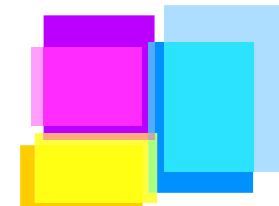
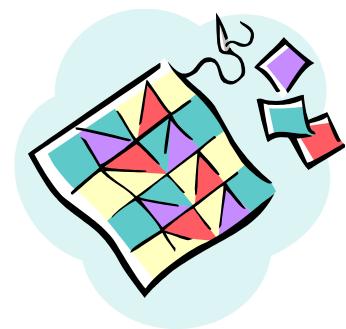
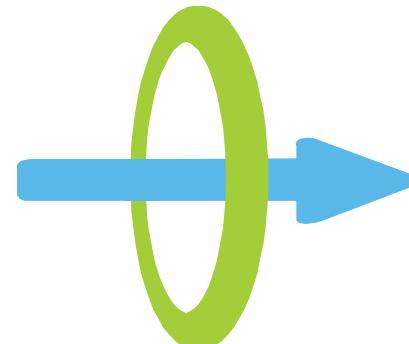
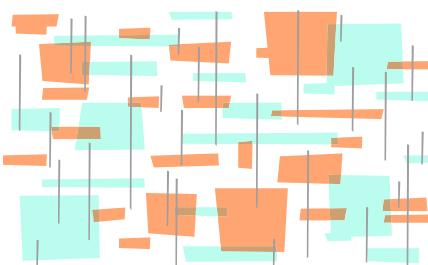


CS I

Introduction to Computer Programming

Lecture 10: October 26, 2015

Testing and Graphics



Last time

- Binary and hexadecimal numbers
- Dictionaries



Today

- Testing your code
- Introduction to computer graphics
 - continued next class
- Keyword arguments to functions



Testing your code

- Code that has never been tested should be considered wrong *by definition*
- Once code is written, the first thing that should be done is to test it
- Different ways to go about this
 - informally
 - systematically



Informal testing

- Informal testing means to run the code interactively
 - on whatever examples come to mind
- Once you're done, what can you say?
 - "It seems to work OK"
- This is rarely good enough



Systematic testing

- Systematic testing means to have some set of *test cases* that you run to test your code *automatically*
- If you change your code, can re-run the test cases easily to see if you broke anything



Exhaustive testing?

- No amount of testing can guarantee that your code is 100% free of bugs
- "Exhaustive testing" (testing for every single possible bug) has been called "Exhausting testing"
- Better strategy: when a bug comes up, write a new test case
 - So it won't come up again!
 - Get better *code coverage* over time



Unit tests

- Tests that apply to a particular module are usually called *unit tests*
- They test the code in a module in isolation from all other code
- These are the easiest tests to write



Integration tests

- Tests that check if multiple modules work properly in combination are usually called *integration tests*
- These are more complicated, and we won't cover them in this course
- But with good enough unit tests, you can still eliminate huge numbers of bugs!



Aside: Type checking

- Many languages also allow the programmer to specify the input and output types for each function
- These can be checked before the program has even been run, potentially catching large numbers of bugs
- This is called "*static type checking*"



Aside: Type checking

- Static type checking is a wonderful feature, but Python doesn't have it ☹
- Tradeoff:
 - Python is more dynamic than most languages
 - Python doesn't check types until programs are actually run
- [Take CS 4, CS 115 for more on static type checking]



Testing in Python

- Testing code in Python is easy
- With the **nose** module, it's *really* easy!
- Need to understand:
 - the **assert** statement
 - **if __name__ == '__main__': ...**
 - How to write individual tests
 - How to write and run test modules



The `assert` statement

- `assert` is a special Python statement which is useful for both debugging and testing
- It tests whether a boolean expression is **True** or **False**
- If it's **True**, nothing happens
- If it's **False**, an error occurs



The assert statement

- Examples:

```
>>> assert True
```

[nothing happens]

```
>>> assert False
```

AssertionError

```
>>> assert 1 + 1 == 2
```

```
>>> assert 2 + 2 == 5
```

AssertionError



The assert statement

- Can also add a message to an **assert**:

```
>>> assert 1 + 1 == 2, 'addition test 1'  
>>> assert 2 + 2 == 5, 'addition test 2'  
AssertionError: addition test 2
```

- This is useful as documentation



Using `assert` for debugging

- `assert` is often used as a way to make functions self-checking:

```
def square_root(x):  
    # code to compute the square root  
    root = ...  
  
    assert (abs(x - root * root) < 0.001)  
  
    return root
```



Using assert for debugging

- Now, every time you use `square_root` it will check that the number it computed is the actual square root
 - within a tolerance (floating point math is never exact)
- Problems:
 - tests are mixed in with the regular code
 - tests slow down the running of the code



Using `assert` for unit testing

- Using `assert` this way is OK
- Another approach is to write the code without `asserts`, and then to write a separate *test module* to test each function you wrote separately
- Such test modules are *unit tests* (where the "unit" is a module)
- Before we get to that, we need to explain one more thing...



name

- Inside a module, the variable name is set to be
 - the name of the module (when the module is imported)
 - the special name 'main' (when the module is run directly by Python)
- Let's see how this works



name

- Here's a simple module: `example.py`

```
# module: example.py
```

```
print 'My name is: %s' % __name__
```

- It defines a Python module called `example`
- When it's imported, this code will be run
- Let's import it from other Python code:

```
>>> import example
```

```
My name is: example
```

- Nothing strange so far...



name

- What happens when the file `example.py` is executed directly by Python instead of being imported?
 - e.g. at the terminal command line
 - or by loading into WingIDE and hitting the run button

% `python example.py`

My name is: main

- This says: `example.py` is the first thing executed by Python



```
if __name__ == '__main__':
```

- We can use this to define some code in a module which only executes if the module is the first thing Python executes:

```
# example.py
if __name__ == '__main__':
    print 'This is the main module.'
```



```
if __name__ == '__main__':
```

- Now, at the terminal command line (or after loading into WingIDE and hitting the Run button):

```
% python example.py  
This is the main module.
```



```
if __name__ == '__main__':
```

- But when importing the module in the Python shell:

```
>>> import example
```

- Nothing happens!
- Can use this trick to execute some code *only* when a module is loaded as the main module



Writing test modules

- A test module has this form:

```
import nose
```

```
from <module to test> import *
```

```
def test_<function 1>:
```

```
    ...
```

```
def test_<function 2> :
```

```
    ...
```

```
if __name__ == '__main__':
```

```
    nose.runmodule()
```



Writing test modules

- Example module to test: `zero.py`

```
# zero.py

def first_zero(lst):
    '''Return the index of the first zero
    in a list, or -1 if no zeros.'''
    for i, e in enumerate(lst):
        if e == 0:
            return i
    return -1
```



Writing test functions

- Function to test `first_zero`:

```
def test_first_zero():
    assert first_zero([]) == -1
    assert first_zero([0]) == 0
    assert first_zero([1, 0]) == 1
    assert first_zero([1, 1, 1]) == -1
```

- If `first_zero` is correct, calling `test_first_zero` will do nothing
 - Otherwise, an error will occur
- Could add more tests if desired



The test module

- The test module: `test_zero.py`:

```
import nose
```

```
from zero import *
```

```
def test_first_zero():
```

```
    # as before
```

```
if __name__ == '__main__':
```

```
    nose.runmodule()
```

- That's it!



Running the test module

- Run `test_zero.py`:
% `python test_zero.py`

.

`Ran 1 test in 0.001s`

`OK`

- Success!



Running the test module

- Can have multiple tests in a test module
 - typically one per function in the module being tested
- If a test fails, get output describing which test failed and at which line
- Test failures may indicate
 - a bug in the code
 - a bug in the tests!
- Either way, need to fix it



Summing up

- How the **nose** module actually runs all these tests has not been described
 - uses some advanced Python features ☺
- However, *using nose* is very simple
- Writing tests may seem tedious, but will save you *enormous* amounts of time on bigger programming projects
- Need to get comfortable with this now



Interlude

- Python!



Topic 2

- Computer graphics!



What does "graphics" mean?

- Graphics is (loosely speaking) the process by which you create visual images on a computer screen
- Graphics also involves the process of *interacting* with these visual images in a meaningful fashion



Text-based programming

- The programming we've done so far has been *text-based*
- We've written games that work from the terminal (Mastermind), programs that read and write text files, etc.
- Text-based programming has a simple notion of...



User interface

- A *user interface* refers to how you (the user) interact with a program
- Text-based programs tend to have very simple user interfaces
- Programs that create and use graphics can have much more complex user interfaces



Text-based user interfaces

- A text-based program typically has one of two kinds of user interface:
 - *batch mode*
 - *interactive mode*



Batch mode

- A *batch mode* program is run without any user involvement
 - Example: compute and print π to 1000 decimal places
 - The computer computes a while and then prints out **3.1415926...** and finally halts
 - Once the program has started, the program's user just waits for the answer



Interactive mode

- An *interactive mode* program is run with the help of the user
 - Example: Mastermind game from lab 2
 - You enter a guess
 - The computer tells you how good it was
 - You enter another guess
 - etc. until you guess correctly
- You and the computer "take turns"



Graphical user interfaces

- Programs that do graphics usually don't fit into either of these categories
- Instead, they have a *graphical user interface (GUI)* which users interact with directly



Graphical user interfaces

- The program provides various visual entities (called *widgets*) that you can interact with
 - buttons, menus, sliders, scrollbars, etc.
 - drawing surfaces for drawing
- The program also displays output visually
 - images, animations, etc.



Example

- The "desktop" of a computer running Linux





Blackjack



Chess



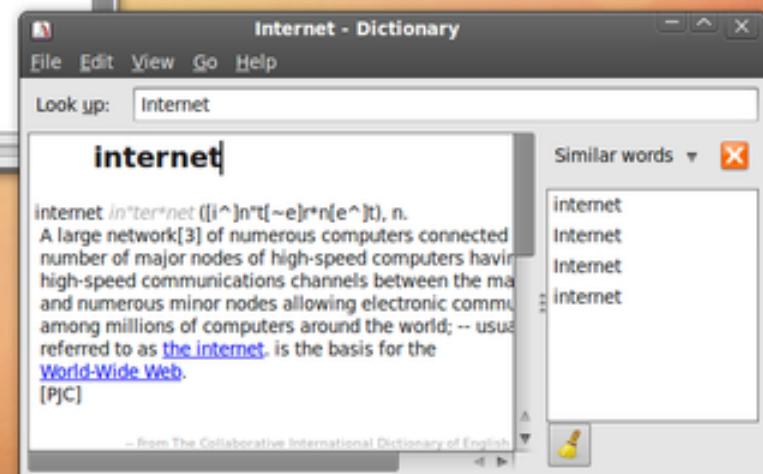
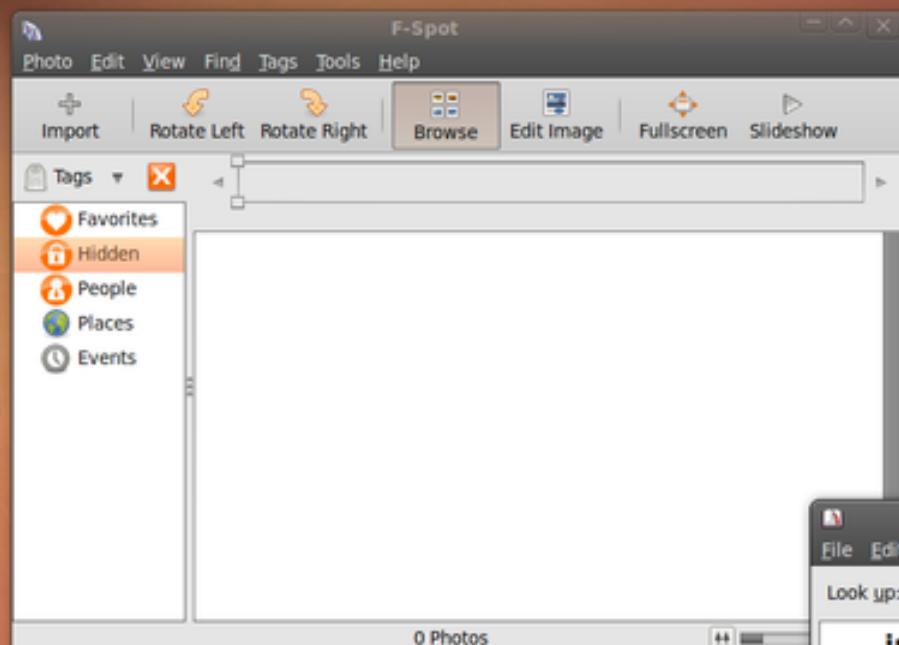
GIMP Image Editor



Dictionary

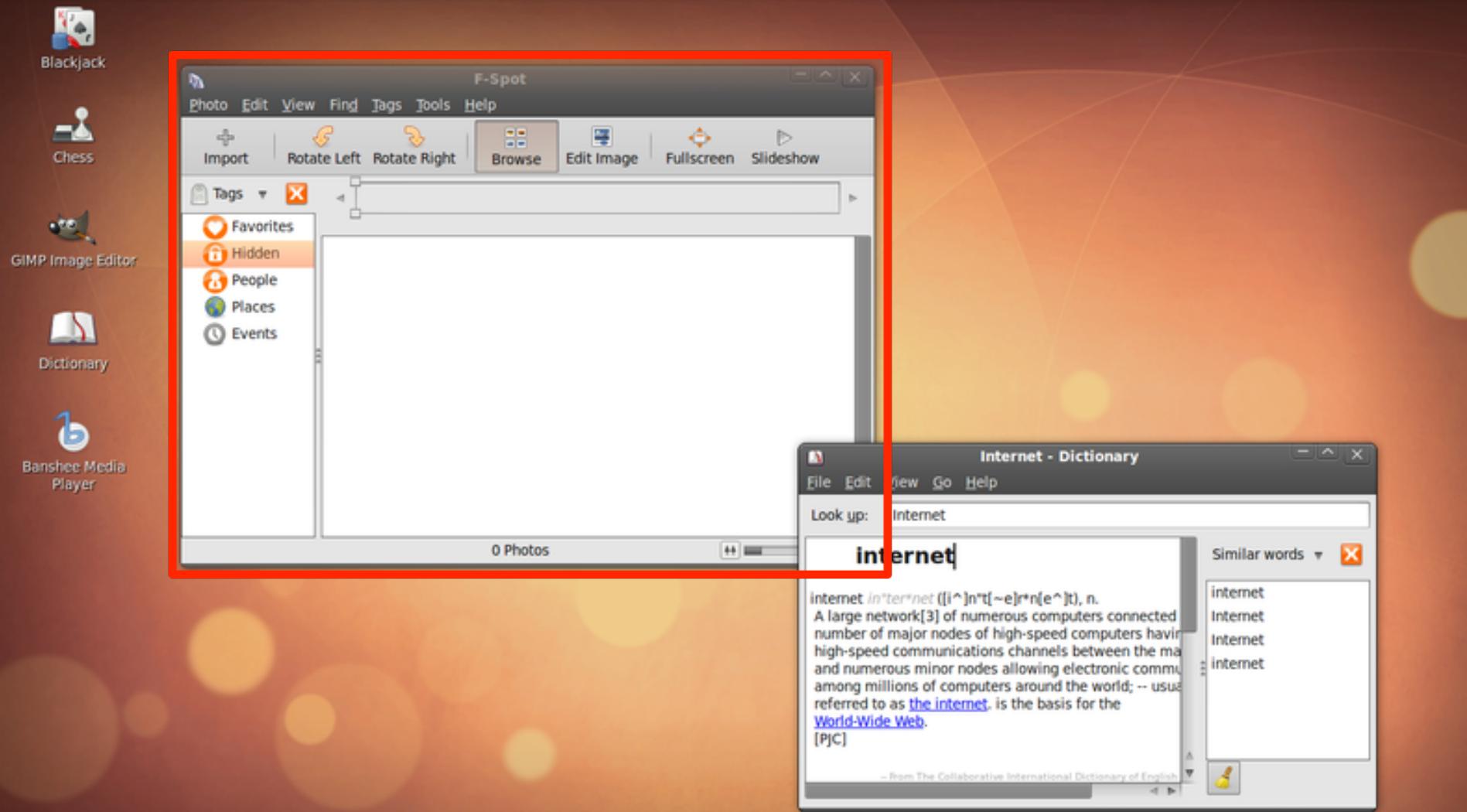


Banshee Media Player



Application (program)

Applications Places System ☰ 📁 ? Sun May 3, 11:02 AM Jaunty Jackalope ☰



Menu

Applications Places System ☰ 📁 ? Sun May 3, 11:02 AM Jaunty Jackalope ☰



Blackjack



Chess



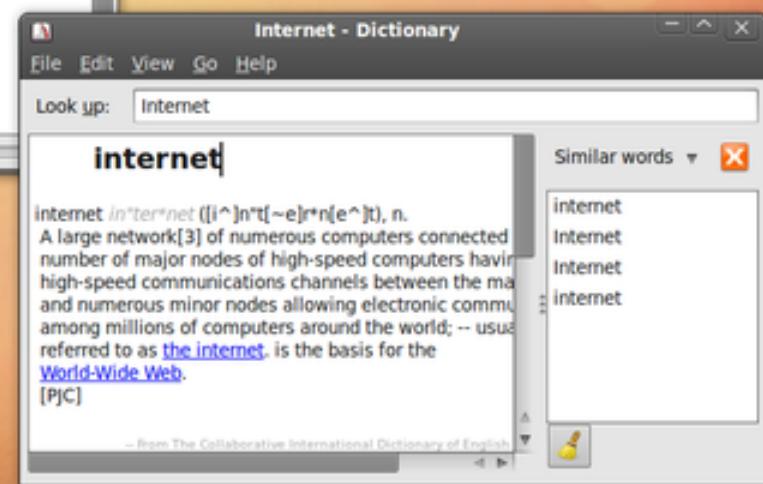
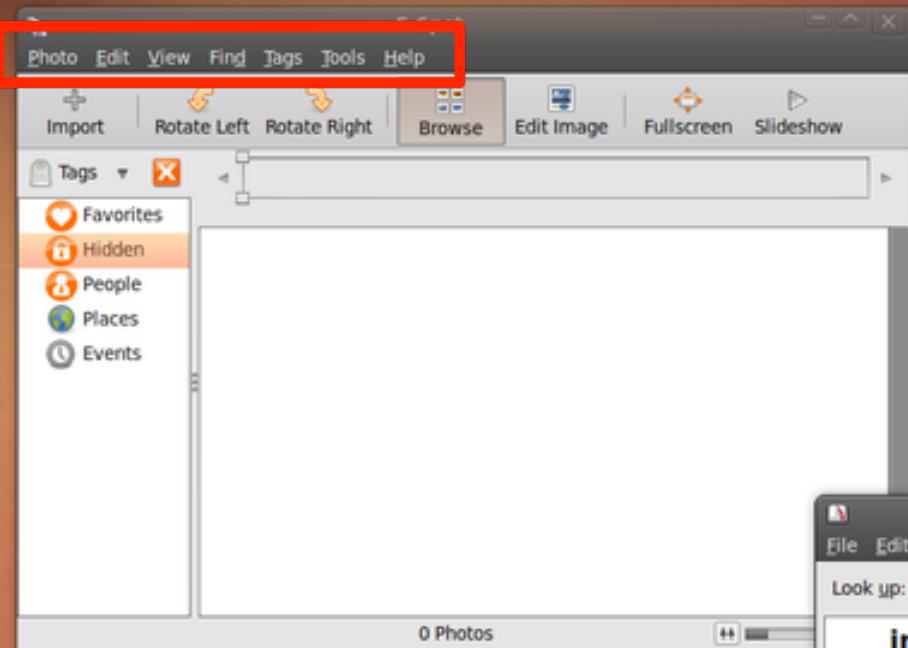
GIMP Image Editor



Dictionary



Banshee Media Player



Button

Applications Places System ☰ 📁 ? Sun May 3, 11:02 AM Jaunty Jackalope ☰



Blackjack



Chess



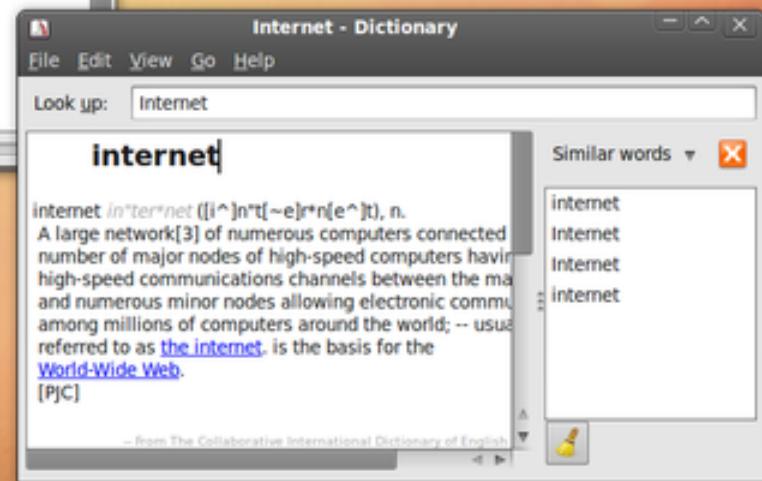
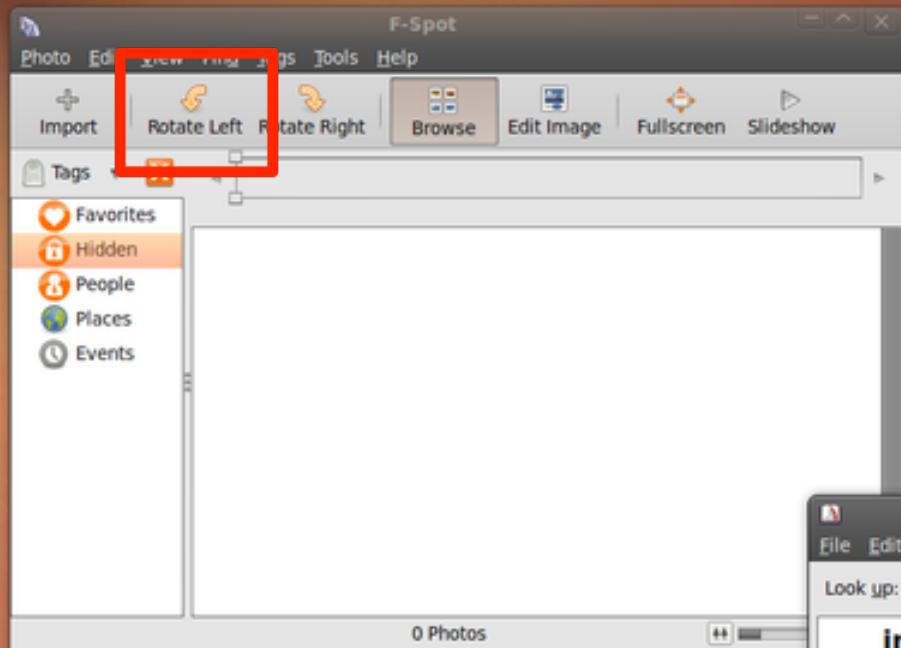
GIMP Image Editor



Dictionary



Banshee Media Player



Graphical interfaces

- Graphical interfaces are so common today, we don't even notice them
 - unless they go wrong ;-)
- But their interfaces are very complex
- They are designed to make it feel "natural" for you to interact with the program



Event loop

- Typically, a program will wait for the user (you) to activate one of its widgets
 - push a button, select a menu item, draw on a drawing surface etc.
- Then it will do something in response
- Then it will go back to waiting for you to do something again
- This process is called an *event loop*
 - your actions are the "events" the program is waiting for



Programming GUIs

- Programming graphical user interfaces (GUIs) is somewhat laborious
 - (some people find it boring)
- The programmer must anticipate every reasonable thing the user might want to do with the program
 - then provide a graphical object (widget) to allow that to happen
- Good news: graphics programming is about more than this!



2-D and 3-D graphics

- Aside from graphical user interfaces, there are two other broad categories of graphics programming:
- **2-D** (two-dimensional) graphics: drawing pictures (or animations) on a two-dimensional surface
- **3-D** (three-dimensional) graphics: drawing pictures (or animations) to resemble three-dimensional objects



2-D graphics example



3-D graphics example



3-D graphics example



Today

- We will only be dealing with 2D graphics today
 - With a teensy bit of user interface thrown in for good measure
- We'll continue and expand our examples in later lectures



Python and graphics

- Many kinds of graphics libraries are available in Python
 - 2-D, 3-D, GUI, etc. (many of each)
- However, none are built-in
 - all require that you add the libraries to the basic Python installation
- Today, we'll look at the most commonly-used graphics library in Python



Tkinter

- The most common graphics library in Python is called **Tkinter**
 - perhaps because all the sensible, meaningful, pronounceable names were taken?
- It's a Python **interface** to a system called "**Tk**" (which stands for "graphics **Toolkit**") written in a different language



Tkinter

- Tkinter provides a number of tools for writing programs that use graphics:
 - many GUI widgets
 - buttons, menus, labels, scrollbars etc.
 - a **canvas** widget on which arbitrary drawings can be created
 - using lines, circles, rectangles, ovals, images, text, etc.
 - ways to capture user interaction
 - key presses, mouse clicks, etc.



Tkinter

- We will concentrate on the canvas widget and drawing simple 2-D pictures
- We'll also show how to get a program to respond to actions (key presses) on the canvas



Simple Tkinter program

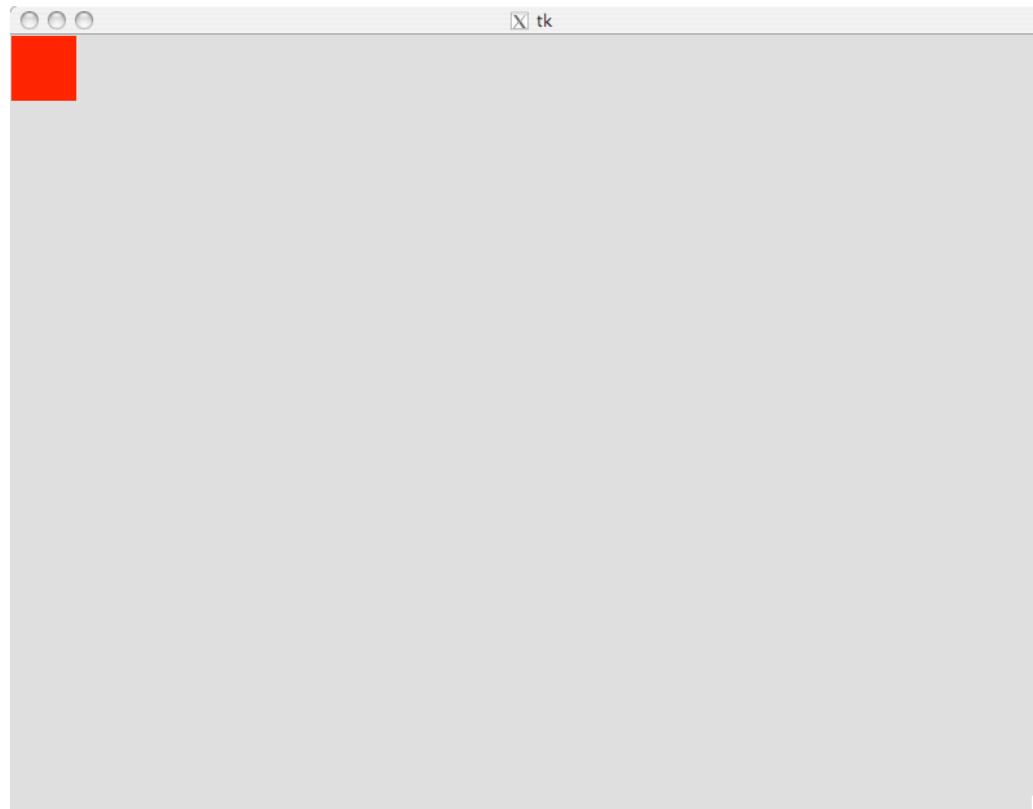
- In file **tkinter1.py**:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
                      fill='red', outline='red')
raw_input("Press <return> to quit")
```



Simple Tkinter program

- This program brings up a window with a red rectangle drawn in one corner:



Simple Tkinter program

- To understand how this works, we first have to understand
 1. pixel coordinates
 2. windows
 3. Python keyword arguments



Pixel coordinates

- To the computer, the entire screen is a 2-dimensional grid of tiny colored boxes called *pixels*
- Most computer screens have large numbers of pixels
 - e.g. 1280x800 pixels on this computer
 - or about 1 million pixels
- With millions of possible colors per pixel



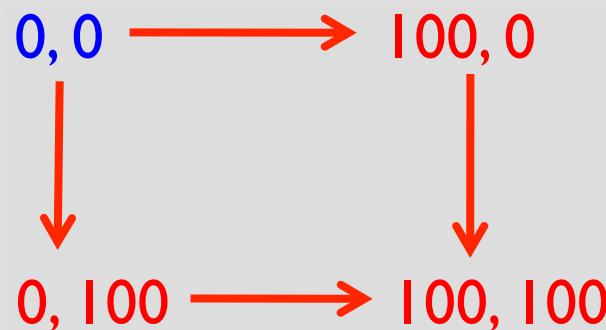
Pixel coordinates

- The pixel in the upper left-hand corner is pixel $(0, 0)$ (called the *origin*)
- The first pixel in the pair represents the horizontal dimension
 - so $(100, 0)$ would be to the right of $(0, 0)$
- The second pixel in the pair represents the vertical dimension
 - so $(0, 100)$ would be below $(0, 0)$



Pixel coordinates

- Visually:



Windows and pixels

- Most computers run multiple programs at one time, each in a separate *window*
- Pixel coordinates can be
 - absolute
 - relative to a particular window
- Absolute coordinates means the upper left-hand corner of the monitor is (0, 0)
- Relative means the upper left-hand corner of a particular window is (0, 0)
- Almost always use relative coordinates



Keyword arguments

- Last time, talked about *dictionaries*
 - store key/value pairs in a single data structure
 - keys are usually strings
- Python also allows functions to get key/value pairs as arguments to functions
 - as long as the key is a string
- All the key/value pairs are put into a dictionary before the function sees them
- These are called *keyword arguments*



Keyword arguments

```
# Keyword argument example:  
def foo(x, y, **kw):  
    print x, y  
    print kw  
  
>>> foo(1, 2, width=100, height=200)  
1 2  
{ 'width' : 100, 'height' : 200 }
```



Keyword arguments

- In the definition of `foo`:

```
def foo(x, y, **kw) :
```

- the `**kw` means that all the keyword arguments will be put into a dictionary called `kw`
- the `**kw` has to come at the *end* of the argument list
 - for boring technical reasons



Keyword arguments

- When calling the function `foo`:

```
foo(1, 2, width=100, height=200)
```

- the keyword arguments are `width` and `height`
- Inside the function `foo`, they get put into the `kw` dictionary, which becomes:

```
{ width: 100, height: 200 }
```



Keyword arguments

- Keyword arguments are useful in functions where you want to be able to specify arguments by name
- **Tkinter** uses keyword arguments a lot
- Usually the meaning is intuitive
 - e.g. **height** means height in pixels, **width** means width in pixels



Back to the example

- We had these lines:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- Let's see what they mean...



Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line means: import all names from the **Tkinter** module
- Use **from Tkinter import *** form because writing **Tkinter.<name>** for every name would be very tedious to write and to read



Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line means: create the *root window* of the program
- This is the window in which all the other graphical components of the application will be placed
- It is a Python object, so has methods



Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line calls the **geometry** method on the root object
- This line means: set the size of the root window to be **800** pixels wide (horizontal dimension) by **600** pixels deep (vertical dimension)



Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- You could run this as a whole program
- If you did, a blank window of size 800 by 600 would appear on the screen and then go away almost immediately
- Need a way to make the screen stay up!



Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
raw_input('Press <return> to quit.')
```

- If this is the whole program, you get a blank window of size 800x600 which stays up until you press the return key in the terminal window



To be continued...

- Need to continue this next class
- Finish discussion of example program
- Also talk about *event handling*
 - how to make graphical programs respond to key presses, mouse clicks, etc.

