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

Lab 1: Python Interpreter and first Python statements

OBJECTIVES

In this lab you will learn:

- How to use the Python interpreter
- Basic arithmetic expressions
- Representing and operating on numbers
- Boolean expressions, using the range statement and lists

1. Using the Python interpreter

Start up a Python programming environment. When using PortablePython, you can use PyScripter or SPE (see the PortablePython Guide for more information). Either environment provides an interpreter window which allows you to type and execute instructions.

We start by typing simple instructions in the interpreter window. The goal is to get comfortable with basic Python instructions. Note that >>> appears on the left in the window. This means Python is ready for your instructions (you don't type these three symbols). Type into the interpreter window:

```
>>> 2+3
```

Hit enter to let the interpreter know to execute the instruction.

```
>>> 2+3
5
```

You instructed the computer to calculate the value of the instruction **2+3** and display the result on the screen; not different from how you would use a calculator.

Luckily for us, we can use the computer to do more than just calculate arithmetic expressions. Try the following:

```
>>>  apples = 2+3
>>>
```

Notice that the outcome of this instruction is different from that of **2+3**. What you just did is instruct the computer to define a **variable** named **apples,** and store into it the value of the arithmetic expression **2+3**. You may choose any name (legal in Python) for your variables, assign a value to it and use this value later.

Some rules about legal names in Python:

[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.

- Names can contain numbers and letters.
- Names cannot contain spaces.
- The underscore character _ is legal (e.g., my_count)
- The first character has to be a letter.
- Upper and lower case letters are different (e.g., Apples is not the same as apples).
- Words that are part of the Python language cannot be used.

If you try to give a variable an illegal name, you get a syntax error.

To see the current value of the variable result, type its name:
>>> **apples**
5

Some more examples:
>>> **oranges = 3*2 - 5*6**
>>> **oranges**
-24
>>> **apples **3**
125

The last example raises the value of apples (which is 5) to the power of 3 (in others words 5^3), which we know as exponentiation. In Python, ** represents exponentiation.

Notice that the value of apples remains unchanged as we execute the statement. If you want to change the value of a variable, it needs to appear on the left side of the statement:
>>> **apples = 4*7**
>>> **apples**
28
>>> **apples = apples - 2**
>>> **apples**
26
>>> **apples = apples + oranges**
>>> **oranges**
-24

2. The Python math module

We can do many interesting things with variables. If we want to use mathematical functions, we need to import them first (as done below):
>>> **from math import sqrt**
>>> **a = 4**
>>> **b = 3**
>>> **c = sqrt(a**2 + b**2)**
>>> **c**
5.0

In order to use the **sqrt** function, we needed to **import** it from the **math** module. This is what the statement **from math import sqrt** does.

We will import functions from modules as we need them. Here is a list of other commonly used function which have to be imported from the math module (for a complete listing see

http://www.python.org/doc/2.2.1/lib/module-math.html):

Function	*returns*
sin(*x*)	the sine of *x*, in radians
cos(*x*)	the cosine of *x*, in radians
tan(*x*)	the tangent of *x*, in radians
acos(*x*)	the arc cosine of *x*, in radians
asin(*x*)	the arc sine of *x*, in radians
atan(*x*)	the arc tangent of *x*, in radians
log(*x*)	the natural logarithm of *x*
log10(*x*)	the base-10 logarithm of *x*
pi	an approximation of the mathematical constant *pi*

A function needs to be imported only once in a session or in a program. If one wants to import all functions in the math module, write **from math import ***

Start a new interpreter session to see what happens if one forgets to import a function used:

```
>>> sqrt(44)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    sqrt(44)
NameError: name 'sqrt' is not defined
```

Let's take a closer look at arithmetic expressions:

```
>>> 5+12*3
41
>>> (5+12)*3
51
```

Notice the mathematical order of operations is followed. What does 1 / 2 equal?

```
>>> 1 / 2
0
```

Now try:

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

Without the statement **from __future__ import division** (that's two underscores to the left and the right) Python does integer division with truncation and **1/2** returns **0**. This is called floor division. We would like true division, so we will start all of our programs with this import statement. It must be the first statement in the program.

You can input numbers in Python using scientific notation as you would on a calculator. For example, 5e10 is equal to 5×10^{10}.

```
>>> 5e10
50000000000.0
```

Two important issues about representing numbers in a computer:

- **Numbers may be approximated**.
  For example, what is the value of 1/3? Is it 0.3, or 0.33 or 0.333333? The answer is: it depends (on

the language, the hardware, the libraries used). The more decimal places we have, the better **approximation** we can calculate for the true value of 1/3, but we can never really represent it as a decimal number.

- **All numbers are represented as binary numbers**.
  If you have seen binary numbers before, you know that 5 is 101 in binary. An integer can be represented accurately in binary. However, not all real numbers can. For example, 1/10 cannot be represented accurately in binary. This is discussed more later.

Exercises

Start a new interpreter session and set **w=0, x = 15, y = 6, z = 30.**
Then write each of following expressions in Python. Write the numerical answer generated by the interpreter off to the side.

1. $x*y + x^2$ _____
2. x/y _____
   Import true division and try again. Now what is the answer? _____
3. What is an expression involving division that you would expect to generate an error?
   _____
   Run it so you can see how Python handles it.
4. $\sqrt{}(y+x*w)$ _____
5. What is an expression with the square root function that you expect to generate an error?
   _____
   Run it so you can see how Python handles it.
6. sin(z) _____
   What is the answer?_____
   What did you expect the answer to be? _____
   What conclusion can you draw about the default setting of trigonometric functions in Python?
   _____
7. $\log_{10}y$ _____
8. 7×10^{-3}
   What is the answer? _____
   What did you expect the answer to be? _____
   Why isn't the answer exact? _____

3. Boolean expressions: True or False

In addition to arithmetic expressions, the programs we write will use **Boolean** expressions. These are expressions with only two possible values, **True** or **False**. Below are some examples.
Note that in Python equal (=) is written as == and not equal (≠) is written as !=.

```
>>> 10<100
True
>>> 100<100
False
>>> 100<=100
True
>>> 100==100
True
>>> 100!=100
False
```

We can also join two Boolean expressions together using **and, or** operations.

```
>>> a = 10
>>> b = 6
>>> a>0 and b<100
True
>>> a>0 and b<5
False
>>> (a==0) or (a==10)
True
>>> (b==0) or (b==10)
False
```

Exercises
In the interpreter window, set **myage = 17**, **driving = 16**, **drinking = 21**.
What do the following statements produce?

1. >>> **myage < 17**
   _____
2. >>> **myage > driving**
   _____
3. >>> **myage == drinking**
   _____
4. >>> **(myage >= driving) and (myage < drinking)**
   _____
5. >>> **3 * 0.1 == 0.3**
   _____
   What did you expect the answer to be? _____
   Why do you get the answer that you do? (Try evaluating the right and left hand sides of the expression individually.) _____

4. Strings and Basic Printing

A **string** is a sequence of characters and strings are used to represent text. We use the **print** statement to output strings to the console.

```
>>> print "this is a string"
this is a string

>>> myString = "I am a variable and I contain a very long string"
>>> print myString
I am a variable and I contain a very long string
```

The print statement will accept multiple values separated by commas as input and will output them with spaces in between. Values which are not strings will first be converted to their string representation before being printed.

```
>>> apples = 25
>>> print "apples =", apples
apples = 25
```

Execute the same instructions without any quotes. You will see that Python does not allow you to type plain text without quotes.

Exercises
In the interpreter window, set the values of two variables: total = 55, count=6.

1. What does the following print statement generate?
   >>> **print total, count**
   _____
2. Write a print statement that prints the word "and" (without quotes) between the numbers (print needs to use the variables).
   >>> **print** _____
   55 and 6
3. Write the print statement generating the following output (the print statement needs to use the variables)
   >>> **print** _____
   The 6 elements sum up to 55
4. What does the following print statement generate?
   >>> **print "the sum of total and count is", total+count**
   _____

5. The Range Statement

The **range** function is a very useful statement in Python. It produces a sequence of **integers**:

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
By default, the sequence starts at 0 and has a step size of 1. It does not reach the limit given (10 in the example).

>>> range(4, 10)
[4, 5, 6, 7, 8, 9]
Sets the starting value to 4, step size remains 1, limit of 10 is not reached.

>>> range(10, 0, -2)
[10, 8, 6, 4, 2]
Sets the starting value to 10, limit of 0 is not reached, and step size is set to -2 (counting backwards). Range allows us to skip over numbers, odd numbers in this case, as well as to count backwards.

Note: there exist other versions of the range statement allowing one to generate lists of non-integers. They are not covered in this class.

Exercises
In the interpreter window, write range statements producing the following sequences of integers:

1. [0,1,2,3,4] _____
2. [7,6,5,4,3,2,1,0] _____
3. [4,5,6,7,8,10] _____
4. [10, 50, 90, 130, 170, 210, 250] _____
5. [18, 16, 14, 12, 10, 8] _____

6. Lists

A **list** is a collection of "things" that Python can use. Each item in a list can be accessed through the name of the list and its position in the list.
We have already seen one example of lists: the range statement generates a list:

```
>>> range(5, 12)
[5, 6, 7, 8, 9, 10, 11]
```

This is a list of numbers from 5 through 12 (not inclusive). Another way to create this list is to simply list the elements:

```
mylist = [5, 6, 7.5, 7.5, 8, 2]
```

What can we do with lists? We can determine their size (i.e., the number of elements in it). The function **len( )** (length) returns this integer:

```
>>> len(mylist)
6
```

We can **access** an **element** in a **list**. Note that counting starts at zero!

```
>>> mylist[0]
5
>>> mylist[2]
7.5
>>> mylist[4]
8
```

The first element of a list lives at position 0. In the example, the last element is at position 5 (not 6, as we start counting at 0).

Accessing a position in the list that does not exist results in an error:

```
>>> mylist[6]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    mylist[6]
IndexError: list index out of range
```

One can change the elements in a list as shown below. Other operations on lists will be introduced when needed.

```
>>> mylist[4] = 19
>>> mylist
[5, 6, 7.5, 7.5, 19, 2]
```

An element in a list can be treated like a variable:

```
>>> mylist[2] = 2*mylist[4]
>>> mylist[2]
38
```

One list operation we will need is adding an element to a list. For example, for a list L = [4,-8.8, 9], add an element to the end, say 100, so that L = [4,-8.8, 9, 100]. We will always add to the end of a list using the operation **append:**

```
>>> L =[14, 5.5]
>>> L
[14, 5.5]
```

>>> L.append(-8)
>>> L
[14, 5.5, -8]

The rules for using append are: the list you want to operate on, a dot, the word **append** and the element to add as the new last element. You will see other operations where this format is used.

Exercises

1. In the interpreter window, set **myL = [3, 9, 27, 2, 4, 8, 16]**.
   Give instructions, each operating on an individual element of the list to change list myL to [9, 27, 81, 2, 4, 8, 16]. Your answer should be a sequence of instructions you execute in the interpreter (setting myL to the desired value does not count).
   _____
   _____
   _____

2. Now you have **myL = [9, 27, 81, 2, 4, 8, 16].**
   Write one statement adding a new last element to myL resulting in myL = [9, 27, 81, 2, 4, 8, 16, 32].
   _____

3. Write a statement that returns the length of **myL**.
   _____

You are almost ready to start writing meaningful programs!
Before we introduce needed constructs of loops and conditionals, we introduce you in Lab 2 to another module, VPython. Portable Python loaded this module for you and you import it writing **from visual import *.**