

NOTES ON LOOPS & ANIMATION

Some comments on Python and programming

variables are the names associated with objects. Variable names are any string of letters, numbers, and underscores (e.g., **myvariable_43**), but can't start with a number (**3bin** is illegal.)

Some objects are simple, like *integers* (1,2,3, -29), *floats* (3.1415).

Some are more complicated and have many attributes, like a **box** object in Visual.

key words or *reserved words* are little words like **for**, **in**, **while**, **print**, etc., that are illegal variable names, since they have special meanings.

procedures or *functions* are things that look like variables, but have parentheses in addition, like **range(10)**, **range(200,300,2)**, **sin(4.3)**, **box(x=9,color=color.green)**. The stuff, if any, inside the parentheses are the *arguments* to the function. The meaning of the *arguments* is determined either by position, like in **range**, or by name, like in **box**. These procedures do things, like make a list or make a box or calculate a mathematical function. We will see examples as we go along.

for loops

Loops are crucial to writing computer programs. The purpose is to have a computer do something, maybe hundreds of times, maybe billions of times, without having to write out the same command hundreds or billions of times. You need to be able to set up repetitive tasks, like “check my e-mail every second, all day long, for new mail” or “simulate the future of the solar system by calculating the position of the Earth every year, for the next billion years.”

Loops are in all computer languages. How do they look in Python? In outline form, they look like the following.

```
for _____ in _____ :  
    statement1  
    statement2  
    statement3, etc.
```

The **for**, **in**, and **:** are required to say to Python this is a for loop.

In the first blank goes a variable name, which can be any legal name, such as **i**, **myvariable**, **kristy**, **a**, **doornumber3**, or **uv40**, for example.

In the second blank goes a LIST. A list can be made up by hand, as in

```
mylist = [1,2,3,10]  
anotherlist = ['a', 43.9, 7]
```

Note that a list is a list of objects, separated by commas and enclosed by square brackets. Another way to make a list is to use a FUNCTION that makes lists. An example of such a function that is built into Python is **range()**. For example, the expression (or function call) **range(10)** makes a list with 10 items, namely, **[0,1,2,3,4,5,6,7,8,9]**, while the function call **range(2,16,3)** counts from 2 to just less than 16, by 3's: **[2,5,8,11,14]**.

The indented statements (or single statement) after the **for** statement is what gets repeated over and over. So if you run the following program:

```
for j in range(10):  
    print "Hello"
```

you will get 10 line of "Hello" printed out. This is because this set of statements means the following:

- ?? Compute what **range(10)** means. This gives the list **[0,1,2,3,4,5,6,7,8,9]**.
- ?? Set **j** to the first item in the list. That is **j=0**.
- ?? Execute the block statement, that is, print out "Hello".
- ?? Set **j** to the second item in the list. That is, **j=1**.
- ?? Again execute the block statement: print out "Hello".
- ?? And so on ... until the computer is done with the list.

Note that in this last example, the value of **j** was not used in the block. It was just a placeholder whose value was not used.

We can also write a **for** loop that *does* use the value of the index variable. Look at this:

```
for j in range(10):  
    print j
```

This will print out the numbers from 0 to 9.

Consider updating a bank balance for a year, where each month you pay \$230 rent and earn \$5 in interest. Suppose you have both a checking and savings account. You could keep track by trying this program.

```
checking = 4500.00           # initial checking balance  
savings = 500.00           # initial savings balance  
  
# now define the monthly changes  
rent = 230.00  
interest = 5.00  
  
# use a for loop to update these balances over 12 months  
for month in range(12):  
    checking = checking - rent
```

```

savings = savings + interest
print "Checking balance: ", checking, "Savings balance: ", savings

```

This program will print out a series of 12 reports, 1 line each, on the status of the checking and savings balances. After setting up the initial amounts and setting how quickly the balances change, the changes are applied 12 times. The block consists of the last 3 statements. What happens is:

```

?? range(12) gives the list [0,1,2,3,4,5,6,7,8,9,10,11].
?? month is set to the first item in the list, so month = 0.
??      checking is updated
??      savings is updated.
??      The balances are printed out.
?? month is set to 1
??      checking is updated.
??      savings is updated.
??      The balances are printed out.
?? month set to 2
?? ..... and so on, until all of the list items have been gone through.

```

If you are having trouble with loops, I suggest you try the above program out and modify it some – changing the number of months, printing out the month number in addition, or other changes.

Changing objects

When we are using Python objects, such as the **box** and **sphere** objects in VPython, we want to be able to modify their attributes, such as color or position or size. For example, to make a sphere and then change its radius, we can use the program:

```

from visual import *

a = sphere(radius=2)      # make a sphere of radius 2, assign it to a
a.radius = 3             # now change its radius to 3

```

This will do the trick, but note that it will happen too fast to see.

So let's use a **for** loop to make a growing sphere. Here's a program that will do that.

```

from visual import *

a = sphere(radius=2)

for j in range(1000):
    rate(20)
    a.radius = a.radius + 0.01

```

This is just like the checking balance program, in that the value of something is being changed, except we can *see* the change via Visual. 20 times a second, the radius of the sphere **a** will be increased by 0.01. Summarize in English, this program means to the computer:

Import the visual tools, so I can draw. Then make a sphere of radius 2, named **a**. Repeat this action 1000 times: increase the radius of **a** by 0.01. But the **rate(20)** statement reminds me to do this at most 20 times a second, so those slow humans can see it happen.

MAPS

Time moves forward. As time moves forward, things change. If we believe in physical determinism, the future is solely determined by the present. If I know where the planets are today, I can predict where they will be tomorrow. The idea of a *map* is based on this ability to predict some time into the future.

For example, if I know that the water level in a lake decreases by 0.5 meters a year, I can write a map to compute next year's water level, based on the current water level:

$$\text{next year's water level} = \text{this year's level} - 0.5$$

In Python, I might write

```
waterlevel = waterlevel - 0.5
```

to update the water level value.

Using a loop, like for the bank balance, would then let me follow the water level over time:

```
waterlevel = 5000.0  
for year in (1900,2050):  
    print "At the start of year", year, "the waterlevel is", waterlevel  
    waterlevel = waterlevel - 0.5
```

When plotted as a function of year, the water level simply decreases linearly in time (as a straight line.)

Another example of a map might be a slowing car. Suppose that the car's speed decreases by 10% each second. That is, the new speed is the old speed multiplied by 0.9. How would we calculate its speed in a loop, over 20 seconds?

Another example is interest rates: consider a bank account that grows by 5% each year. The magnitude of the balance increases in time in a predictable fashion.

The maps that we have discussed so far are not chaotic. They do not have a "butterfly effect".

The lab today will look at how we do *not* see a butterfly effect. Starting next time, we will look at maps that *do* have chaotic behavior.

PHY307, Science and Computers I

Lab #3, September 10, 2002 (due by end of lab period)

Playing with Maps

Summary of what you will do:

You will use loops to compute the effects of maps. The maps will be simple; you will see how to *visualize* the maps and to see for yourself that the maps are non-chaotic. You will also have time to polish your programming knowledge.

PYTHON NOTES:

- ?? Graphing will be introduced here. We will also visualize the maps using “hopping” objects.
- ?? When watching hopping objects, set `scene.autoscale = 0`, so that you can see what happens more clearly (if you don't do this, the camera viewpoint will be changing.)

LOOPING

1. Log in to your workstation using your CMS/SUnix name & password.
2. Open up some editor or word processor to start your report. Put in a title, your name, and date. As you write the report, make it clear what each section is. At the end of the lab, print your report out, staple it together, and hand it in.
3. Get IDLE for VPython going by going through the Menu: Start ? SU Academic Departments ? Physics.
4. Open a program file editor by “New” under the “File” menu and save your (currently empty) work to your I: or K: drive as “loop1.py” or a similar name.
5. Write a program that will make a cone that hops around. The x -position of the cone will be the negative of the old x -position. For a map, we write this as “ x goes to $-x$ ” or “ $x \rightarrow -x$ ”. Have the cone hop 6 times a second. Start the cone at position $x=5$.
6. Does the butterfly effect show up in this map? Create two cones, say **a** and **b**, separated by a small distance, say $x=5.0$ and $x=5.1$. Have both cones hop (in the **for** loop, set $a.x = -a.x$ and $b.x = -b.x$.) Remember to have a rate limiting statement inside your **for** loop. Do the cones stay together or diverge? In your report, include your program and describe what you see.
7. Now do the same with another map. In this map, have the cones x position multiplied by **0.99** each time step. In your report, include your program and describe what you see.
8. Now we will repeat these same two maps, but plotting, at the same time, the positions on a graph. I'll lecture on how to do this, today.