Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 1: Making Shapes

Saying *hello*

The first goal in learning any new computer programming environment is always the same: Write and run a program to print (or display) a brief message, traditionally "Hello, world!"

Why bother with such a boring program? Because the steps required to write and run even the most trivial program can be quite intricate. For some programming environments you may need to install software, configure the software to work with your computer's directory system, and then learn to use various software tools for editing, compiling, linking, and launching your program. Then you need to learn enough about the programming language, and about the associated software libraries for producing the type of output you want, in order to type in the code needed to produce that output. The number of things that can go wrong during this whole process is enormous.

Fortunately, the GlowScript-VPython system is one of the easiest of all possible programming environments for getting started. There are still some things that can go wrong, but even if they do, I don't think they'll take long to resolve. Here are the steps:

1. Launch a web browser (Google Chrome is recommended, but most others should work), and go to the site `glowscript.org`.
2. Sign in with a Google account. Your WSU email credentials should work, or you can use a personal Google account.
3. Follow the link where it says "your programs are **here**."
4. Click on the "Create New Program" link.
5. In the dialog box that appears, type the title "MakingShapes", then click the "Create" button.
6. You will now see an editing space that is blank except for the line "GlowScript 2.6 VPython" (possibly with a different version number). Click in the white space below that line and type the following, verbatim:

```
print("Hello, GlowScript-VPython!")
```

(Actually you can put almost any message you like between the quotes.)

7. Then click the "Run this program" link at the top of the page. Your editing space should then vanish, replaced by a new text area with the words "Hello, GlowScript-VPython!" (or whatever message you put in your code).

Well, did it work? If so, congratulations! You're up and running. If not, don't worry; if you can't resolve the problem yourself in a few minutes, your instructor, or perhaps a classmate, can probably help. (The most likely stumbling block is with the Google account sign-in. If that went smoothly but your program doesn't work, be sure to double-check your spelling, capitalization, and punctuation.)

GlowScript-VPython is a cloud-based system that stores your programs on Google's servers. The interface is bare-bones, and I hope you won't have any trouble seeing how to create new programs, copy, rename, and delete them, and organize them into folders. If you have any questions about these procedures, be sure to ask your instructor.

Now let's look at your two-line program in a bit of detail. Click the "Edit this program" link in order to see your code again. The first line, which you don't even need to type, tells the GlowScript system what language (VPython) and version (2.6) you want to use. This line is required, and for this course you should always leave it alone.

The next line, which you typed, "calls the `print` function and passes it a string of text characters." Sorry about the jargon words *call*, *function*, *pass*, and *string*. If you're an experienced programmer then these words are probably already in your vocabulary; if this is your first time writing code, then you'll need to pay attention to these words and practice using them correctly. For now, please notice two aspects of Python syntax:

- The string of text is enclosed in double-quote marks. Single-quote marks would also have worked, as long as the beginning and ending quotes match.
- The information passed to the function (its *parameter*) is enclosed in parentheses. In this case there is a single parameter (a string), but we'll soon see examples of functions that take multiple parameters, separated by commas.

The idea of a *function*, by the way, is to hide the details of what's happening from your code (or at least from this portion of it), so all you need to know is the function name, what parameter(s) to pass, and what the function does—but not how it works. The `print` function has to do an awful lot to make those words appear on your screen, but you needn't know the details.

Your first shape

Next, on a new line below your `print` instruction, type the following simple instruction:

```
box()
```

Here you're calling a function called `box`, and passing it no parameters at all (but notice that the parentheses are still required). Run the program again, and you should see a black rectangular area (called a *canvas*) containing a light gray square (the box). Again, this function is doing a great deal of work, but you needn't worry about how.

The box that you see lives in an imaginary *three*-dimensional space, but you're viewing it from one side, at relatively close range. To change the perspective, you can do two things:

- **Rotate:** Press the right mouse button and drag one way or another, or, if you don't have a right mouse button, press the *control* key and drag using your mouse or trackpad.
- **Zoom:** Use the scroll wheel on your mouse, or, if there isn't one, press the *alt* or *option* key and drag using your mouse or trackpad. (On an Apple trackpad you can also drag using two fingers.)

I hope you're impressed by how hard that little `box` function is working!

You can change the attributes of your box by passing some parameters to the `box` function. Try this:

```
box(pos=vector(1,0,0), size=vector(.5,.3,.2), color=color.red)
```

Here you're providing three parameters, separated by commas, and you're identifying them by their names (a neat feature of Python), which means you can provide them in any order. The `pos` parameter specifies the position of the center of the box; the `size` parameter specifies its dimensions; and the `color` parameter is self-explanatory. In the first two cases, the parameter values are three-dimensional vectors, which you create using the `vector` function. This function in turn takes three parameters, `x`, `y`, and `z`, which you needn't name as long as you provide them in that order. Initially (before you rotate the scene), the x direction points to the right; the y direction points up; and the z direction points directly outward, toward you (or toward the "camera").

To learn more about colors, click the **Help** link at the upper-right corner of the window. (I suggest opening the help page in a new browser window or tab.) Then, from the second drop-down menu in the left sidebar, choose "Color/Opacity". There you'll find a list of pre-defined colors, and also see how you can use the `vector` function to create arbitrary colors.

Exercise: Through trial and error if necessary, figure out how to change the color of your box to purple. Do this in your program code, and write down how you did it here:

Exercise: Create at least two more boxes, so you'll have a total of at least three, each with different positions, sizes, and colors. Keep their positions within the range -5 to 5 in each dimension, and keep their sizes small enough to leave plenty of room for more shapes within that range.

More shapes

VPython provides functions for creating quite a variety of shapes, but in this course you'll need just two others: spheres and cylinders. Try this instruction to create a sphere:

```
sphere(radius=0.25)
```

The `sphere` function can also accept the `pos` and `color` parameters, so use those now to change the defaults according to your taste. Don't forget to separate the parameters by commas!

Exercise: What happens if you omit the `radius` parameter when you call the `sphere` function?

Exercise: Create a second sphere, again within the range -5 to 5 in each dimension, with a (reasonably small) radius and color of your choosing.

Now try this instruction to create a cylinder:

```
cylinder(axis=vector(0,1.5,0))
```

Here the `axis` parameter is a displacement vector that takes you from one end of the cylinder to the other. You can also provide the `pos`, `radius`, and `color` parameters, so please put in all three at this time, to change the defaults to suit your taste.

If you check carefully, you'll discover that the `pos` of a cylinder is located at one of its ends, rather than in its center as for a sphere or a box.

Exercise: It would be helpful if your 3D space had some coordinate axes, right? So create some now, in the form of very skinny cylinders running from -5 to 5 in each of the three dimensions. Color them light gray.

Errors and debugging

You may have already had to deal with some error messages, alerting you to typographical errors (sometimes called *bugs*) in your program. Now let's take a deliberate look at one of these messages, and what you should do about it.

Exercise: Remove one of the commas that separate the parameters in one of your `box`, `sphere`, or `cylinder` function calls. Then try to run your program, and write down the error message that appears (or at least the beginning of it, including the line number). Click the "Edit this program" link to go back to your code. Is the line number in the message the same as the line in which you introduced the error?

In my own experience, the line number is usually the most useful part of an error message. The rest of the message is often unhelpful, or at least unnecessary. But even the line number can be off by a little, so when you're looking for the error in the code, be sure to look at the surrounding lines as well.

Besides missing commas, some other common types of Python errors are other incorrect punctuation, incorrect spelling, and inconsistent capitalization (since all words in Python are case-sensitive).

Variables and arithmetic

The various shape attributes that you've been setting—`pos`, `radius`, `color`, and so on—are examples of what we call *variables*. A variable, in computer programming, is a named location in the computer's memory in which some information can be stored. The equals sign tells the computer to store the information on its right in the variable whose name is on its left.

Besides these pre-named variables, you can create your own. The next exercise demonstrates this.

Exercise: To keep track of your shapes, you can give them variable names. Name your x axis by inserting "`xaxis = `" at the beginning of the line, like this:

```
xaxis = cylinder( . . . )
```

Similarly, name the other two axes `yaxis` and `zaxis`.

These variable names make your code easier to read, even if you never use them in any other way. But here's a way to make use of one of them:

Exercise: Your three axes probably all have the same radius and the same color. So instead of writing out the same specifications three times, put them into only the instruction that creates the x axis, and then in the other two, say this:

```
radius=xaxis.radius, color=xaxis.color
```

Now if you decide to change the radius or the color, you'll need to make the change in only one place (try it!).

Exercise: Create a dumbbell shape, by combining a cylinder with two spheres. Make the cylinder first, with any suitable attributes, naming it `bar`. Then, when you create the two spheres, simply set the position of one of them to `bar.pos` and the position of the other to `bar.pos + bar.axis`. This is an example of how you can use the `+` sign to do `vector` addition. Also set the radius of each sphere to `bar.radius*3`, and set the color of each sphere to `bar.color`.

The previous exercise included your first examples of addition and multiplication. Python also provides the `-` operator for subtraction and the `/` operator for division, as well as other operators that you'll learn later. You'll need to do some subtraction in the next exercise.

Exercise: Create a small table (the furniture kind) in your simulated 3D space, consisting of a box for the top plus four cylinders for the legs, oriented with the y direction up. For convenience and flexibility, start by setting the values of some variables:

```
tablex = 2.5
tabley = -2
tablez = -1
```

Don't feel obligated to use the same numbers as these; I'm just suggesting some variable names and showing how to assign values to them. Similarly, introduce and set the variables `tableLength`, `tableWidth`, `tableHeight`, `legRadius`, and `tableColor`. Instead of providing an explicit number for `legRadius`, use multiplication or division to make it a small fraction of `tableWidth`. Then create the box that will be the table's top, centering it at `vector(tablex, tabley, tablez)` and setting its length and width to the corresponding variables. The height of the table's *top*, however, should be only a small fraction of its total height. (If the line of code to create the tabletop gets too long, you can break it into two; for readability you should then indent the second line by a couple of tabs.) Finally, create the leg cylinders, using arithmetic to position them near the table's four corners no matter what the values of all your variables are set to. Try changing some of these values to make sure your table still looks like a table for any reasonable values.

Exercise: Here's something easier. The "canvas" in which your shapes appear has its own variable name: `scene`. This variable, like `color` and `xaxis`, is a so-called *object* with its own attributes. (An attribute is essentially a sub-variable that

is associated with some larger, more inclusive variable.) One of the attributes of `scene` is the background color, which is black by default. Change it to white by inserting this line:

```
scene.background = color.white
```

Check that this works, then change the background color to a pale sky blue, or to some other very light color that won't use a lot of toner when you later print your scene. Also set the variable `scene.range` to 5; this will set the initial zoom level to put $y = 5$ at the top edge of the canvas and $y = -5$ at the bottom edge.

Comments

By now your program is getting long enough to require some organization—not for the computer's sake, but for your own as you look at the code.

Exercise: Divide your lines of code into logical groups, separated by blank lines. Then, at the top of each group, insert an introductory *comment* such as

```
# Create a table out of a box and four cylinders:
```

A comment in Python begins with the `#` character and continues until the end of the line. To create a multi-line comment, you need to put the `#` character at the beginning of each line. The computer completely ignores comments when it runs your program.

Exercise: Put another comment at the top of your program, right after the line "GlowScript 2.6 VPython", to indicate the name of your program, your own name, and the date when you created it, and to give a one-sentence description of what it does. (You should include a similar comment at the top of every program that you write.)

Animation

In this course you'll need to depict not just static objects but also physical processes that play out in time. That calls for *animation*, and one of the advantages of VPython is that it makes animation extremely easy.

Exercise: Create a small box near $x = -5$ and call it `movingBox`. Then insert the following code and see what it does:

```
# Move a box across the scene in a straight line:
while movingBox.pos.x < 5:
    rate(50)
    movingBox.pos.x += 0.05
```

Be sure to indent the last two lines; you can use the tab key for this.

If this code works, you should see the box move smoothly across the scene, stopping when it reaches $x = 5$. The code that accomplishes this magic is called a *loop*, which in general means a block of code that the computer executes repeatedly. Some languages use curly braces to denote which lines of code are part of the loop, but in Python this is indicated by indentation. This particular type of loop begins with the special word `while`, followed by the *condition* that must be true as long as the loop continues, and then followed by a colon. The condition can involve any of the comparison operators `<`, `>`, `<=`, `>=`, `==` (for "is equal to"), and `!=` ("is not equal to"), as well as the "boolean" operators `and`, `or`, and `not`.

The two indented lines themselves also require explanation. The first of them, `rate(50)`, tells the computer how fast to try to execute the loop—in this case, 50 times per second. The next line contains the two-character operator `+=`, which tells the computer to add the quantity on the right *onto* the variable on the left. Saying `x += 42` is completely equivalent to saying `x = x + 42`. (When appropriate, you can similarly say `-=`, `*=`, and `/=`.)

Exercise: Change the parameter `50` in the `rate` function to some other number, and make sure this change has the expected effect.

Exercise: Add a line or two to your code to make the box move along a diagonal, rather than directly from left to right. Make sure the box doesn't move too far in some other direction before the motion stops.

Exercise: Add code to your program to create a small sphere and then move that sphere at a steady speed around a circle in the xy plane, centered at the origin. Use a variable called `theta` for the angle around the circle, and set this variable equal to zero before your `while` loop begins. Also use a variable called `r` for the circle's radius. Then you can calculate the x coordinate of your sphere using an instruction like

```
movingSphere.pos.x = r * cos(theta)
```

and similarly use the `sin()` function to calculate the y coordinate. As in most computer languages, the trigonometric functions assume that the parameter passed to them is in *radians*. Keep this in mind when deciding how much to change theta during each loop iteration, and in deciding what condition to use in the `while` statement.

Exercise: Look in the VPython documentation (via the Help link, if you don't already have it open) for instructions on how to "attach a trail" to an object as it moves. Attach a trail of points to your moving sphere, using the `interval` attribute to space them a little farther apart than the default.

Exercise: Remember the `print` function? Insert a couple more calls to it in your code, to print out suitable messages when each of your animation loops (one for the box, one for the sphere) has finished.

Graphing

Often in science we want to visualize a phenomenon not in physical space but instead in a "space" of other variables such as time, velocity, and so on. While we `could` use a 3D VPython canvas to make such an abstract graph, VPython also provides a `graph` object that's usually more suitable.

As an example, let's graph the x and y positions of your moving sphere as functions of "time". Here is the code to set up the graph, with two data sets that will be plotted as dots in different colors:

```
graph(width=400, height=250)
xDots = gdots(color=color.green)
yDots = gdots(color=color.magenta)
```

(I've set the `width` and `height` attributes to make the graph a little smaller than the default, so it will fit more easily on small screens and printed pages.) After this initial setup, you add a dot to the graph by saying something like

```
xDots.plot(t,x)
```

and similarly for `yDots`.

Exercise: Insert the code to set up a graph just before your existing code to move the sphere around in a circle. Also, in that existing code, insert code to create a variable `t` to represent time, initially equal to zero and increasing by 1 during each loop iteration. Then, also inside the `while` loop, insert the lines that plot each (t, x) and (t, y) pair, in green and magenta, respectively. Run the program to make sure it all works.

Exercise: Look up the other attributes of the `graph` object in the VPython documentation, then set the graph's background color to white, label both axes appropriately, and give it a suitable title, centered above it. Notice in the documentation that you can also specify the ranges of values for the graph to show—but for your current graph, leave these unspecified to enable "auto-scaling".

Finishing up

Congratulations! You now know how to use GlowScript VPython to do basic arithmetic, draw and animate shapes, create graphs, and produce text output. You'll get plenty of practice with all of these tasks in later projects. For now, just answer a few more questions and then you'll be finished with this assignment.

Exercise: At the top of the next page, describe one specific bug (error) that you had to fix while working on this project, or, if you prefer, one specific point at which you became frustrated or confused. If you can't think of an example of either, then browse the VPython documentation and describe one interesting feature that you haven't used in this project.

Exercise: When you type your code in the GlowScript editing window, it automatically tries to use colors to distinguish different categories of words and symbols. List these colors below and describe what each seems to be for, keeping in mind that the process is rather buggy so some of the coloring can be inconsistent.

Exercise: List the names of all the VPython functions you used in this project. (Hint: Look in your code for parentheses.)

Before turning in this assignment, please look over your program and make sure it is well organized and easy to read. The code should be divided up into logically distinct groups, separated by blank lines, each introduced by a helpful comment.

Finally, run your program one last time and use your browser's print command to print the window contents, showing your 3D scene (with a light-colored background!) with the graph and text output below it. Make sure that all of this fits on a single printed page. Staple that page to the back of this instruction packet and turn it in to your instructor.

To turn in your actual code, you have two choices. The easier choice is to copy it into a public GlowScript folder and write that folder's URL (web address) below, so your instructor can view and run it. However, if for privacy reasons you do not wish to put your program into a public folder, you may also carefully copy and paste your code into an email message sent to your instructor, or copy and paste it into a plain text file and then email that file as an attachment (use the file extension .txt or .py). Please use "Physics 2300 Project 1" as the subject line of your email.