Name:_____

Introduction to Programming in Python

Lab material prepared for Harrison High School Physics courses in collaboration with Purdue University as part of the project "Science Education in Computational Thinking (SECANT)", http://secant.cs.purdue.edu/.[1]

August 2009

Common mistakes, pitfalls, and debugging hints

This document describes common mistakes made when starting to program in Python. It also gives pointers on how to find bugs in programs and lists some general programming tips. To get the greatest benefit, read the document more than once. As you become more familiar with Python, you will understand the tips and guidelines better.

1. Common mistakes and pitfalls

Programming is an art. There exists no recipe for writing code without bugs, but there are guidelines you can follow to help you write correct code. In this section, we give examples of common programming mistakes and pitfalls you may encounter. Most have been mentioned in earlier labs.

Example 1: Remember the rules about arithmetic operations with integers and representing numbers. Now look at the commands listed below:

```
>>> 1/3
0
>>> -1/3
-1
>>> 1.0/3
0.33333333333333331
>>> 1/3.0
0.33333333333333331
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    1/0
ZeroDivisionError: integer division or modulo by zero
```

If the first statement in your program is the import shown below, arithmetic operations will produce floating point numbers.

```
>>> from __future__ import division
>>> 1/2
0.5
>>> 1/3
```

[1] This work is supported by the National Science Foundation under Grant No. CCF-0722210. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

0.33333333333333331

See Lab 1 for more information on importing the math module. Remember:
>>> from math import sqrt, sin
>>> sqrt(4)
2.0
>>> sqrt(-4)

Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
sqrt(-4)
ValueError: math domain error
>>> sin(90)
0.89399666360055785
>>>

- Division by 0 is a common mistake (in the programs you write, check for it using an if-statement if there is the possibility of division by 0).
- Operations involving all integers will generate integers unless you have included **from __future__ import division**
- Almost all functions imported from the math library return floats, essentially turning an integer into a float (for example, sqrt will always return afloat, even if the argument is a perfect square).
- In Python, trigonometric functions are evaluated in radians: http://docs.python.org/library/math.html - trigonometric-functions

Example 2: Here is a program we wrote earlier, except we introduced a mistake.

```
i = 0
while i < 5:
   print i
i = i + 1
```

What happens when you run this program? Where did we go wrong?
Pay attention to the indentation! Environments like SPE will help reduce indentation errors and will underline incorrect statements. Make use of them. Do not use Word or similar word-processing software to write a Python program (they include many control characters Python chokes on).

Example 3: Lab 1 describes how numbers are represented and the approximation of real numbers by floats. Here is a common mistake related to number representation when writing while loops:

```
i=1
while i != 1.6:
   i = i+0.1
   print i
```

Theoretically, the while loop should be executed six times. However, running the program produces an infinite loop whose initial printed lines are
>>>
1.1
1.2

1.3
1.4
1.5
1.6
1.7

Our intent was to stop this loop when the value of i became 1.6. But we made the mistake of assuming that fractions are represented exactly. You have already seen that 1/10 is not represented exactly. And, 1.6 is not represented as 1.6. Terminating the while loop when equality is achieved is numerically impossible and leads to the infinite loop. Mistakes like this one can cause serious problems (imagine an air-traffic control system having distance decisions based on equality).

How should this problem be solved? The most common fix is to use inequalities in the stopping criteria:

```
i=1
while i <= 1.6:
   i = i+0.1
   print i
```

One final comment: Now the program prints 6 values, but it does not print its initial value of 1. Why? Because we print after increasing i.

Example 4: Assume we have a list containing numbers and we now want to print the numbers in the list. One way to do this is using a for-loop and the function **len** (which returns the number of elements in a list). You have seen this in a number of lab programs.

```
mylist = [1.0, 1.5, 2.0, 9.25, 3.0, 4]
for i in range(len(mylist)):
        print mylist[i]
```

1.0
1.5
2.0
9.25
3.0
4

Here is an attempt to do the same with a while loop:

```
mylist = [1.0, 1.5, 2.0, 9.25, 3.0, 4]
i = 1
while i < len(mylist):
   print mylist[i]
   i = i+1
```

>>>
1.5
2.0
9.25
3.0
4

Why does it not print the first element of the list?
Our mistake was that we started counting at 1, when we should have started at 0. To fix it, set i=0.

A similarly flawed program:

```
mylist = [1.0, 1.5, 2.0, 9.25, 3.0, 4]
i = 0
while i <= len(mylist):
   print mylist[i]
   i=i+1
```

```
1.0
1.5
2.0
9.25
3.0
4
Traceback (most recent call last):
  File "h:\My Documents\programs\pitfalls.py", line 20, in <module>
    print mylist[i]
IndexError: list index out of range>>>
```

Our mistake is that we counted too far. There exists no element mylist[5]! Look back at Lab 1 for additional explanations.

When using a while-loop, always check the logic of the starting and ending conditions. Remember that the digital world generally starts counting with 0, not 1.

A final comment on accessing the elements in a list. Python allows you to access the elements in yet a different way:

```
>>> mylist = [1.0, 1.5, 2.0, 9.25, 3.0, 4]
>>> for item in mylist:
        print item

1.0
1.5
2.0
9.25
3.0
4
>>>
```

Variable **item** takes on every value in list L, starting with list[0]. This form of iterating through the elements of a list does not explicitly index into the list and is generally easy to read. It is a more "pythonic" way of writing code.

Example 5: When creating and naming VPython objects, the following mistakes are common

- **Relationships between objects don't look right.**
  A common reason for this is an incorrect use and interpretation of arguments as well as the default values of objects. For example, what point does attribute **pos** refer to? If this happens, verify your assumptions (visiting http://www.vpython.org/webdoc/visual/index.html can help). You can also print the values of attributes to see whether your interpretation is correct.

- **Not using a vector when one is expected.**
  The material you have seen has consistently used the work **vector** when VPython expects a vector, even when it is not necessary to say vector. For example,
  sphere(pos=vector(0,4,0), radius=0.40, color=color.red) is equivalent to
  sphere(pos= (0,4,0), radius=0.40, color=color.red) as VPython will automatically interprets the triple (0,4,0) as a vector. However, there are situations when one needs to say **vector**. To avoid any confusion, we recommend the approach we have taken.

- **Reusing a name for VPython objects.**
  In Lab 3 you saw a program that drew many arrows when it intended to draw only one. When an object is created and given a name and the name is used later again, one is no longer able to access the first object (it will however remain in the VPython window). Be careful when using the same name twice.

2. My program isn't working – what shall I do?

You will make both syntax errors and mistakes leading to run time errors. Debugging a program is part of programming. This section describes mistakes we expect are likely to happen and basic ideas on how to find and correct runtime errors.

Syntax Errors
The earlier labs discussed the syntax of Python statements and that syntax rules have to be followed. If they are not, you will see syntax errors. Syntax errors are probably the easiest errors to fix. When typing code in environments like SPE, the editor will already alert you to some of them. Pay attention to responses by the editor like having red underlined code after having hit return. Syntax errors like missing a parenthesis, typing a semi-colon instead of a colon, having too many or not enough quotes in a print statement can be frustrating to spot, but you will probably get quite good in detecting them.

Syntax errors that don't become apparent to you until you run the program produce messages like "*SyntaxError: invalid syntax*", which tend to not be very helpful. But you will be given a line number in the program that will narrow your search for the mistake. Note that the syntax error might be in the line just before the given line number.

How to reduce the syntax errors in your program?
- Learn the syntax rules. Python has few rules compared to other languages and learning them is a very good idea.
- If you are not sure how to correct a syntax error, look up the rules. Or, try the statement in the interpreter environment.
- Don't copy code from Word processing files. Copying a single statement is often okay unless the statement contains quotes.
- Make use of the fact that environments like SPE will pop up a help window while you are typing (e.g., after typing **range(** a window will tell you the rules of range and how to use it).
- When writing a program, write it incrementally. Write small portions and make sure they contain no syntax errors.

Runtime Errors
When you run your program, you want it to produce the correct output. In programs you have seen and will write, output will be printed or visualized in a VPython window. Quite frequently, running your

program for the first time produces incorrect results . To help reduce runtime errors, develop and test a program incrementally.

Your program may produce no output at all, it may be stuck in an infinite loop, it may crash and provide a possibly not very helpful error message, or it may produce the wrong result. Each such error could be easy to find or difficult to find. Understanding the logic in your code and eliminating possible mistakes will help you figure it out.

- **No output is produced**
  Take a look at the first version of the quadratic equation program from Lab 5 that did not produce any output when the equation had no real roots. The code written was correct, but it provided no output as no print statement was ever executed. If your program produces no output, add print statement so you can trace what statements are actually executed and what the values of the variables are during the execution. Doing so will usually provide insight into the flow of execution and point you towards the actual problem to fix. When your VPython objects are not moving, follow the same philosophy.

- **Stuck in an infinite loop**
  Your infinite loop may be caused by a while loop and you will have to kill the program. Placing print statements in an infinite loop is probably of limited use as the program may just print and print. Take a close look at the condition in the while loop and check that it tests what you want it to test. Run your program with other parameters. The programs written in this class do not read input and values are generally assigned at the beginning of the program. Change the values of the variables and run the program with simple values. Programs can run correctly in some cases, but fail on others. If you still don't see what is causing the infinite loop, add a condition that executes the loop only for a fixed number of times and print the variables changing inside the loop. Now you can trace the execution and you may see that the code is not making progress as expected.

- **Crashing with an error message**
  Python prints a message, provides the line of code where the error occurred, and stops execution. In some cases such an error is easy to fix; like when you forgot to initialize a variable before using it, when you mistyped a variable name, or when there is a division by zero. In other cases, finding the cause of the error can be a challenge.
  If you can't find the error, add print statements to the program so you are better able to trace its execution. In larger programs, you will want to make sure that individual parts of code are tested before they are included into the larger program.

- **Generating the wrong results**
  Your program may have a mistake that is easy to spot and fix, or it could have a serious logic problem. Most programs you will write have a similar logic and this will help you get better in finding mistakes. Sometimes it helps to have someone else look at your program with a "fresh set of eyes." Any better answer goes beyond the scope of this course. If you are interested in learning more, we recommend taking a computer science programming class (like CS 177 and 180 at Purdue).

3. Programming Tips

- **Develop Incrementally**

Don't try to write your entire program at once. Instead, break the problem down into smaller pieces or simplify the problem. Get that working first, then expand the program to do more and more. This approach will make debugging easier.

- **Use Assertions**

Often, you make assumptions about the problem you are solving while writing your program. For example, you may assume that an array index will never exceed the size of the array. Python allows you to insert assertions into your code:

assert i < len(a)

to verify that your assumptions are correct. The statement above says that at this point in the program, I assume that i is less than the length of a. Python checks this assertion and stops execution (with an error message) if the assertion is not true.

- **Use Print Statements**

As mentioned earlier, print statements can also be used to test your assumptions. Is execution reaching this point? What is the value of this variable or expression? Use print statements while writing and debugging your code, then comment them out (using the "#" character) when they are no longer needed.