
Visual Python

KMUTT

CS101

June 2003

Sidney Raffer
sraffer@it.kmutt.ac.th

Python is a general-purpose programming language that resembles Java in many ways, but is easier to get started with. Visual python is a three-dimensional graphics library designed for python. Our objective is to cover enough of the language so that you can use the visual library.

There are two python books on your CD: “How to Think Like a Programmer” and the “Quick Reference Guide.” Use them to look up topics in these lecture notes that give you trouble. From python help you can access a book-length tutorial written by the inventor of the language, and also the “Visual Python Manual.” Make sure you can find all of these resources.

Read these notes in front of a computer. Type in the examples and experiment with them. *Play.* You cannot learn a programming language just by listening to lectures. You need personal experience.

To install python on your computer, open the “Welcome” page on your CD. Click on “Python” and then “Vpython.” Be sure that you install python before vpython.

To run python, go to the start menu and select “python 2.2” and then “IDLE for vpython.” Make sure you choose IDLE for vpython and *not* “IDLE for python.” To run a program in the IDLE, hit the F5 function key.

Here is a python program. It is on your CD in the python folder as ball.pyw

```

from visual import *
scene=display()
scene.autoscale=0
s=sphere()
s.radius=.5
b=box()
b.pos=[0,-5,0]
b.size=[10,.3,11]
b.color=color.red
x=0
while 1:
    s.pos=[0,4.5*sin(x),0]
    x=x+.01
    rate(400)

```

Load this file into the python interpreter and run it, just to make sure everything works. You should see a little movie of a silver ball bouncing on a blue surface. You can move closer to the scene by holding both mouse buttons down and dragging. You can fly around the ball as it bounces by dragging with the right mouse-button.¹

¹If you have a three-button mouse these functions might be different. Experiment.

1. Assignment Statements. We begin our study of python with assignment statements. These have the form

$$\text{Variable} = \text{Expression.}$$

A variable is any sequence of letters and numbers starting with a letter. Expressions denote values, such as $2 + 2$. The symbol $=$ *doesn't mean equals*. It means that the value of the expression on the right will be calculated and then assigned to the variable on the left. Any other value the variable may have had will be discarded.

Example.

```
a=2
b=a*3
```

assigns 2 to a and then 6 to b .

Example.

```
a=3
a=a-2
```

assigns 3 to a and then 1 to a . In the end a is 1.

The right-hand-side of each statement is evaluated before anything happens to the variable on the left. No variable is modified while the right-hand-side of an assignment statement is being evaluated. For example, in the last instruction the quantity $a - 2$ is computed “on scratch paper” to be 1, without changing the value of a . Only then is the number 1 assigned to the variable a .

Example. (Assume a and b have already been assigned the values 1 and 2.)

```
c=a
a=b
b=c
```

When this is all finished a is 2 and b is 1. In general these three lines have the effect of switching the values of a and b . Think this through. Why do we need c ?

2. Print and Input. When a program runs you don't see the commands as they are executed, except for the ones that display something on the screen. The **print** statement displays results in the output window.²

Type in and run the following.

```
a=0
b='This is a one: '
print b, a+1
```

²In more advanced python programs there is no “output window” and no “vpython IDLE.” Programs run in their own windows with menus and text-areas and other features created by the programmer. In fact, “vpython IDLE” is written in python. If you are interested in this, try a Google-search on “Tkinter.”

Note that multiple items can be listed in the print statement, and they will all be printed on one line. Each new print statement begins a new line.

The sentence in quotes is a **string**. Strings can be assigned to variables. There are four commands for manipulating strings that we will find useful. String addition works as follows:

```
'dog'+ 'cat'='dogcat'
```

To convert an integer to a string, use *str*.³ To convert a string to an integer use *int*. To count the number of characters in a string use *len*.

Example.

```
print 1+int('1')
```

gives 2, and

```
print str(1)+'1'
```

gives 11, and

```
print len('mississippi')
```

gives 11.

The input command displays a prompt and waits for the user to enter a value. The value is assigned to a variable and then the program resumes.

Example.

```
a=input('Yo! Give me an integer: ')
print a*a
```

This program prompts the user 'Yo! Give me an integer,' and then squares the input.

You can enter any expression that evaluates to a number, such as 2 or 2.7 or even 1+2.7. You can also enter a your input as a string by enclosing it in quotes. The prompt string is optional.

3. Numbers and Expressions. The arithmetic operations +, * and – behave as expected. The symbol / for divide has some surprises. Run this:

```
print 1/2, ' and ', 1./2
```

Python responds with

```
0 and 0.5
```

Why the zero? There are two types of numbers in python, integers and floats. Numbers without decimal points are integers. Numbers such as 2. or 1.1 are floats. The word “type” in the last sentence refers to the way numbers are coded in memory by 0’s and 1’s. The numbers 2 and 2.0 are coded by entirely different sequences of 0’s and 1’s. We shall study this in detail in the Chapter on assembly language.

³Actually, *str* will turn any object into a string. This will become relevant later.

The symbol `/` does division as usual with floats. The symbol `//` does **truncated division** with integers. Informally, the truncated quotient a/b is the number of piles of b beans you can make from a pile of a beans. Formally, $a//b$ is the largest integer less than or equal to the actual quotient a divided by b . Try to predict the output of the program

```
print 11/5, -11/5, 5*(11/5)
```

Then run it to check your answers.

If you combine integers and floats python turns everything into a float. Thus to get the usual quotient of 3 divided by 7 it is enough to write

```
print 3./7
```

The operator `%` gives remainders. The program

```
print 11%7
```

prints out 4, because 4 is the remainder on dividing 11 by 7. Informally, $a\%b$ is the number of beans left over after you make as many piles of b beans from a beans as possible. Formally,

$$a\%b = a - b * (a/b)$$

What does this say? Well, a/b is the number of piles of b beans you can make from a beans. Thus $b(a/b)$ is the number of beans in all the b -bean piles you have made. Therefore $a - b * (a/b)$ is the number of beans left over.

The operator `**` does exponentiation. Thus

```
print 2**3, ' and ', 4**.5
```

gives

```
8 and 2
```

The operator `int` truncates floats.⁴ Formally `int(x)` is the greatest integer less than or equal to x .

4. Lists. A comma-separated list of items of any kind enclosed by square brackets is treated by python as a single object, called an **array** or **list**.

Example. This assignment statement

```
a=[1, 2, 'dog', 'cat']
```

assigns the list on the right to the variable a . The items in the list are named in the following way:

```
a[0] a[1] a[2] a[3]
```

The numbers in the square brackets indicate *position* on the list a , starting from 0. Variables and expressions with integer values can be used inside square brackets to indicate position. For instance

```
a=[1, 2, 'dog', 'cat']
b=1
```

⁴Note that *int* has two meanings, one for strings and one for floats.

```
print a[2*b], a[b+2]
```

prints out “dogcat.”

Expressions of the form

```
a[numerical expression]
```

can appear on the left-hand-side of assignment statements.

Example.

```
a=[1, 2, 'dog', 'cat']
b=1
a[b+1]='frog'
print a
```

produces

```
[1,2,'frog', 'cat']
```

Here are some useful operations on lists.

- a. `len(a)` gives the number of elements of the list a .
- b. `del(a[b])` deletes the b -th element of the list a .
- c. `range(n)` is a list of all integers from 0 to $n - 1$.
- d. `range(n,m)` is a list of all integers from n to $m - 1$.
- e. `range(n,m,k)` is a list of integers starting at n and going in steps of k as far as $m - 1$.
- f. `a.pop()` gives the last element of the list a , and has the side-effect of deleting that element.
- g. The operation `+` is used to join two lists.

Example.

```
print range(2,7)
```

prints out

```
[2, 3, 4, 5, 6]
```

Example.

```
a=range(4)
b=[4,5,6]
print a.pop(),a,a+b
```

prints out

```
3 [0, 1, 2] [0, 1, 2, 4, 5, 6]
```

Many more operations on lists can be found in the python tutorial.

Lists can have other lists as elements.

Example.

```
a=[[1, 'dog', 3], 1, 0, 3]
```

defines a to be a four-element list. Here $a[0]$ is the list

```
[1, 'dog', 3],
```

$a[1]$ is the number 1, etc. What do you think that the statement

```
print a[0][1]
```

would display? What about the statement

```
print a[a[2]]?
```

What about⁵

```
print a[a[2]][1]?
```

5. Functions. Here is a python **function**.

```
def f(x):
    y=x+1
    return y
```

The function f can be used in a program in the following way:

```
def f(x):
    y=x+1
    return y
print f(2)
z=4
print f(z**2)
```

Type in this last program and run it. The output window should show the numbers 3 and 17.

The first line announces that you are defining a function. The symbol f is the name we have chosen for the function. The variable x in parentheses is the input or **argument** of the function. When the function is used the argument x is given a value.

The remaining part of the function, called its *body*, is indented. When the indentation stops python assumes that the definition of the function is finished.

There are two commands in the body of the function. The first adds 1 to the variable x and puts the result in the variable y . The second command is a *return* statement. This tells python what value to give the function when it is used, and ends the execution of the function.

The remaining three unindented lines are not part of the definition of the function. They are the program that uses the function. The function f is used (we say “called”) twice, but we don’t have to write all of the instructions twice. That is why we need functions.

⁵The answers to these three questions are: 'dog', [1, 'dog', 3], and 'dog.'

When the above program runs, python reads the definition of the function and says “OK, now I know what f means.” That is all. No calculations are performed. The first instruction that actually does a calculation is the first print statement. When $f(2)$ is calculated the number 2 is used for the variable x in the definition of the function. When $f(z**2)$ is calculated the variable z has already been assigned the value 2, therefore $2 * 2$ or 4 is used for the variable x in the definition of the function.

Functions can be applied to other functions. For example the following program prints 16. Work this out by hand.

```
def f(x):
    return 2*x
def g(x,y):
    return x+x*y
print g(f(2), g(1,f(1)))
```

Functions need not have return statements. Some functions are used to perform an action rather than to return a value. Functions need not have arguments (i.e. input variables.)

Example. Here is a program with a function whose job it is to print “hello.”

```
def f()
    print ‘‘hello’’
f()
```

Functions with no arguments are called with an empty set of parentheses. When this program is run, python reads the definition of f and then executes the single command $f()$, which has the effect of printing “hello.”

Strings and lists can be used as the arguments of functions.

Example. Here is a function that adds the word ‘dog’ to the end of a list:

```
def f(x):
    return x+['dog']
```

The statement

```
print f([1,2,3])
```

displays

```
[1,2,3,'dog'].
```

Suppose we want a function that usually squares its argument and adds 1, but sometimes we want it to add something other than 1. We can accomplish this as follows:

```
def f(x,add=1):
    return x**2+add
```

```
print f(2)
print f(2,add=3)
```

The output of this program is the two numbers 5 and 7.

The function f has one ordinary argument x and one **default argument** add . If f is called with a single argument, as in $f(2)$, then add is assigned the value 1, as specified in the first line. This is the *default* value of add . The caller can specify the value of add in this way:

```
f(2, add=3)
```

which causes f to return the value 7.

Now we shall talk about local and global variables. You may need to read this section very slowly.

If a variable appears *anywhere* in a function f on the left-hand-side of an assignment statement, or as an argument of f , then *all* appearances of that variable in f are said to be **local** to f .

For example in

```
def f(y):
    z=2
    y=y+z+u
```

the variables y and z are local to f but u is not.

If a variable x is local to a function f , then the definition of x does not extend beyond the definition of f . What does this mean? If a local x also appears as a variable name outside of f , then the “outside” x is treated as a different variable. If the function f changes the local x , no “outside” x changes. And no change in any “outside” x has any effect on the local x .

Variables that appear outside of any function are called **global** variables. If x is a global variable, then the definition of x extends to any function in which x is not a local variable.

Here are some examples.

The program

```
def f():
    x=79
f()
print x
```

results in an error, because the global x in the print statement has not been defined. What about the statement that sets x equal to 79? *That* x is local to f . As far as statements outside of f are concerned the x local to f doesn't exist.

The program

```
def f():
    x=x+1
print x
```

```
x=2
f()
```

also results in an error, because the variable x on the second line is local to f , and f is trying to compute $x + 1$ before x has been defined. Of course there is a global x that has been defined to be 2, but *that* x doesn't exist for f because f has a local x .

The program

```
def f():
    print x
x=1
```

is perfectly fine. The two occurrences of x name the same global variable, and a 1 is displayed by the print statement.

The program

```
1. def f(x):
2.     x=x+1
3.     print x
4. x=2
5. f(x)
6. print x
```

prints 3 and then 2. Why? Well, the first line to be executed is line 4, where a global x is assigned the number 2. Then f is called with the value 2. Inside of f is another x , local to f , which begins life with value 2. This local x becomes 3 at line 2. Next the print statement on line 3 prints the value of the local x , which is 3. Then the function f is finished, and execution resumes at line 6. The print statement on line 6 prints the value of the global x , which has remained 2 all along.

There is more to the story of local and global variables in python than I have told you here. In particular there are some subtleties concerning lists that I am hiding. But this is a start.

Typically one writes a number of functions in the same program, using one inside the other. For example, suppose I want a function f that inputs a list of three two-digit numbers and returns a list of their digit-sums. Here is an example of how f would work:

```
f([12, 13, 14])=[3,4,5].
```

First I write a function g that computes the digit-sum of a two digit number:

```
def g(x):
    return x/y+x%y
```

Then I can use g to define f

```
def f(x):
    return g(x[0])+g(x[1])+g(x[2])
```

A complete program that tests these two functions would look like this:

```
def g(x):
    return x/y+x*y
def f(x):
    return g(x[0])+g(x[1])+g(x[2])
a=input('Give me a 3-element list ')
print f(a)
```

6. Visual Classes. Visual python is a library of graphics functions. To use the library, begin your program with

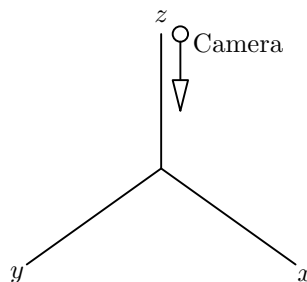
```
from visual import *
```

It will simplify our discussion if all of our scenes have the same fixed scale. Therefore we shall always begin our program with the lines

```
from visual import *
scene=display()
scene.autoscale=0
```

Don't forget the last two lines or you may get confusing results.

A 'scene' is a chunk of space seen through an imaginary camera. Each point in space is named by a list of three numbers, representing a point in a three-dimensional Cartesian coordinate system.



To visualize this, imagine an ordinary two dimensional coordinate system drawn on the screen. The third coordinate measures distance perpendicular to the screen. Positive third coordinates indicate points in front of the screen, and negative third coordinates indicate points on the other side of the screen. The camera is looking at the point $[0,0,0]$ and is about 10 units away, at the point $[0,0,10]$. In other words, the camera is approximately where your eye is.

To summarize, when you look at the screen, the positive z direction goes from the screen to your eye, the positive x direction is to your right, and the positive y direction is up. When you fly around the scene or zoom in using the mouse, nothing in the scene changes, only the position of the camera.

The objects you can place in a scene are known in python (and other languages) as **classes**. A class is a box with compartments, each of which holds a function or variable. If a class c holds a

function f then $c.f$ is the name of that function. If a class c holds a variable x , then $c.x$ is the name of that variable. The variables held by a class are called its **attributes**, and the functions its **methods**.

For example, there is a sphere class. To create a sphere, pick a variable, s let us say, and assign a sphere to it as follows:

```
s=sphere()
```

The reason for the parentheses is that the word “sphere” is actually the name of a function, the sphere-constructor, that creates a sphere. It can be called with no arguments.

A sphere has an attribute pos that determines its center. The following instructions put the center of the sphere s at the origin:

```
s.pos=[0,0,0]
```

The sphere-constructor can also be called with pos as a default argument, as in:

```
s=sphere(pos=[0,0,0])
```

I will stick to the first form, and you should too because your code will be easier to read and debug.

Spheres also have a $radius$ attribute. It defaults to $.5$. The following instructions make the radius of the sphere s equal to 1:

```
s.radius=1
```

We would like a red sphere. A **color** in python is a three element list describing how much red, how much green, and how much blue, on a scale from 0 to 1. Thus for a red sphere we write:

```
s.color=[1,0,0]
```

You can make any color by mixing red green and blue.

Since mixing colors can be tedious, there is a color class called “color” with attributes red green blue white yellow etc., which you can use in place of a triple of numbers. For example this instruction makes a red sphere:

```
s.color=color.red
```

To summarize, we have the following program:

```
from visual import *
scene=display()
scene.autoscale=0
s=sphere()
s.pos=[0,0,0]
s.radius=1
s.color=color.red
```

We can make a number of spheres without writing these lines again and again. Define a function

```
def f(x,y,z,col=color.red,rad=.5):
    s=sphere()
    s.pos=[x,y,z]
    s.radius=rad
    s.color=col
```

Then we can make spheres at different positions, of different sizes and colors.

```
f(0,0,0)
f(0,4,0,col=color.green)
f(4,0,0,rad=3,col=color.blue)
```

Note the *default arguments*. Make sure you understand the effect of each of these function calls. There are other objects besides spheres. Boxes are created as follows:

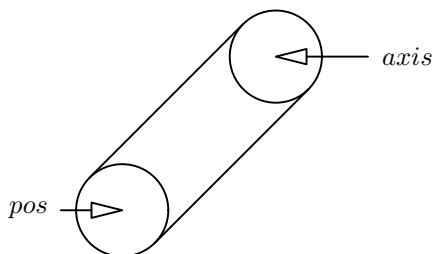
```
b=box()
b.pos=[1,2,3]
b.size=[4,5,6]
b.color=color.green
```

Boxes are drawn with sides parallel to the coordinate axes.⁶ Boxes have a *pos* attribute that determines their center. They also have a *size* attribute. The *size* attribute is a 3-element list that determines the dimensions in the *x y* and *z* directions. When you use *size* for boxes, remember that *x* is left and right, *y* is up and down, and *z* projects out of the screen. Boxes also have a color attribute.

Cylinders are created in this way:

```
b=cylinder()
b.pos=[1,2,3]
b.axis=[4,5,6]
b.radius=.2
b.color=color.green
```

The following diagram explains the meaning of *pos* and *axis* for cylinders.



⁶Actually you can draw tilted boxes. See the visual python manual.

Curves are created with the commands:

```
b=curve()
b.pos=[[1,2,3], [4,5,6] , [7,7,7]]
b.radius=.2
b.color=color.green
```

Here *pos* is a list of points and the curve is a sequence of pipes joining those points. After you learn about loops you will be able to make such a thing look like a curve, by joining hundreds of skinny little pipes.

7. Comparatives. The comparatives `<`, `<=`, `==`, and `!=` are used to test for order and equality. Note the use of the *double* equals sign. Think of the numbers 0 and 1 as representing *false* and *true* respectively. Thus

```
print 1==2
```

prints the number 0, because the statement “1 equals 2” is false.

The symbol `!=` means “not equals.” Thus

```
print 1!=2
```

prints the number 1, because the statement “1 is not equal to 2” is true.

The comparative *in* is used to test for membership in a list. For example

```
print 1 in [1,2,3], 4 in [1,2,3]
```

prints the numbers 1 and 0.

Comparatives are used in **if-then** and **if-then-else** statements to control the sequence of commands that are executed in a program. This is an amazing capability. Instead of executing instructions one after the other, your program can look at the value of a variable and use it to choose among alternative sequences of instructions to execute.

Example. The program

```
x=731
if x%17==0:
    print 'yes ', x, ' is divisible by 17'
else:
    print 'no ',x, ' is not divisible by 17'
```

tests the number 731 to see if it is divisible by 17. There are two commands in this program. The first assigns the number 731 to the variable *x*. The second is an if-then-else command spanning multiple lines. Directly following the “if” is a comparative in parentheses. The comparative says, in effect,⁷ that *x* is divisible by 17. If the comparative evaluates to 1 (i.e. “true”) then the print statement

⁷Remember that `x%17` is the remainder on dividing *x* by 17. If this is 0 then *x* is divisible by 17.

following the “if” is executed. Otherwise the print statement following the “else” is executed. The “else” part is optional.

It happens that $731 = 17 \times 43$. Therefore the first print statement is executed, and it displays

```
731 is divisible by 17
```

Note that the sequence of commands to be executed in the *if* case must be indented. Similarly for the *else* case.

Example. We define a function that returns the larger of two numbers.

```
def f(x,y):
    if x<y:
        return y
    else:
        return x
print f(2,3), ' ', f(5,3)
```

This prints out

```
3 5
```

because $3 > 2$ and $5 > 3$.

8. Loops. Here is a *while* loop.

```
x=1
while x<=50:
    print x
    x=x+1
print 'All done'
```

There are three commands in this program. The first assigns 1 to the variable x . The second command is the while loop. It spans three lines. The while-loop is made from a *sentinel*,⁸ which in this case is the comparative

```
x<=50
```

and a *body*, the two indented commands. When the *while* statement is executed, the sentinel is tested for truth. If it is true then the body of the while statement is executed. This process repeats forever, or until the sentinel becomes false. *The sentinel is tested before the body is executed.*

Thus the variable x starts at 1. The sentinel asks “Is it true that $x \leq 50$?” Yes. Therefore the number 1 is printed and x is increased by 1. Back to the sentinel. “Is it still true that $x \leq 50$?” Yes, therefore 2 is printed and x becomes 3. And so on. When x becomes 50, the number 50 is printed on the screen. Then x becomes 51 and the sentinel becomes false. The while-statement

⁸A sentinel is a guard.

is now terminated and control passes to the third command, the print-statement. In the end, the numbers 1 through 50 have been printed on the screen, followed by “All done.”

Any operation that we want to repeat 50 times can be in this while-loop. And if we want to do different things each time that is also possible: We can make what happens each time depend upon the variable i , or depend upon what happened the last time through.

Study the following examples carefully. It may help to make a table with columns headed by each of the variables in the program, and rows showing the values of the variables each time through the loop.

Example. This program gets an integer a and computes the sum $1 + 2 + \dots + a$.

```
def f(x):
    sum=0
    count=1
    while count <=x:
        sum=sum+count
        count=count+1
    return sum
a=input('Give me an integer: ')
print f(a)
```

Example. This program gets an integer a and prints out the divisors of a .

```
def f(x):
    count=1
    while count <=x:
        if x%count==0:
            print count
        count=count+1
a=input('give me an int ')
f(a)
```

Example. Here we ask the user for an integer n , and put n^2 balls the screen, each touching its neighbors, in the form of an $n \times n$ square. This is an example of a while loop inside of a while loop. We have omitted the “autoscale=0” command so that the whole construction is visible.

```
from visual import *
def makesquare(n):
    count1=0
    while count1<n:
        count2=0
        while count2<n:
```

```

s=sphere()
s.radius=.5
s.pos=(count1,count2,0)
count2=count2+1
count1=count1+1
n=input('An integer: ')
makesquare(n)

```

You should make a table and work through the execution of this program in the case that $n = 2$. In the exercises you will learn a more elegant way to do the same thing using “for-loops.”

9. Animation. Here is the program from Page 1 of these notes.

```

from visual import *
scene=display()
scene.autoscale=0
s=sphere()
s.radius=.5
b=box()
b.pos=[0,-5,0]
b.size=[10,.3,11]
b.color=color.red
x=0
while 1:
    s.pos=[0,4.5*sin(x),0]
    x=x+.01
    rate(400)

```

The first three lines set up a scene, as described in Section 6. The next two lines create a sphere s of radius .5. The next four lines create a red box b . Then the number 0 is assigned to the variable x and we enter a while-loop.

The sentinel is 1, meaning “true.” Therefore the while-loop will never terminate. Two instructions are executed repeatedly. The first sets the position $s.pos$ of the sphere s . The second increases x by .01.

Look at $s.pos$. The coordinates in the horizontal direction and in the direction perpendicular to the screen do not change. The middle coordinate, which controls the up-and-down position, is $4.5 \sin x$. This varies between approximately -4.5 and 4.5 as x changes. Therefore the sphere moves up and down.

In order to prevent the sphere from moving too quickly, we add a delay to the loop. The command

```
rate(400)
```

does this. The length of the delay grows shorter as the argument grows larger. Therefore you can think of the argument of as a measure of speed.

Now suppose we want to make the ball circle above the table instead of bouncing. We shall change the line in the while-loop defining `s.pos`. The new line is

```
s.pos=[2,3*cos(x),3*sin(x)]
```

The idea here is that as x increases the point $(3 \cos x, 3 \sin x)$ describes a circle in the plane of radius 3 centered at the origin. We have transferred this motion to 3-dimensional space by keeping the first coordinate constant and using $3 \cos x$ and $3 \sin x$ for the second and third coordinates.

To get a differently oriented circle, we can use this for `s.pos`.

```
s.pos=[3*cos(x),3*sin(x),2]
```

Can you predict how this would look?

Now we shall create a little yellow moon that spins around the sphere s in a circle of radius 1 as the sphere s circles above the table. We assume that the line

```
s.pos=[3*cos(x),3*sin(x),2]
```

is present in the while-loop, so that the sphere s is circling above the table. Before the while-loop, add the commands

```
moon=sphere()
moon.color=color.yellow
moon.radius=.1
```

Now anywhere inside the while-loop, add the line

```
moon.pos=[4*cos(x), 3*sin(x),2+sin(x)]
```

We now have a circling moon. Why does this work? We want the moon to go in a circle of radius 1 parallel to the “floor”. The center of the circle is `s.pos`, which is

```
s.pos=[3*cos(x),3*sin(x),2]
```

Therefore we have to add to `s.pos`, coordinate by coordinate, the values

```
[cos(x),0,sin(x)]
```

This is how we found the point assigned to `moon.pos`.

Try to add a second moon that rotates in an orbit perpendicular to the first.

Next we explain how to process mouse-clicks. I assume that, as usual, the second line of your program is

```
scene=display().
```

The variable `scene.mouse` is a class that contains current information about the mouse. To make something happen when the mouse is clicked, do the following:

```
m=scene.mouse
```

```

if m.clicked:
    m=m.getclick()
    .
    .
    Put here whatever is supposed to happen.
    .
    .

```

The variable *m* as used here will be a class with the following attributes:

1. *m.pos* is the current position of the mouse.
2. If the user clicks on an object in the scene then *m.pick* will be that object.

As an example of how to use this, here is a program that displays a blank screen, and adds a sphere of radius 1 wherever the user clicks.

```

from visual import *
scene=display()
scene.autoscale=0
while 1:
    m=scene.mouse
    if m.clicked:
        m=m.getclick()
        s=sphere()
        s.pos=m.pos
        s.radius=1

```

You can respond to mouse-clicks while an animation is going on. For example, here is a rotating sphere. Wherever the user clicks, a new sphere is added.

```

from visual import *
scene=display()
scene.autoscale=0
x=0
s=sphere()
while 1:
    s.pos=[sin(x),cos(x),0]
    x=x+.005
    m=scene.mouse
    if m.clicked:
        m=m.getclick()

```

```
s1=sphere()
s1.pos=m.pos
s1.radius=1
rate(100)
```

Finally, here is an example of how to use `m.pick`. Three spheres are displayed. When the user clicks on a sphere with the right mouse button, that sphere is erased.

```
from visual import *
scene=display()
scene.autoscale=0
sphere(pos=[1,2,2])
sphere(pos=[0,0,0])
sphere(pos=[3,3,3])
while 1:
    m=scene.mouse
    if m.clicked:
        m=m.getclick()
        s=m.pick
        s.visible=0
```

We used here for the first time the attribute, “visible,” shared by all shapes in vpython. If `s` is a sphere then when `s` is created, `s.visible` is 1 (i.e. “true.”) To make `s` invisible, set `s.visible` to 0.

Problems

Sections 1 ,2, 3

- (1) Show by example that the instructions

```
a=b
b=a
```

do not in general switch the values of a and b .

- (2) In the following program

```
x=input('Give me an integer ')
y=x%10
print y
```

what happens if the user inputs 123? 124? 125? Check your answers. Explain how to instantly predict the output for any given input.

- (3) In the following program

```
x=input('Give me an integer ')
y=x/10
print y
```

what happens if the user inputs 123? 1234? 12345? Check your answers. Explain how to instantly predict the output for any given input.

- (4) By thinking about the answers to the previous two problems, write a program that inputs an integer and prints out the second-to-right-most digit. For example if 123 is the input then 2 is the output. Hint: first strip off the rightmost digit of the input and then print the new right-most digit.

- (5) Predict the output of the following program

```
print 21/22
print (21+0.0)/22
print 22*(21/22)
```

Check your answer. Why is the first number different from the second? Why is the third number not 21?

- (6) One day after Sunday it is Monday. Two days after Sunday it is Tuesday. Write a python program that prints out the day 987654 days after Sunday. HINT: This is a “bean” problem. Think of days as beans and weeks as 7-bean piles.
- (7) Explain what the following program does, for any two integer inputs.

```
a=input('Give me an integer ')
b=input('Now give me another ')
print str(a)+str(b)
```

- (8) Write a program that inputs 3 numbers and prints out their average. Test it to make sure that it works.
- (9) Write a program that inputs 3 numbers a , b and c , and prints out the quantity

$$\frac{1}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c}}$$

Test your program. If you input 1, 2 and 3 the output should be 1.803...

- (10) The **fractional part** of a positive real number is obtained by discarding everything to the right of the decimal point. For example the fractional part of 1.2 is .2. Write a program that inputs a real number and prints out its fractional part.
- (11) Write a program that inputs an integer n and prints out the the number of beans in all the 2-bean piles you can make from n beans.
- (12) The following program multiplies its input by 10 using only four additions and no other arithmetic operations.

```
input('Enter an integer: ')
y=x+x
y=y+y+x
y=y+y
print y
```

Write a program that multiplies its input by 9 using at most four additions and no other arithmetic operations. Type it in and test it.

Section 4

- (1) What in general does this command do to the list a ?

```
b=a.pop()
a=[b]+a
```

- (2) Write a program that inputs a list, and moves its first element to the end. For example if the input is [1, 2, 3] then the output is [2, 3, 1].
- (3) Write a program that inputs a list and switches its first two elements. For example if the input is [1, 2, 3] then the output is [2, 1, 3].
- (4) Write a program that inputs a list with an odd number of elements and prints out the middle element. For example if the list is [1, 2, 30, 41, 52] then the output is 30.

- (5) What in general does the following do to the list a ?

```
a=a+[a[len(a)-1]]
```

- (6) Given a list a of three integers in the range from 0 to 9, what exactly does the following print out?

```
print 100*a[0]+10*a[1]+a[2]
```

- (7) I haven't explained the way python interprets the product of a list and a number. Type in the following and run it.

```
a=[4]
print 30*a
```

Now write a program that asks for an integer n and prints out a list of n 4's.

- (8) If a and b are positive integers, with $a < b$, what exactly is the output of the following:

```
print len(range(a,b))
```

- (9) Write a program that outputs a list of all the multiples of 7 starting at 7 and ending at 700.
Hint: Use the third form of "range" in Section 4.

Section 5

- (1) In the following function, which variables are local?

```
def f(z):
    x=x+y+z+w
    y=z+w
```

- (2) Predict the output.

```
def f(z):
    print x+z
x=1
f(3)
```

- (3) Predict the output.

```
def f(x)
    x=x+1
x=3
f(x)
print x
```

- (4) Predict the output.

```
def f(x):
    x=x+1
    return x
x=2
print x+f(x)
```

- (5) Predict the output.

```
def f():
    print x
    x=2
x=1
f()
```

- (6) Write a function f that inputs a two-digit number and returns a list of its digits. For example

```
f(23)=[2, 3]
```

Write a program to test your function. Hint: Look at Problem 4 from Section 1.

- (7) Write a function that inputs a positive integer n and prints out what day it is n days after Sunday. Write a program to test your function. Hint: Define a list whose elements are the days of the week. Make the positions on the list correspond to remainders.
- (8) Run the following weird program.

```
def f(z):
    return z(2)
def g(x):
    return x*4
print f(g)
```

Try to explain the output.

Section 6

- (1) Write a function `ball(x,y,z)` that puts a ball of radius 1 on the screen centered at the point $[x, y, z]$.
- (2) Use `ball` from the previous problem to make two balls, one just touching the other.
- (3) Use `ball` to make three balls, each just touching the other two. Hint: The center of these balls form an equilateral triangle of side 2. Put one vertex of the triangle at $[0, 0, 0]$ and another vertex at $[2, 0, 0]$. Put the third vertex at $[?, ?, 0]$.
- (4) Write a function `legs(x,y,z)` that draws a box centered at $[x, y, z]$ with dimensions $1 \times 4 \times 1$ in the x y and z directions respectively. It should look tall and skinny.

- (5) Write a function `top(x,y,z)` that draws a box centered at `[x,y,z]` of dimensions $7 \times .5 \times 7$.
- (6) Use `legs` and `top` to make a table with four legs.
- (7) Put a red sphere and a blue cylinder on the table.
- (8) Make a green floor for the table to rest on.
- (9) Put all of your code for the table and floor and sphere and cylinder into a single function called `table(x,y,x)`, where `[x, y, z]`, is some reference point, and use the function `table` to make four identical floors, one above the other. Give your building a back wall and paint the wall a nice color. HINT: Add the arguments `x`, `y` and `z` to the numbers that determine the positions of the table, sphere and cylinder.

Section 7

- (1) What is printed? Check.

```
print 1<2, 1!=12%11, 1 in [1,2]
```

- (2) The words *not*, *and* and *or* can be used to combine comparatives. For example

```
print not 1 in [2, 3]
```

displays 1,

```
print 1<2 and 2<1
```

displays 0, and

```
print 1<2 or 2<1
```

displays 1.

Note that *or* means one or the other or *both*.

What is printed? Check.

- a. `print not(1<2 and 5<3)`
- b. `print not(1<2 or 5<3)`
- c. `print not(1<2) and not(5<3)`
- d. `print not(1<2) or not(5<3)`

- (3) There are three people x , y and z who make different salaries. You are not allowed to ask them their salaries. But you can ask any two of them who makes the most. You are allowed two questions of this kind, and you must find the one who makes the highest salary. Exactly how do you do it?
- (4) Write a function that inputs three different numbers and returns the largest. Test your function.
- (5) Here is a function that takes a list of two different numbers and returns the same list arranged in increasing order.

```
def f(x):
    if x[0]<x[1]:
        return x
    else:
        return [x[1], x[0]]
```

Write a function that takes a list of three different numbers and returns the same list arranged in increasing order. Write a program to test your function.

- (6) Write a function with one argument that takes a list of three numbers and prints out “yes” if some number occurs more than once and “no” otherwise. For example [1, 2, 1] should produce a “yes” and [1, 2, 3] should produce a “no.”
- (7) Write a function f of two arguments that behaves as follows: If you give it any two of the numbers 1, 2, 3 it returns the third. For example, $f(1, 2) = 3$ and $f(1, 3) = 2$. Test.
- (8) Write a function of one argument that inputs a 3 element list and returns a list of all *different* elements of the input list. For example if the input list is [1, 2, 2] then the function returns [1,2]. Test your function.

Section 8.

- (1) Write a program that gets a positive integer n and prints the product of all integers from 1 through n .
- (2) Write a program that inputs any *list* of numbers and prints out the sum. For example if the input is [1, 2, 3] then the output is 6.
- (3) For the “ball” example of Section 8, make a table with a column for each variable and rows showing the values of these variables each time through the loop. Assume the input is 2.
- (4) Write a program that gets an integer n and prints the *sum* of all of the divisors of n . For example if $n = 4$ then the program prints 7 because this is $1 + 2 + 4$.
- (5) Write a program that gets an integer n and prints the *number* of divisors of n . For example of n is 4 then the output is 3.
- (6) This is a **for-loop**:

```
for x in [1, 'dog', 3]:
    print x
```

The variable x runs through each item on the list, and for each value of x all indented commands that follow are performed. For example here is a program that computes the sum $1 + 2 + \dots + 10$.

```

sum=0
for x in range(1,101):
    sum=sum+x
print sum

```

In general, to make x run through all the positive integers *less than* n use `range(1, n)`.

Rewrite Problems 4 and 5 using for-loops instead of while-loops.

- (7) Rewrite the ball example of Section 8 using for-loops.
- (8) Using a for-loop, write a program that inputs a list of any length and prints out the largest element. HINT: Use a variable to keep track of the largest element seen so far.

Section 9.

- (1) Add the second moon, as requested at the end of Section 9.
- (2) In this problem you will create an alien solar system. Make a yellow sun, of radius 3, and two planets that orbit the sun in the same plane. The inner planet is red and has radius .5. The outer planet is green and had radius 1. The outer planet has a silver moon of radius .25 that orbits in a plane perpendicular to the plane of the planets. The inner planet orbits twice for every orbit of the outer planet.
- (3) Write a program that puts a sphere of radius .2 on the screen. Every time the user clicks the mouse anywhere on the screen, no matter where, that sphere doubles in radius.
- (4) Put a sphere of radius 3 on the screen. Every time the user clicks anywhere on the screen, the sphere grows a little darker, until it disappears. Hint: Remember we talked about colors as RGB triples. The value of black is `[0, 0, 0]`, i.e., no red, no green and no blue.
- (5) Put three silver spheres of radius 1 on the screen. Every time the user clicks on a sphere, it starts to blink. Every time the user clicks on a blinking sphere, it stops blinking.
- (6) Write a program that puts two spheres on the screen centered at `[-1, 0, 0]` and `[1, 0, 0]`. The position of the spheres does not change. As the program runs, the spheres expand until they just touch each other and then shrink to points and then expand and then shrink, etc.