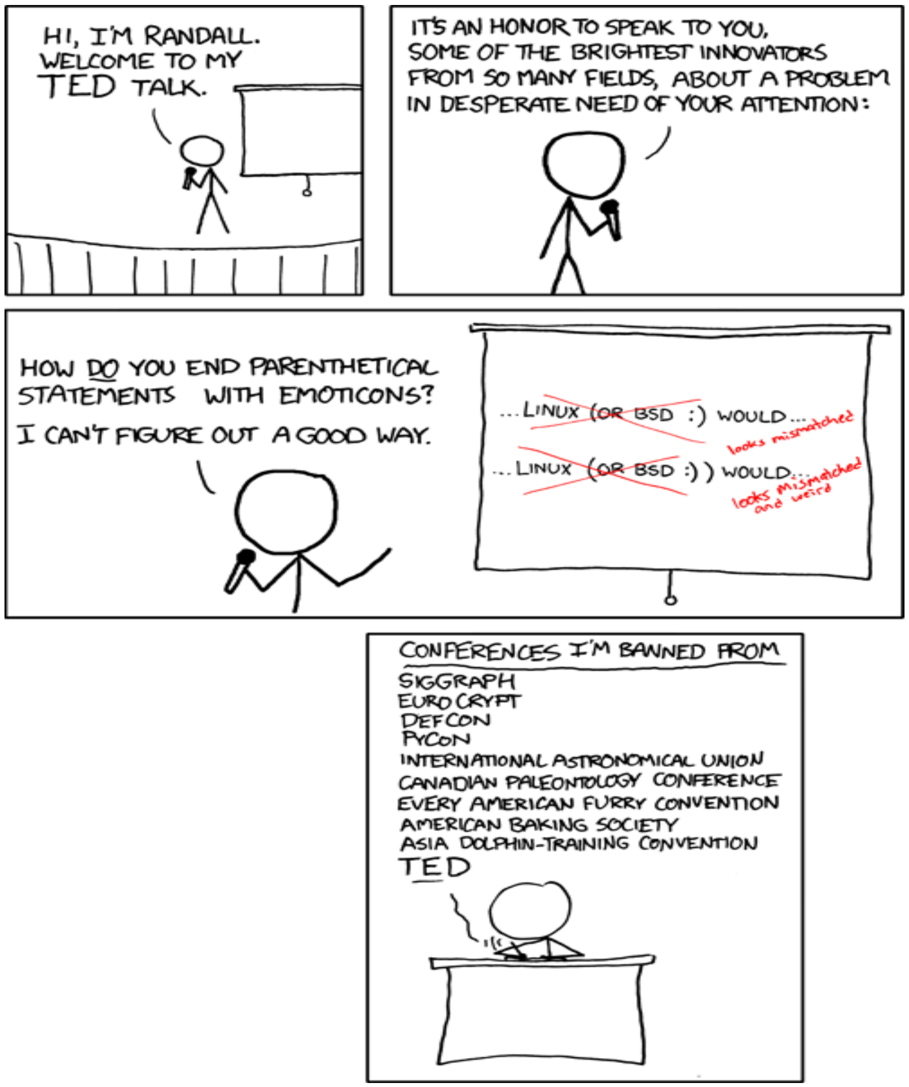
CS3101 Python: Lec 5


HI, I'M RANDALL. WELCOME TO MY TED TALK.
IT'S AN HONOR TO SPEAK TO YOU, SOME OF THE BRIGHTEST INNOVATORS FROM SO MANY FIELDS, ABOUT A PROBLEM IN DESPERATE NEED OF YOUR ATTENTION:
HOW DO YOU END PARENTHETICAL STATEMENTS WITH EMOTICONS? I CAN'T FIGURE OUT A GOOD WAY.
...LINUX (OR BSD :) WOULD...
looks mismatched
...LINUX (OR BSD :)) WOULD...
looks mismatched and weird
CONFERENCES I'M BANNED FROM
SIGGRAPH
EUROCRYPT
DEFCON
PYCON
INTERNATIONAL ASTRONOMICAL UNION
CANADIAN PALEONTOLOGY CONFERENCE
EVERY AMERICAN FURRY CONVENTION
AMERICAN BAKING SOCIETY
ASIA DOLPHIN-TRAINING CONVENTION
TED

This week

- Working w/ zip files
- Modules and packages
- Subprocesses
- Serialization
- GUIs
- Databases

Working with compressed files (code is Python 3.x, should work w/ 2.x as well)

See: http://docs.python.org/library/zipfile.html

Working with Compressed Files

- Python provides libraries to work directory with data in zip, gzip, bz2, tars, etc
- Most libraries offer decompression of individual files on the fly (i.e., unnecessary to extract the entire archive to make modifications)
- Third party libraries are available to handle pretty much any format

Ex

- Simpsons.zip:
  - 2 files:
    - simpsons.txt (homer, marge, bart, lisa, maggie)
    - donuts.txt (jelly, grape, etc)

Reading a zip archive, simple right?

```
import zipfile
f = zipfile.ZipFile('simpsons.zip')
for name in f.namelist():
   bytes = f.read(name)

   contents = bytes.decode()

   print ('File', name, 'contains', len(bytes), 'bytes')

   print ('It\'s contents are', contents)
```

```
$python3 foo.py

File donuts.txt contains 29 bytes
It's contents are:
jelly
sugar
chocolate
grape

File simpsons.txt contains 52 bytes
It's contents are:
....
```

Creating an archive, simple again

```
import random, zipfile

paths = ['file_a', 'file_b', 'file_c'] # create three random files
for path in paths:
    out = open(path, 'w')
    out.write(str(random.random()))
    out.close()

# create an archive: default compresssion is deflate
z = zipfile.ZipFile('foo.zip', 'w')
for path in paths[:-1]:
    z.write(path)
    z.close()

# append a file to an existing archive
z = zipfile.ZipFile('foo.zip', 'a')
z.write(paths[-1])

print (z.namelist())
z.close()
```

- Serializing data (code is 3.x)
- See:

http://docs.python.org/3.1/library/pickle.html

Serializing Data using cPickle

- Basic operations: dump and load
- Dump: store arbitrary data structure
  - Supports text and binary forms (where might you prefer binary data?)
- Load
  - Compatibility is guaranteed from one Python release to the next
- Machine and implementation independent
- In between dumps and loads you can
  - Store to a database, compress, send over a network,etc

Ex. – pickling an object

```
import pickle
donuts = {'homer': 12, 'lisa' : 0, 'marge' : 1}
text = pickle.dumps(donuts) # serialize (note the 's')
print (text)
del donuts
donuts = pickle.loads(text)
print (donuts)
```

```
python3 foo.py
```

```
b'\x80\x03}q\x00(X\x05\x00\x00\x00homerq\x01K\x0cX
  \x04\x00\x00\x00lisaq\x02K\x00X\x05\x00\x00\x00margeq
  \x03K\x01X\x04\x00\x00\x00bartq\x04K\x04u.'
```

```
{'homer': 12, 'lisa': 0, 'marge': 1, 'bart': 4}
```

Pickling multiple, arbitrary objects

```
import pickle
# An arbitrary collection of objects supported by pickle.
data = {'a': [1, 2.0, 3, 4+6j], 'b': ("character string", b"byte string"), 'c':
   set([None, True,
                       False])}
more_data = 'abcdef'

def store(path, *objects): # store an arbitrary sequence of objects
    out = open(path, 'wb')
    for obj in objects:
        pickle.dump(obj, out, pickle.HIGHEST_PROTOCOL) # new in Python 3.x
    out.close()

def retriever(path):
    f = open(path, 'rb')
    while True:
    try:
        yield pickle.load(f)
    except EOFError as e:
        print ('No more objects to retrieve')
    f.close()

path = 'foo.pickle'
store(path, data, more_data)
f = retriever(path)
print (next(f))
print (next(f))
print (next(f))
```

Pickling class instances

```
class Donut():
    def __init__(self, tasty):
        self.tasty = tasty

chocolate_donut = Donut(tasty=True)
asparagus_donut = Donut(tasty=False)

store(path, chocolate_donut, asparagus_donut)
x = retriever(path)
d = next(x)
print (d.tasty)
d = next(x)
print (d.tasty)
```

Modules

Modules

- A typical Python program is made up of several source files
- Recall that each source file is a *module*
  - Groups code and data for reuse
- Modules should be independent of one another
- To communicate between modules
  - Use import and from keywords
  - Global variables are not global to all modules
    - They are attributes of a module's namespace
    - Different from some programming languages
    - Not a viable way to communicate between modules

Extensions (preview)

- Python supports *extensions: components can be written in other languages for use in Python*
  - Use C, C++, Java, C#, etc.
- Extensions are seen as modules
  - Python code that uses extensions are called client code
  - Client code does not care if modules are pure Python or an extension
- Advanced topic: covered in the next lecture

Module Objects

- Recall that code for a module resides in a file with the same name (minus the filename extensions)
  - E.g. module name corresponds to name.py
- Modules are first-class objects
  - Can be treated like other objects
    - Passed as arguments in a function call, can be returned by functions, bound to a variable, etc.

The import Statement

- Syntax:
  - import modname [as varname], [, ...]
  - Import keyword followed by comma delimited module specifiers
- When the statement is executed, a module object is bound to a variable
- When a variable name (varname) is specified, the module (modname) is found but it is bound to varname instead
- Example:
  - import os, sys, csv as comma_separated_values

Module Body

- Sequence of statements in the module's source file
- No syntax is required to specify a file as a module
  - This is automatic; every source file can be used as a module
- Executed the first time it is imported
- If the first statement is a literal string, it is used as a docstring
  - Recall: accessible as the __doc__ attribute of a module

Doc Strings

```
def complex(real=0.0, imag=0.0):
"""Form a complex number.

  Keyword arguments:
  real -- the real part (default 0.0)
  imag -- the imaginary part (default 0.0)

  """
  if imag == 0.0 and real == 0.0:
      return complex_zero
  ...
```

- See: http://www.python.org/dev/peps/pep-0257/

Module Attributes

- Import creates a new namespace containing attributes of the imported module
- To access an attribute in this namespace, use the name of the module and the dot operator
  - import Module
  - Module.foo()
  - import Module as Alias
  - Alias.foo()
- Normally, attributes in a module are bound by statements in the module‟s body
- You can also bind / unbind attributes outside a module
  - For clarity, this is not recommended

Private Variables

- How to make variables “private” to a module
- No variable is truly private
  - Recall that encapsulation is not Python‟s strong suit
- Begin a variable name with an underscore to signify that it is private
  - Convention, not enforced by Python, so it is up to programmers to follow this convention
  - Some IDEs will respect this convention, and not show attributes when performing code completion

The from Statement

- Allows you to import specific attributes of a module into the current namespace
- Syntax:
  - from modname import attrname as varname
- Alternate syntax:
  - from modname import * (ewww)
  - Imports all attributes from module modname into the current namespace
  - The asterisk (*) requests that all attributes of the module some_module be bound to the current namespace
  - Considered bad form, it obfuscates the namespace!
- Better:
  - from numpy import vector as vec

Searching for a Module

- How does Python search for a non-built-in module in the filesystem?
- First, Python looks at the items of sys.path
  - Each item is the path of a directory or archive (zip)
  - Initialized at start-up using the environment variable PYTHONPATH (if present)
  - First item is always the path of the main script
  - You can rebind sys.path at runtime, but it will not affect modules already loaded
- Modules can be loaded from:
  - dll (Windows) or .so (Unix) libraries
  - .py (pure Python code)
  - pyc, .pyo: bytecode compiled Python modules
- Lastly, for module M, if there exists file M/__init__.py is loaded

Loading a Module

- If a file M.py is found, it is compiled into M.pyc (or M.pyo, if the optimize –O flag is used)
  - Unless the compiled file already exists, or is newer than M.py
  - If M.py is in a writable directory, then M.pyc is stored there and is not recompiled in the future
- Once the compiled bytecode is found, Python executes the module body to initialize the module
- Circular Imports
  - Python allows circular imports
  - E.g. module a.py imports b.py, while b.py imports a.py
  - Messy, avoid if possible
    - but generally no side effects if modules ONLY contain definitions rather than executable code in the outer body

Main

- what happens if you import a module (source) that has commands in the module namespace?
  - They're executed
- If you use the if __name__ == "__main__": block, those statements will only execute if this module is the first loaded
- __name__ attribute of main program is always "__main__", otherwise the module name

Main

```
#homer.py
print (“Name is”, __name__)
print ("Mmmm homer")
if __name__ == "__main__":
    print ("Arghggggg")
```

```
#donut.py
import homer
print ("Mmmm donut")
if __name__ == "__main__":
    print("Arghggggg")
```

Understanding imports and __main__

```
$python3 homer.py
name is __main__
Mmmm homer
Arghggggg

$python3 donut.py
name is homer
Mmmm homer
Mmmm donut
Arghggggg
```

Packages

- Python packages are modules that contain other modules
- Packages may contain subpackages
- A package resides in a subdirectory of the same name in one of the directories in sys.path
  - Packages may also be stored in zip files
- Module body of package P is stored in P/__init__.py
  - The file __init__.py is required, even if it is empty to indicate that P is a package
  - Loaded when you first import P
- You can import module M in package P using:
  - import P.M
  - More dots let you navigate the package hierarchy
  - Package body is always loaded before module in that package
- You can also use:
  - from P import M
  - from P import * # not recommended!

Distributing Python Programs

- Python modules, extensions, programs can be packaged and distributed
- Supported forms:
  - Compressed archives (.zip, .tar.gz)
  - Self-unpacking, self-installing (.exe)
  - Self-contained, ready-to-run (.exe, .zip with a script prefix on Unix)
  - Platform-specific installers (.msi, .rpm, .deb)
  - Python Eggs: a third party extension
- Python provides utilities for packaging
  - distutils

The subprocess module

See:

http://docs.python.org/library/subprocess.html

Subprocesses

- Say you’d like to call an existing program, pass it input, and capture the output
- The subprocess module defines one class, Popen(), and convenience functions
- To run a command, without interacting with it,

```
import subprocess
subprocess.call('ls -l’)
```

Shell

- Expands variables in the command string using the system’s environmental variables

```
import subprocess
# Command with shell expansion
subprocess.call('ls -l $HOME', shell=True)
```

Communicate

```
#You can communicate with processes by
#piping stdout, in, err

from subprocess import Popen, PIPE
cmd = 'ls'
p = Popen(cmd, stdout=PIPE, stderr=PIPE)
# comm. waits for the process to finish
stdout, stderr = p.communicate()
stdout = stdout.decode()
print (stdout)
```

Passing arguments

```
from subprocess import Popen, PIPE
p = Popen(['echo', 'foo'], stdout=PIPE,
  stderr=PIPE)
stdout, stderr = p.communicate()
result = stdout.decode()
print (result.strip())


$python3 foo.py
foo
```

GUIs (code is Python 3.x, except for the image, menu, and list slides which are 2.x)

See:

http://wiki.python.org/moin/TkInter

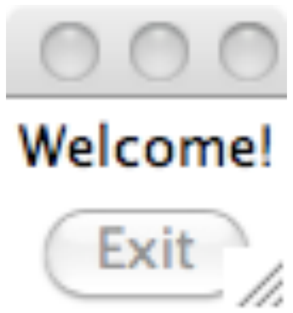
GUIs and Python

- Programmed through a toolkit
  - Library supplying controls (widgets)
  - Toolkit lets you compose controls, display them and interact with them
- A number of toolkits are available for Python
  - Tkinter, wxPython, PyQt, PyGTK
- Tkinter is included with Python
  - We will only cover the basics here

Tkinter

- Object-oriented wrapper around Tk
  - Tk is a cross-platform toolkit which can be used with other scripting languages like Tcl, Ruby, Perl
- Cross platform
  - Runs on Windows, Unix-like platforms, Mac
  - More or less mimicks native look, feel
- Note on examples
  - Meant to be run as stand-alone scripts
  - If you run from within IDLE or other scripts with GUIs, anomalies may pop up

Fundamentals



- Import Tkinter module
- Development process
  - Create, configure and position widgets
  - Enter Tkinter main loop
- Your application becomes event driven
  - User interacts with widgets, firing events
  - Application responds via handlers you write for these events
- A first example (from Python in a Nutshell)
  - import sys, Tkinter as tk # capitialized in Python 2.x, lower case in 3.x
  - tk.Label(text='Welcome!').pack()
  - tk.Button(text='Exit', command=sys.exit).pack()
  - tk.mainloop()
- In this simple case, we do not need to bind widgets to named variables
- Configurations specified as named arguments
- No parent window(s) specified, so widgets are placed on main window
- Calling pack() passes off layout handling to a default layout manager

The Main Loop

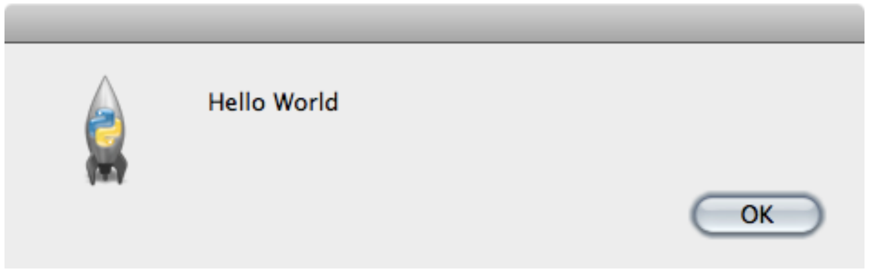
- Calling the Tkinter.mainloop() function enters the Tkinter main loop and the program becomes event driven
- Tkinter will respond, as expected, to user-driven events
  - E.g. moving the window, minimizing, maximizing, etc.

Dialogs

- Tkinter provides modules to define dialogs (modal boxes)
- Some commonly used modules
  - tkMessageBox
    - Message dialog, with simple input options
  - tkSimpleDialog
    - Subclass to create your own dialogs
  - tkFileDialog
    - Dialog for choosing files or directories
  - tkColorChooser
    - Dialog for choosing colors

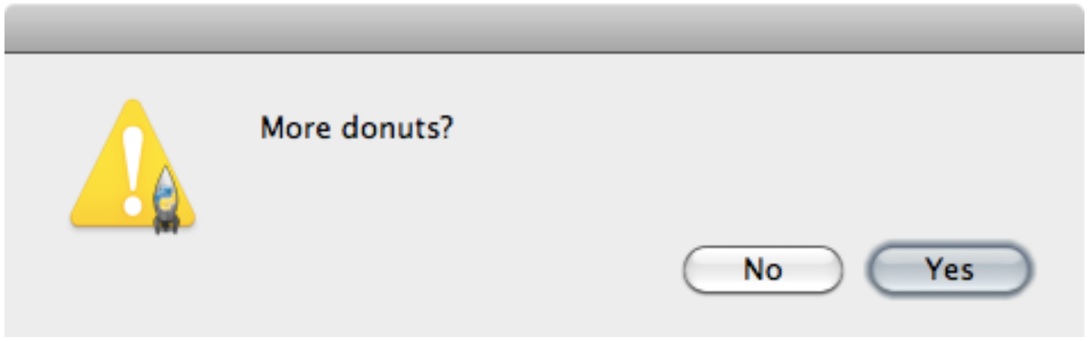
Dialog Box

```
import tkinter as tk
import tkinter.messagebox as box

top = tk.Tk()
def hello():
  box.showinfo("Say Hello", "Hello World")

b1 = tk.Button(top, text = "Say Hello", command = hello)
b1.pack()

top.mainloop()
```



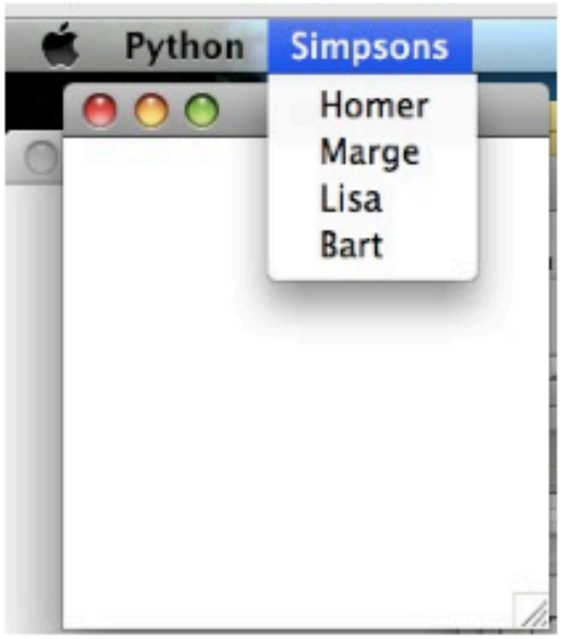
Hello World

Yes No


More donuts?

```
import tkinter as tk
import tkinter.messagebox as box

top = tk.Tk()
def ask():
   r = box.askyesno("Homer", "More donuts?")
   #box.showinfo("Say Hello", "Hello World")
   print (r)
b1 = tk.Button(top, text = "Ask", command = ask)
b1.pack()

top.mainloop()
```

Menus



```
import Tkinter

root = Tkinter.Tk()

menubar = Tkinter.Menu()

def handle_click(menu, entry): print menu, entry

filemenu = Tkinter.Menu()

for x in 'Homer', 'Marge', 'Lisa', 'Bart':

    filemenu.add_command(label = x, command=lambda
x=x:handle_click('Simpsons', x))

menubar.add_cascade(label='Simpsons', menu=filemenu)

root.config(menu=menubar)

Tkinter.mainloop()
```

```
Simpsons Marge
```

Widgets

- A widget is a class which contains code to display a common gui object (e.g., a button, or a listbox)
- When instantiating, first argument is the parent window (master) of the widget
  - If the first argument is omitted, the application main window is the master
  - All other arguments are named
- To change a option on an existing widget, use the widget‟s config(...) function
  - E.g. for widget w, function form is: w.config(option=value)
  - Alternatively, each widget is a mapping object, so you can also do: w[„option‟] = value

Common Widgets

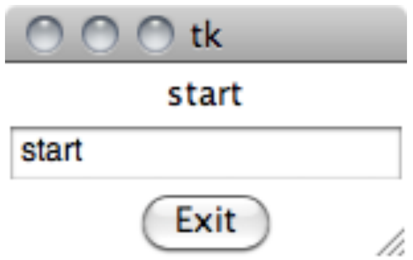
- Tkinter provides some common widgets for simple GUIs
  - Button
  - Checkbutton
  - Entry
  - Label
  - Listbox
  - Radiobutton
  - Scale
  - Scrollbar
- And more

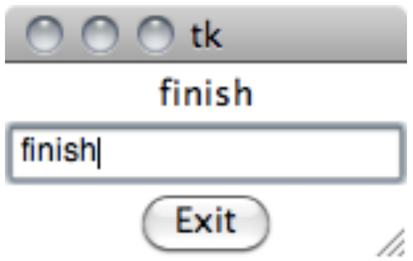
Tk Variables

- Tkinter provides classes whose instances represent variables for some data types
  - E.g. DoubleVar for floats, IntVar for integers, StringVar for strings
- Sort of like mutable versions of the built-in immutable types
- Variable objects can be passed as textvariable or variable in configuration options for widgets
- When the variable is changed, the widget will automatically update
- Instantiate one of these classes to get a variable object
  - For variable object x, x.set(...) and x.get() sets and returns the value of x, respectively

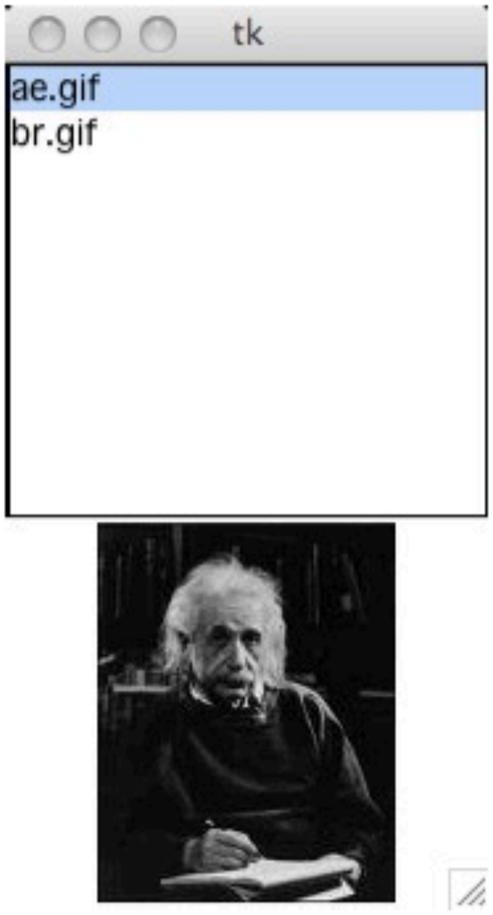
Tk Variables and Widgets

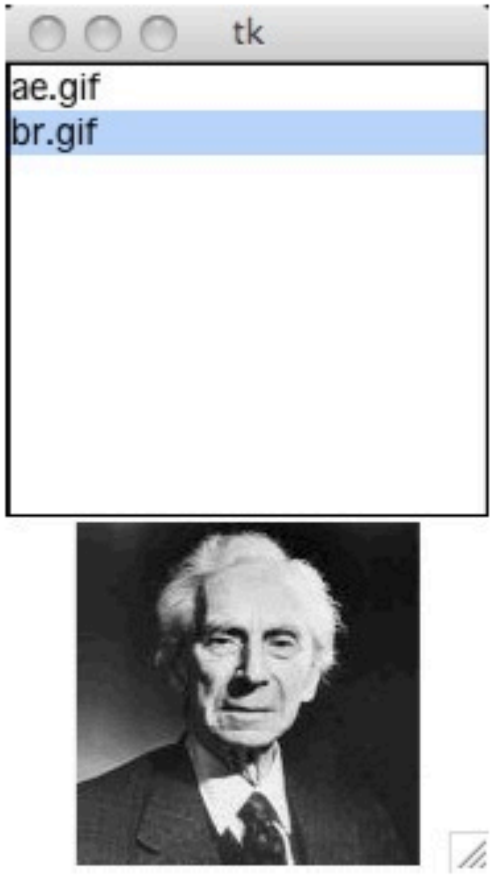
```
import tkinter as tk
import tkinter.messagebox as box

root = tk.Tk()
tv = tk.StringVar()
tk.Label(textvariable=tv).pack()
tk.Entry(textvariable=tv).pack()
tv.set('start')
tk.Button(text='Exit', command=root.quit).pack
  ()
tk.mainloop()
```





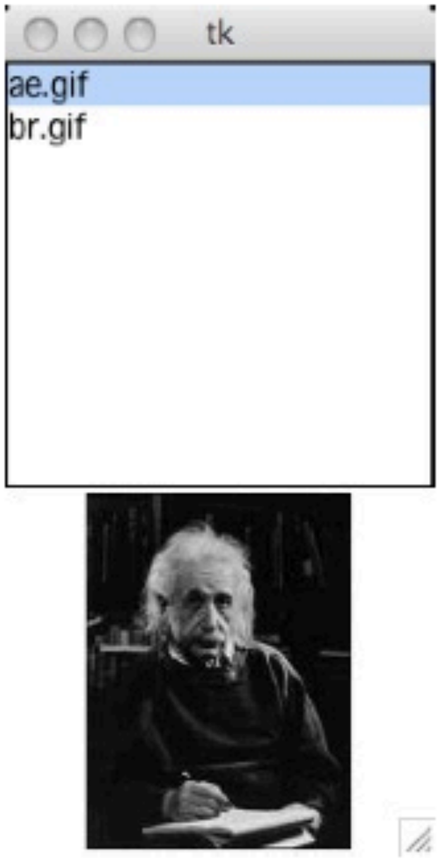
Tkinter Lists, Images, and Clicks





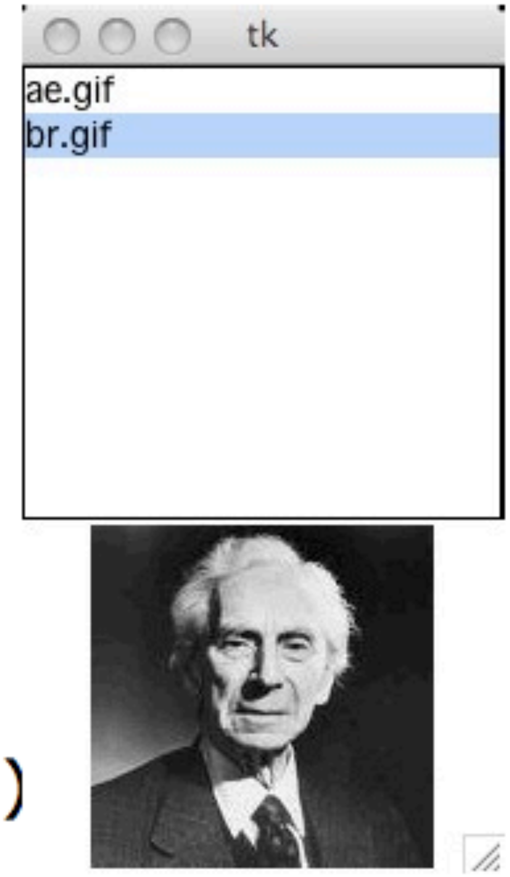
Tkinter Lists, Images, and Clicks

```
import os,  Tkinter

root = Tkinter.Tk()
L = Tkinter.Listbox(selectmode=Tkinter.SINGLE)
imgdict = {}
path= '/Users/josh/Desktop'
for name in os.listdir(path):
    if not name[-3:] == 'gif': continue
    imgpath = os.path.join(path, name)
    img = Tkinter.PhotoImage(file=imgpath)
    imgdict[name] = img
    L.insert(Tkinter.END, name)

L.pack()

...
```



Tkinter Lists, Images, and Clicks

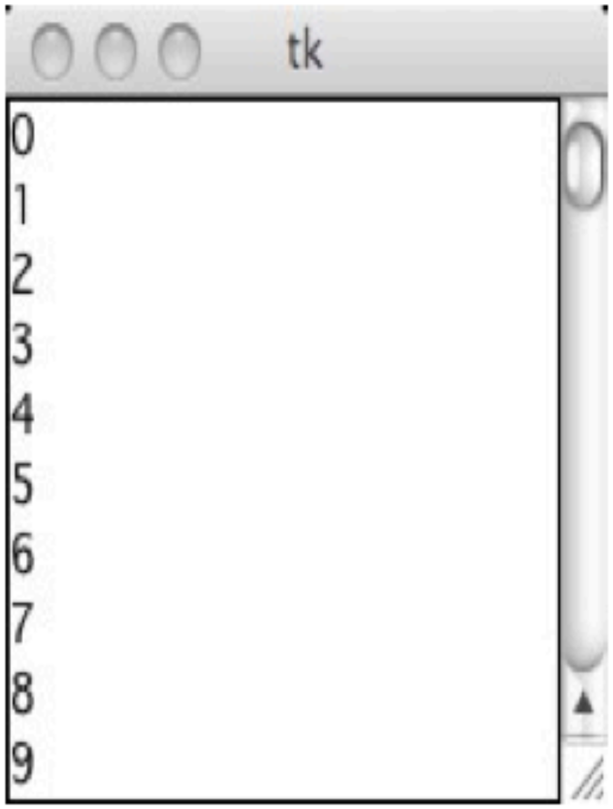
```
...

label = Tkinter.Label()
label.pack()


def list_entry_clicked(*ignore):
    name = L.get(L.curselection()[0])
    label.config(image=imgdict[name])


L.bind('<ButtonRelease-1>',list_entry_clicked)

root.mainloop()
```



Tkinter Lists and Scroll bars

Listbox can display textual items, selection capability
L.delete(0, END) #clear the box
L.insert(End, foo) #insert a string to the back

```
import Tkinter
S = Tkinter.Scrollbar()
L = Tkinter.Listbox()
S.pack(side=Tkinter.RIGHT, fill=Tkinter.Y)
L.pack()
S.config(command=L.yview)
L.config(yscrollcommand=S.set)
for i in range(100):
    L.insert(Tkinter.END, str(i))
Tkinter.mainloop()
```

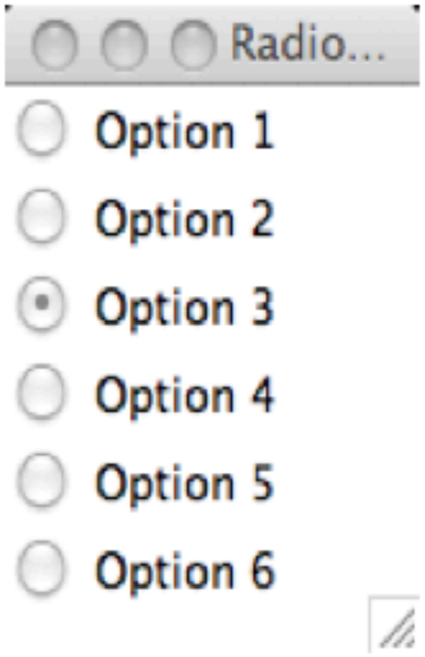


```
0
1
2
3
4
5
6
7
8
9
```

Tkinter Radio Buttons

```
from Tkinter import *
root = Tk()
root.title('Radiobutton')
opts =[('Option 1', 1), ('Option 2', 2), ('Option 3', 3),
      ('Option 4', 4), ('Option 5', 5), ('Option 6', 6)]
var = IntVar()
def which():
   print var.get(), 'selected'
for text, value in opts:
   Radiobutton(root, text=text, value=value,\
     variable=var, command=which).pack()
var.set(3)
root.mainloop()
```

```
4 Selected
5 Selected
```



Tkinter reference

- http://docs.python.org/library/tkinter.html
- http://www.pythonware.com/library/tkinter/introduction/
- http://wiki.python.org/moin/TkInter

Databases

The Python Database API (DBAPI)

- Python specifies a common interface for database access, but the standard library does not include a RDBMS (relation db management sys) module - why?
- Designed to encourage similarity between database implementations – pick a module, same patterns apply
- Defines common connection objects, cursor objects, types, etc

Implementations

- There are many free third-party modules (including XML db support).
- Pretty much these all work the same way programatically
- Differences are mostly in SQL variations
- PostgreSQL: http://www.initd.org/
- PostgreSQL: http://pybrary.net/pg8000/
- MySQL http://sourceforge.net/projects/mysql-python
- MSSQL: http://pymssql.sourceforge.net/

DBAPI Pattern

- Download and install the DBAPI implementation
- Import the module and call the connect function (when you're finished, remember to close it)
- Specify the server address, port, database, and authentication
- Get a cursor, use it to execute SQL (cursors are emulated for DBs which do not support them)
- Fetch results as a sequence of tuples, one tuple per row in the result, where tuple indexes correspond to columns
- Cursors pretty much work the way you expect in other languages, just with less code.
- Standard methods on cursors: fetchone(), fetchmany(), fetchall()

Accessing a MySQL Database, getting column names

```
import MySQLdb

#create a connection object
con = MySQLdb.connect('127.0.0.1', port=3306,
   user='tomato', \ passwd='squish', db='test’

cursor = con.cursor() #get a cursor

sql = 'SELECT * FROM Simpsons #some quick sql
cursor.execute(sql) #execute and fetch

results = cursor.fetchall()
print (results) # returns a list of column, value
   dictionaries

con.close() #close the connection
```

Insert data into a MYSQL Database

```
import MySQLdb # Open database connection

db = MySQLdb.connect
  ("localhost","testuser","test123","TESTDB" )

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = “INSERT INTO SIMPSONS(TITLE, AUTHOR) VALUES (’The
  Tao of Homer', ’HJS’)”

try: # Execute the SQL command
  cursor.execute(sql)
  db.commit()
except: # Rollback in case there is any error
  db.rollback()
# disconnect from server
db.close()
```

References

- http://wiki.python.org/moin/DatabaseProgramming/
- http://www.tutorialspoint.com/python/python_database_access.htm