanced: you can use the *global* statement if you want to change a global variable from within a function
  - Dangerous, but useful. We’ll talk about it in a later lecture
- Names not assigned to in a function are assumed to be globals



## Still More on Scopes

- What all this boils down to is...
  - Local variables (those first assigned to within a function) serve as temporary names you need only when a function is running
  - This helps modularity of your program ("hide" details within a function)
- But:
  - You need to be careful when using a name within a function that's defined outside
  - Subtle and hard to track bugs...



# Scoping Gotchas

- Subtly different than some other languages

- 1. Local scopes don't nest

```
def outerfunc(x, y):
```

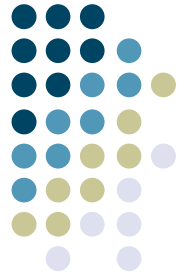
```
    def innerfunc(z):
```

```
        if z > 0:
```

```
            print x, y
```

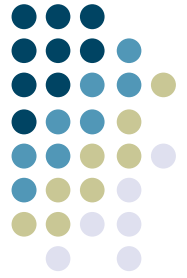
```
    innerfunc(x)
```

- $x$  and  $y$  aren't available inside the local scope for `innerfunc`
- 2. There are actually *three* scopes: global, local, and `__builtin__`
  - First, the local scope is checked
  - Then, the global scope
  - Finally, the scope defined by the module called `__builtin__`
    - `len`, `abs`, `max`, `min`, ...



# Modules

- *Modules* are the highest level building blocks in a Python program
- Usually correspond to a single file of code
- Let you organize your code more creatively:
  - Reuse code by storing it in files, callable by other files
  - Partition the variable and function namespace (so that not everything has to be at the “top level”)
  - Create functionality or data that can be shared across programs
- You *import* a module to gain access to its functionality



# Modules and Scoping

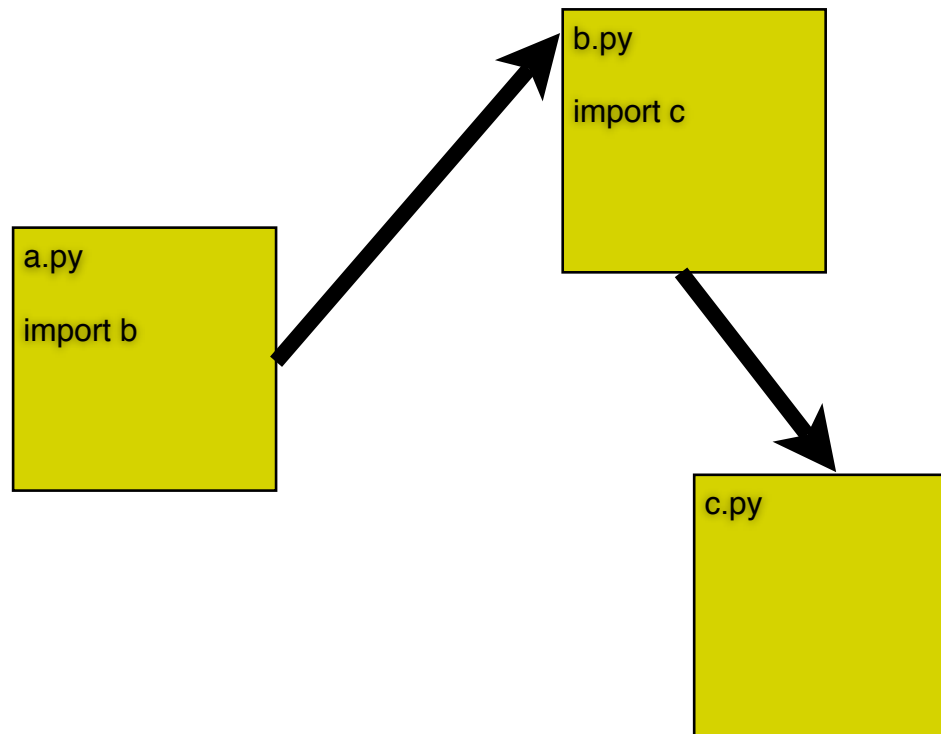
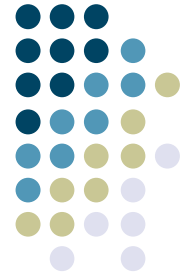
- Each module actually defines its own global scope
- Within a module, you can refer to names without using any extra qualification
- To refer to names outside a module, you first import the module to make it available to you
- Then refer to the name using dot notation

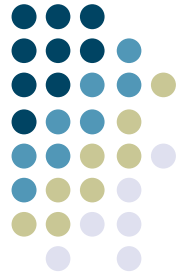
- Example:

```
import os
```

```
os.listdir("/Users/keith/Desktop")
```

# Breaking Your Program into Separate Files



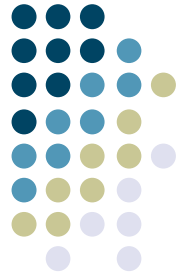


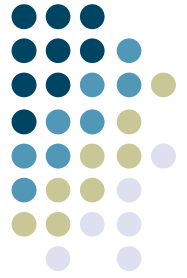
# Imports

- Each *import* statement defines a new namespace
  - Imagine a file *networking.py*, containing a function *broadcast()*
  - In your code:
    - `import networking`
    - `networking.broadcast()`
- You can assign more convenient names at the time of import
  - Example: *networking* is too long to type repeatedly, or collides with another name in you program
  - In your code:
    - `import networking as net`
    - `net.broadcast()`
  - Or:
    - `import javax.swing as swing`
    - `list = swing.JList()`

# Classes and Objects

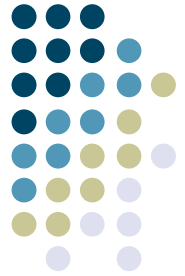
Georgia  
Tech





# Objects

- Objects are simply a way to group together a set of functions with the data that they operate on
- The built-in Python types are *already* objects!
  - Strings, integers, lists, dictionaries, etc.
- You can also create your own
  - You first have to write a “blueprint” for what the object will do
  - This is called the object’s *class*
  - Defines what operations are available on it, what data it contains, etc
  - You can use the blueprint to make *instances* of the class
- Terminology:
  - Instances are the actual objects
  - Classes are just the blueprints for making instances



# Defining a New Class

```
class Counter:
```

```
    def __init__(self):  
        self.count = 0
```

```
    def increment(self):  
        self.count = self.count+1  
        return self.count
```

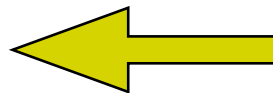
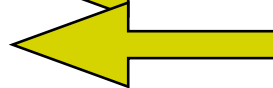
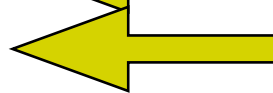
```
>>> c = Counter()
```

```
>>> c.increment()
```

```
1
```

```
>>> c.increment()
```

```
2
```



You define a new class via the *class* keyword

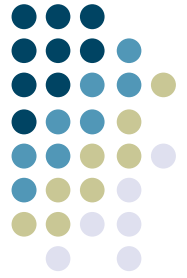
`__init__()` is a special function that is called when an instance of the class is created

Classes can contain functions

Every function in a class takes an additional argument, called *self*, which refers to the object on which the function is being called.

Within a class, you must refer to data in the class explicitly by scope: `self.count`

Creating an instance of a class looks like using the class name as a function call



# Each Instance is Separate

- Each instance has its own copy of the data, and its own namespace:

```
>>> c1 = Counter()
```

```
>>> c2 = Counter()
```

```
>>> c1.increment()
```

```
1
```

```
>>> c1.increment()
```

```
2
```

```
>>> c2.increment()
```

```
1
```

- Object-oriented programming lets you create reusable chunks of code and data
- Each copy is separate from the others
- Advanced: there are ways to have instances of a class share data



# Classes and Scoping

- Classes add a few more scoping rules to Python
  - Each *instance* is its own scope
  - Within a class, methods define local scopes just like functions
  - Example:

```
class Test:
```

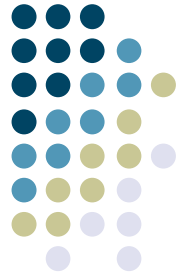
```
    def someMethod(self):
```

```
        self.daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

```
        length = len(self.daysOfWeek)
```

```
        for i in range(0, length):
```

```
            print self.daysOfWeek[i]
```



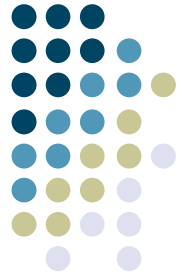
# Coming Full Circle...

- In Python, everything makes use of the same simple mechanisms:
  - Modules are really *dictionaries* that map from names (of variables and functions) to the data and code in those modules
    - `import os`
    - `print os.__dict__`
    - `{'listdir': <function at 15905785>, ...}`
    - `dir(os)` -- shows values in dictionary
    - `print os.__doc__`
  - Classes use the same mechanisms under the cover
    - `print Counter.__dict__`
    - `{'__module__': '__main__', 'increment': <function increment at 8963605>, '__doc__': None, 'count': 0}`
  - These dictionaries just define the names that are valid within the module or class

# GUI Programming

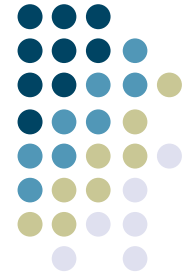
Georgia  
Tech



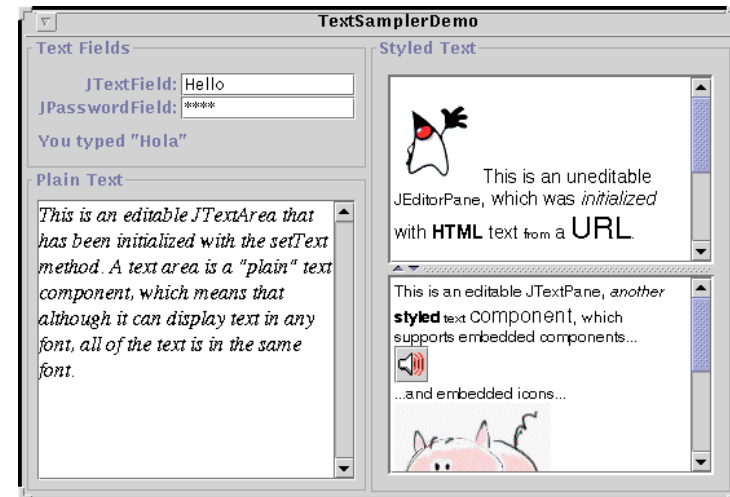
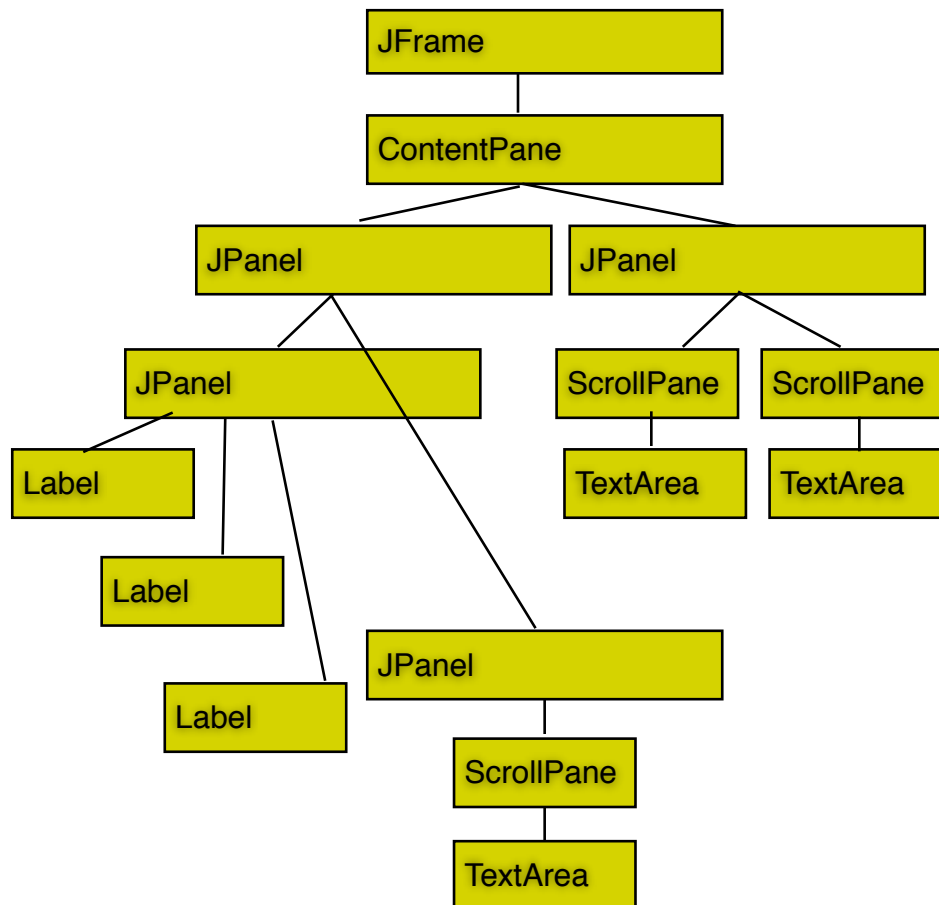


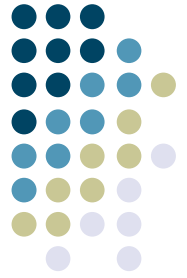
# GUI Programming 101

- The most important thing:
  - GUIs are layed out as *trees*
- There is a toplevel container, usually a window
- Inside this are multiple panels (often invisible), used to control layout
- For page layout people, think of the grid
  - Decompose interface into rectangular regions
  - May need many (invisible) sublevels to get it all right



# An Example

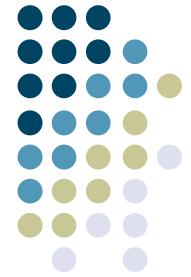




# Some Common Swing Widgets

- *Swing*: Java's GUI programming toolkit, callable in Jython
- On today's menu:
  - JFrames, JPanels
  - Layout Managers
  - JLists
  - JButtons
  - JLabels, JTextFields, JTextAreas
- This is an overview *only*
- You can do much more than I've shown here with each of these widgets, plus there are many more widgets than these

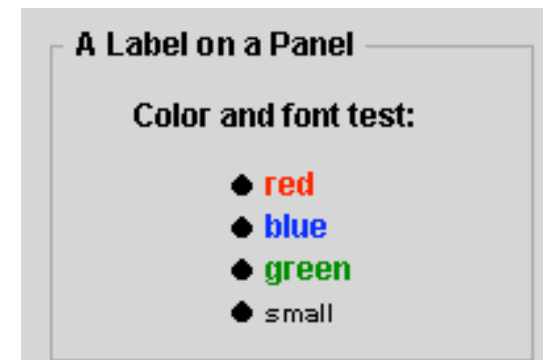
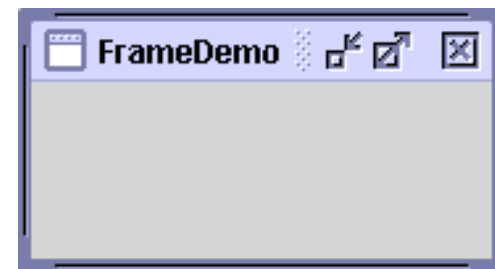
# Swing Widgets in Jython: JFrames and JPanels



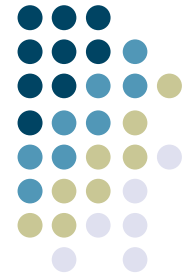
- JFrames are top-level windows
- JPanels allow grouping of other widgets
- Each JFrame has a panel into which the frame's contents must go: the *contentPane*  

```
window = swing.JFrame("FrameDemo")  
window.contentPane.add(new JButton)
```
- You must *pack* and *show* a JFrame to display it  

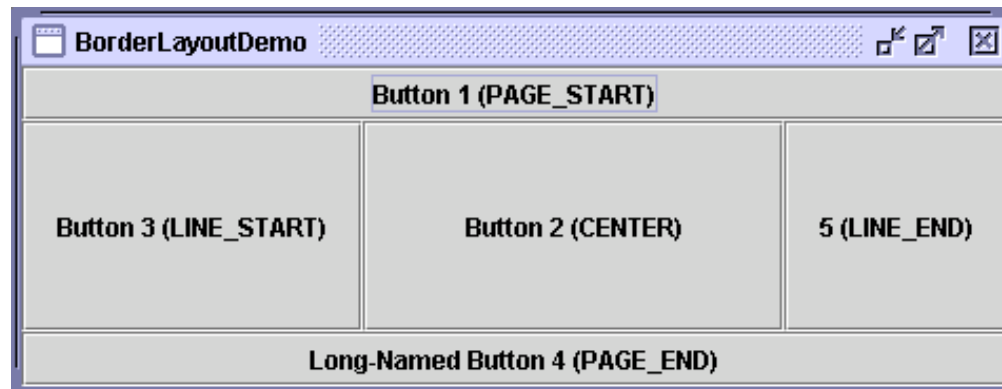
```
window.pack()  
window.show()
```



# Swing Widgets in Jython: Layout Managers

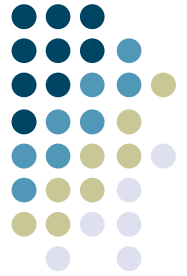


- Layout Managers control the placement of widgets in a JPanel
- Simplest by far: `awt.BorderLayout`  
`window.contentPane.layout = awt.BorderLayout()`  
`window.contentPane.add("Center", swing.JButton("Button 2 (CENTER)"))`
- Five regions:
  - North, South: expand horizontally
  - East, West: expand vertically
  - Center: expands in both directions



# Swing Widgets in Jython: JLists

Georgia  
Tech



- JLists are collections of widgets
  - `list = swing.JList()`
- Put JLists in a JScrollPane to make them scrollable
  - `window.contentPane.add(swing.JScrollPane(list))`
- JLists contain a *listData* member with the contents
  - `list.listData = ['January', 'February', 'March', ...]`
- *selectedValue* contains the selected item!
  - `>>> print list.selectedValue`
  - `'March'`

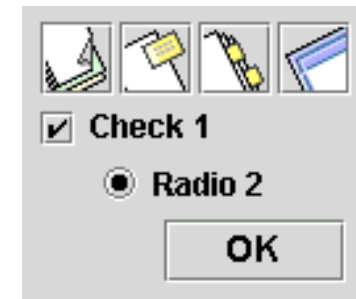


# Swing Widgets in Jython: JButtons

Georgia  
Tech



- JButtons have many fancy features...
  - Images, labels, tooltips, etc.
- Basic use is very simple:
  - Supply a label when you construct the button
    - `button = swing.JButton("This is my label!")`
  - Provide a function to use as a callback
    - `def callbackFunction(event):`
      - ...
    - `button.actionPerformed = someCallback`
  - NOTE: when the function is a *method*, you must handle it slightly differently!
    - `def callbackMethod(self, event):`
      - ...
    - `button.actionPerformed = self.someCallback`



# Swing Widgets in Jython:

## JTextFields, JTextAreas, and JLabels

Georgia  
Tech



- JLabels are the world's simplest widgets  
years = swing.JLabel("Years")



- JTextFields are used for single-line text entry

```
yearValue = swing.JTextField()
```

```
print yearValue.text
```

```
30
```

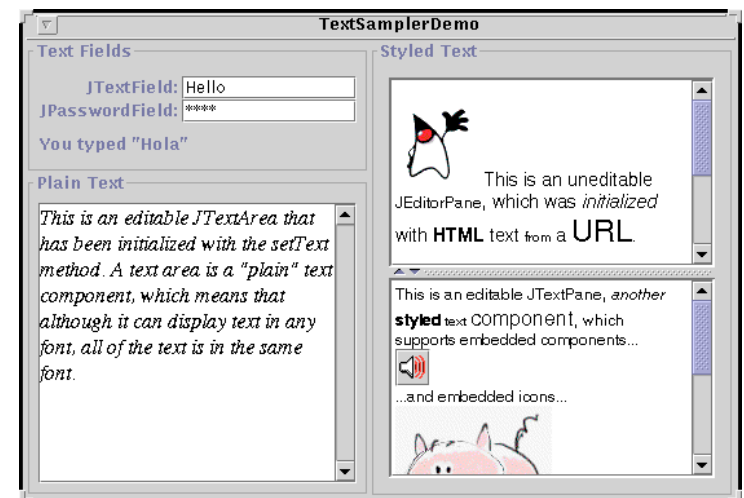
- JTextAreas are used for longer pieces of text

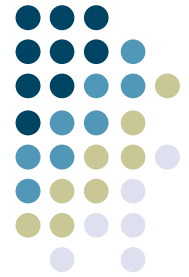
```
area = swing.JTextArea(24, 80)
```

```
area.setEditable = 0
```

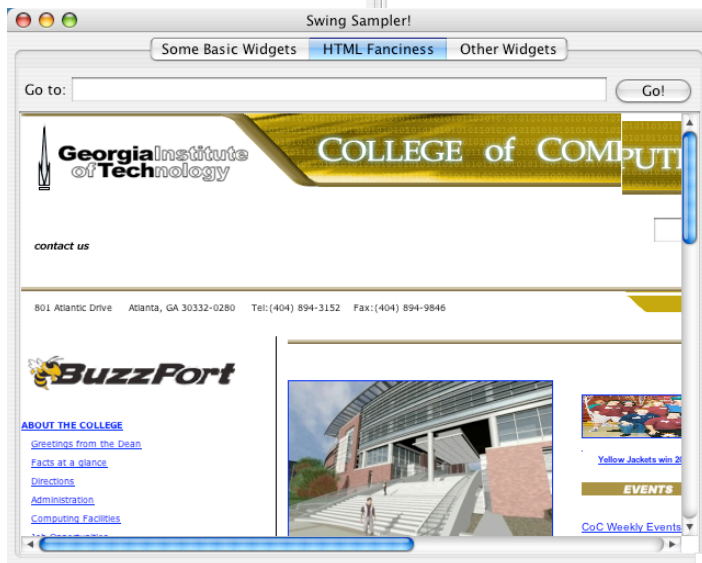
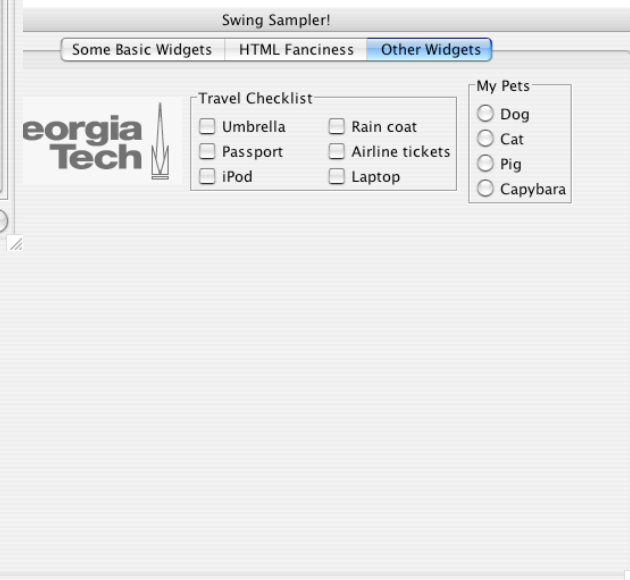
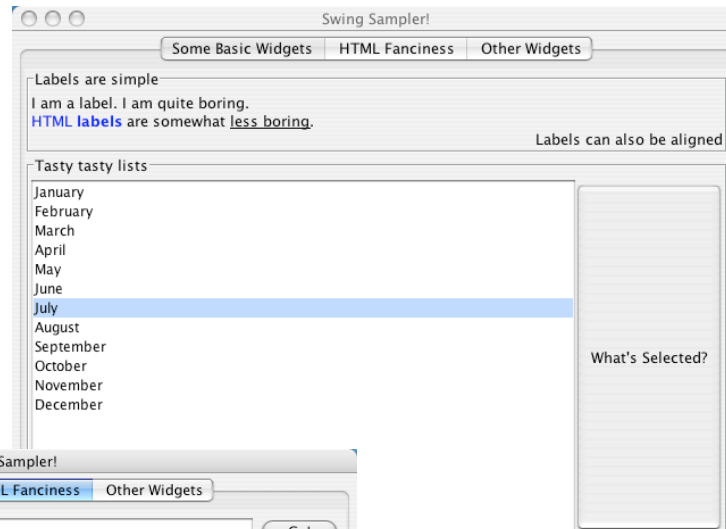
```
print area.text
```

```
area.text = area.text + "One more string"
```



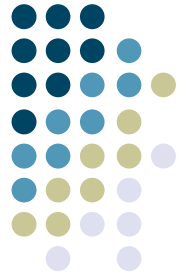


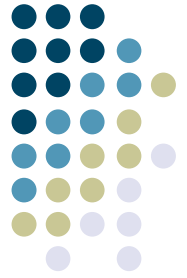
# Putting it All Together



# Code Walkthrough and Demo

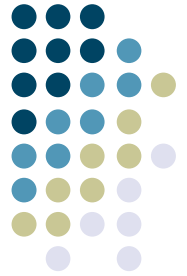
Georgia  
Tech





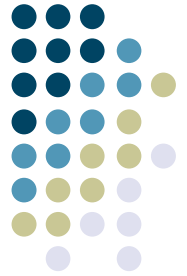
# Useful Odds-and-Ends #1

- How do you make a “main” program?
  - Analog to `void main()` in C, `public static void main()` in Java
- In Jython, the system variable `__name__` will be set to the string “`__main__`” in any file passed directly on the command line to Jython
- Example:
  - `if __name__ == “__main__”:`
    - `sampler = SwingSampler()`
  - On command line:
    - `jython swing-sampler.py`



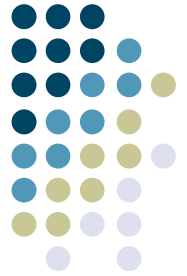
## Useful Odds-and-Ends #2

- How do you get the name of the user running your program?
- Useful in, e.g., a Chat program if you don't want to require users to log in explicitly
- *Note: for testing, you probably want some way to override this, so that you can simulate multiple users on the same machine*
  - `import java.lang as lang`
  - `me = lang.System.getProperty("user.name")`
- Returns login name



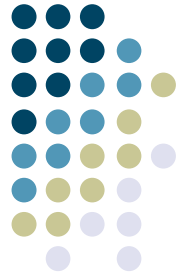
## Useful Odds-and-Ends #3

- How do you pass arguments on the command line?
- Useful, for example, to override the user name or set other parameters explicitly
- The variable `sys.argv` is the “argument vector”--the list of arguments passed on the command line
- The first element (`sys.argv[0]`) is *always* the name of the Jython file
- Example:
  - `import sys`
  - `if __name__ == “__main__”:`
    - `if len(sys.argv) > 1:`
      - `print “Got an argument”, sys.argv[1]`
    - `else:`
      - `print “Got no arguments”`



## Useful Odds-and-Ends #4

- Wacky Python syntax
- Multi-line string constants
  - `"""this is a multi-line string constant"""`
- Multiple assignment
  - `a, b, c = 1, 2, 3`
  - `for key, value in dict.items():`
- Default parameters
  - `def func(a, b=0, c="Fred", *d, **e):`
    - `*d` is a “catch-all” -- captures in a tuple any excess arguments
    - `**e` is a second-level catch-all -- captures in a dictionary any keyword arguments not already specified in the argument list
- And, of course, indentation denotes blocks...



# Useful Odds-and-Ends #5

- Easy bridging between Java and Jython
- Can import and use arbitrary Java classes from within Jython
  - `import java.util.Date as Date`
  - `d = Date()`
  - `print d`
- Can subclass Java classes from Jython
  - `class MyUI(swing.JFrame):`
- Automatic type conversion between many Java and Jython types
  - e.g., Jython lists to and from Java arrays
- Detection and conversion of common code patterns

<code>setFoo(); getFoo()</code>	<code>foo = 12; print foo</code>
<pre> JButton close = new JButton("Close Me") close.addActionListener(new ActionListener() {     public void actionPerformed(ActionEvent ev) {         java.lang.System.exit(0);     } }); </pre>	<pre> close = swing.JButton("Close Me") close.actionPerformed = self.terminateProgram  def terminateProgram(self, event):     java.lang.System.exit(0) </pre>