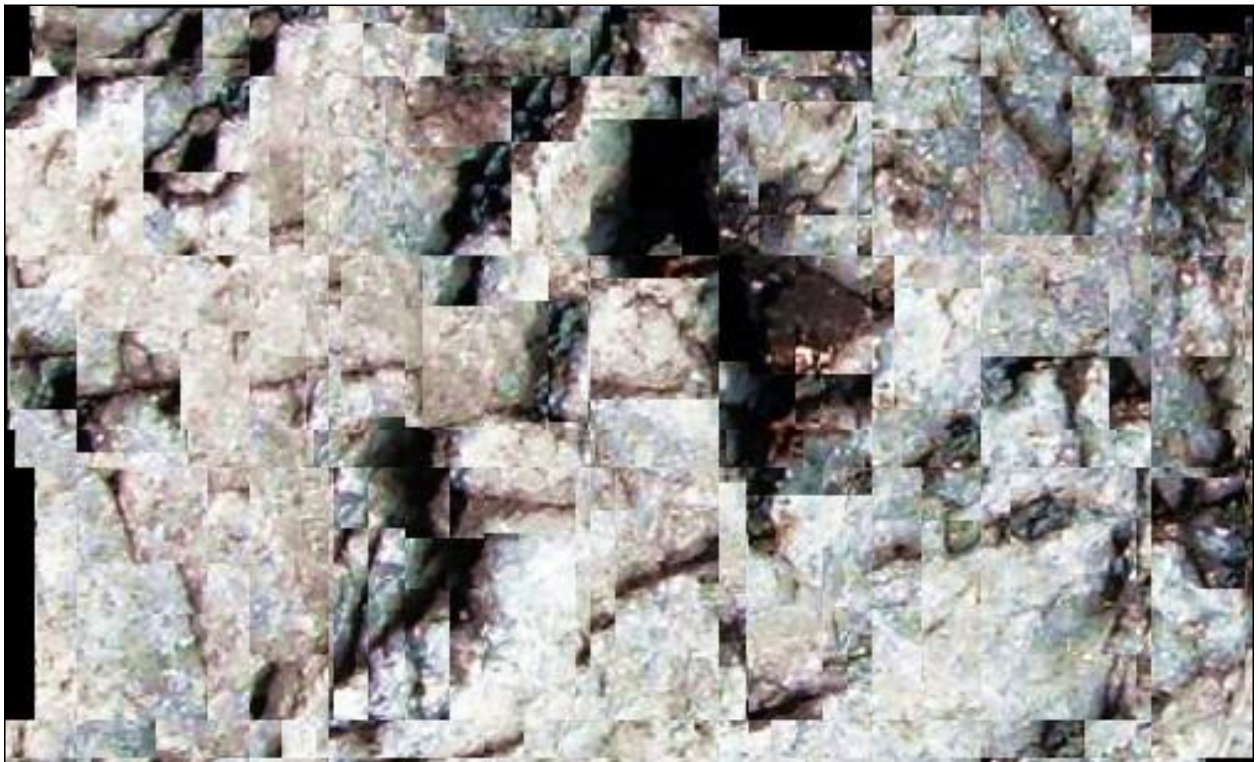
Computer Programming with Python™ and Multisim™ / 3E

Laboratory Manual



James M. Fiore

Laboratory Manual

for

Computer Programming

with Python™ and Multisim™, Third Edition

by

James M. Fiore

Version 3.1.4, 22 September 2016

Published by James M. Fiore via dissidents

For more information or feedback, contact:

James Fiore, Professor
Electrical Engineering Technology
Mohawk Valley Community College
1101 Sherman Drive
Utica, NY 13501
jfiore@mvcc.edu

or via www.dissidents.com

Cover photo *Squarer for Bear* by the author

Introduction

This manual is intended for use in an introductory computer programming course for electrical engineering technology students. It begins with a basic explanation of schematic capture and simulation tools and proceeds to the Python programming language. Python (version 3.X) was chosen for several reasons. First, it is a modern, open-source programming environment. Second, it has a relatively shallow learning curve meaning that new programming students can get up and running fairly quickly, yet the language is fairly deep and powerful. It is by no means a “toy” language. Third, it is free and multi-platform, available for Windows, Mac and Linux. This third edition is updated to Multisim 13 (V3.1.1 and higher include a few differences relating to the latest version, 14).

The programming applications presented tend to be electrical circuit based although some lean closer to quality control issues and a few are intended strictly as a way of stretching out and having some fun. As the language’s designer and developers are fans of Monty Python, it is helpful to at least watch a few of their movies in order to appreciate the embedded jokes. Most of the exercises are designed to be completed in a single practicum period of two or three hours, however, a few are a bit more involved and will require more time (such as *Caerbannog* and *Functions and Files*).

Other manuals in this series include DC and AC Electrical Circuits, Semiconductor Devices (diodes, bipolar transistors and FETs), Operational Amplifiers and Linear Integrated Circuits, and Embedded Controllers Using C and Arduino. Texts also available for Embedded Controllers (first edition) and Operational Amplifiers (third edition), and will be available for Semiconductor Devices in early 2017.

A Note from the Author

This work was borne out of the need to create a lab manual for the ET154 Computer Programming course at Mohawk Valley Community College in Utica, NY, part of our ABET accredited AAS program in Electrical Engineering Technology. Another important aspect was to come up with an affordable solution for the students. As both the programming language and the manual are free, this much is certainly covered. I am indebted to my students, co-workers and the MVCC family for their support and encouragement of this project. While it would have been possible to seek a traditional publisher for this work, as a long-time supporter and contributor to freeware and shareware computer software, I have decided instead to release this using a Creative Commons non-commercial, share-alike license. I encourage others to make use of this manual for their own work and to build upon it. If you do add to this effort, I would appreciate a notification.

“Without deviation, progress is not possible”

- Frank Zappa

Table of Contents

1

Introduction to Multisim

Objective

The objective of this exercise is to become familiar with the Multisim™ electrical circuit simulation package in order to create simple schematics and perform basic simulations. The differences between virtual, real, and 3D components will be examined along with the use of virtual instruments to make simulated measurements. Please note that the precise look of the windows, menus and dialog boxes may be slightly different from that pictured depending on the version of Multisim that is being used. Regardless of appearance, the functionality remains.

Procedure

After logging into the computer, open Multisim. If a desktop shortcut is not available Multisim may be accessed via the Programs menu under the Electronics Workbench menu item. This is a large program and may take a minute or two to load. Eventually, you will be greeted with something similar to the screen shown in Figure 1-1. As the toolbars are customizable, the precise look of the program may be a little different from that shown. In general, there are a series of toolbars along the top. These are used to select different components and editing or viewing functions. Multisim's schematic capture facility is object based, that is, you "draw" a circuit by selecting predefined objects such as resistors and transistors, and drag them onto the workspace. They are then wired together using the mouse. You can zoom into or out of the workspace using the mouse.

By default, along the left edge is the Design Toolbox browser window. This may be closed to create more working room for the schematic. At the bottom is the tabbed Spreadsheet View that shows simulation data, components, etc. Clicking on entries will highlight the corresponding elements on the schematic worksheet. Along the right edge is a vertical toolbar that contains virtual instruments.

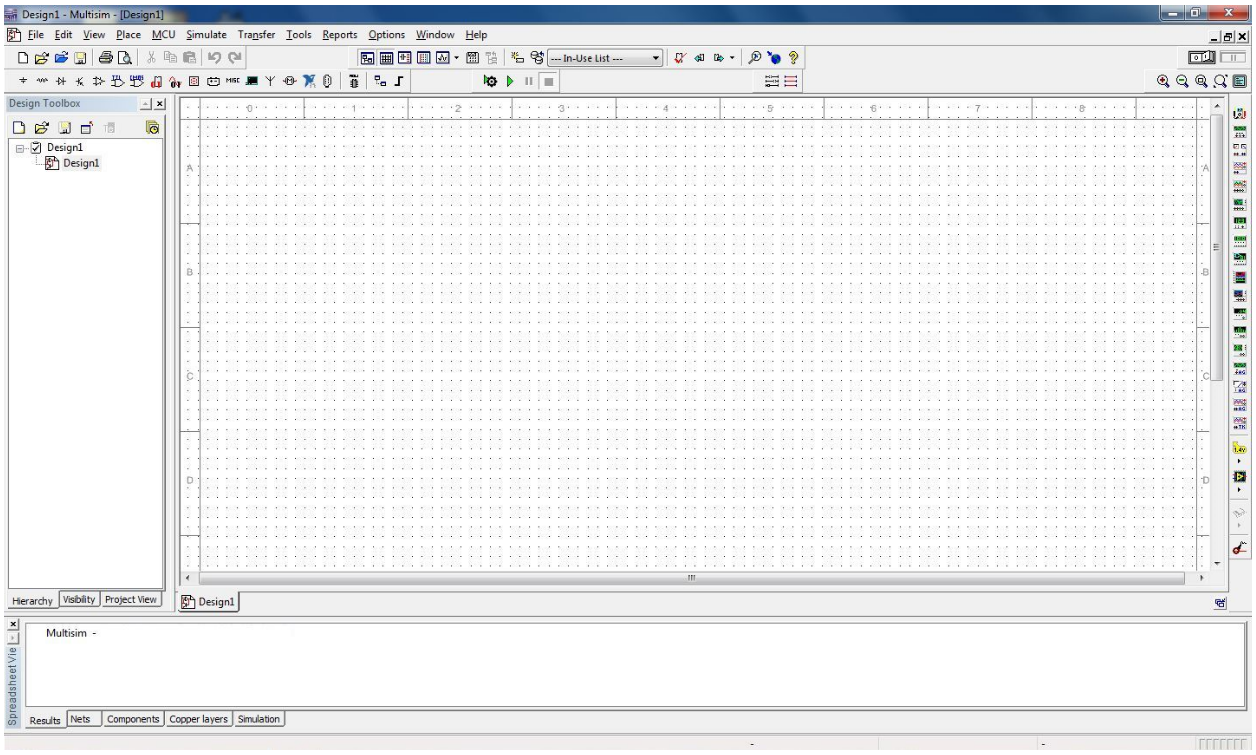

Design1 - Multisim - [Design1]
File Edit View Place MCU Simulate Transfer Tools Reports Options Window Help
--- In-Use List ---
Design Toolbox
Design1
Design1
Hierarchy Visibility Project View
Design1
Multisim -
Spreadsheet View
Results Nets Components Copper layers Simulation

Figure 1-1

Multisim uses three different kinds of components to create schematics. They are virtual basic and rated components, real (or manufacturer's) components, and 3D virtual components. Virtual and real components use the standard industry schematic symbols. By default, components with physical footprints (most real components) are colored blue and components without a physical footprint (virtual components) are colored black. In contrast, 3D components look more like photographs of the actual component. Examples are shown in Figure 1-2. Although 3D components add a certain amount of color and false reality to a circuit, they are non-standard and not generally used for simulations. We shall not discuss them further.

The difference bewteen virtual and real components is that real components reflect items from a manufacturer's database. The items include physical parameters, such as size and pinouts, which are required for designing proper printed circuit boards. Also, real behavioral models for semiconductor devices such as op amps will be more accurate than the virtual models. Finally, the values of real passive components (resistors, capacitors and inductors) are limited to the nominal values specified by the manufacturer. In contrast, the values of virtual components can be set to almost anything, however, there are no corresponding physical data. As a consequence, if a PCB is needed, virtual components are not the appropriate choice. In practice, if the goal is to create a production circuit, real components will be used. If the goal is to simulate a lab exercise, virtual components will be used for the passives (rated resistors, capacitors and inductors) and reals will be used for the active components (transistors, diodes, op amps, etc.).

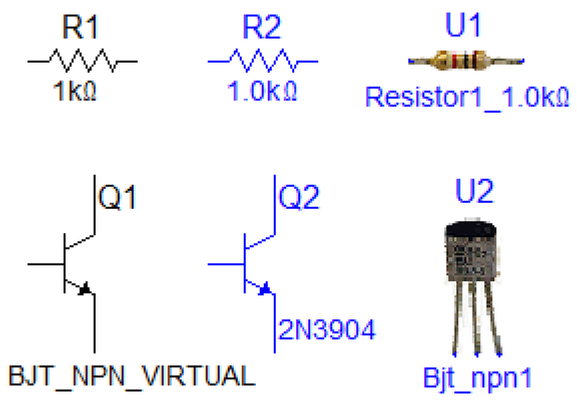
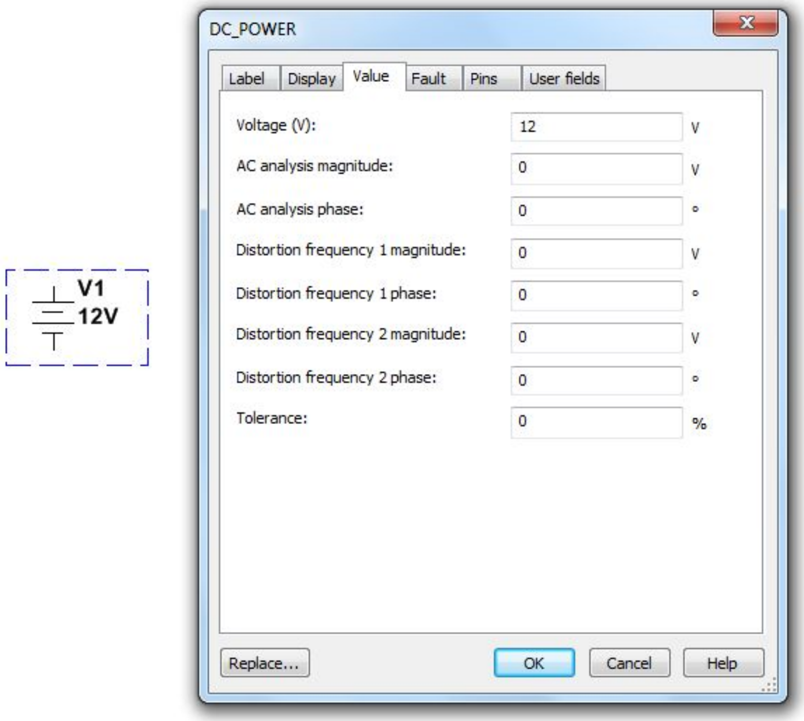


Figure 1-2

To illustrate the use and editing of components, drag a DC voltage source onto the workspace. It will show up with a default voltage value and label. Double click the symbol. A dialog box will pop up next to it as shown in Figure 1-3.



Voltage (V):	12	V
AC analysis magnitude:	0	V
AC analysis phase:	0	°
Distortion frequency 1 magnitude:	0	V
Distortion frequency 1 phase:	0	°
Distortion frequency 2 magnitude:	0	V
Distortion frequency 2 phase:	0	°
Tolerance:	0	%

Figure 1-3

From this dialog you can change a variety of attributes, the most important of which are the voltage value and the name. Double clicking any component will bring up this settings dialog box although the precise items contained within it will vary from component to component. For example, for a resistor there will be a resistance setting instead of a voltage setting. If you need to remove a component, simply select it and hit the Delete key.

Editing the position and orientation of a component is straight forward. Once the item is selected (shown by a surrounding dashed line) it may be moved using either the mouse or the cursor keys. If you need to move a group of components, the mouse may be used to select several items by clicking and then dragging the mouse over them. Every component within the selection box will be highlighted and will move as a group. Also note that it is possible to select the text labels of components and just move them. This can be handy if a label becomes obscured by a wire.

Components may also be rotated and flipped. These commands can be accessed from the main menu, however, it is handy to remember certain keyboard shortcuts (such as CTRL-R). You can also customize the toolbars and add these commands as their own buttons for easy access. The Customize dialog is shown in Figure 1-4 and is accessed via the Options menu.

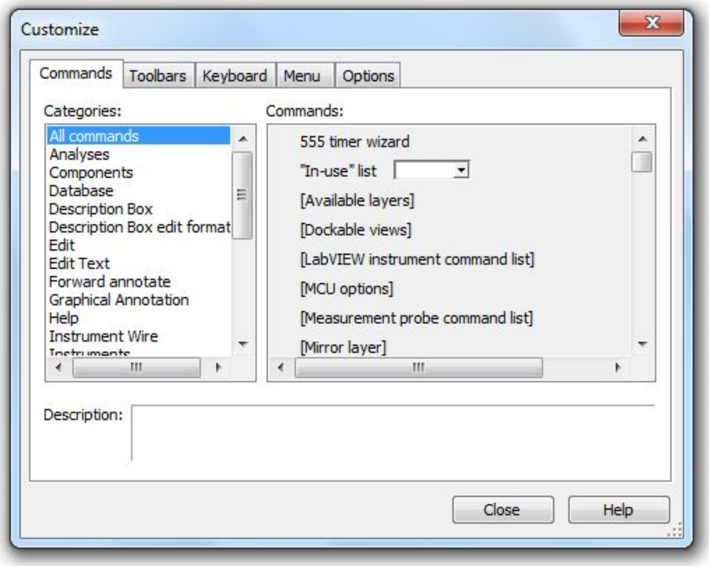
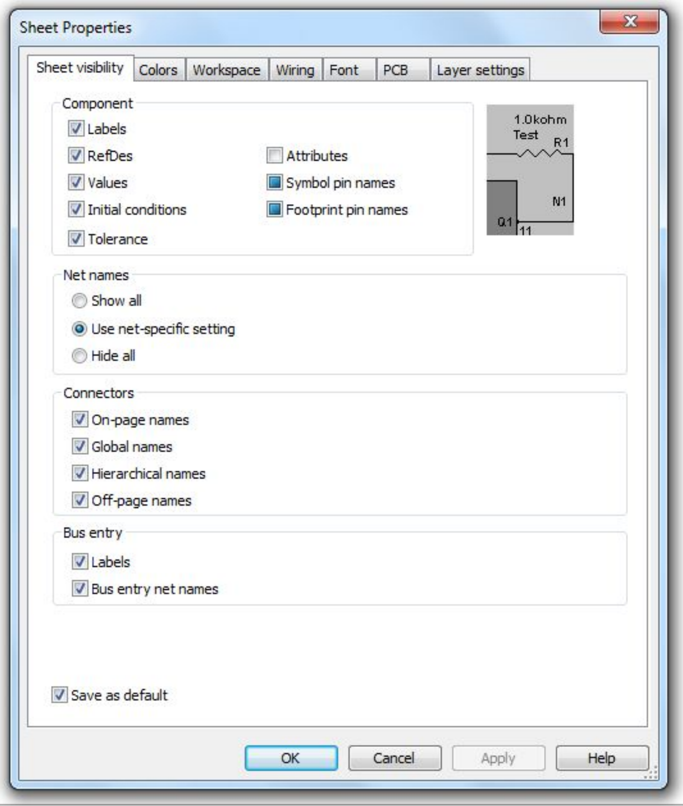


Figure 1-4

Along with editing the components and customizing the toolbars, you may also customize the look of the workspace. Go to the Options menu and select Sheet Properties. From here you can select a variety of color schemes for the components and wiring. You can also select which component items (labels, values, etc.) will be displayed. Fonts may be altered as well. Be fore-warned, it is possible to spend a great deal of time trying to make the workspace look pretty instead of doing truly productive work. Don’t fall into this trap. Before we close this dialog, there is one important setting to note and that is the section labelled “Net Names”. For now leave it as it is. We shall revisit this in the future.


Sheet Properties

Sheet visibility | Colors | Workspace | Wiring | Font | PCB | Layer settings

Component
- [✓] Labels
- [✓] RefDes
- [✓] Values
- [✓] Initial conditions
- [✓] Tolerance
- [ ] Attributes
- [■] Symbol pin names
- [■] Footprint pin names

Net names
- ○ Show all
- ● Use net-specific setting
- ○ Hide all

Connectors
- [✓] On-page names
- [✓] Global names
- [✓] Hierarchical names
- [✓] Off-page names

Bus entry
- [✓] Labels
- [✓] Bus entry net names

[✓] Save as default

Figure 1-5

OK, let's create a circuit and perform a simulation. You should already have a DC voltage source on your workspace. From the virtual (blue) components toolbars, select two resistors and an earth ground symbol. We shall make a series loop of the three elements with the negative end of the power supply at ground. One resistor will need to be rotated 90 degrees (one horizontal and one vertical). In order to wire the items together, simply click on the free lead of one component and move to the desired lead of another. While moving, Multisim will draw a ghost line. Clicking on the second component will create a proper wire (by default, colored red). Wires are always drawn along the horizontal and vertical with 90 degree bends, not directly from point to point. This is the proper way to draw a schematic in the vast majority of cases.

It is possible to click on the middle of a wire in order to tie multiple items together. A small node circle will be drawn at the connection point. To delete a wire, click on it to select it. You will see a set of small box "handles" around it. To remove the wire, simply hit the Delete key. Note that you can also move the wire with those handles if desired. In fact, it is possible to align wires at odd angles with these handles (again, this is not typical).

Note that if you move a component, Multisim will automatically move the wires along with it. You do not have to rewire it. Sometimes, especially if components are too close, the wire may be draw in an odd, looping form. The easiest solution is to simply separate the components a little more.

Once the components are in place and wired, double-click each component to set their values as shown in Figure 1-6. We shall then add two the two voltmeters. Please note that it is perfectly acceptable to change the component values immediately after dragging them onto the workspace; you don't have to wait until they are wired together. It is suggested, though, that you decide upon a particular workflow and stick to it otherwise the chances of forgetting to change components from their default values increases.

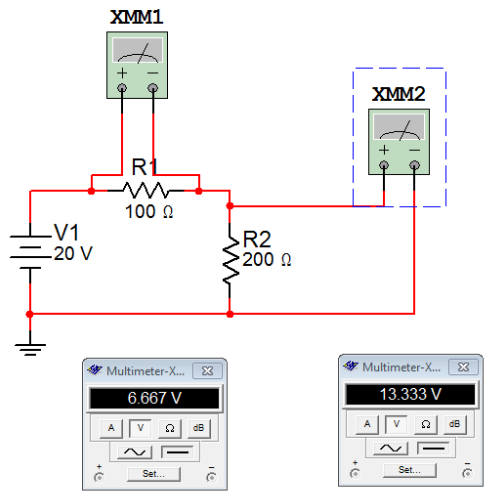


Figure 1-6

To add the voltmeters (XMM1 and XXM2 in Figure 1-6), select a virtual DMM from the top of the Instruments toolbar along the right edge of the screen. Drag this onto the workspace and wire it across R1 as shown in Figure 1-6. Grab a second virtual DMM and repeat the process for R2. Now double click on the DMMs. Two small windows will pop up (they might overlap each other, if so, move one over). Select the V (voltage) button and the straight line (DC) button on each of them. The circuit is now ready to perform a simulation.

To start a simulation, you can select the the rocker switch located in the upper right corner of the menu area. This is the virtual on-off switch. Alternately, you can select Run from the Simulate menu or use the green Run button on the simulation toolbar. In a moment, you should see voltages appear on the two virtual DMMs. **In order to edit the circuit, the simulation must be turned off.** So, if we wish to add or delete components, we must remember to "power down" the circuit just as we would in a real lab. To be safe, do this now.

Multisim has a wide variety of virtual instruments. Some of them are fairly simple such as the DMM just used. Other instruments are virtual recreations of real-world test instruments. For example, from the Instruments toolbar select the Tektronix Oscilloscope. You may place this anywhere on your schematic. Now double click on the small icon for this device (XSC1). A rather ornate window opens which appears to be the front panel of a Tektronix TDS 2024 oscilloscope, very similar to the models we have in the Circuits and Digital labs, right down to subtle shadows around the knobs! See Figure 1-7. This has the advantage of immediacy (assuming you've used this type of oscilloscope before), however, it is not the ultimate way to perform a simulation. We will look at even more powerful and flexible ways of creating simulations in the next exercise. For now, you may wish to delete this new instrument from the workspace and then save the existing schematic. Remember to always save to either your student account on the H drive or to a USB drive. Never save directly to the C drive.

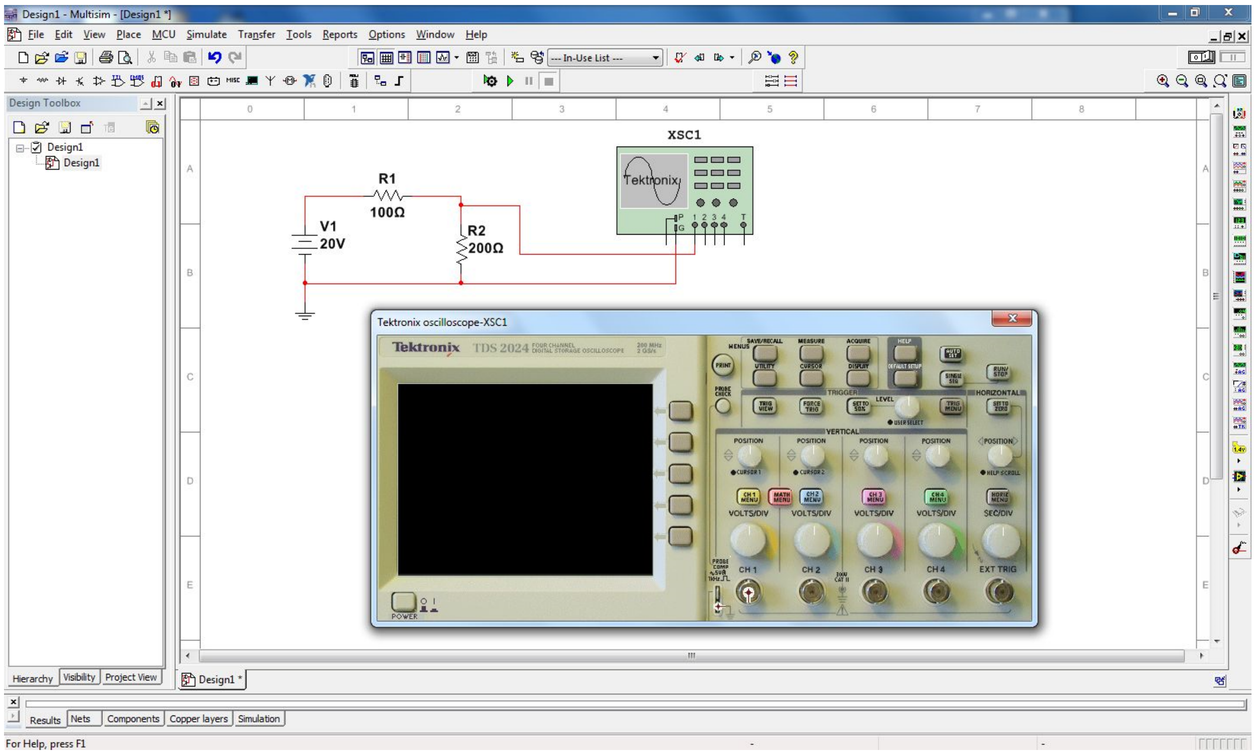

Design1 - Multisim - [Design1 *]
File Edit View Place MCU Simulate Transfer Tools Reports Options Window Help
Design Toolbox
Design1
XSC1
R1
100Ω
V1
20V
R2
200Ω
Tektronix oscilloscope-XSC1
Tektronix TDS 2024
Hierarchy Visibility Project View
Results Nets Components Copper layers Simulation
For Help, press F1

Figure 1-7

There are three very good “every day” uses for Multisim during your studies: First, it is a very handy tool for verifying lab results. That is, you can recreate a lab circuit, simulate it, and compare the simulation to both your theoretical calculations and lab measurements. Second, it is a handy tool for checking homework if you get stuck on a problem. Third, it is convenient for the creation of schematics, for example, for a lab report. An easy way to do this is to draw the circuit, capture the screen image (Windows key+Print Screen key copies it into the clipboard) and then paste the image into your favorite image manipulation program. From there you can crop it, change contrast, etc. as needed and then paste the modified image into your lab report.

At this point you may wish to experiment a bit by rewiring the circuit to measure current or to try building new circuits. As with any tool, continued practice will hone your skill. Once you are done and have saved the file (the extension should be .msX where X is the Multsim version number), close Multisim and then shut down the computer.

2

Multisim Extensions

Objective

The objective of this exercise is to become more familiar with the Multisim electrical circuit simulation package in order to use more generalized simulations via the Grapher Window.

Procedure

In previous work we have examined the basic functionality of Multisim, namely basic schematic capture functions such as component selection, placement, and parameter editing, along with simple simulations using virtual instruments such as a Digital Multimeter (DMM) to measure DC voltage. While virtual instruments are quick and easy to use, and offer some amount of familiarity, they are necessarily limited in other aspects. Some of the issues with virtual instruments include:

1. Limited measurements per unit, for example a single measurement for a DMM, requiring multiple units for multiple measurements.

2. The need to rewire the instruments (and hence the schematic) in order to take different readings.

3. Excessive amount of workspace area obscured by the instrument(s) with accompanying clutter.

4. No convenient way of storing and recalling prior simulations.

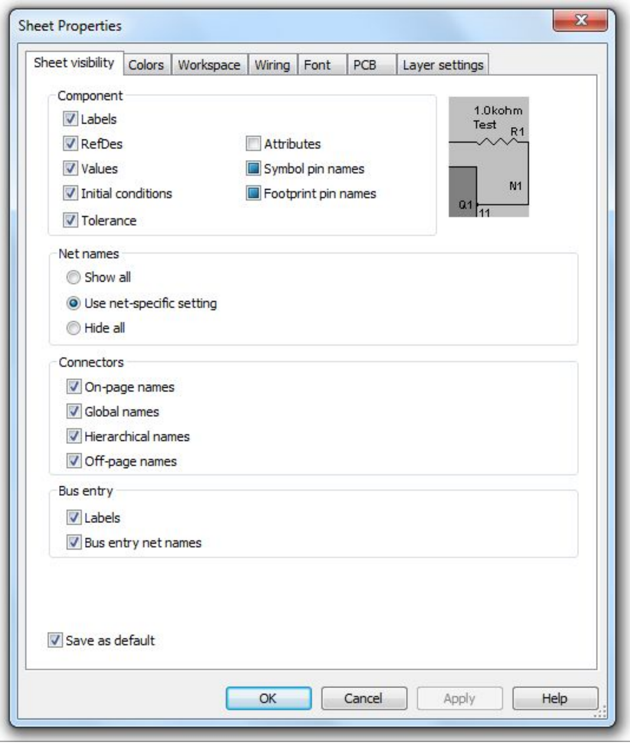
5. No convenient means of exporting simulated measurement data to other programs.

To address these issues Multisim allows non-instrument simulations through the use of the Grapher window. The Grapher is a single, general purpose window that presents simulation data in both text and graphical form, as appropriate. Large amounts of data may be displayed simultaneously. The display itself is highly customizable (titles, axis ranges, colors, fonts, etc.). Each simulation is given its own sheet or tab and these may be saved for future reference. To top it off, nothing needs to be wired into the existing schematic and the Grapher may be minimized for greatest access to the schematic. The Grapher may be used for both DC and AC simulations, and includes interactive measurement tools for certain simulation types.

Because nothing is wired to the schematic, some means of identifying the measurement points is required. Multisim, like virtually all other simulation programs, does this through the use of numbered *nodes*. A node is simply a connection point in the circuit. Ground is always node number 0. The numbers increase as connections are made. Consequently, if the same circuit is entered by two different people who wire the components in a different order, the node numbers will not be the same between the two circuits. This is generally not a problem. As a side note, Multisim will not backfill nodes that have been deleted. For example, if a circuit has been wired up to node ten and then node five is deleted, the next node to be wired will be numbered eleven, not five. Node number five will simply remain unused. Again, this is not a problem. It is important to note that, internally, Multisim always refers to connections via their node numbers, whether you use them or not.

When using virtual instruments, node numbers can add a certain amount of clutter to the schematic, consequently they are not shown by default. To show node numbers, select Sheet Properties from the Options menu. The dialog box of Figure 2-1 opens. This dialog box should be familiar as it was used in the prior exercise to set schematic colors and the like.

If the dialog box looks a little different from Figure 2-1, make sure that the Sheet Visibility tab is selected (leftmost tab). The center area is labeled Net Names. This controls whether or not node numbers will be visible on the schematic. To make node numbers visible, select Show All. Select OK to close the dialog box. We will now proceed to create a simple schematic.



Component
- [✓] Labels
- [✓] RefDes
- [✓] Values
- [✓] Initial conditions
- [✓] Tolerance
- [ ] Attributes
- [■] Symbol pin names
- [■] Footprint pin names

Net names
- ○ Show all
- ● Use net-specific setting
- ○ Hide all

Connectors
- [✓] On-page names
- [✓] Global names
- [✓] Hierarchical names
- [✓] Off-page names

Bus entry
- [✓] Labels
- [✓] Bus entry net names

[✓] Save as default

Figure 2-1

Using virtual components, drag a DC voltage source, an earth ground and two resistors onto the workspace. Connect them in a series loop and edit the component values to 20 volts for the source and 4000 and 6000 Ohms for the two resistors. Once completed, the circuit should look something like Figure 2-2.

Note that ground is node zero. This is always the case. If the components were wired from left to right, the other two nodes will be one and two as shown. If the wiring steps were reversed, the node numbers will be reversed.

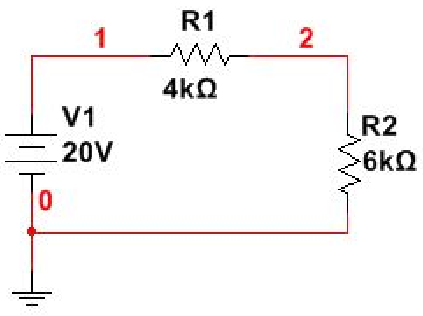


Figure 2-2

We shall perform a simulation to inspect a few DC voltages. Instead of wiring in virtual DMMs, we shall select the Analyses and Simulation item under the Simulate menu (see Figure 2-3).

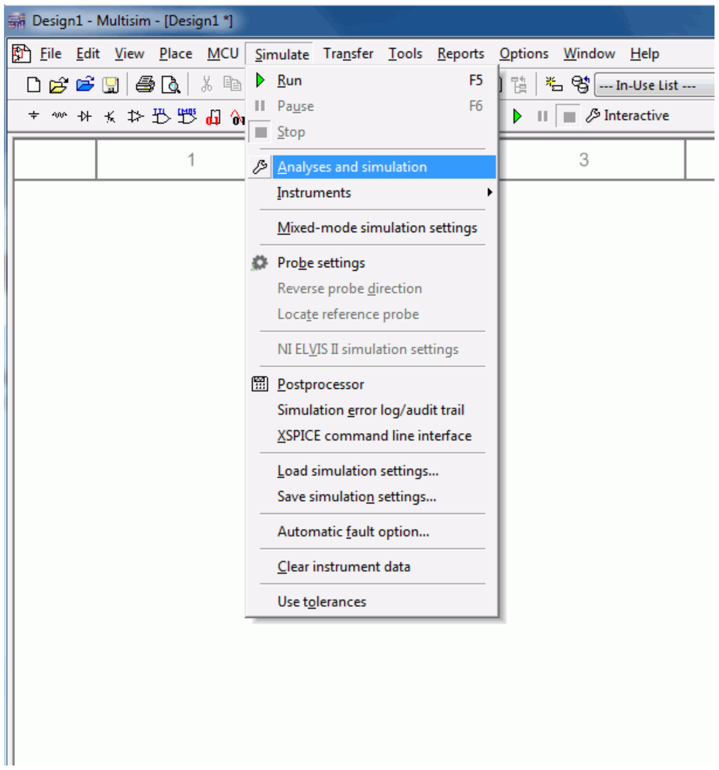


Figure 2-3

A dialog box will open (see Figure 2-4). There are well over a dozen analyses available. To find DC voltages or currents in our circuit, select DC Operating Point from the Analysis list. The dialog box will update and now contains two list areas. On the left will be a listing of available node voltages along with branch currents. The right side contains those items currently chosen for analysis. Node voltages will be shown with a V prefixed to the node number as in V(1). Branch currents will use an I prefix. In version 13 and earlier of Multisim the Analyses menu has a submenu of analysis choices. Simply select the analysis type from the submenu and then continue with the settings as described.

Select both nodes one and two on the input list with the mouse. To transfer them to the output list, select the Add button between the two lists. For now, this is all we need to concern ourselves with but it is worth mentioning that it is possible to create mathematical expressions of variables as well. For example, one node voltage could be subtracted from another in order to find the voltage across a single component or a group of components. Further, a node voltage could be squared and divided by a circuit resistance to determine the power dissipation.

Select the Run button at the bottom of the dialog to start the simulation.

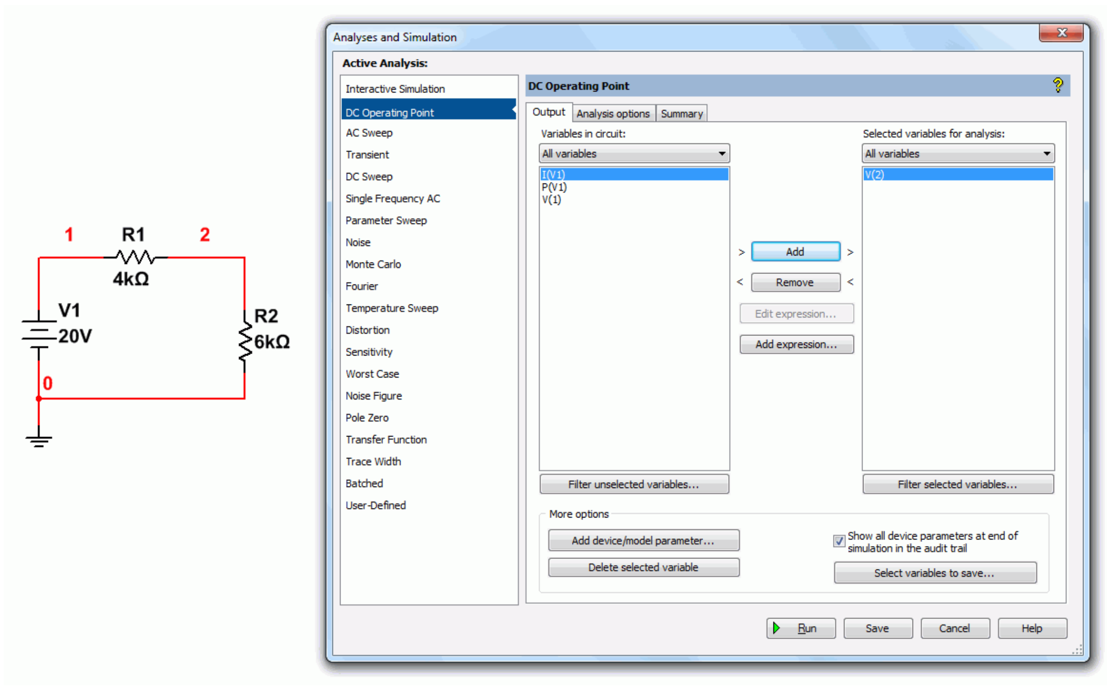
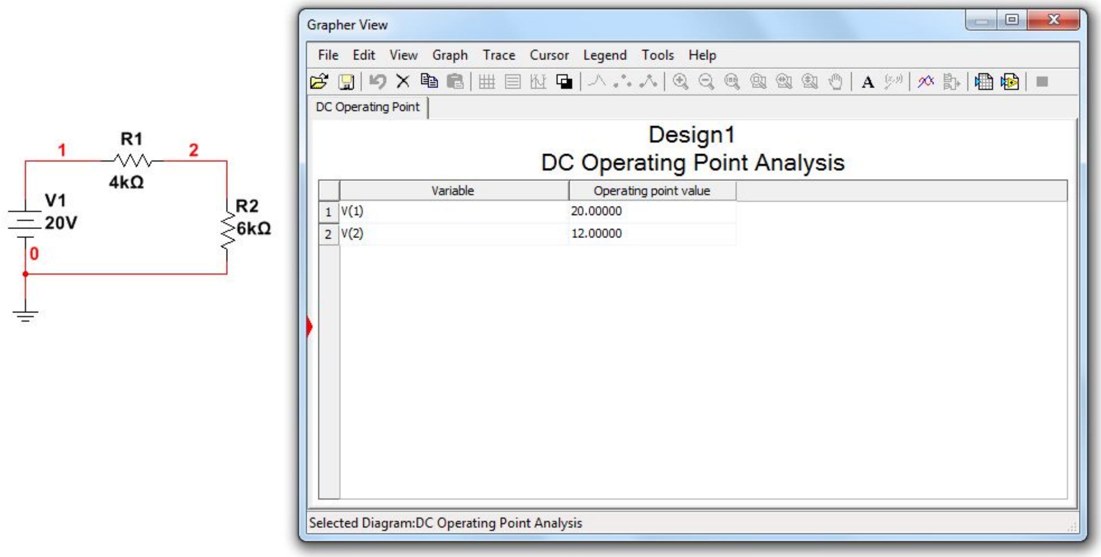


Figure 2-4

After a moment, the Grapher window will appear as shown in Figure 2-5. For this simulation, the Grapher simply lists the node number and the corresponding voltage in two adjacent columns. This is a nice, compact display, much nicer than the individual DMM windows, especially if a large number of voltages are needed.



Design1
DC Operating Point Analysis

	Variable	Operating point value
1	V(1)	20.00000
2	V(2)	12.00000

Figure 2-5

Along the top of the Grapher window is a toolbar area for saving, cutting, customizing, and so forth. As useful as this is, the Grapher comes into its own when used with AC simulations.

Close the Grapher and modify the circuit to replace the DC source with an AC signal source as shown in Figure 2-6. Make sure that the new source is set to 20 volts with a frequency of 1000 Hz (1 kHz).

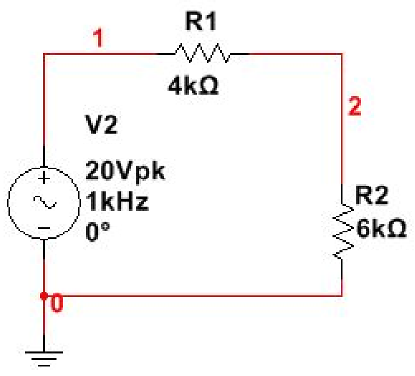
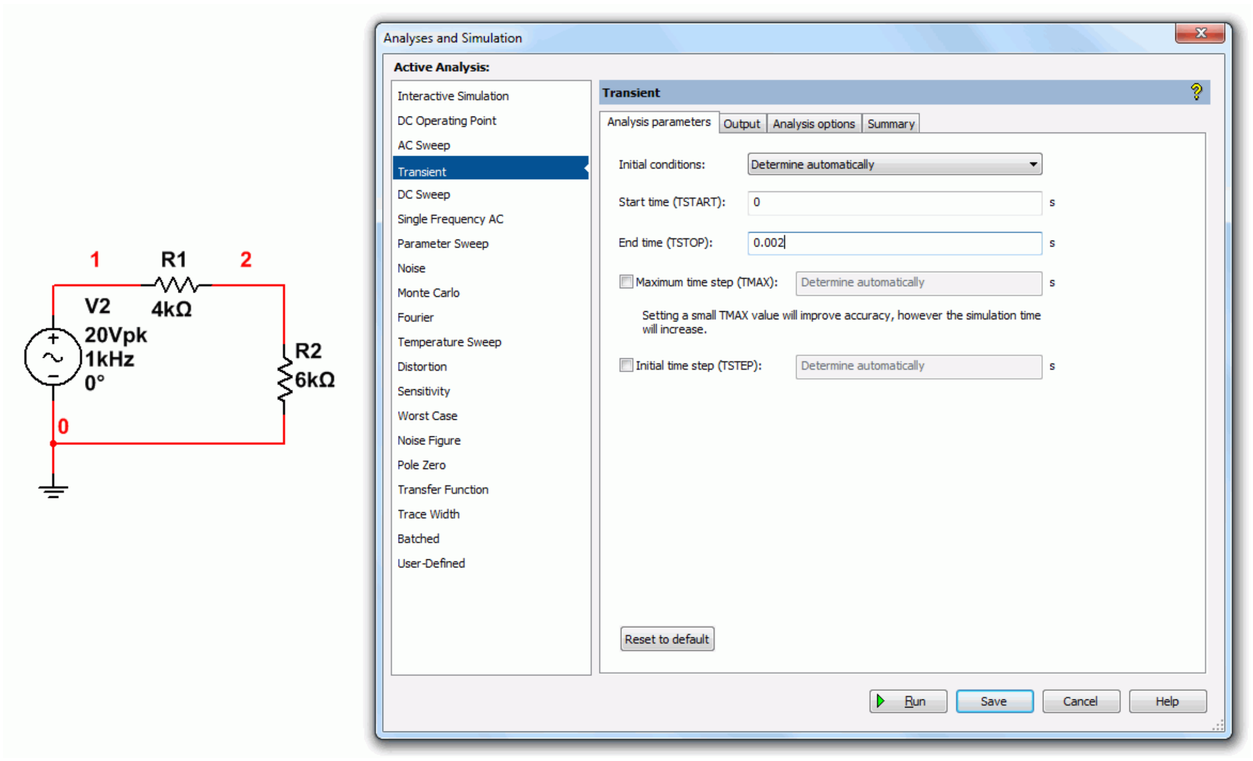


Figure 2-6

We are going to examine the node voltages once again, however, this time the voltages will vary over time. If we were to use virtual instruments for this we'd choose once of the virtual oscilloscopes. In this case, however, we shall choose Transient Analysis from the Analyses and Simulation menu. A dialog box will open like that of Figure 2-7.



Initial conditions: Determine automatically

Start time (TSTART): 0 s

End time (TSTOP): 0.002 s

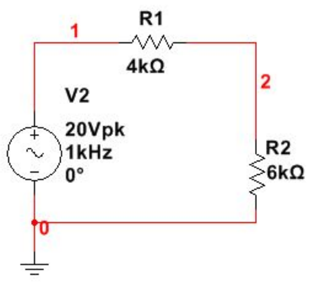
Maximum time step (TMAX): Determine automatically s

Setting a small TMAX value will improve accuracy, however the simulation time will increase.

Initial time step (TSTEP): Determine automatically s

Figure 2-7

The first thing we need to do is select the range of times we wish to see. Set the Start time to 0 and the End time to two milliseconds (0.002 seconds). Now select the Output tab. This is the same tab you saw back in Figure 2-4. Make sure that voltage nodes one and two are in the output analysis (right side) list. Now select Simulate. The Grapher will reopen with a display similar to that of Figure 2-8.



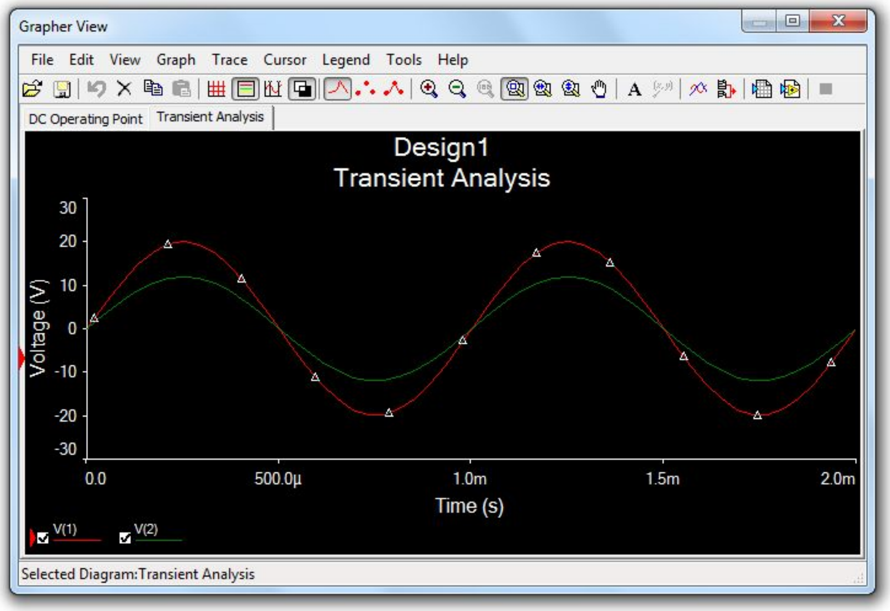
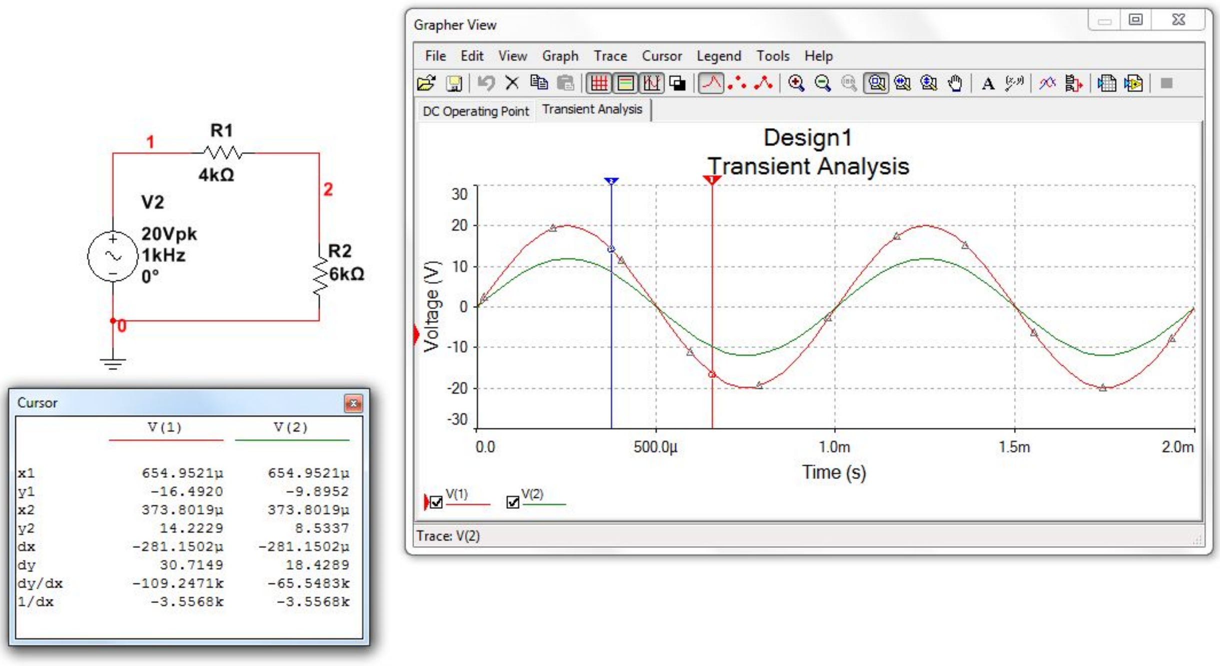


Figure 2-8

The red and green traces graph the voltages at nodes one and two as they change through time. Note that there are many ways to customize this display. For example, selecting the white/black double square toward the middle of the toolbar will toggle the background color and labels between black and white. The grid pattern will overlay a measurement grid and the set of horizontal lines will bring up a color-node number legend. You can also click on the labels and axes to change them.

One of the more useful items is the pair of interactive cursors. The button for this is between the Legend and the Magnify buttons. Selecting this will bring up a pair of vertical cursors which may be grabbed and moved with the mouse. A separate cursor window will open. This will list the coordinates of the cursors on the plot lines along with the differences between the points (that is, how far apart the points are in time and in voltage). It will also list maxima and minima along with other useful data. An example of the Grapher with many of these settings in place is shown in Figure 2-9.



```
              V(1)          V(2)

x1        654.9521µ     654.9521µ
y1         -16.4920       -9.8952
x2        373.8019µ     373.8019µ
y2          14.2229        8.5337
dx       -281.1502µ    -281.1502µ
dy          30.7149       18.4289
dy/dx    -109.2471k     -65.5483k
1/dx       -3.5568k      -3.5568k
```

Figure 2-9

Finally, note that the original DC Operating Point tab is still there. That is, you can quickly recall the original DC analysis by just selecting this tab. It should be clear by now that while the Grapher takes a little more to set up, it is far more flexible and useful than simple virtual instruments.

Assignment

Recreate the transistor amplifier schematic shown in Figure 2-10. Try to make it as close to this drawing as possible. All parameters must be the same. Device designators must also be the same (after all, these will be cross referenced to the PCB layout and bill of materials). The only items to change are the date (use today's date) and the "Designed by" entry (insert your name).

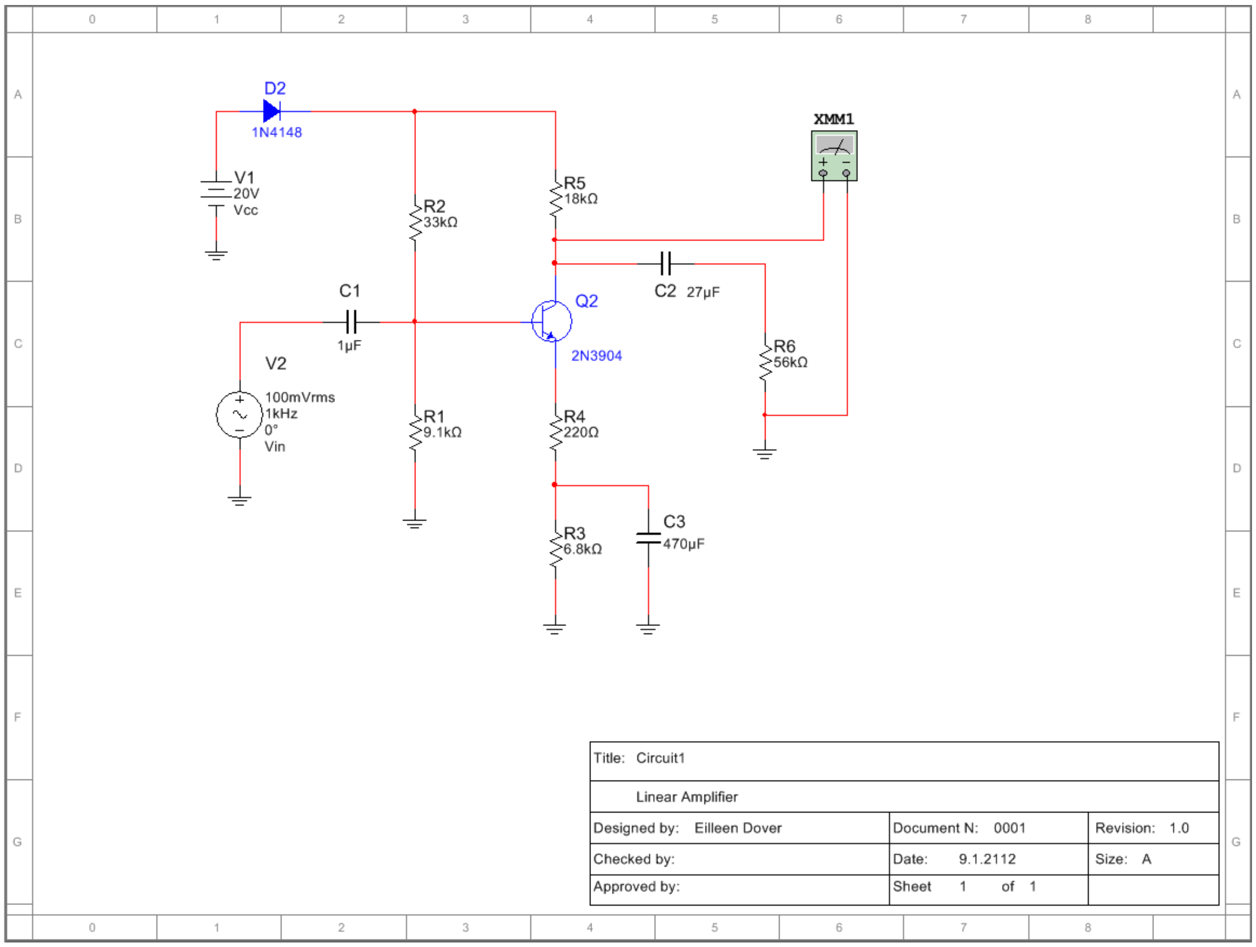

D2
1N4148
V1
20V
Vcc
R2
33kΩ
R5
18kΩ
XMM1
C1
1µF
Q2
2N3904
C2 27µF
R6
56kΩ
V2
100mVrms
1kHz
0°
Vin
R1
9.1kΩ
R4
220Ω
R3
6.8kΩ
C3
470µF
Title: Circuit1
Linear Amplifier
Designed by: Eilleen Dover
Document N: 0001
Revision: 1.0
Checked by:
Date: 9.1.2112
Size: A
Approved by:
Sheet 1 of 1

Figure 2-10

3

Introduction to Python

Objective

The objective of this exercise is to become familiar with the Python IDE for version 3.X while introducing basic mathematical operations, variable types, and printing options.

Background

Virtually all modern programming languages make us of an IDE, or Integrated Development Environment, which allows the creation, editing, testing, and saving of programs and modules. In Python, the IDE is called IDLE (like many items in the language, this is a reference to the British comedy group Monty Python, and in this case, one of its members, Eric Idle).

Before opening IDLE, it is worth recalling that there are three basic types of simple variables in Python: integers (whole numbers, commonly used as an index), floats (that is, numbers with a decimal point, AKA real numbers), and strings (collections of alphanumeric characters such as names, sentences, or numbers that are not manipulated mathematically such as a part number or zip code). A legal variable name must start with a letter. It is then optionally followed by some collection of letters, numerals and the underscore. It cannot contain any other characters or spaces, and cannot be a *reserved word* (i.e., a word with a special meaning in the language such as a command or operator). In Python, variables may be created by simply declaring them and assigning a value to them. Examples include:

```
a=2.3
name=”Joe”
```

It is best to think of the equal sign as “gets”. That is, think of the first example as “the variable `a` gets the floating point value 2.3” and the second as “the variable `name` gets the string *Joe*”. An assignment command such as these literally reserves space in the computer’s memory for the variable and tags it with the variable name. Then, it stores the appropriate value at that location for future use.

Procedure – Output Window

Open IDLE by selecting Python from the Start menu and then choosing the option to open *IDLE (Python GUI)*. Do NOT open the command line. Alternately, open the *IDLE (Python GUI)* icon on the desktop. A simple text window known as the Python shell will open. It should have a white background with a text message at the top and immediately below that, a cursor prompt
>>>
The shell serves two functions. First, it can serve as a sort of scratch pad to try snippets of code (shown in the steps below). Second, it can serve as the text output window for larger programs. **Do not try to use this window for the creation of complete programs that you wish to save.** We shall use this window to create a few variables and perform some basic manipulations on them. Type the following and then hit the Enter key:

```
a=5
```

The >>> should reappear. This command defines a variable called `a` and copies the integer value `5` into it. In similar manner, type in the following commands:

```
b=13
x=5.0
y=13.0
m="Mary"
n="Nancy"
```

It is very important that the "`.0`" portions be included. This is how integers and floats are distinguished: floats always have a decimal point, integers don't. Also, it is possible to define the strings using the apostrophe `'` versus the quote `"`. This can be handy if you need to have a string that includes a quote or apostrophe within it; merely define the string with the other character. In any case, the computer's memory now looks something like this:

name	value
a	5
b	13
x	5.0
y	13.0
m	Mary
n	Nancy

The trick now, of course, is to access these values, manipulate them, and see the results. An important command for this process is the `print` command. `print` will print what follows it, either variables or expressions, on to the output window. Note that like all built-in commands and functions in Python, this command is all lower case. Capitalizing it will generate an error. Also, note that commands will be color coded orange-red.

At the prompt, type the following:

```
print( a )
```

The output should be the integer `5`
Now type:

```
print( a, x, m, n )
```

In this case, the following sequence should result:

```
5 5.0 Mary Nancy
```

Continue with the following expression:

```
print( a + b )
```

This results in the value `18`. This line retrieves the values of `a` and `b` from memory, adds them together, and prints the result on the output window. Neither `a` nor `b` are altered in the process. Alternately, we could have created a brand new variable and printed it. The result will be the same. Enter the following two lines to verify this:

```
c = a + b
print( c )
```

The only difference is that this version adds a new "slot" called `c` to the memory map above. It is worth noting that once a variable is created, its value may be recomputed over and over if desired. For example, type the following:

```
c = 20 + a
print( c )
```

The result should be 25. The first line computes a new value which then overwrites the prior value of 18.

Besides addition, the other main math operators are -, * (multiplication), / (division), ** (exponents, which can also be performed using the function pow(x,y) for x^y), % (modulo), and // (floor divide). Parentheses () may be used to force the execution of some operations before others. Parentheses have the highest precedence and are followed by multiplication, division, addition and subtraction. That is, the expression a=b+c*d will multiply c by d before b is added. To force the addition first, use parentheses: a=(b+c)*d Remember, think of the equal sign as "gets" as in "a gets the value computed by…". It is an assignment, not a true mathematical relation. That is, if at some point in the future the value of b was to change, a will not automatically be altered to reflect that change. This allows you to do the following:

```
c = c + 1
```

Type this in. What do you think the result will be?

The line above may appear a little odd. After all, how can something equal itself plus one? Remember, this is an assignment, not a mathematical relation. What it says is, "Retrieve the current value of c, add one to it, and store the result back in c (overwriting the original value). Print out the value of c. You should be get 26 (the prior value of 25 plus one).

Continuing with the other math operators, type:

```
print( y/x )
```

The result should be 2.6. Now try the following:

```
print( b/a )
```

The result is also 2.6 even though both variables are integers (an integer, of course, can't contain a fractional portion). In essence, Python *promotes* the variables to floats in order to maintain precision, producing a floating point answer. Now try:

```
print( b/x )
```

In this case the answer is again `2.6`. This is because in a mixed calculation between a float and an integer, the integer is again promoted to a float in the calculation in order to maintain the precision of the floating point variable. You can force a variable to be promoted (or demoted) by using the `float()` and `int()` functions.

Now try:

```
print( b%a )
```

The result should be `3`. The modulo operator produces the remainder of the divide, that is, `a` goes into `b` two whole times with `3` left over. Finally, we have floor divide:

```
print( 18.2//4.1 )
```

The result should be `4.0`. You can think of floor divide as like integer divide for floats. That is, `4.1` goes into `18.2` `4.0` times (with `1.8` left over, which you can verify with `print( 18.2%4.1 )`).

What do you expect the results to be from the following?

```
print( y//x )
print( y%x )
```

Type in the above lines and see if you were correct.

At times it is useful to limit the number of digits that are printed. By default, Python uses up to 18 digits if required. Try this:

```
print( x/y )
```

The result is `0.38461538461538464` To limit this to fewer digits, the `round()` function may be used. The first argument is the value of expression to be rounded and the second is the number of digits after the decimal point. Now enter this:

```
print( round(x/y,3) )
```

The result should be `0.385` (rounding to the third digit).

There are also limited operations on strings. The + and `*` operators when used with strings perform *concatenation* and *iteration*. That is, combining strings and repeating strings. Type:

```
print( m, n )
```

You should see:

```
Mary Nancy
```

Now try:

```
print( m+n )
```

The result should be:

```
MaryNancy
```

Note the lack of a separating space. Now type:

```
print( m*3 )
```

The result should be:

```
MaryMaryMary
```

The string is repeated three times. Note that it doesn't make sense to ask for things like `m*2.6` or `m-n`. We can't have a fractional copy of something and what would it mean to subtract "Nancy" from "Mary"? These sorts of statements will generate errors.

Procedure – Editor Window

As useful as the output shell window is, you will not be able to easily edit and save programs from it. For this, you'll need an editor window. From the File menu of the output window select New File. A second window will pop open. It will not contain a cursor. This is a simple text editor. It will not interpret lines as you type them. The edit window is where you will normally write your programs (occasionally going back to the output to see results or to use its "scratch pad" feature).

One of the most useful operators in Python is `#`, which is used for comments. That is, Python will ignore anything on a line that follows this symbol; it's for human consumption only. This is how you can place documentation inside a program. By doing so, the code and documentation can never get separated. Type in the following two line program:

```
# This is my first Python program
print( "Hello Word!" )
```

The editor works like any other text editor that you might have used. That is, you can insert, delete, cut, copy, paste, etc. Unlike a word processor, there is no selection for font, margins, and the like. After all, the point is to write down commands. Python and the computer don't care how pretty those lines appear.

To run the program, go to the Run menu and select Run Module. You will be prompted to save the program. NEVER save your code to the hard drive on a lab computer. ONLY save to either your student account space (H drive) or to an external USB drive. It is suggested that you create a folder on your student account for Python programs and store everything there, using a USB drive as a backup.

When naming a Python program, a `.py` extension must be used. Failing to do so will result in code that will not be recognized by the system as a Python program. For this exercise, it is suggested that the program be saved as `hello.py`

After the filename is entered, select Save. Python will now load your code and start executing it (i.e., performing the commands you entered). Move back to the output window. You should see the following at the bottom:

```
Hello World!
```

Note that Python will not perform any spell or grammar checking for you. So if you spelled World as *Whirled*, that's what it will print out.

Go back to the edit window. You will note that your code is now color coded. It will not do this until the program is saved as a `.py` file. Consequently, it is suggested that after typing in the initial comment header (name, date, program title and description), the program should be saved in order to engage the color coding. This can be very useful for spotting typos and syntax errors once you get used to the color scheme.

In general, the process of developing a program will involve entering and editing code and then saving and running it. The output is then examined to see if it is proper. If not, the code is edited or added to, resaved, and rerun until the output is correct. This process may be repeated many, many times. In larger programs, the task is usually broken into smaller and more manageable chunks, each tested successfully before continuing to the next portion.

Let's edit this program. Sometimes it is useful to print out an entire block of text formatted a certain way (program directions for the user, for example). This can be accommodated through a triple quoted string. Go to the editor window, alter the existing lines and add the new lines so that your program looks like this (include extra spaces as shown in the second `print` statement:

```
# This is my second Python program
print( "Hello Word!" )
print( " " "
Look at the odd formatting of   these    lines.
     They will show up as defined!
" " " )
```

Save and run the program. Do you get the results you expected?

It is a good idea to periodically resave your code if you have made several modifications since the last execution. Few things are more frustrating than losing code because a computer locked up.

At this point, try experimenting with different assignment and print statements. This is a good habit to get into. You can't "break" the computer by typing in improper code. Usually the worst that will happen is that you'll get a syntax error. Consequently, one of the best ways to remember program statements and syntax is to try little snippets of code and see if they do what you expect. The simple act of typing in code will help you remember the details. For this reason, do not make use of cut and paste, at least not until you have mastered the syntax of the language.

Once your coding is done for the time being, save your code, make sure that you close any Python windows and then proceed to shut down the computer.

4

Obtaining User Data

Objective

Interactive programs require data from the user (i.e., the person running the program, who is not necessarily the programmer). In this exercise, the function `input()` will be examined in order to create a simple Ohm's Law calculator

Introduction

The most general means of obtaining information from the user is the `input()` function. When Python executes this command it will wait for the user to enter a string of characters until the user hits the Enter key. These characters are then assigned as a string to a variable. Usually, some form of user prompt will be required (i.e., the question posed to the user). This can be accomplished via a separate print statement or as an argument to the function. Below is a simple example which asks the user for their name and then prints it back out.

```
print( "What is your name? " )
n = input()
print( "Hello", n )
```

Alternately, this can be shortened with the following:

```
n = input("What is your name? ")
print( "Hello", n )
```

It is important to remember that this function always returns a string variable. If the entered data is numeric, it must be turned into either a float or integer. This can be accomplished via the `float()` and `int()` functions. For example:

```
p = float(input(“What is your weight in pounds? ”))
# one kilogram is approximately 2.2 pounds
kg = p / 2.2
print( “You weigh approximately”, kg, “kilograms” )
```

Let’s consider how we might create a simple Ohm’s Law calculator. Before we start coding, we must define exactly what we wish the program to do and create a logical outline. This outline is not written in python but rather a simplified form of English which shows the steps required to solve the problem. One line of pseudo code might correspond to one line of Python. Alternately, it might correspond to many lines of Python. Pseudo code is not tied to a specific language.

In our example, we shall use Ohm’s Law in the form V=I*R. Here’s the pseudo code:

```
1. Give the user directions
2. Ask the user for current in amps
3. Ask the user for the resistance in ohms
4. Compute the voltage from V=I*R
5. Print out the resulting voltage in volts
```

Step one might be very short or very detailed. It all depends on the complexity of the program. Note that in steps two, three and five, we have specified the units. This is important. The user should not assume that it’s OK to enter a current in milliamps, for example. It is worth noting that `input()` can deal with an exponent so a value such as 1.2 milliamps can be entered as `1.2e-3` So here’s the Python code based on this pseudo code.

```
# Programmer’s name, date, etc.

print( “\t\tOhm’s Law Calculator\n” )
print( “This program will determine the voltage given a current
and resistance.\n” )

I=float(input(“What is the current in amps? ”))
R=float(input(“What is the resistance in ohms? ”))
```

```
V=I*R

print( “The result is”, V, “volts” )
```

The `\t` and `\n` sequences are used to enter a Tab and a Newline, respectively. This makes the print out look a little nicer. Enter this program, save it as OhmsLaw.py and run it. Try the values 2 amps and 10 ohms. The result should be 20 volts. Also try some large and small values, for example, test the program with a current of 3 milliamps (3.0e-3) and 5 k ohms (5.0e3). The result for this should be 15 volts. **It is important to always test your code!** Never assume that it runs properly without exhaustive testing.

Assignment

Based on the example, create two new programs using Ohm’s Law. The first version should ask for current and voltage to determine and print the resistance. The second should ask for voltage and resistance to determine and print the current. It is suggested that you name these OhmsLawR.py and OhmsLawI.py. Test the first program using 4 amps and 20 volts. Test the second program with 12 volts and 5 k ohms.

To enhance these programs, you might consider using a triple quoted string to build a simple version of a circuit containing a voltage source and resistance. For example, a single resistor can be created using front and back slashes as follows:

```
print( “---/\/\/\/---“ )
```

5 Conditionals: if

Objective

The objective of this exercise is to become familiar with the concept of *branching*. We introduce the *if* conditional statement for simple decision making. We shall also introduce the concept of menu-driven programs. In the process, we shall create a program which estimates battery life.

Introduction

Our prior programs could be classified as simple linear or straight-line programs. The program flow was fairly straight forward: Give the user directions, ask the user for data, perform a few calculations based on those data and then print out appropriate results. The next level up in sophistication is the concept of branching. That is, the execution path through the code can vary depending on certain conditions. You might think of this as the program making decisions to do one thing or another. The fundamental conditional operation is the `if` statement. It looks something like this:

```
if conditional expression:
     resulting action
```

The *conditional expression* is some manner of test, for example to see if one variable is larger than another. The tests include = = (same as), != (not same as), >, <, >= and <=. The logical directives `and` and `or` are also available. The *resulting action* is any legal block of Python code. It may be a single line or a multitude of lines. So, if the conditional expression is true, the resulting action is performed. If the expression is not true, the action is skipped. In either case, program execution picks up at the next line after the resulting action block. **It is extremely important to note that the resulting action block must be indented.** All lines of the block must be indented by the same amount. This is how Python recognizes that it is a single block of code.

As an example, suppose we'd like to test to see if variable A is larger than variable B. If it is, we'd like to print out the message: "It's bigger". After this, we want to print out the message "Done", whether or not A was larger.

```
if A > B:
     print( "It's bigger" )

print( "Done" )
```

Because the second print statement is not indented, it is not part of the block, therefore it is always executed. If the second print statement had been indented instead, then "Done" would only be printed if A was larger than B. A common beginner's syntax error is to forget the colon at the end of the if statement.

For another example, consider that you have a floating point variable named T that represents a computed time in hours. Instead of printing this out as hours with a fraction, you prefer to present it as hours and minutes. A floor divide can be used to obtain the whole hours:

```
h = T // 1.0
```

Similarly, a modulo can be used to obtain the fractional portion, which when multiplied by 60.0 will yield the minutes:

```
m = (T % 1.0) * 60.0
```

So, you could print out the result as follows:

```
print( "The time is", h, "hours and", m, "minutes" )
```

Of course, what it the minutes portion works out to zero? Reading something like "The time is 5 hours and 0 minutes" looks a little strange. We'd prefer to leave off the "and 0 minutes" portion. This can be achieved with a simple set of if tests:

```
if m != 0.0:
     print( "The time is", h, "hours and", m, "minutes" )
```

```
if m == 0.0:
     print( "The time is", h, "hours" )
```

This sort of "one-or-the-other" construct is fairly common. To make life a little simpler, we can use the `else` clause:

```
if m != 0.0:
     print( "The time is", h, "hours and", m, "minutes" )
else:
     print( "The time is", h, "hours" )
```

If `m` is non-zero, the full print statement is used, otherwise (else) the simplified version is used. Enter the completed program below and try it with several different values, some whole numbers, others not, and inspect the results:

```
T = float(input("Please enter a time value: "))

h = T // 1.0
m = (T % 1.0) * 60.0

if m != 0.0:
     print( "The time is", h, "hours and", m, "minutes" )
else:
     print( "The time is", h, "hours" )

print( "Done!" )
```

If you look carefully, you might note that under certain circumstances the printout may still be less than satisfactory (hint: what about seconds?). How might this issue be fixed?

Assignment

Now let’s consider our battery life estimator program. It is suggested that you name this program BatteryLife.py. The goal is to create a simple utility where the user can select a battery from a list of choices, specify a current draw and have the program print out the expected lifespan of the battery. This hinges on the concept of battery amp-hour ratings. That is, the expected life of a battery in hours is equal to its amp-hour rating divided by the current draw in amps. The amp-hour rating depends on the size of the battery and its chemical composition (for example, D cells are higher than C cells and alkaline cells are higher than zinc-carbon cells). Here are some examples in amp-hours:

Type	AAA	AA	C	D
zinc-carbon	.2	.4	1.5	3.0
alkaline	.5	.8	3.2	6.2

The expected life is simply the desired amp-hour rating above divided by the current draw:

```
life = amphours / I
```

Here is an appropriate pseudo code for the program:

```
1. Print out program description and user directions
2. Ask user for the current draw
3. Print out the battery choice list
4. Ask the user to choose a battery from the list
5. Look up the corresponding amp-hour rating for the chosen battery
6. Compute the expected battery life
7. Print out the expected battery life
```

Some of these lines we have already seen. For example, the “Ask” directives of steps two and four imply the use of `input()` statements. For example, for step two we might use:

```
I = float(input(“What is the current draw in amps? ”))
```

The printouts and computations of steps one, six, and seven should also be familiar (you may find the prior time printout example to be useful here). Step three is really nothing more than either a set of `print` statements or a single `print` statement using a triple quoted string to display all of the battery choices. It is safest to present the menu of battery choices numerically as it is easy for the user to select an item by simply typing in a number versus spelling out the entire description of the battery (it turns out to be easier for the programmer, too). For example, the first few statements of this step might look like:

```
print( “Please choose a battery from the list below” )
print( “1. AAA zinc-carbon” )
print( “2. AAA alkaline” )
print( “3. AA zinc-carbon” )
# and so forth
```

Step four would then look something like:

```
battery = int(input(“Please enter the battery choice number, 1
through 8: ”))
```

The trick is step five, looking up the corresponding amp-hour rating. For this step a series of `if` statements may be used:

```
if battery == 1: # AAA zinc-carbon
     amphours = 0.2
if battery == 2: # AAA alkaline
     amphours = 0.5
if battery == 3: # AA zinc-carbon
     amphours = 0.4
if battery == 4: # AA alkaline
     amphours = 0.8
# and so forth
```

There are other ways to structure this “chain of choices” as we shall see, but this technique is perfectly serviceable for now. Along with your code, hand in two trial runs: The first should be a zinc-carbon D cell with a .1 amp draw and the second should be a AAA alkaline cell with a .6 amp draw.

6

More Conditionals

Objective

In this exercise, the concept of conditional statements is expanded. We expand on the *if* conditional statement with the use of the logical operators `and` and `or`. These can be used to make branches that are dependent on more than one condition. In the process, we shall create a program that can be used to check the tolerance of resistors.

Introduction

Our initial examination of the *if* conditional allowed for the conditional execution of code. A single item (variable) may be tested in the following manner:

```
if conditional expression:
    resulting action
```

If multiple conditions need to be met together, they may be handled with successive tests, as in:

```
if conditional expression:
    if conditional expression 2:
        resulting action
```

This idea can be cascaded on and on. It is important to note, though, that each successive test needs to be indented one more level. Failure to do so will ruin the statement block hierarchy and lead to unpredictable code execution (or syntax errors).

For example, to see if the variable `A` is larger than the variable `B` and is also smaller than the variable `C`, we might use the following snippet of code:

```
if A > B:
     if A < C:
          print( "It's just right!" )
```

Note that it is possible to use the `else` clause for a cascade in order to catch conditions that are not met. For example:

```
if A > B:
     if A < C:
          print( "It's just right!" )
     else:
          print( "A is greater than B but not smaller than C" )
else:
     print( "A is not greater than B" )
```

It is extremely important to note the manner of indentation. Note that each `else` aligns vertically with its parent `if`.

So, the above format is applicable if several different conditions must be met simultaneously. An alternate possibility is that at least one condition of several needs to be met. For example, we might define success as the variable `A` being larger than `B` or if `A` is smaller than `C`, but not necessarily both. Two simple `if` statements appear to work at first, but there is a problem:

```
if A > B:
     print( "Success!" )
if A < C:
     print( "Success!" )
```

The problem is that if both conditions are met, *Success!* will be printed out twice. An `else` can alleviate this problem:

```
if A > B:
     print( "Success!" )
else:
     if A < C:
          print( "Success!" )
```

A more compact form uses the `elif`, which is just a contraction of `else` and `if`:

```
if A > B:
     print( "Success!" )
elif A < C:
     print( "Success!" )
```

This removes the potential double print out but it still leaves a redundant `print` statement.

The easiest way around this modest mess is to make use of the logical operators `and` and `or`. These are used to make compound tests. The `and` operator is used to make test which require all parts of the expression to be true while the `or` operator is used when at least one part needs to be true. Here are the updated versions of the two examples above. Note how compact they are in comparison:

```
if A > B and A < C:
     print( "It's just right!" )

if A > B or A < C:
     print( "Success!" )
```

It is important to note that any `and` can be turned into an `or` by negating the individual tests and the outcome (i.e., reversing their sense) :

```
if A <= B or A >= C:
     print( "It's no longer just right!" )

if A <= B and A >= C:
     print( "Not successful!" )
```

Finally, multiple `and` and `or` operators may be used in a given expression. This may require the use of parentheses.

```
if (A > B and A < C) or X != 1:
     print( "Done" )
```

In the example above, *Done* is printed if `X` does not equal 1, regardless of the value of `A`.

Assignment

Now let’s consider a resistor tolerance program. It is suggested that you name this program ResistorTolerance.py. The goal is to create a simple utility where the user enters a nominal resistor value and tolerance (i.e., based on the color code) along with a measured resistance. The program will then determine if the measured resistor is within specs. If it is, a success message is produced, if not, a failure message is produced along with the actual tolerance. The pseudo code might look something like this:

```
1. Give user directions
2. Ask user for nominal and measured resistor values in ohms along with the tolerance as a percent.
3. Compute the legal maximum and minimum resistor values.
4. Compare the measured value to the legal max and min. If it’s within, print out a success message. If it’s outside the range, compute the actual tolerance and print it out along with a failure message.
```

There are other ways of designing this program. For example, you could compute the actual tolerance immediately and then compare that to the stated tolerance to see if the resistor is within specs.

Creating the program consists of many steps we’ve already seen. For example, giving the user directions involves simple `print` statements while obtaining the measured and nominal resistor values can make use of `input()` statements. The tolerance is specified to the user as a percentage as that’s easiest for them, however, computation requires that it be expressed as a factor. You can divide the tolerance by 100 after obtaining it but it might be easier to do it all in one step:

```
tol = float(input(“Enter the percent tolerance: ”))/100.0
```

The upper and lower limits can then be found by determining the offset in Ohms and then adding and subtracting this value to/from the nominal:

```
Roffset = Rnominal * tol
Rupper = Rnominal + Roffset
Rlower = Rnominal - Roffset
```

We are now ready for step four. Note that in order for a resistor to be considered good, it must meet two requirements: It must be equal to or greater than the lower limit and also be equal to or less than the upper limit. The `and` operator is perfect for this:

```
if Rmeasured >= Rlower and Rmeasured <= Rupper:
     print( "This device is within tolerance." )
```

Conversely, the resistor could be checked to see if it's bad. That is, we can use the `or` operator by reversing the sense of the test:

```
if Rmeasured < Rlower or Rmeasured > Rupper:
     print( "This device is out of tolerance." )
```

There remains the issue of computing the actual deviation of a failed device. This may be done via a simple formula:

```
Actualdev = 100.0 * (Rmeasured - Rnominal) / Rnominal
```

So, combining the elements above, step four may be performed in two basic ways. First, by checking to see if the resistor is good and second by checking to see if it's bad:

```
if Rmeasured >= Rlower and Rmeasured <= Rupper:
     print( "This device is within tolerance." )
else:
     Actualdev = 100.0 * (Rmeasured - Rnominal) / Rnominal
     print( "The device is out of tolerance." )
     print( "The actual deviation is", Actualdev, "percent" )
```

or

```
if Rmeasured < Rlower or Rmeasured > Rupper:
     Actualdev = 100.0 * (Rmeasured - Rnominal) / Rnominal
     print( "The device is out of tolerance." )
     print( "The actual deviation is", Actualdev, "percent" )
else:
     print( "This device is within tolerance." )
```

Trials

Along with your code, hand in two trial runs: The first should be a nominal 100 Ohm 10% measured at 105 Ohms and the second should be a 4.7 k 5% measured at 4320.

7

Random Numbers

Objective

The objective of this exercise is to become familiar with the generation of random numbers. One possible use is in the area of circuit simulation. To visualize this, a simple simulation of a voltage divider circuit will be created making use of randomization to simulate variances in resistance values from the nominal value.

Introduction

The generation of random numbers lends itself to a variety of applications. In general, games rely heavily on random numbers. Without randomization, games would be ultimately predictable, and consequently, boring. It is the unexpected variation which keeps the mind engaged and the player interested. Another area of use is circuit simulation. Even that most simple of components, the resistor, has some level of uncertainty associated with it. For example, we might purchase one thousand 220 Ohms resistors rated at 10% tolerance. This means that the acceptable range of variation is +/- 22 Ohms. Consequently, any particular resistor may have a value between 198 and 242 Ohms. An obvious question is whether or not this variation will adversely affect a particular circuit design. Because every resistor (along with every other component) will exhibit some variance in its value, Ohm's law tells us that this will create some variance in the associated voltages and currents. Further, it is **not** true that if each component has a tolerance of, say 10%, that the overall circuit variation can be no greater than 10%. It is quite possible for multiple 10% tolerance components to be off in such a way that their combined effect will be much greater than 10%.

For large circuits, computing the worst case variations along with the typical variations caused by component tolerances can be a tedious job. Simulation programs such as Multisim have analyses for just such circumstances. One of particular interest here is called Monte Carlo Analysis. A Monte Carlo analysis randomizes components in a simulation based on their tolerance, performs the simulation, re-randomizes the components to perform another simulation, and on and on for perhaps dozens or even hundreds of simulations. Examination of the results will yield a decent idea of what a typical production spread would be. Our goal in this exercise is to write a very small version of this in order to understand how to use random numbers.

Before we go any further it is important to note that we will be producing what are properly known as *pseudo random sequences*, not true random numbers. Truly random numbers are in no way related to each other. That is, given a list of truly random numbers, no pattern or function exists that would allow you to predict with any certainty what the next number will be. Fair coin flips are a good example of the process. The flip of a coin has no bearing on any subsequent flip of the coin. Even if we flip ten heads in a row, the likelihood of the next flip being heads is still 50-50. Because computers are inherently deterministic instead of random, it's very difficult to get truly random sequences out of them. This deterministic quality is normally a good thing. After all, we tend not to like machines and tools that "behave" in an unpredictable manner. You rightly expect that stepping on the brake will stop your car on dry pavement 100% of the time, not that it will occasionally cause the vehicle to accelerate suddenly, turn on the radio, or cause your windows to open.

Most computer languages, Python included, use mathematical techniques to create sequences of numbers that appear to be uncorrelated. That is, they appear to be random. If you run the sequence long enough, however, you will note that it repeats. In other words, it becomes predictable, and therefore not random. Properly done, the size of these sequences is very, very long and the values can be treated as random for all but the most sensitive and demanding of cases. (A Monte Carlo analysis would be fine with this.)

Python does not have a built-in random function. Random functions in Python, like many advanced math functions, are found in external modules. You may think of modules as libraries of code written by other people which you can add to your program. In fact, with continued study you will be able to create your own modules eventually. These modules must be imported into your program before you can use them. The import directives usually occur at the very beginning of the program. To import the random module, use the following code:

```
import random
```

There are functions within the random module that you may find useful. The first is `random()`. This returns a floating point value between 0 and 1. Consider the following code snippet:

```
import random

print( random.random() )
print( random.random() )
print( random.random() )
```

This will print out three fractional values. You might get:

```
0.01254372
0.93470061
0.50003267
```

You are just as likely to get some other sequence of three values. Moreover, every time you run the code, you'll get a different sequence. Another useful function is `randrange()`. This will produce an integer value between your stated extremes. For example, the following will produce an integer from 0 up to (but not including) 10:

```
import random

print( random.randrange(10) )
```

You can also include a start point (and other limits). The following produces an integer from 2 up to (but not including) 10:

```
print( random.randrange(2,10) )
```

This function is particularly useful if you want to randomly choose one item from a list.

So, what if we want to pick a random *float* between two extremes? `random.randrange()` only produces integers so it won't work. To do this, we can simply scale the basic `random.random()` function. Remember that this function will produce a value between 0 and 1. To get a different range, all we have to do is scale and offset the result. For example, if we need a value between 0 and 50, we simply multiply the function by 50.0.

```
x = 50.0 * random.random()
```

What if we want a value between 80 and 100? This requires both scaling and offsetting. First, note that the range of values spans 20 (that is, 100-80). So, we scale the function by 20 to give us a random number between 0 and 20. Then we add the lower limit of 80 to offset the result to 80 through 100:

```
x = 80.0 + 20.0 * random.random()
```

In other words, 80 is the lower bound while 20 is the difference between the upper and lower bounds. So the result is that `x` will be a floating point value between 80 and 100.

Assignment

Let’s consider making a miniature Monte Carlo simulator for a two resistor voltage divider. It is suggested that you save this as Monte.py. This circuit will consist of a voltage source E and two resistors in series, first R1 and then R2. For simplicity, we shall assume that the voltage source is perfectly stable. The two resistors, however, will have a stated tolerance. We would like our program to simulate the action of picking two resistors from bins, that is, randomizing their values, and then determine the voltage across R2. The pseudo code would look something like this:

```
0. Don’t forget to import the random module!
1. Give user directions
2. Ask user for voltage source value
3. Ask user for nominal value and percent tolerance of resistor one
4. Ask user for nominal value and percent tolerance of resistor two
5. Produce randomized value for R1 and R2 (R1rand and R2rand)
6. Determine the voltage across R2 using the voltage divider rule:
        VR2 = E * R2rand / (R1rand + R2rand)
7. Print out the randomized resistor values and resulting R2 voltage (R1rand, R2rand and VR2)
```

By now, steps one, two, six and seven should be obvious. Steps three and four may be handled with a set of `input()` statements, for example:

```
R1 = float(input(“Enter the resistance of R1 in Ohms: ”))
Tol1 = float(input(“Enter the percent tolerance of R1: ”))
Tol1 = Tol1/100.0 # Turn this from a percentage into a factor
```

The generation of the randomized resistor values in step five takes a little thought. There are a few different approaches. For example, a randomized percentage could be generated and then applied to the nominal value. Conversely, the maximum and minimums could be calculated and the randomized resistor determined using the scale and offset technique shown earlier. This builds on the prior programming assignment which dealt with resistor tolerance.

```
R1max = R1 + R1 * Tol1
R1min = R1 - R1 * Tol1
R1rand = R1min + (R1max - R1min) * random.random()
```

Compare the third line above to the line of code computing `x` two pages back. The offset value is the lower bound and the scale factor is the difference between the bounds. Because `random.random()` can be anywhere between 0 and 1, the result can be anywhere between `R1min` and `R1max` (just substitute 0 and 1 for `random.random()` and simplify the equation to prove it to yourself).

Another approach is to simply adjust the tolerance itself. Think of it in terms of the resistor having anywhere from -100% to +100% of the stated tolerance. Because we're computing the resistors with factors instead of percentages, all we need is a random float between -1 and +1. Then we multiply this by the stated tolerance to obtain the actual randomized tolerance which can then be used to find the resistor value.

```
r = 1.0 – 2.0 * random.random()
Tol1rand = Tol1 * r
R1rand = R1 + R1 * Tol1rand
```

Whichever method is chosen, `R1rand` represents a randomized version of R1, that is, R1 with tolerance applied. The same approach may be applied to R2 to generate `R2rand`. These values are then used to compute the voltage as indicated in step six of the pseudo code.

Trials

Run three trials of the same circuit: E = 9 volts, R1 = 1k @ 10%, R2 = 2k @ 5%. The nominal result for this is 6 volts. Your three runs should produce results in this neighborhood but it is extremely unlikely that any will produce exactly 6 volts or that any two trials will be identical. Do not round any of the values or the variances may be obscured.

8
Iteration

Objective

The objective of this exercise is to become familiar with the concept of iteration, also known as looping. We shall also investigate the creation of simple text-based graphs. In the process, a program that will illustrate the Maximum Power Transfer Theorem will be created.

Introduction

The ability to repeat a series of instructions with controlled variance is an extremely powerful computing tool. There are a few different ways to do this in Python, each with their own strengths.

The first loop control structure is the `while` loop. At first glance this looks something like an `if` statement:

```
while control expression:
     statement block
```

The statement block, which might be many lines long, will be repeated as long as the control expression is true. Like the `if` statement, this block **must** be indented. The control expression is basically the same as those used in `if` statements: They tend to be simple variable tests. Further, **it is important that one or more of the variables used in the control expression change during the looping process otherwise the loop will try to run forever.** For example:

```
x=1
while x<10:
     print( x )
```

The first time the `while` loop is entered, `x` is checked to see if it is less than 10, which it is. Consequently, the `print` statement is executed and the value 1 is printed. This is the last statement of the block so program flow loops back to the control expression. `x` is checked again to see if it's less than 10. It is, so the `print` statement will be executed again. Because `x` never changes, this loop never stops. The result of this bit of code is the number 1 printed over and over and over, not stopping until the program is aborted (if this happens accidentally, a loop may be aborted by holding down the Ctrl key and pressing the C key). Consider this change:

```
x=1
while x<10:
     print( x )
     x=x+1
```

Now `x` is incremented by 1 each time through the loop. On the tenth time starting the loop the test will fail because `x` will no longer be less than 10 (it will equal 10). The loop terminates and program flow picks up at the first line after the loop's statement block. It is also possible to use the `and` and `or` operators in order to check for multiple conditions.

A second technique to create a loop is through the `for` statement. The template looks similar to the `while` structure:

```
for variable in value list:
     statement block
```

`value list` can be a simple listing of values such as:

```
for x in 1,3,25,17:
```

This loop will execute four times. The first time through `x` will take on the value 1, the second time 3, the third time 25, and 17 for the final iteration. This is very convenient if a variable needs to cycle through a set of very specific values, particularly if those values are not in ascending or descending sequence or related in some obvious way (such as doubling each time). On the other hand, if the variable is changing by a specific amount each time, the `range()` function can be used to simplify the value list. The arguments to the `range()` function are variable. The simplest version just specifies the number of values, starting from zero. For example:

```
for x in range(5):
     print( x )
```

The code above will print the numbers 0, 1, 2, 3 and 4. Separate starting and ending values may also be used:

```
for x in range(3,7):
     print( x )
```

This will print out the values 3, 4, 5, and 6. Next, an increment value may be included:

```
for x in range(3,11,2):
     print( x )
```

This will print out the values 3, 5, 7 and 9. Note that in all cases the termination value is not included in the resulting sequence. Finally, negative values may be used to decrement rather than increment:

```
for x in range(5,2,-1):
     print( x )
```

This results in the values 5, 4 and 3 being printed.

The choice of whether to use a `while` or a `for`, and whether or not to use `range()`, will depend on the requirements of the variable (or variables) being used. Indeed, it is possible to structure virtually any loop as a `while` loop but it is not always the most efficient way to do it. Here are some general rules: If there is a single variable that needs to cycle through a set of values which are based on a simple increment or decrement (i.e., adding or subtracting the same amount each time), a `for` with `range()` is the obvious choice. If the values are unique and cannot be created through a simple increment, decrement or other direct formula (for example, a series of measured resistor values), a `for` with *value list* is probably appropriate. Finally, if the variable changes in an obvious fashion which is not a simple increment (such as doubling) or if there are multiple conditions/variables that need to be examined, the `while` is most likely the way to go.

Assignment

We shall now write a program which will make use of iteration to illustrate the Maximum Power Transfer Theorem. It is suggested that the program be saved as MaxPower.py. This program will feature both tabular and graphical output. The program should ask the user for a voltage source value along with its internal resistance. It will then create a table of results for a range of load resistances. The load resistance will range from one-tenth of the internal resistance to twice the internal resistance, incrementing by one-tenth of the internal resistance each time. For example, if the internal resistance is 200 Ohms, the load will start at 20 and increase by 20 until it gets to 400. This will create ten load values less than the internal resistance and ten loads which are greater. The table should show four columns: The load resistance, load voltage, load current, and load power. It should also indicate the maximum power found. Once the table is complete, the program should draw a graph of the load power versus the load resistance. Here's one possible pseudo-code:

```
1. Give user directions
2. Ask user for source voltage and resistance (E and Rsource)
3. Initialize the maximum power (Pmax) to zero
4. Print out a heading for the table (Rload, Vload, I and Pload)
5. Start a loop which will initialize the load resistance (Rload) at one-tenth of the source
        resistance, increment the load resistance by one-tenth of the source resistance and finish
        when the load reaches twice the source resistance
6. Compute the current (I), the load voltage (Vload) and load power (Pload) from:
        I = E/(Rsource+Rload)
        Vload = E * Rload/(Rsource+Rload)
        Pload = I * Vload
7. Compare the load power just calculated to Pmax. If it's larger, reset Pmax to this value
8. Print out Rload, Vload, I and Pload
9. End of loop (from step five)
10. Print out Pmax.
11. Determine the proper scaling factor for the graph
12. Print out a title for the graph
13. Start a loop which will initialize the load resistance (Rload) at one-tenth of the source
        resistance, increment the load resistance by one-tenth of the source resistance and finish
        when the load reaches twice the source resistance (same as step five)
14. Compute the current (I), the load voltage (Vload) and load power (Pload) as in step six
15. Plot Pload using the scale factor from step 11
```

16. End of loop (from step 13)

A number of steps should be very familiar by now such as those giving the user directions, obtaining user data, and basic computations. Let's focus instead on the new items beginning with step three. We need to initialize `Pmax` to zero. The reason for this may not be obvious. This variable is going to hold the highest load power found so far. Each time through the loop the currently calculated power (`Pload`) will be compared to `Pmax`. If `Pload` turns out to be larger than `Pmax` then we have a new `Pmax` and we'll reset `Pmax` to this new `Pload`. If the newly calculated `Pload` is not larger than `Pmax`, we do nothing. By the time the loop is finished, every `Pload` will have been examined and `Pmax` will indeed be the maximum `Pload` found. This process can be thought of as a software version of a ratchet.

```
Pmax=0
```

Step four is just a print out for the heading. It might be advisable to separate each column with a tab or two (`\t`).

Step five starts the loop. We have a choice between a `for` and a `while`. First, the load resistance `Rload` needs to be initialized. This will be the first value used in the loop. The terminating value will be twice the source resistance. It's possible to do this in one statement using `for` and `range()`.

```
for Rload in range(Rsource/10, 2*Rsource, Rsource/10):
```

There is a practical problem here in that `range()` expects integers and we have floats. Although this issue might be resolved with the `int()` function, it won't work in every case (for example, if `Rsource` is very small, the increment might be zero once it's converted to an integer and this would prevent the loop from operating). In comparison, the `while` loop is always safe.

To use the `while` loop, the load resistance needs to be initialized manually:

```
Rload = .1*Rsource
```

Once this is done the loop termination point may be set:

```
while Rload <= 2.0*Rsource:
```

And finally, the load resistance needs to be manually incremented at the end of the loop (i.e., after all of the computations and printouts):

```
Rload = Rload + .1*Rsource
```

A shortcut way of saying this is:

```
Rload += .1*Rsource
```

Remember, everything which is inside the loop (i.e., the statement block) **must be indented** one level. Failure to do this will mess up the definition of the loop. Following the `while` or `for` is step six which deals with the various computations for `I`, `Vload` and `Pload`, and then step seven which involves `Pmax`, as discussed earlier:

```
if Pload > Pmax:
     Pmax = Pload
```

At this point, the values may be printed out. This is shown below with a single tab for spacing. Some form of rounding may be appropriate here.

```
print( Rload, "\t", Vload, "\t", I, "\t", Pload )
```

Using the `while` loop, the next line would be the increment for `Rload`, which completes step nine. Step 10 is a simple print out of `Pmax`. **Before continuing, note that the loop which starts at step 13 is basically the same as the loop we just examined.** The differences are that we don't need to deal with `Pmax` again and instead of printing out table values we'll print out the graph points. Consequently, we won't rehash this discussion but instead move directly to the issue of creating a graph.

Our technique for creating a graph will rely on simple text output. While definitely a step below modern high resolution graphics, the graph will be perfectly serviceable, even if not a work of art. Indeed, this is the same method that was used with the early text-only versions of the SPICE circuit simulator. To create this graph, the image will be rotated 90 degrees clockwise so that the normal vertical axis (`Pload`) will move to horizontal and increase from left to right while the normal horizontal axis (`Rload`) will be located vertically along the left edge and increasing from top to bottom. The idea is to turn `Pload` into either a point or solid bar that is displaced from left to right for increasing values. One possibility is to turn `Pload` into an integer and use it to iterate a character such as an asterisk:

```
    print( int(Pload) * "*" )
```

This will create a bar of asterisks `Pload` long. That is, if `Pload` is 34 we'll have printed out 34 asterisks in a horizontal row. (Note: It would be more accurate to add .5 to `Pload` before turning it into an integer as the `int()` function will truncate, not round. Adding .5 will effectively round the result.) To create a single dot, simply iterate a blank space for one less and then add the asterisk:

```
    print( int(Pload-1) * " " + "*" )
```

There are two problems with this, one minor and one major. The minor issue is that we might wish to include the `Rload` value (so we know which value produced the peak). This can be done with a small modification (shown using bar version):

```
    print( Rload, "\t\t", int(Pload) * "*" )
```

The larger problem is that `Pload` might be so large that it flows off screen (actually, it will wrap down to the next line which can be very confusing). For that matter, it might be so small (e.g., milliwatts) that it will appear as only a single line at 0 (there is no way to print partial spaces). Fortunately, both these ills may be cured by scaling all of the values. For very large powers we'll scale to make them smaller and for very small powers we'll scale to make them larger. While we could ask the user for a scaling factor, it would be better if we compute an optimal one.

Computing an optimal scaling factor turns out to be fairly easy. All we need to do divide the maximum graphing width by the maximum power (which we have already determined, namely `Pmax`). For example, if the width of the graphing area is 50 characters and `Pmax` is 0.1, then the scaling factor is 500. That is, every value of `Pload` will be multiplied by 500 so that the entire space will be filled. Conversely, if `Pmax` is 200 then the scaling factor will be .25 and every value will be reduced to one quarter of its size in order to fit the entire curve on the screen. 50 is a reasonable screen width for this exercise so the scaling factor is:

```
sf = 50.0 / Pmax
```

Note that this computation is performed *before* the second loop in step 11 prior to printing out a graph title and the start of the second loop. This single scaling factor is needed inside the loop for the plot so it doesn't make sense to compute it inside the loop itself. When it comes time to print the graph inside the second loop (step 16), we modify the `print` statement above to include the scale factor (shown using bar version and without rounding):

```
print( Rload, "\t\t", int(Pload*sf) * "*" )
```

Trials

Test the program with two trial circuits. The first should use a 10 volt source with a 1 Ohm resistance and the second should use a 20 volt source with a 3 k Ohm resistance.

Addendum – Pretty Print

One potential problem when printing tables is that the columns may not line up even if tabs are used for alignment. This is caused by the fact that some numbers will appear "nice", that is, contain few digits, and others won't. For example, one element in a column might be 25.0 while the item beneath it might be 24.83259074. The second value takes up more space and pushes the remaining items in that row over to the right thus causing a misalignment with the row above. If the values are limited to a fairly narrow range this problem can be alleviated through the judicious use of the `round()` function. This will not work, however, if the range of values is very wide: With large values there will still be a large number of digits and very small values may lose precision (in fact, "over rounding" may result in just zeroes being printed).

To get around this, Python uses *format specifiers*. These indicate the number of digits to be used on a print out (values will be rounded, not truncated). The most useful format for scientific numeric data is of the form `"%.3e"`. The `3` indicates the number of digits to be displayed while the `e` indicates that exponent form should be used (i.e., scientific notation). An `f` can be used instead of `e` which specifies ordinary (non scientific) form or a `g` may be used which indicates that the shorter of `e` or `f` will be used (e.g., a small value such as 12.3 will be printed as 12.3 instead of 1.23e+01 but a very large or small value will be printed using exponent form, such as 1.23e+09 instead of 1230000000.0). The specifier is placed before the variable to be printed, separated by a percent sign. For example, to print out the variable `x` to 4 places using the more compact of the ordinary and exponent formats:

```
print( “%.4g” % x )
```

Note that the second percent sign is **not** interpreted as the modulo operator in this case. Python “knows” this by context: It simply doesn’t make sense to perform a modulo operation between a string and a number.

Several specifiers may be used on one line. The following prints the variables `x`, `y` and `z` to 4 places each using exponent format and separates them with tabs:

```
print( “%.4e” % x, “\t %.4e” % y, “\t %.4e” % z )
```

The individual specifiers may be combined into one with the variables joined together with parentheses (this is called a *tuple*, a data type that will be examined in a later exercise):

```
print( “%.4e \t %.4e \t %.4e” % (x,y,z) )
```

Finally, be forewarned that the interpretation of “number of places” is slightly different between formats `e`, `f` and `g`.

9

Caerbannog

Objective

The objective of this exercise is to integrate knowledge of printing, computation, data input, looping, branching, menus and random number generation into a single program.

Introduction

The movie *Monty Python and the Holy Grail* includes a famous scene where King Arthur and his knights attempt to enter the Cave of Caerbannog. Much to their surprise, the entrance is effectively protected by a single rabbit who, it soon becomes apparent, has a "mean streak a mile wide". The ensuing chaos and calamity takes all by surprise, except for Tim the Enchanter. Our goal in this exercise is to create a computer game based on this scene.

A classic computer game is a bit different from the popular video game genre. The two most obvious differences are that computer games are text based rather than video based, and they tend not to be real-time, that is, you can consider your options and actions at a leisurely pace, not one determined by the game. In this regard, a computer game when compared to a video game is like playing chess versus competing in fencing. Consequently, for a computer game to be successful and interesting, it needs to have a good plot and interesting options leading to myriad outcomes.

Assignment

For this version, we shall use the following scenario: The player assumes the role of King Arthur. He or she is given a menu of weapons to use against the rabbit such as sword, dagger, Holy Hand Grenade of Antioch, and so forth. Each weapon will have a different level of effectiveness and reap a different reward (the ever popular "points"). For example, a dagger might only rarely defeat the rabbit; perhaps one in ten tries, while a sword defeats the rabbit two in ten tries and the Holy Hand Grenade works nine times out of ten. The dagger would have a much higher reward though, maybe 1000 points versus the sword's 400 points and the Holy Hand Grenade's meager 50 points. Without this point variation there will be little motivation to ever use anything other than the most effective weapon. The player continues to choose weapons and battles the rabbit (and the rabbit's offspring should the rabbit be killed) until the player loses a certain number of times (normally depicted as "lives" but this could be thought of as simple "loses"). For example, the player might start out with five lives and they will continue to play until they are defeated (killed) by the rabbit five times. At this point, they will have accumulated a certain number of "points", the more the better (not that the player could use them for anything, like say, a sandwich or a Monty Python DVD).

So, a basic pseudo code might look something like this:

```
0. Import the random module!
1. Initialize game variables such as lives=5 and score=0.
2. Give the player directions
3. Start the main battle loop which continues for as long as the player has lives left
4. Show the player the weapon menu
5. Ask the player for their choice of weapon
6. Compute whether or not the weapon worked this time (requires random number)
7. If the weapon worked, tell the player and increment score by the appropriate number of points
8. If the weapon failed, tell the player and subtract a life
9. Tell the player their current score and lives
10. End of loop from step 3
11. Give player final parting message
```

Think of this as a general starting point, the core of the program: Get this much working first. Once the core is up and running, consider expanding some of these ideas to make the game more interesting and entertaining to play. For example, the winner/loser messages can be randomized, a new life could be earned after accumulating a certain number of points, a weapon used too frequently could become "broken" and unavailable, etc. We shall look at a few of these momentarily but let's concentrate on the core first.

The first three steps should be very familiar by now. Take care to put in some effort on step two "Give the player directions". One element that makes computer games entertaining is good, detailed, (and in this case) humorous text. This would be an ideal place to use a triple quoted string. For example, you might consider using the “ASCII graphics” technique explained in the beginning of the text to create a giant image of a rabbit.

Step three is tailor made for a while loop. The game will continue for as long as the player has lives remaining:

```
while lives > 0:
```

Steps four and five should also be familiar. The weapons menu involves one or more `print` statements while the weapon choice will utilize a `input()` statement where the player enters their numeric choice (an integer) from the weapons menu. The more weapons on the menu, the more permutations of the game that exist, and the more entertaining it can be. Once finished, it is suggested that the menu contain at least 10 different weapons. Don’t restrict yourself to conventional weapons, either. Along with swords and arrows the choices might also include a shrubbery or a cow launched from a catapult.

Step six is the most interesting part. Each weapon needs to have two things associated with it: the odds of it working and the points awarded if it does indeed work. One possible way of handling this is to create a large `if/elif` structure to handle each weapon. First, a random number would be generated, let’s say an integer between 0 and 100. Then for each weapon, that number would be compared to the odds for that weapon to determine if it was a winner. An appropriate message would result along with adjustment of the scores and lives. For example, suppose there are ten weapons on the menu. Weapon one is a rock. It has a low chance of winning, maybe 10% of the time (i.e., 10 times out of 100), but it has a high payout, maybe 500 points. The second weapon is a lance. Its odds are better at 15% but it only pays out 200 points.

```
r = random.randrange(100)

if weapon == 1: # rock
     if r < 10: # 10% chance to win
          print( “You killed the rabbit!” )
          score = score + 500
     else:
          print( “You die an embarrassing death: rabbit food!” )
          lives = lives - 1
```

```
elif weapon == 2: # lance
     if r < 15: # 15% winning odds
          print( “The rabbit has been vanquished!” )
          score = score + 200
     else:
          print( “Fool! You end your days as an appetizer!” )
          lives = lives - 1
elif weapon == 3: # next weapon on menu…
```

This would continue for all of the weapons on the menu, each with its own odds, award, and messages. It is particularly useful if some of the losing messages are notably snarky in attitude. This makes it more enjoyable for someone who is just watching someone else play the game. Once this is completed, a status message is needed (step nine) which is merely a print out of the current score and lives remaining. Other items might be added as well, such as the number of attempts or a win/loss percentage (other variables would need to be created and then modified and updated within the loop).

Finally, a parting message (step 11) is added at the very end, once the loop is complete.

Enhancements

What will make or break the game are the enhancements. Again, it is recommended that you start with the core described above, perhaps with only two or three weapons, get that working, and then proceed to enhance the program. The first step would be to expand the armory (i.e., the choices of weapons). Other ideas include randomizing the win/lose messages so that a certain weapon does not always generate the same message. One way to do that is as follows for each weapon:

```
m = random.randrange(3)
if m == 0:
     print( “You won!” )
elif m == 1:
     print( “The rabbit is defeated!” )
else:
     print( “The rabbit was no match for you!” )
```

This technique will work but there is a more efficient method utilizing *tuples* (see the Tuples exercise for a general explanation; the random integer would serve as the index into a tuple of messages thereby removing the need for the `if/elif` structure for much more compact code).

Other ideas include awarding an extra life as well as points for a particularly unlikely weapon, removing points as well as a life if the player loses on a “sure thing” weapon (i.e., The Holy Hand Grenade of Antioch) or preventing the player from using a certain weapon too many times (a variable would need to be added to keep track of the number of times the weapon was used). A popular add-on is the “extra life”. At the end of the loop, the score could be compared to a preset value (say, 5000 points) and an extra life added if that level is exceeded. This is somewhat trickier that it appears to be at first. A more interesting variant is the idea of adding lives for every “so many” points, such as a new life every 5000 points. There are a couple of different ways to do this but this also is not as simple as it at first appears.

For the more adventurous, advanced scores could expand the weapons menu or even introduce a new level, such as battling the Black Knight at the bridge or trying to escape from the Castle Anthrax. Different levels can be thought of as new loops added *after* the first loop (not *within*).

Take your time with the enhancement and try to make something that you and your friends might find enjoyable.

10
Tuples

Objective

The objective of this exercise is to become familiar with tuples and have a little fun besides.

Introduction

In broad terms, Python has two kinds of variables: Those that are simple and contain single items, examples being floats and integers; and compound types that contain many instances of items. These are called sequences. A string is a type of sequence because it is made up of a collection of individual characters. Although strings are often treated as a single item, it is possible to extract individual characters or groups of characters from them (a process known as *slicing*). Another type of sequence is the *tuple* (think of this as a contraction of *multiple*). A basic tuple can contain a group of floats or ints. It could also contain a group of sequences such as strings or even other tuples. For example, consider a tuple that contains a series of voltage settings. It might be initialized like this:

```
V = (3.5, 2.0, 4.5, 6.0, 50.0, 10.0)
```

The sequence is defined with enclosing parentheses and the individual items are separated by commas. Square brackets are used to access any given item or slice with the initial item at location 0, as shown in the examples below:

```
print( V[1] )
print( V[4] )
```

These yield 2.0 and 50.0, respectively. A slice refers to a range of locations. Two values are specified separated by a colon: The first is the starting point while the second is the ending point (which is itself not included). It is also worth noting that a slice is itself a sequence and therefore will be printed with surrounding parentheses. For example:

```
print( V[1:4] )
```

This prints `(2.0, 4.5, 6.0)` If the start point is left off, it is assumed to be 0. Thus,

```
print( V[:4] )
```

yields `(3.5, 2.0, 4.5, 6.0).`

Finally, a third argument may be added which indicates an increment. For example:

```
print( V[0:4:2] )
```

yields `(3.5, 4.5).`

You can determine the number of items in a sequence by using the `len()` function:

```
print( len(V) )
```

This line will produce 6 (the number of items in the original declaration).

Sequences are extremely useful when combined with loops. A loop control variable can be used to access the items in the sequence in order. Consider the program snippet below:

```
for x in range(5):
     I = V[x] / R
     print( I )
```

This would compute and print five different values of I. The first would use V[0], the second would use V[1], and so forth. Note that if you wanted this computation performed on every item in the sequence, you could modify the first line to:

```
for x in range(len(V)):
```

Although the examples above use a tuple filled with floats, the same sort of manipulation can be performed on a simple string:

```
s = "abcdefg"
print( s[1] )
print( s[0:4] )
```

These yield `b` and `abcd`, respectively. As previously mentioned, a tuple may hold other sequences. This means that it's possible to have a tuple filled with strings:

```
c = ('black', 'brown', 'red', 'orange', 'yellow', 'green')
print( c[1:3] )
```

This bit of code will print `('brown', 'red')`. It is possible to "drill down" into these strings (sequences) by using a second pair of brackets:

```
print( c[1][2] )
print( c[1][0:2] )
```

These lines will print out `o` and `br`, respectively.

Assignment

Now for a little fun. This exercise will be a take off on Mad Libs, which is a word substitution game. The idea is to create randomized phrases based on a sentence template and lists of words to be substituted into the template. For example, the template might be "I am a *adjective noun*". The list of adjectives might be items such as *green*, *large*, and *puffy*. The list of nouns might include *man*, *warthog*, and *car*. An item from each list is chosen randomly and inserted into the template. We might wind up with phrases such as "I am a green warthog" or "I am a puffy man". The longer the sentence and the larger the lists, the greater the number of resulting permutations. Some of these permutations will be unexpectedly humorous.

This variation is commonly known as Curses (in this case, curses.py). It will generate wild, non-profane curses. While it is very easy to utter a common "four letter word" curse, they tend to be boring as we've all heard them before. Curses instead seeks to create novel phrases that will cause someone to stop for a moment and consider what has just been said.

One of the keys to good version of Curses is the curse template. You can come up with one by simply creating a unique curse and identifying nouns, adjectives, and so forth for substitution. Consider for example the following phrase an impulsive driver might yell after being cutoff in traffic: “You dirty little weasel! I hope your stupid car breaks!” Here’s a template:

“You adjective adjective noun! I hope your adjective noun verb!”

A list of adjectives might include obvious items such as “little” and “dirty”, but it can also include less obvious (and hopefully humorous in context) items such as “worm-infested”, “vomit-covered” or “odiferous”. Likewise, verbs such as “breaks” can also give way to short phrases such as “gyrates madly” or turns gangrenous”. Nouns, of course, would be given a similar treatment, perhaps with items such as “washing machine” or “younger sister”. So, after substitution, we might end up with a curse such as “You worm-infested, vomit-covered washing machine! I hope your younger sister turns gangrenous!” That phrase will surely make the offending driver do a double-take.

For further fun enhancement, we can make this version of curses create several curses instead of just one. Considering all of the above, we can create tuples to hold strings for each of the items to be substituted (i.e., a tuple of nouns, one of verbs, etc.). These can then be accessed with a simple random integer and printed out to form the sentences of the curse. The random integer generation and print out can be wrapped inside of a loop to generate several different curses. Here’s a possible pseudo code:

```
0. Don’t forget to import the random module (import random)
1. Define tuples for nouns, verbs, etc.
2. Give user directions
3. Ask user for number of curses desired
4. Start a loop for the number of curses
5. Generate a random number for each of the items to be substituted.
6. Print out the curse using the random numbers to access an element of the appropriate tuple
7. End of loop from step 4
```

Definition of the tuples in step one is the major bit of effort in this program. DO NOT resort to common four letter words. That completely ruins the effect of the curse and will result in a downgrading of the assignment. If need be, consult a thesaurus. Here is a short example of some adjectives:

```
adj = ('green', 'dirty', 'vomit-covered', 'self-loathing')
```

It is suggested that the tuples be at least 10 items in length (preferably twice that) because the longer the tuple, the greater the number of possibilities.

In step three, the number of curses will have to be an integer. If we call this `NumCurses`, we can set up a `for` loop in step four as follows:

```
for Curse in range( NumCurses ):
```

Inside of this loop we will have to generate a bunch of random numbers, one to index each slot that needs a substitution. `randrange()` is ideal for this as it creates integers that can then be used to access the tuples. If there are 15 items to choose from in the tuple, the following line would do the trick:

```
     i1 = random.randrange(0,15)
```

Of course, you could also use the `len()` function which can be handy if you're editing the tuples to add items:

```
     i1 = random.randrange(0,len(adj))
```

This would be repeated for as many indices as needed for your template. At this point, we're at step six, printing out the curse itself. This simply involves a mix of fixed strings and indexed tuples, something like the line below:

```
    print( "You", adj[i1], adj[i2], noun[i3]+"!" )
```

Trials

Run curses and generate at least a half-dozen unique curses. Make sure that the resulting curses are grammatically correct.

11

Functions and Files

Objective

There are three major objectives to this exercise: First, to become familiar with programmer-created functions, second, to utilize lists, and third, to explore methods to read and manipulate data contained in external files.

Introduction - Functions

Programmer defined functions are very useful as a means of compartmentalizing and reusing code. Once a bit of code has been created that performs a certain task, it can be reused over and over. This also makes the subsequent code more readable. If these functions prove to be widely applicable, they may be placed into custom modules so that they can be used conveniently in other programs. In Python, these functions must be defined (or imported if they're in a module) before they are called within the main program. They may or may not have arguments and they may or may not return a value (possibly more than one).

By comparison, think of a typical function such as `round()`. It takes two arguments: The variable you wish to round and the number of places to round to. It returns a single value, namely the rounded version of the original argument. For example:

```
x = round( y, 2 )
```

`y` and `2` are the arguments to the function and it returns a result which we then assign to the variable `x` (that is, `x` gets `y` rounded to `2` places). Suppose you wish to create a function that produced (returned) the parallel equivalent of two resistors. The function would look something like this, using product-sum rule:

```
def parallel( r1, r2 ):
     rp = r1*r2/(r1+r2)
     return rp
```

`def` indicates that this is the beginning of a function definition. Also note that the body of the function is indented, just like when using conditionals and loops. The name of the function is `parallel()`. It takes two arguments and returns a result based on the product-sum rule. Note that the variables `r1`, `r2` and `rp` are effectively place-holders or copies of the variables that are passed. It would be used like this:

```
Rx = parallel( Ra, Rb )
```

The values of `Ra` and `Rb` are copied over to `r1` and `r2`. Similarly, the resultant `rp` is copied to `Rx` when the function completes. This is known as *passing by value* (there is another technique, *passing by reference*, which we shall not explore). It is important to note that the variables `r1`, `r2` and `rp` are *local* to the function, that is, they are distinct from another variable that might be named `r1` or `r2` in another part of your program (or in another function). So any time two resistors need to be combined in parallel, this function may be called instead of performing the calculation in-line. It makes the code much more readable. While this function is very short, functions can be hundreds of lines long and contain conditionals, loops, and all manner of code. **Functions must be defined before they are called**, therefore they are usually found at the beginning of programs.

Introduction - lists

The second item of interest is the *list*. Lists are sequences like tuples, but unlike tuples, lists are mutable, that is, the elements within a list maybe changed. Tuples are best thought of as a collection of constants in comparison. Lists are **defined** using square brackets `[]` instead of using parentheses `()` like a tuple. Like tuples and strings, lists are accessed by using square brackets `[]`. Lists can be sliced just like tuples.

```
T = (12,43,17)      # This is a tuple definition
L = [12,43,17]      # This is a list definition
print( T[0] )       # print first element of tuple
print( L[0] )       # print first element of list
```

Note that it is not obvious if the item is a tuple or list when it is accessed. It is only obvious when it is defined. Also note that the entire contents of a list may be printed out as follows:

```
print( L )
```

The line above produces the entire list contents within brackets, each element separated by a coma:

```
[12,43,17]
```

Because lists are mutable, contents may be changed. Using the prior example:

```
L[1] = 4.3     # This is legal
T[1] = 4.3     # This is not legal
```

Because lists are mutable, there are a number of functions and *methods* that may be applied to them (think of methods as functions that apply to that particular object). One useful function is `len()`. This will let you determine how many items are in the list. Two useful methods are `sort()` and `append()`. `sort()` will rearrange the items in the list into ascending or descending order while `append()` will allow you to add new items to the list. Methods are thought of as belonging to the object in question so they are accessed in a slightly different manner from functions. In the example lines below, assume that the list `L` exists. The function call for `len()` should look familiar, however, note the new syntax for the two methods:

```
n = len(L)
L.append(x)
L.sort()
```

The first line determines how many elements are in the list `L` and assigns the result to the variable `n`. The second line adds a new element to the list `L` with the value of `x`. The third line sorts the list `L` from smallest to largest (to get the reverse ordering, use the optional `reverse` argument, as in `L.sort(reverse=True)` instead).

Introduction - Files

Often, it is not practical to hold data within a program or expect the user to re-enter the data each time the program is used. Imagine how impractical a word processor would be if documents could not be saved external to the program itself. This is where the concept of files comes in. Ultimately, there are only a handful of things programmer's normally need to do with files: Open them (i.e., gain access to them, often exclusively, through the use of an appropriate filename and/or path), read data from them, write data to them, move around in them (for example, to skip unneeded sections), and release or close them (so that other programs can gain access). While any program that deals with files will have to open and close them, whether or not they are read, written to and moved within will depend on what the program needs from the file. In the exercise that follows, only opening, reading, and closing will be used. Also, files can be generally of two types, binary and text. Binary files are potentially more powerful but text files are usually easier to use. In the exercise that follows we shall only look at text files. These files contain strings that may be read with any text editor while binary files are generally not readable with ordinary text editors.

In order to gain access to a file, it must first be opened:

```
fil = open( filename, mode )
```

`filename` is the literal disk file name such as `C:\myfolder\myfile.dat`. This is a string that can be hardcoded as a constant (rarely) or more commonly obtained from the user via a `input()` statement. `mode` is a string that describes how the file is to be accessed. `“r”` is used for reading and `“w”` may be used for writing. There are other modes as well. `fil` is a *file object* that is returned to you. It will be needed for subsequent read and write calls. Note that Python allows several files to be open at once, hence the need for file objects. So, a read mode access might look like this:

```
fn = input(“Please enter the file name: ”)
fil = open( fn, “r” )
```

Once the file object is obtained, data may be read from the file. Data can be read character by character or line by line. Line oriented files are easily read as follows:

```
str = fil.readline()
```

`readline()` is a file object method. It returns a string that corresponds to the next line of text in the file. Subsequent calls will read subsequent lines. For example, assume you open a basic text editor (such as Notepad) and create a text file that contains the following lines:

```
Information is not knowledge
Knowledge is not wisdom
Wisdom is not truth
3.14159
```

You then save this file as `“C:\myfile”`. Consider the following snippet of code:

```
fil = open(“C:\myfile”, “r” )
str1 = fil.readline()
str2 = fil.readline()
str3 = fil.readline()
str4 = fil.readline()
print( str4 )
print( str3 )
print( str2 )
print( str1 )
```

The result would look like:

```
3.14159
Wisdom is not truth
Knowledge is not wisdom
Information is not knowledge
```

Note that `str4` is a string. If you want to turn this into a number so that you can perform math on it, convert it with the `float()` function:

```
mypi = float(str4)
```

or, if you don’t need the string and just want the floating point value, do it in one line:

```
mypi = float( fil.readline() )
```

When you are done with the file, you need to close it. `close()` is another file object method:

```
fil.close()
```

Finally, files need to have some manner of internal organization. These can range from very sophisticated "chunk oriented" file structures to simple fixed layouts. Fixed layouts, where every line has a predetermined definition, are easy to access and use when the data are fairly well fixed with few, if any, options or variations. This is the form the assignment shall use.

Please note, in the interest of brevity, file checking code is not included in the code snippets above. It is possible, for example, for a file to fail to open or for a read operation to result in an error. Any commercially viable program should have these checks but we shall ignore them for now.

Assignment, Part One

The concept of quality control is very important. Suppose you work for a company that makes resistors. The resistors would have to be tested on a regular schedule to ensure that they are within appropriate manufacturing tolerances. A few hundred items might be pulled from a batch and measured. The sample would then be analyzed statistically and compared to the standard. If it meets the standard, all is good. If it doesn't meet the standard, the process needs to be investigated to find the source of the error(s). The assignment will mimic this. A group of measured resistors will be analyzed by the program. The results will include the number of resistors tested, the maximum and minimum values found, the arithmetic mean (i.e., average), the median (i.e., the value such that 50% of the sample is larger and 50% is smaller), the percentage of parts that fall outside the lower and upper tolerance limits, the percentage of parts that fall within the lower tolerance limit and the nominal, and the percentage of parts that fall within the nominal and the upper tolerance limit.

This is a fair amount of analysis and will require some testing to make sure that it works properly. The program will be split into two versions. The first will focus on the statistical analysis using a small number of resistors that will be entered manually. The second version will replace the manual entry with access to a file that contains a large number of resistor values (possibly thousands). While it's possible to do both at the start, splitting it will make it easier to find possible errors.

Let’s start with a pseudo code, we’ll use R for **r**esistor values and N for **n**umber of items:

```
1. Establish list for resistors, R, and four variables to keep track of the number of resistors found in the four tolerance zones (Ntoolow, Nlow, Nhigh, Ntoohigh)
2. Ask the user for the number of resistors being tested: N
3. Ask the user for the nominal value and tolerance: Rnom, Tol
4. Start a loop that will run for as many resistors as stated
5. Ask the user for a resistor value
6. Append the resistor to the list, R
7. End of step 4 loop
8. Determine the upper and lower tolerance limits: Rlowerlimit, Rupperlimit
9. Determine actual number of resistors entered
10. Sort the list
11. Determine the largest and smallest resistor values: Rmax, Rmin
12. Determine the mean and median values: Rmean, Rmedian
13. Start a loop that will cycle through the list, R
14. Compare the current item in the list to the lower and upper limits, and the nominal to determine which of the four zones the resistor falls within, and then increment the appropriate region variable
15. End of step 13 loop
16. Turn the zone totals into percentages
17. Print out the total number of parts tested, the largest and smallest resistors, the mean and median values, and the four percentages
```

Now for some Python code. Step one is basic initialization. The four zone totals are set to zero while the list is initialized as empty. Note the multiple assignment shortcut in the second line:

```
R=[]
Ntoolow = Nlow = Nhigh = Ntoohigh = 0.0
```

Steps two and three should be familiar `input()` statements with `int()` or `float()` as appropriate. Steps four through seven are perfect for a `for` loop:

```
for x in range(N):
     rn = float(input(“Enter the next resistor value: ”))
     R.append(rn)
```

Step 8 should be familiar from prior work, simply using `Rnom` and `Tol` to determine the upper and lower tolerance limits for the resistor, namely, `Rlowerlimit` and `Rupperlimit`. The total number of parts entered (step 9) may be found via the `len()` function, and then sorted (step 10) via the `sort()` method:

```
N = len(R)
R.sort()
```

Strictly speaking, N is already known and the first line is not needed, however, this will be needed for part two so we might as well add it here. Once the list has been sorted, the largest and smallest resistors will be located in the first and last positions of the list (step 11):

```
Rmin = R[0]
Rmax = R[N-1]
```

To determine the mean and median values in step 12 we'll call a couple of functions (yet to be written):

```
Rmean = find_mean(R)
Rmedian = find_median(R)
```

For steps 13 through 15, a `for` loop is again ideal, this time with a twist:

```
# As R is a list, rn will cycle through all of R's values
# automatically, from first to last
for rn in R:
     if rn < Rlowerlimit: # out of tolerance on low side
          Ntoolow += 1
     elif rn < Rnom: # must be between lower limit and nominal
          Nlow += 1
     elif rn <= Rupperlimit:
          Nhigh += 1
     else:
          Ntoohigh += 1
```

Note that by "stacking" the range tests with `elif`, we don't have to explicitly test for both "edges" of the middle zones. Step 16 is relatively straightforward. To turn any total into a percentage, just divide by the total number of resistors and multiply by 100 (note that the four `N` values were initialized as floats so there isn't an integer divide problem here). For example, to introduce new percentage variables:

```
Ptoolow = 100.0 * Ntoolow / N
```

The step 17 print out(s) merely need to be arranged to make the output report clear and logically presented.

Time to look at those two functions: They make writing the code easier in that we think "top down", that is, the big picture first. Now we have to drill down into the detail. The code for determining the mean is fairly easy, simply sum up all the values and divide by the total:

```
def find_mean( q ):
     tot = 0.0
     c = len(q)
     for x in range(c):
          tot += q[x]
     return tot/c
```

A slightly more compact version is:

```
def find_mean( q ):
     tot = 0.0
     for rx in q:
          tot += rx
     return tot/len(q)
```

Finding the median is a bit trickier. The following will only work with a list that has already been sorted (note steps 10 and 12 in the pseudo code). The trick is to find the middle-most occurrence in the list. If it's an odd sized list that's a single value but if it's an even sized list we'll need to average the middle two. In either case, we'll need to find the size of the list and cut that in two in order to get to the middle occurrence. Remember, when accessing any member of a sequence (such as our list), the index must be an integer. The `int()` function will be useful for this.

```
def find_median( q ):
     c = len(q)
     if c%2:   # remainder means c is odd
          return float(q[int(c/2)])
    else:      # it’s even, average the two middle values
          c /= 2
          return (q[int(c)]+q[int(c-1)])/2.0
```

Before looking at part two, make sure that the code above is operating correctly. It is suggested that you name it ResTolManual.py. Try the program with a couple small sets of resistors that you can test quickly by hand. Make sure you try both even and odd sized sets and sets with all legal resistor values, no legal values, and a mix of values. Only when you are satisfied that the program works should you head to part two.

Assignment, Part Two

In this section we shall modify the program from part one in order to utilize external data files. The data file will have a very simple structure. The first item will be the nominal value. The second line will contain the percent tolerance. After that, each line will contain a single measured resistor value. So, a file with three resistors, nominally 10k and 5 percent might look like:

```
10000.0
5.0
10234.1
9978.5
9863.2
```

Much of the earlier code will be reused. Generally, anything that required a `input()` statement will be replaced by a `readline()` method. In fact, there will be only one `input()` statement, and that’s to get the name of the data file from the user. So, instead of asking for the number of resistors, the nominal value and percent tolerance, instead we ask for the filename and then open the file in read mode:

```
fn = input("Enter name of resistor file to be processed: ")
fil = open( fn, "r" )
```

Once the file is opened we can read the first two lines which are the nominal value and percent tolerance:

```
Rnom = float( fil.readline() )
Tol = float( fil.readline() )
```

The main resistor input loop needs to be reorganized. We no longer know how many resistors are in the data file. While it's possible to ask the user to count them (a tedious waste of time), using a `while` loop can solve the problem. If we call `readline()` and there are no more lines to be read (i.e., we've already read the final resistor in the file), `readline()` will return a null (empty) string (`""` with no internal space). Think of the loop as "going until there are no more resistors". We establish a variable, `keepgoing`, as `True`, and reset it to `False` if we get back a null string (that is, nothing more to read). If we do get something, we turn it into a float and append it to the resistor list. Once the loop terminates we close the file:

```
keepgoing = True
while keepgoing:
     s = fil.readline()
     if s == "":
          keepgoing = False
     else:
          R.append( float(s) )

fil.close()
```

Trials

First, it is only necessary to hand in the completed program once part two is finished. Part one is an interim exercise necessary for initial testing only. It is suggested that this program be saved as ResTolFile.py.

Use the res3300 file from the course web page for processing. It will be most handy if this file is copied to the root directory of a USB drive so that it may be accessed via the form "X:\res3300.txt", where X is the USB drive letter. This file contains over 500 randomized resistor values.