The cPickle module contains functions for storing objects to files and loading objects from files:

cPickle module functions	
Example	**Description**

`import cPickle`
`fileObj = open(fileName, 'w')`
`cPickle.dump(objectToSave, fileObj)`
	Writes the `objectToSave` to the previously opened `fileObj`.

`import cPickle`
`fileObj = open(fileName, 'r')`
`myObj = cPickle.load(fileObj)`
	Reads the object from the `fileObj`.

1. Using the simple Die and AdvancedDie classes previously discussed, write a program that:

- opens a file for writing
- creates 25 AdvancedDie objects all with 100 sides and puts them into a list
- prints the rolls of all 25 die objects
- writes these 25 AdvancedDie objects to a file using cPickle.dump,
- closes the file
- re-opens the file for reading
- reads 25 Advanced Die objects into a list of AdvancedDie objects
- prints the rolls of all 25 die objects

2. The `try-except` statement (buried in section 8.3.8) is useful for validating user input. Its syntax is:

```
try:
      <statements which might raise an exception>
except <exception type>:
      <statements to perform upon the exception>
```

We can use a `try-except` statement to write a better int_input function as:

```
def int_input(prompt):
    while True:
        try:
            userString = raw_input(prompt)
            myInt = int(userString)
            return myInt
        except Exception, e:
            print "Integer input error: please try again"

x = int_input("Enter an integer: ")
print "x =",x
```

Write a similar function definition for float_input that uses a try-except statement to validate a float input.

Intro. to Computing (810:051) **Lecture 20** **Name:_____**

Much of today's software uses a point-and-click graphical user interface (GUI). The standard library modules `Tkinter` and `Tix` allow for portable, event-driven, GUI development in Python. A Python/Tkinter program is entirely event-driven, i.e., the program

- constructs tailored GUI "widgets" via inheritence (LabelDemo inherits Frame) to suit the program's needs,
- registers associated event-handles (*call-back functions*) for "widgets" with the Python virtual machine, then
- waits for the user to generate events (e.g., clicking the mouse on, entering some text into a "field", typing some key at the keyboard, etc.) that the event-handles react to

A very simple example would be to display a Label that's tailored as:

```
"""  File: labeldemo.py """
from Tkinter import *

class LabelDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Label Demo")
        self.grid()
        self._label = Label(self, text = "Hello world!")
        self._label.grid()

def main():
    """Instantiate and pop up the window."""
    LabelDemo().mainloop()

main()
```

Conceptually, the mainloop of Tkinter performs the following algorithm:

```
def mainloop():
     while the main window has not been closed:
          if an event has occurred:
               run the associated event handler function
```

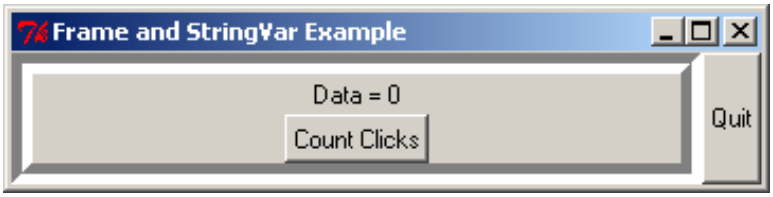
A Button can have an associated event-handler (the `command` argument) that gets called when it is pressed.

```
"""  File: buttondemo.py """
from Tkinter import *

class ButtonDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Button Demo")
        self.grid()
        self._label = Label(self, text = "Hello")
        self._label.grid()
        self._button = Button(self, text = "Click me",
                              command = self._switch)
        self._button.grid()

    def _switch(self):
        """Event handler for the button."""
        if self._label["text"] == "Hello":
            self._label["text"] = "Goodbye"
        else:
            self._label["text"] = "Hello"

def main():
    """Instantiate and pop up the window."""
    ButtonDemo().mainloop()

main()
```

After running and pressing the button a couple times, the window's label toggles from "Hello" to "Goodbye".

The `.grid()` method is a layout manager for your GUI widgets that allows you to treat every window/frame as a two-dimensional grid of rows and columns. By default the width of a column is the width of its widest cell, similarly for rows. Widgets too small for their cells can expand (or not) by using the `sticky` option in the `.grid()` method, and specifying directions (N+S+E+W) for expansion. Widgets can also span multiple cells by specifying rowspan and columnspan options in the `.grid()` method.

You can group a set of related widgets into a frame and nest it inside an outer frame. Each frame has its own grid layout, so the gridding of widgets within each frame works independently. Below is a simple example that constructs a frame (myFrame), then packs a label ("Data = 0") on row 0 and a button ("Count Clicks") on row 1. This frame is placed into the enclosing frame on row 0 and column = 0, and a quitButton on row 0 and column = 1 of the enclosing frame. The quitButton is sticky to the North and South so it extends from the top and bottom of the enclosing frame. Upon executing the below code we get the following window:




```
# NestedFrames.py
from Tkinter import *
import tkMessageBox

class Hello(Frame):                          # Extend Frame class
    def __init__(self, parent=None):
        Frame.__init__(self, parent)         # call superclass init
        self._top = self.winfo_toplevel()    # find top-level window
        self._top.title("Frame and StringVar Example")
        self.grid()
        self._data = 0
        self._showDataString = StringVar()   # create a String control variable
        self._showDataString.set('Data = %d' % (self._data))
        self._make_widgets()                 # attach widgets to self

    def _make_widgets(self):
        """Create GUI widgets and place them in the frame"""
        myFrame = Frame(self, borderwidth=10, relief=GROOVE)
        myFrame.grid(row=0, column=0)
        countLabel = Label(myFrame, textvariable=self._showDataString,
                           width=50, justify=CENTER)
        countLabel.grid(row = 0)
        myButton = Button(myFrame, text='Count Clicks', command=self._message)
        myButton.grid(row = 1)

        quitButton = Button(self, text='Quit', command=self._quit)
        quitButton.grid(row=0, column = 1, sticky=N+S)

    def _quit(self):
        print "Quit"
        quitAnswer = tkMessageBox.askokcancel(message='Do you really want to quit?',
                                              parent=self)
        if quitAnswer:
            self._top.quit()
        else:
            print "Quit cancelled"

    def _message(self):
        self._data += 1
        self._showDataString.set('Data = %d' % (self._data))
# end class Hello(Frame)

def main():
    Hello().mainloop()

main()
```

Every time the "Count Clicks" button is pressed, the "Data = #" label is updated using the *control variable*, showDataString. A control variable is a special object that can be shared by several different widgets and updated automatically on the screen by using the `.set( value )` method. Control variables also have a `.get()` method to return its current value. Control variables of type integer and float can be gotten using using the constructors `IntVar()` and `DoubleVar()`, respectively.

Some examples where you might want to use control variables are:

- The `Entry` widget allows you to enter a line of text, which is normally linked to a control variable.
- A group of radiobuttons normally share a single control variable that allows them to tell which one gets selected and allows it to be cleared.
- Checkbuttons use a control variable to hold its current state: on or off.

An example using the Entry widget to allow a line of text to be entered is:

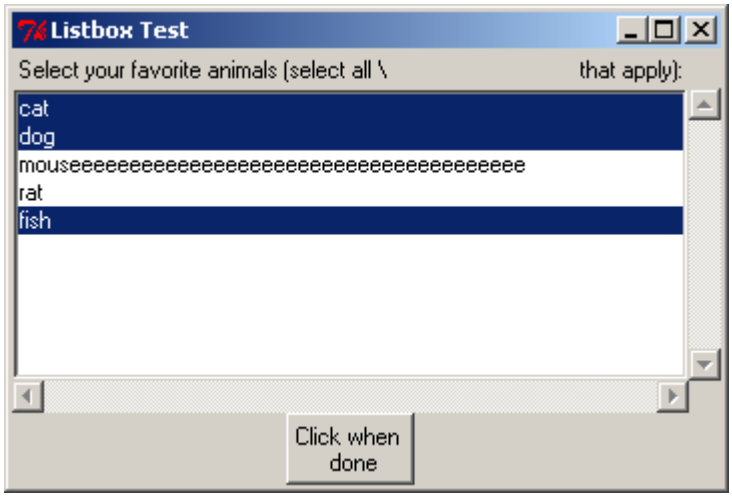
```
# EntryTest.py
"""Test the Entry widget"""

from Tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid()
        self.name = StringVar()
        self.name.set("Enter name here")
        self.age = IntVar()
        self.age.set("Enter age here")
        self.createWidgets()

    def createWidgets(self):
        self.textEntry = Entry(self, takefocus=1,
                               textvariable = self.name, width = 40)
        self.textEntry.grid(row=0, sticky=E+W)

        self.ageEntry = Entry(self, takefocus=1,
                               textvariable = self.age, width = 20)
        self.ageEntry.grid(row=1, sticky=W)


# end class Application

myApp = Application()
myApp.mainloop()
```

The corresponding window is:

The `takefocus=1` allows you to <tab> between these items.

1. Modify the above code to add a "Done" button to the EntryTest.py program that prints the contents of both Entry widgets to a `tkMessageBox` using the `showinfo(`*`title, message`*`)` function. (You would need to import the `tkMessageBox` module as done in the textbook in Table 9.2.)

2. A Listbox widget allows the user to select several lines of text at the same time.



For the selections shown in the window, the "Click when done" button causes the following output:

```
('0', '1', '4')
Favorite Animals:
0 cat
1 dog
4 fish
```

The following code illustrates a Listbox widget that allows selection of several lines of text.

```
"""ListboxTest.py   Test the Listbox widget with event handling"""
from Tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.master.title("Listbox Test")
        self.grid()
        self._createWidgets()

    def _createWidgets(self):
        prompt = Label(self, text="Select your favorite animals (select all that apply):")
        prompt.grid(row=0, sticky=W+E)
        yScroll = Scrollbar(self, orient=VERTICAL)
        yScroll.grid(row=1, column=1,sticky=N+S)

        xScroll = Scrollbar(self, orient=HORIZONTAL)
        xScroll.grid(row=2, column=0,sticky=E+W)

        self._myListbox = Listbox(self, xscrollcommand=xScroll.set,
                                  yscrollcommand=yScroll.set, selectmode=MULTIPLE)

        self._myListbox.grid(row=1, column=0, sticky=N+S+E+W)
        self._myListbox.insert(END, 'cat')
        self._myListbox.insert(END, 'dog')
        self._myListbox.insert(END, 'mouseeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee')
        self._myListbox.insert(END, 'rat')
        self._myListbox.insert(END, 'fish')

        xScroll["command"] = self._myListbox.xview
        yScroll["command"] = self._myListbox.yview
        doneButton = Button(self, text='Click when \n done', command=self._readSelections)
        doneButton.grid(row=3)

    def _readSelections(self):
        selectionsTuple = self._myListbox.curselection()
        print selectionsTuple
        print "Favorite Animals:"
        for i in selectionsTuple:
            print i, self._myListbox.get(i)

# end class Application

def main():
    """Instantiate and pop up the window."""
    Application().mainloop()

main()
```

Refactor the Python code so that it populates the Listbox from a list of animals, e.g., ['cat', 'dog', ...].