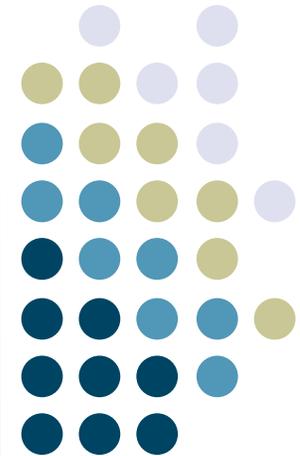


Week 2: Quick and Dirty Jython Refresher

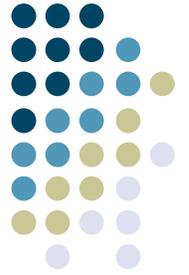
(Prelude to Asynchronous Programming)



**Georgia
Tech**

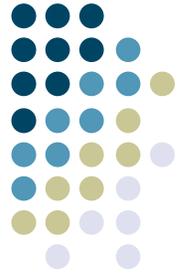


CS6452



Today's Menu

- Review of lo-fi prototyping homework
- 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

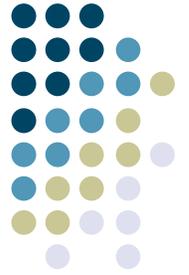


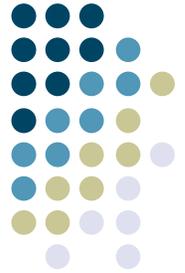
Review of First Assignment

- Great job!
- Wide diversity in techniques, level of detail, etc.
- Some issues:
 - Online versus offline users
 - Authentication
- What's supported by the protocol, and what goes into the client?

Jython Collection Classes

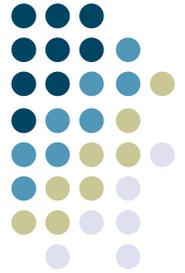
Georgia
Tech





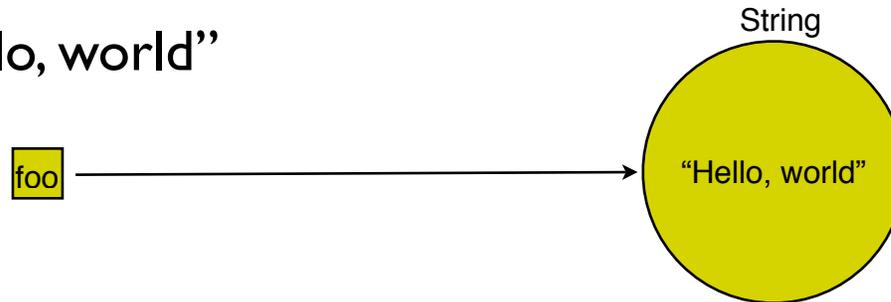
Collections

- 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



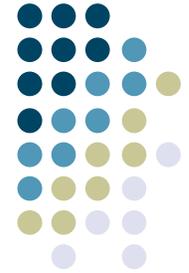
Variables and References

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

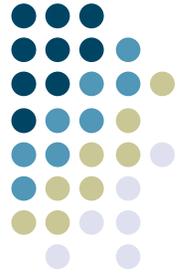


- Variables can be reassigned, and multiple variables can refer to the same thing.
- Stashing a reference in a variable gives you a way to name it, and get at it later.

The Need for More Complex Data Structures

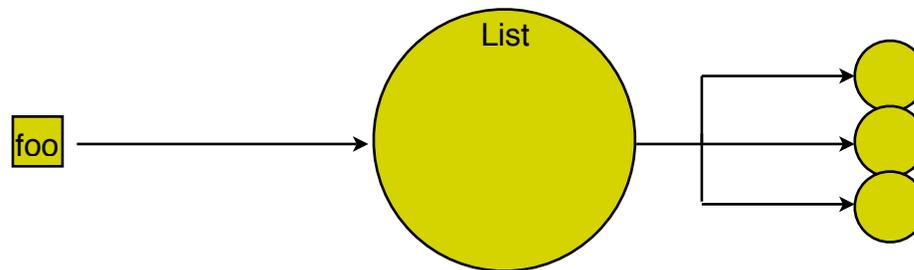


- Some more complex structures are hard to represent by just a named variable though.
- Example: you want to keep track of all of the users in a chat.
 - user1 = "Steven"
 - user2 = "Amy"
 - ...
- This is too *static*. Would you just create 1000 variables in case you ever had that many users? How would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

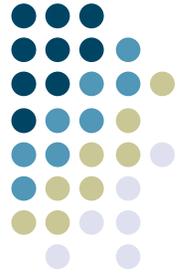


Lists to the Rescue

- 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 Jython 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')
```

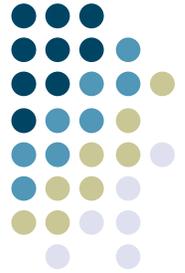
```
>>> print L
```

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

- Iteration

```
for item in L:
```

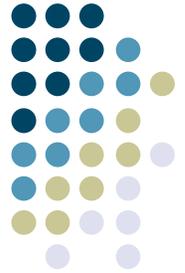
```
    print item
```



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 related data items that are fixed:
 - `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



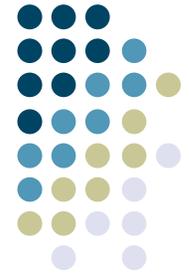
- Sometimes, you need to associate one item with another one
 - Example: hours worked on each day of the week:

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

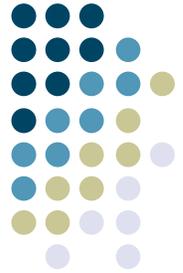
- You could do this with variables, as long as there's a fixed set of them:
 - `sunday=4.5`
 - `monday=8`

Associating Data Items With Each Other (cont'd)

Georgia
Tech

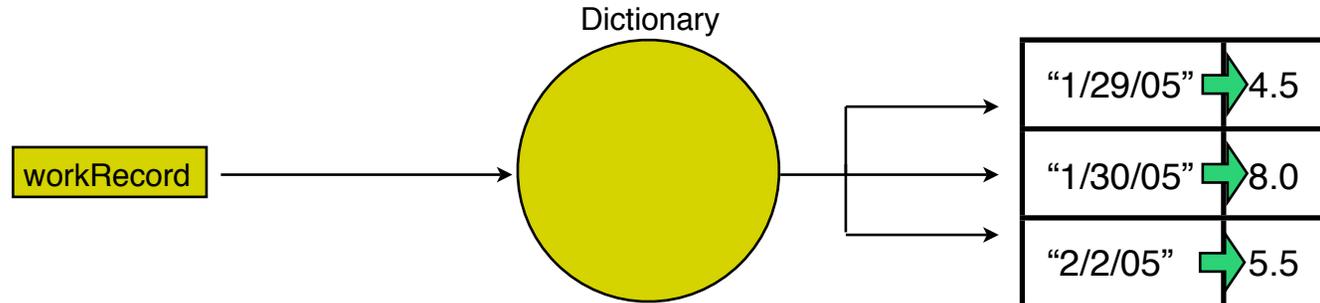


- If you don't know the associations you might have up front, you could use parallel lists:
 - `workDates = ["1/29/05", "1/30/05", "2/1/05", ...]`
 - `workHours = [4.5, 8, 5.5, ...]`
- Then, iterate through the first list to find the date you're looking for, then look for the item with the corresponding 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

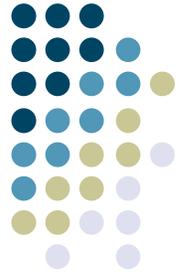
- Dictionaries associate *values* with *keys* (you *lookup* a value given its key)
- Both are references to data items
- `workRecord = { "1/29/05": 4.5, "1/30/05": 8, "2/2/05": 5.5 }`



- *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:

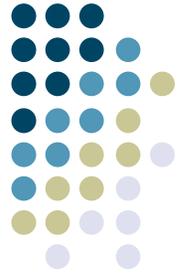
```
>>> del dict["one"]
```

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

```
3
```

Scoping and Modules

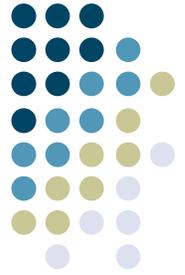
Georgia
Tech





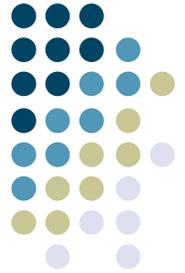
Scoping

- 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

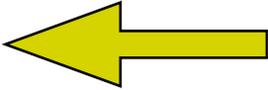
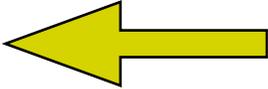
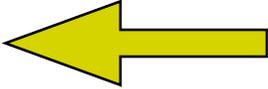


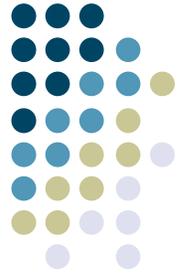
An Example: Scoping Error

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



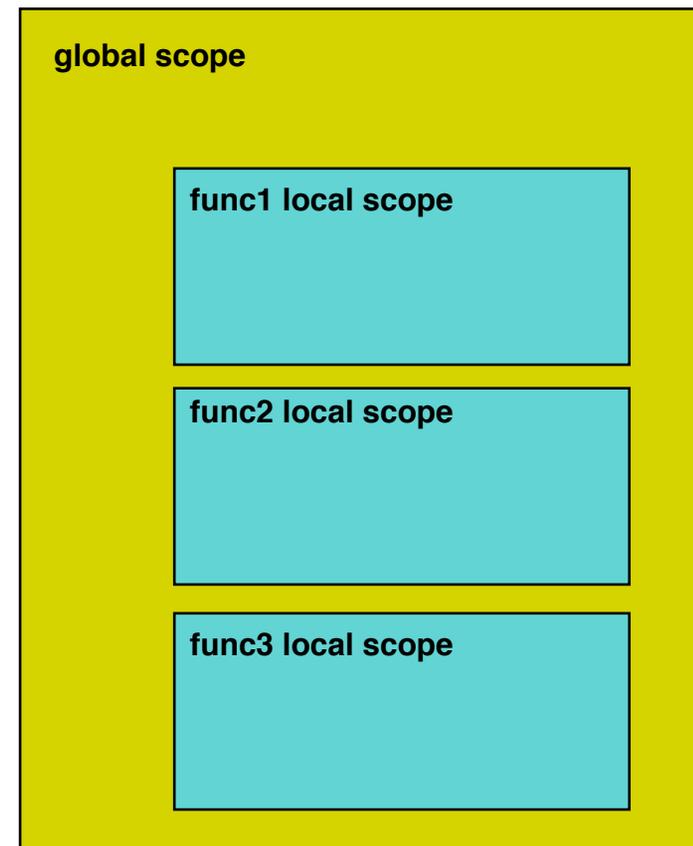
An Example: Scoping Error

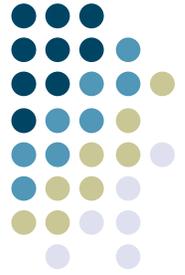
- `welcomeMsg = "Hello!"`  `welcomeMsg` is defined in the **global scope**
- `def changeWelcomeMsg():`
 - `welcomeMsg = "Bonjour!"`  This line defines a *new* variable with the same name, in the **local scope!**
 - `print "New welcome message is", welcomeMsg`
- `changeWelcomeMsg()`
- `>>> New welcome 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 statement in that scope
- The first **assignment** to a variable determines the scope it is in





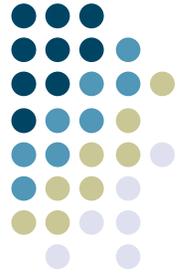
More on Scopes

- “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
- Names not assigned to in a function are assumed to be globals



Still More on Scopes

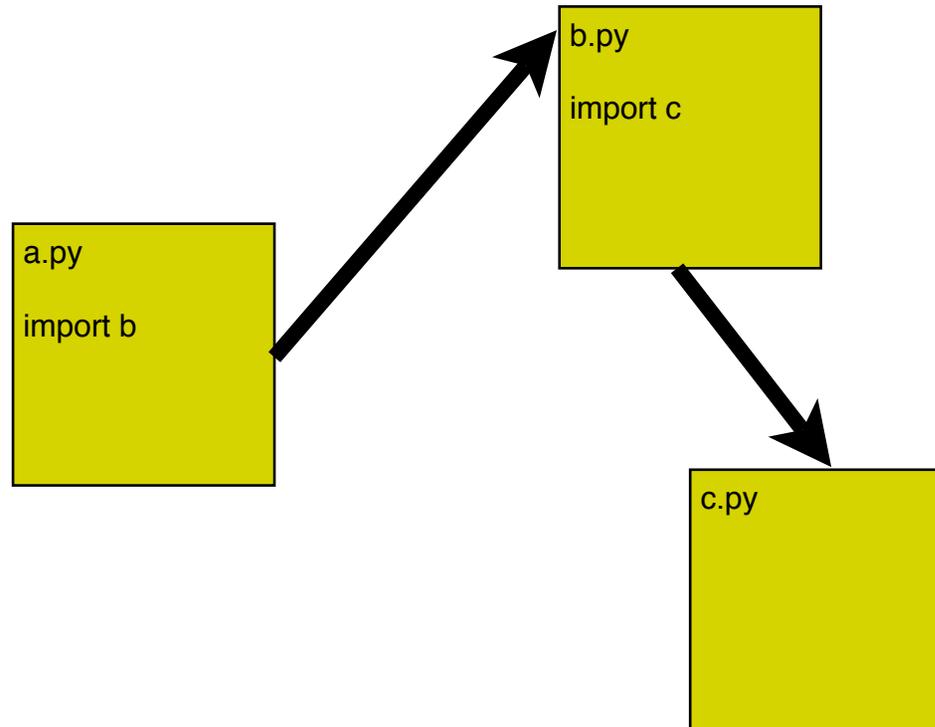
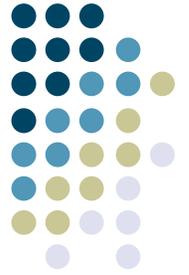
- What all this boils down to is...
 - Local variables (those first assigned to within a function) serve as temporary names you need only when a function is running
 - This helps modularity of your program ("hide" details within a function)
- But:
 - You need to be careful when using a name within a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

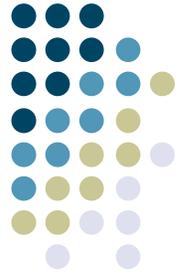


Modules

- *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

Breaking Your Program into Separate Files



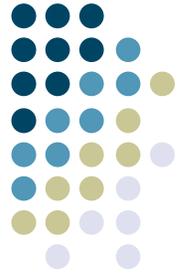


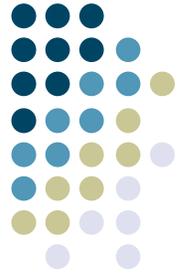
Imports

- 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

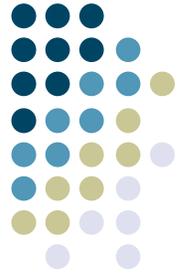
Georgia
Tech





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
```

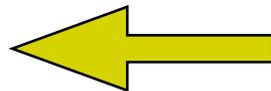
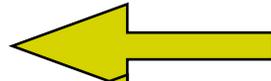
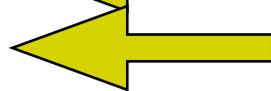
```
>>> c = Counter()
```

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

```
1
```

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

```
2
```



You define a new class via the *class* keyword

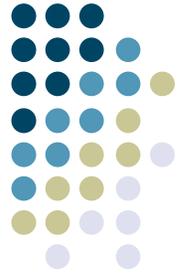
`__init__()` is a special function that is called when an instance of the class is created

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
```

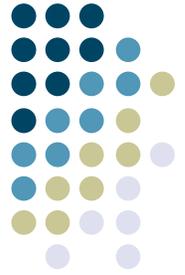
```
>>> c1.increment()
```

```
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



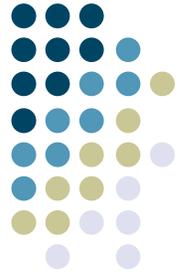
Coming Full Circle...

- 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 networking`
 - `print networking.__dict__`
 - `{'broadcast': <function at 15905785>}`
 - 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

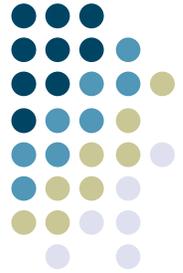
Georgia
Tech



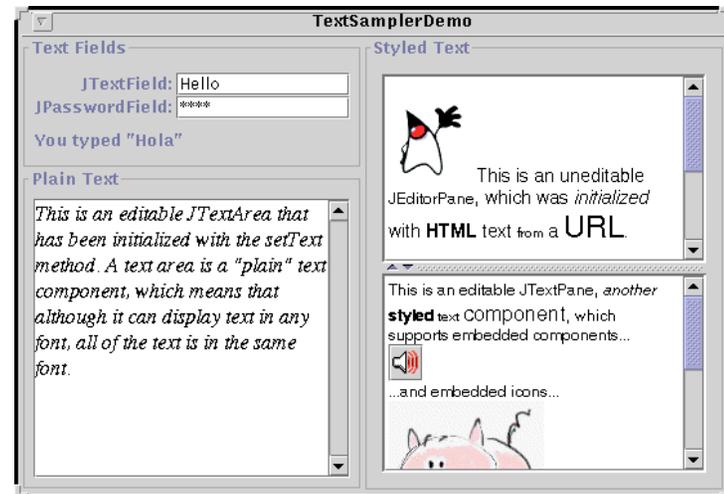
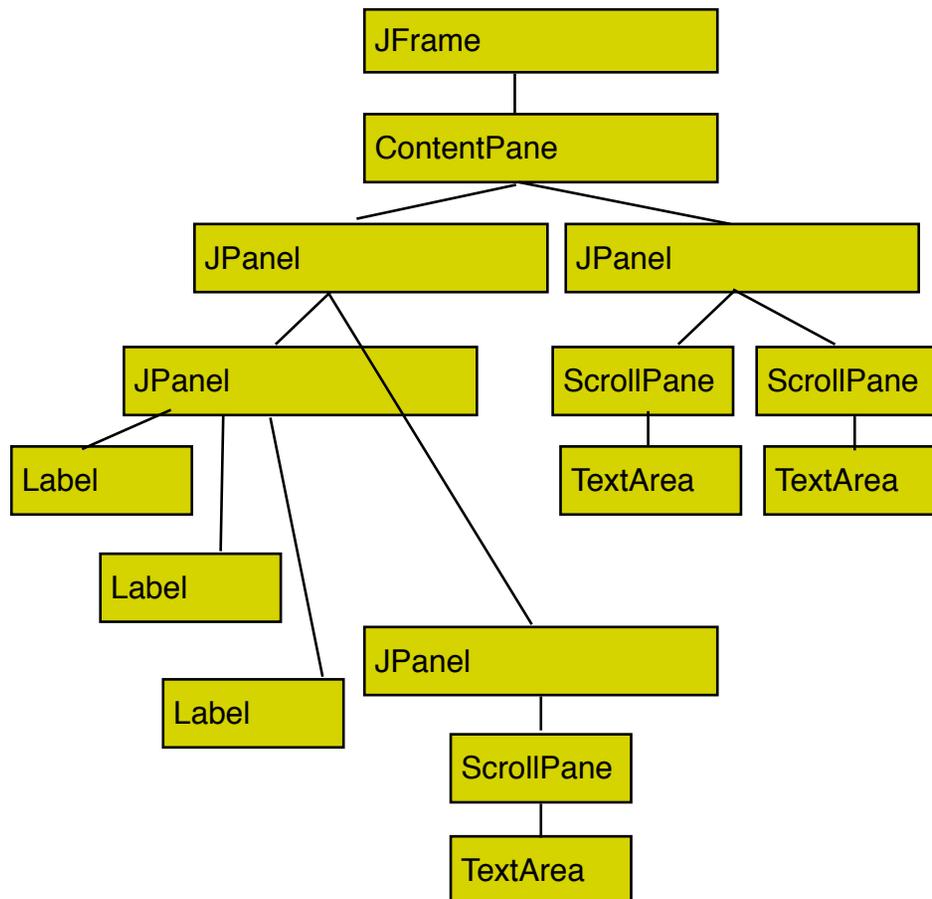


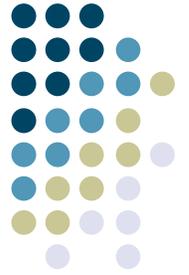
GUI Programming 101

- 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



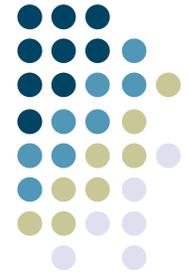


Some Common Swing Widgets

- *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

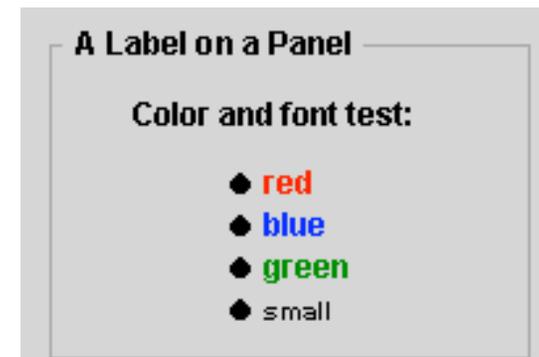
Georgia
Tech



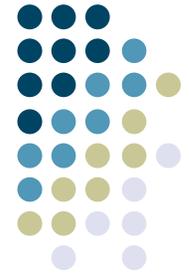
- JFrames are top-level windows
- JPanels allow grouping of other widgets
- Each JFrame has a panel into which the frame's contents must go: the *contentPane*

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

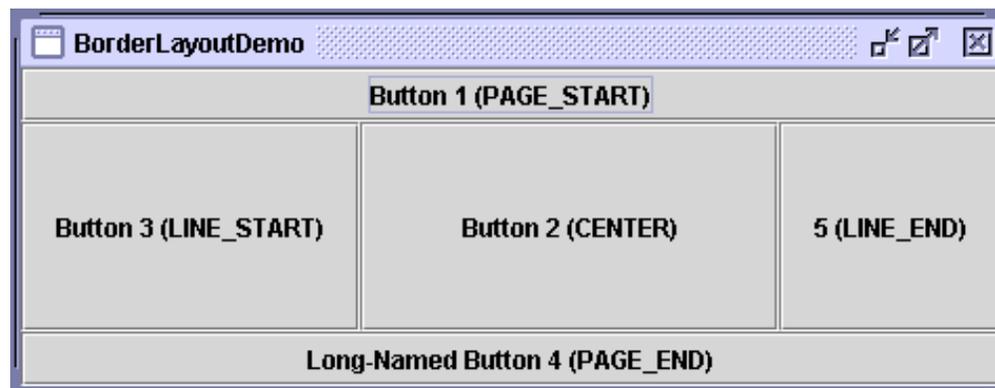
```
window.pack()  
window.show()
```



Swing Widgets in Jython: Layout Managers

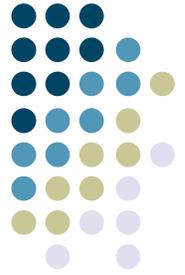


- Layout Managers control the placement of widgets in a JPanel
- Simplest by far: `awt.BorderLayout`
`window.contentPane.layout = awt.BorderLayout()`
`window.contentPane.add("Center", swing.JButton("Button 2 (CENTER)"))`
- Five regions:
 - North, South: expand horizontally
 - East, West: expand vertically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



Swing Widgets in Jython: JLists

Georgia
Tech

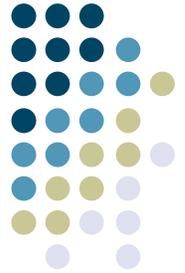


- JLists are collections of widgets
 - `list = swing.JList()`
- Put JLists in a JScrollPane to make them scrollable
 - `window.contentPane.add(swing.JScrollPane(list))`
- JLists contain a *listData* member with the contents
 - `list.listData = ['January', 'February', 'March', ...]`
- *selectedValue*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



Swing Widgets in Jython: JButtons

Georgia
Tech



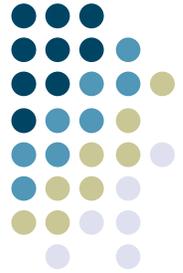
- JButtons have many fancy features...
 - Images, labels, tooltips, etc.
- Basic use is very simple:
 - Supply a label when you construct the button
 - `button = swing.JButton("This is my label!")`
 - Provide a function to use as a callback
 - `def callbackFunction(event):`
 - ...
 - `button.actionPerformed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

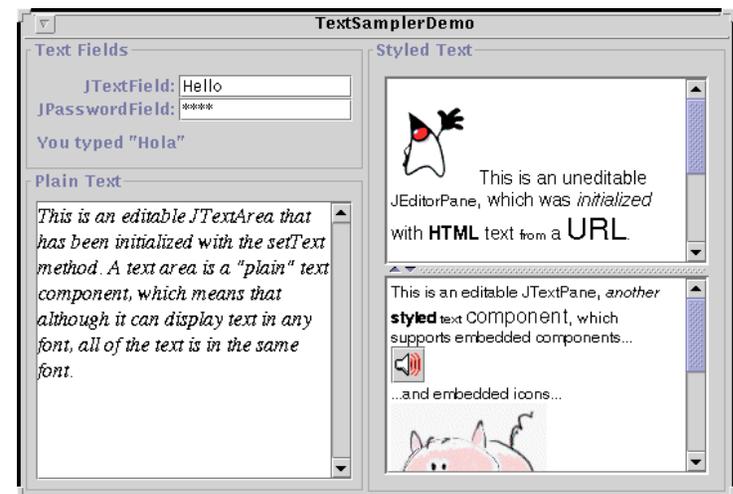
```
area = swing.JTextArea(24, 80)
```

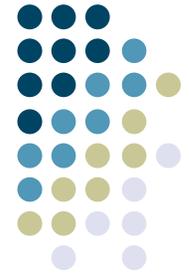
```
area.setEditable = 0
```

```
print area.text
```

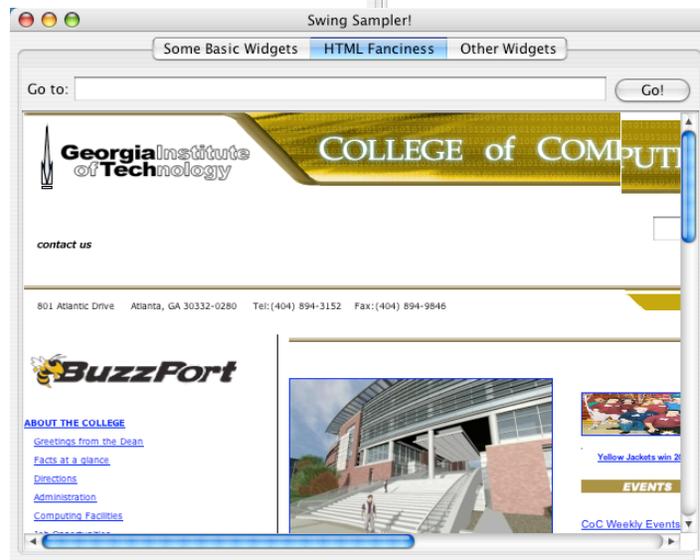
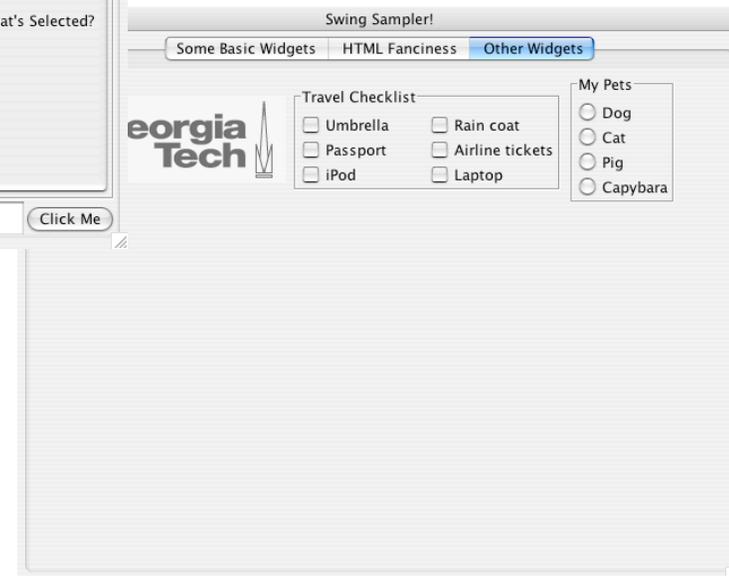
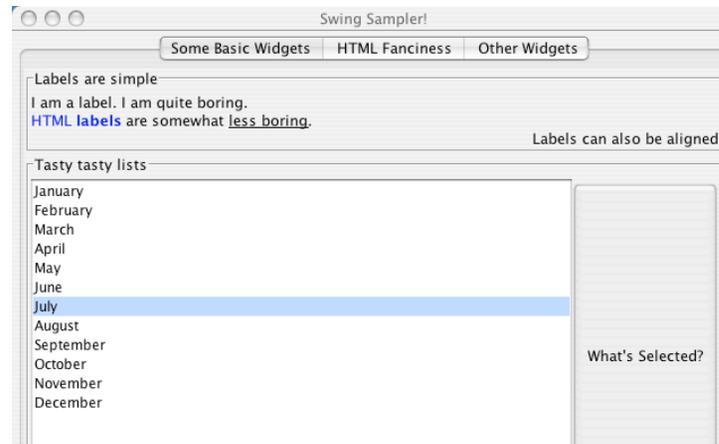
```
area.text = area.text + "One more string"
```

Years:



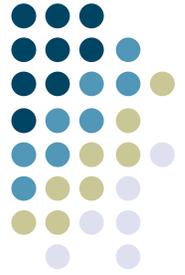


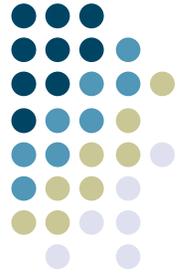
Putting it All Together



Code Walkthrough and Demo

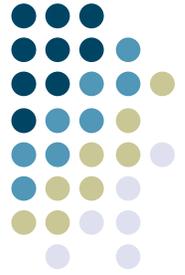
Georgia
Tech





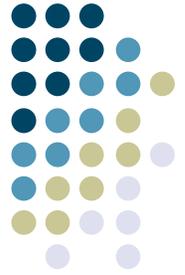
Useful Odds-and-Ends #1

- 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()`



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”`