Python for Scientific Computing 3

Konrad Hinsen
Centre de Biophysique Moléculaire, CNRS
Rue Charles Sadron; 45071 Orléans
E-Mail: hinsen@cnrs-orleans.fr

10 October 2003

Overview

- Advanced array operations
- GUIs with Tk

1 Advanced array operations

Purpose:

- high-level operations simplify many programs
- loop-free treatment of large data sets for efficiency
- avoid having to use C/Fortran modules in many cases

2 Arrays revisited

Multidimensional arrays:

```
a[0, 1:2, ::-1, 2:3:10]
```

Conceptual model:
list [of lists [of lists ...]]
One-dimensional structure operations therefore act by default on the first index, treating an n-dimensional array as a list of $n-1$-dimensional arrays.
Example: looping

```
for sub_a in a:
    print sub_a
```

3 Advanced indexing

```
from Numeric import *

a = array([[[1, 2], [3, 4]],
           [[11, 22], [33, 44]]])

# last axis
print a[..., 1]
```

```
[[ 2  4]
 [22 44]]
```

```
# first and last axis
print a[0, ..., 1]
```

```
[2 4]
```

4 Advanced indexing

```
# insert axis
print a[0, ..., 1][NewAxis, :]
```

```
[[2 4]]
```

```
print a[0, ..., 1][:, NewAxis]
```

```
[[2]
 [4]]
```

5 Reference semantics

Arrays consist of two parts: a data space that holds the elements, and a bookkeeping part that stores size and dimension information.
Indexing creates a new array object with its own bookkeeping part but using the same data space. The elements are not copied.
Advantage: saves space and time when working with big arrays.

Watch out when

- changing array elements (other arrays may indirectly be changed as well)
- keeping a reference to a subarray of a deleted big array (the big data space is kept)

6 Array arithmetic

```
b = array([[6, 5],
           [4, 3],
           [2, 1]])

# add arrays elementwise
print a+b
```

```
[[7 7]
 [7 7]
 [7 7]]
```

7 Array arithmetic

```
# add row to array
print a+b[0]
```

```
[[ 7  7]
 [ 9  9]
 [11 11]]
```

```
# add number to array
print a+b[0, 0]
```

```
[[ 7  8]
 [ 9 10]
 [11 12]]
```

8 Array arithmetic

```
# add column to array
print a+b[:, 0, NewAxis]
```

```
[[7 8]
 [7 8]
 [7 8]]
```

```
# comparison
print a > b
```

```
[[0 0]
 [0 1]
 [1 1]]
```

9 Array arithmetic

```
# in-place addition
a += 3
print a
```

```
[[4 5]
 [6 7]
 [8 9]]
```

10 Math functions

- Module `math`: float objects
- Module `cmath`: complex objects
- Module `Numeric`: many object types
  - Numbers: int (cast to float), float, complex
  - Arrays of numbers
  - Sequences of numbers: cast to arrays
  - Class instances: method call

⇒ always use `Numeric`.

11 Binary functions

```
from Numeric import *

a = arange(5)
b = a[::-1]
c = zeros(a.shape, a.typecode())

# standard addition
print add(a, b)
```

```
[4 4 4 4 4]
```

```
# sum over elements
print add.reduce(a)
```

```
10
```

12 Binary functions

```
# cumulative sum
print add.accumulate(a)
```

```
[ 0  1  3  6 10]
```

```
# outer product sum
print add.outer(a, b)
```

```
[[4 3 2 1 0]
 [5 4 3 2 1]
 [6 5 4 3 2]
 [7 6 5 4 3]
 [8 7 6 5 4]]
```

13 Binary functions

```
# addition to preallocated array
add(a, b, c)
print c
```

```
[4 4 4 4 4]
```

```
# watch out!
add(a, b, a)
print a
```

```
[4 4 4 7 8]
```

14 Structural operations

```
from Numeric import *

a = (1+arange(4))**2
print a
```

```
[ 1  4  9 16]
```

```
# selection by index
print take(a, [2, 2, 0, 1])
```

```
[9 9 1 4]
```

```
# selection by value
print where(a >= 2, a, -1)
```

```
[-1  4  9 16]
```

15 Structural operations

```
# reshaping/resizing
print reshape(a, (2, 2))
```

```
[[ 1  4]
 [ 9 16]]
```

```
print resize(a, (3, 5))
```

```
[[ 1  4  9 16  1]
 [ 4  9 16  1  4]
 [ 9 16  1  4  9]]
```

```
# element replication
print repeat(a, [2, 0, 2, 1])
```

```
[ 1  1  9  9 16]
```

16 GUI basics

Python knows many GUIs
Popular GUI libraries for use with Python are Tk (discussed here), wxWindows, and Qt. Each has its particular strong points. We use Tk because it is available by default in every Python installation. For big programs, Qt is often a better choice.

Event-driven computing
In GUI programs, user interactions cause specific "events" (mouse clicks, menu selections, ...) that the program has to handle. The events determine the program flow.

17 Tk's world view

Tk knows windows and widgets (from "window" and "gadget"), e.g. buttons, labels, menus, ...

Widgets form a hierarchy, every widget is contained within its master (another widget or a windows).

The layout of widgets within the same master is defined through a geometry manager. There are three of them, but only the simplest one (grid) will be used here. It places widgets on a rectangular grid within their master.

18 A simple example

```
from Tkinter import *

def report():
    print "The button has been pressed"

object = Button(None,
                text = 'A trivial example',
                command = report)
```

```
object.grid()
object.mainloop()
```

19 Arranging widgets

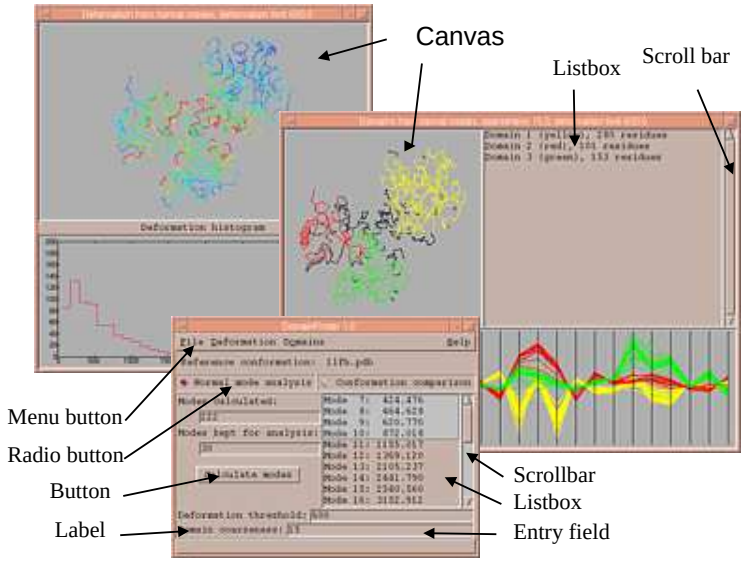
```
from Tkinter import *

def report(label):
    print "Button %s" % label

outer_frame = Frame(None)
outer_frame.grid()

top_frame = Frame(outer_frame)
top_frame.grid(row=0, column=0)
button1 = Button(top_frame,
                 text='Button 1',
                 command=lambda: report('1'))
button1.grid(row=0, column=0)
button2 = Button(top_frame,
                 text='Button 2',
                 command=lambda: report('2'))
button2.grid(row=0, column=1)

bottom_frame = Frame(outer_frame)
bottom_frame.grid(row=1, column=0)
button3 = Button(bottom_frame,
                 text='Button 3',
                 command=lambda: report('3'))
button3.grid(row=0, column=0)

outer_frame.mainloop()
```

20 Widget overview

- Frames contain other widgets.

- Buttons cause a specific action.
- Labels display short texts.
- Check buttons allow a yes/no choice
- Radio buttons allow a one-out-of-several choice.
- List boxes show lists of items and allow selection.
- Text objects provide text browsers and editors.
- Scroll bars permit scrolling in texts and list boxes.
- Canvases are for graphics displays.
- Entry fields let the user enter data (one line).
- Menu buttons and menus create pop-up menus.

21 Widgets in action



22 Defining widgets

Widgets are standard Python objects defined by classes. Special-purpose versions can be created by subclassing. In particular, compound widgets can be created by subclassing `Frame`.

A compound widget may contain not only GUI code, but also computations executed in response to GUI events. However, it is usually better to separate computation and GUI.

23 Widget definition

```
from Tkinter import *

def report(): print "A button has been pressed"

class ButtonLine(Frame):
  def __init__(self, master, button_data):
    Frame.__init__(self, master)
    i = 0
    for text, command in button_data:
      button = Button(self, text=text,
                      command=command)
      button.grid(row=0, column=i)
      i += 1

outer_frame = Frame(None)
outer_frame.grid()

first = ButtonLine(outer_frame,
                   [('Button 1', report),
                    ('Button 2', report)])
first.grid(row=0)

second = ButtonLine(outer_frame,
                    [('Button 3', report),
                     ('Button 4', report)])
```

```
second.grid(row=1)

outer_frame.mainloop()
```

24 Windows

A new window is created by `Toplevel()`. It can be used as a master widget for its contents. Usually one defines the contents as a compound widget subclassed from `Frame`.

```
class WindowContents(Frame):

   ...

new_window = Toplevel()
new_window.title(filename)
WindowContents(new_window, text).grid()
```