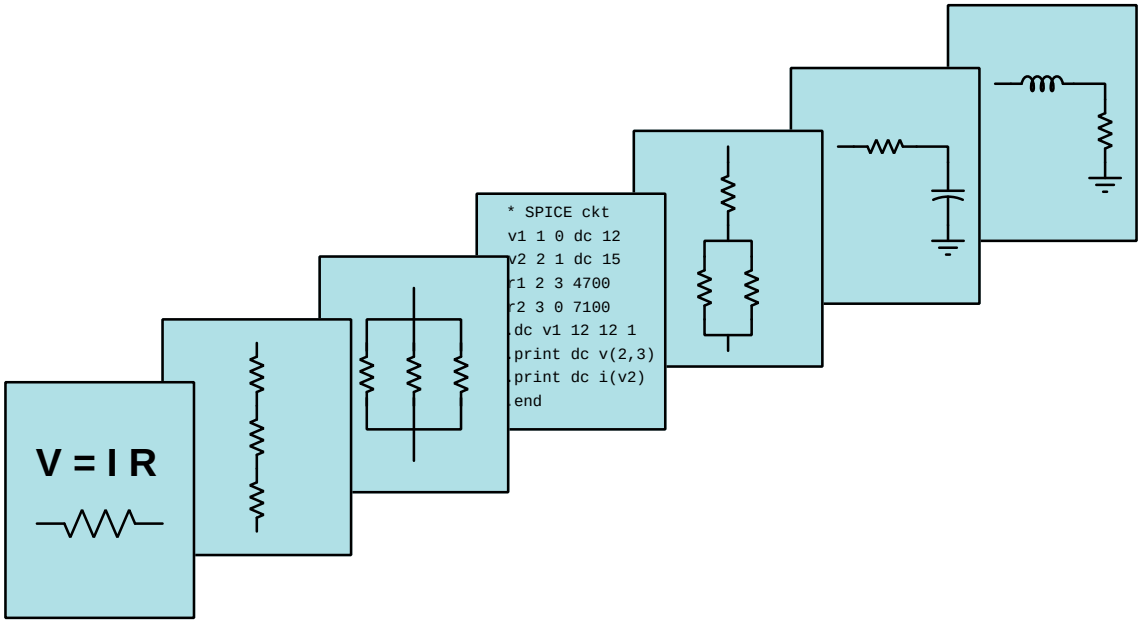
Modular Electronics Learning (ModEL) project



Introduction to Python Language Programming

© 2023-2025 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 2 July 2025

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Case Tutorial **3**
- 1.1 Example: running Python programs . . . 4
- 1.2 Example: simple uses of the print() function . . . 7
- 1.3 Example: using input() for numerical user entries . . . 8
- 1.4 Example: importing libraries . . . 9
- 1.5 Example: for() and while() loops . . . 10
- 1.6 Example: complex-number calculations . . . 14
- 1.7 Example: generating .csv output . . . 15
- 1.8 Example: Python functions . . . 16
- 1.9 Example: using lists . . . 17
- 1.10 Example: creating and using Python objects . . . 19
- 1.11 Example: . . . 22
- 1.12 Example: using Python to control a LabJack model U3 DAQ . . . 23

A Problem-Solving Strategies **33**

B Instructional philosophy **35**
- B.1 First principles of learning . . . 36
- B.2 Proven strategies for instructors . . . 37
- B.3 Proven strategies for students . . . 39
- B.4 Design of these learning modules . . . 40

C Tools used **43**

D Creative Commons License **47**

E References **55**

F Version history **57**

Index **57**

Chapter 1

Case Tutorial

The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

1.1 Example: running Python programs

First, you will need to visit `www.python.org` on the internet and download Python to your computer. After that, you will need to install that downloaded software. When installing Python, I *highly* recommend selecting the "Add python.exe to PATH" option, as this will make certain future uses of Python much easier!!

Next, you will need to write your Python "source" code using a text editor program (Notepad++ is my editor of choice when using Microsoft Windows) and save the resulting file to a filename ending with `.py` so that the operating system knows to associate that file with the Python interpreter software.

It is important to understand that Python is an *interpreted* programming language, unlike C and C++ which are *compiled* languages. Compiled languages require the use of a piece of software called a *compiler* which translates the source code into the native "machine language" of your computer. Another process called *linking* follows compilation, where the compiled "object code" gets linked to libraries of code pre-installed on your computer to make it a complete executable program. Only after compiling and linking will you be able to run that executable file output by the compiler. By contrast, an interpreted program merely needs to be read by a piece of software called an *interpreter* to run.

Generally speaking, compiled programming languages produce executable code that is smaller and runs faster, while interpreted programming languages are easier (especially for those new to programming) to administer.

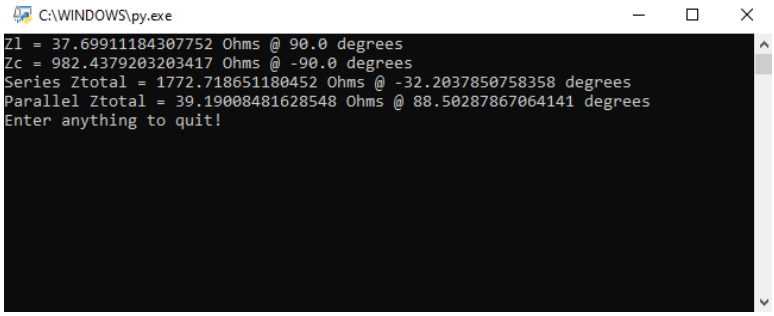
To run your Python source code, you have two easy options:

1. Simply double-click on your source code file's icon (it should appear as a Python logo if properly named with a `.py` extension) and the installed Python interpreter will run it
2. Use your computer's file navigator to locate the source code file, then right-click on that file's icon and choose to open it in a command-line environment (e.g. Terminal), then type `python` followed by your source code's filename and press Enter
3. Open up a command-line window (type `cmd`) into Microsoft Windows' search bar, then at the command line navigate to the proper directory (folder) where your Python source code file is located, then type `python` followed by your source code's filename and press Enter

Be aware that if you choose the first option you will need to have some provision in your source code to cause Python to hesitate prior to closing the window it opens upon execution. Otherwise, that window will close automatically as soon as the program is finished, which in most cases means you won't get an opportunity to see what it did!

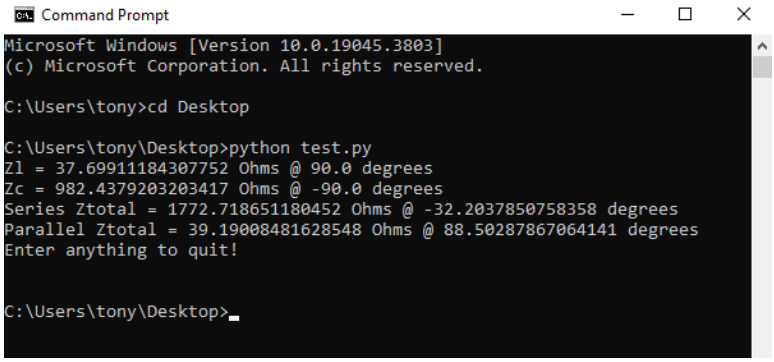
Shown below are some visuals for option number 1. First we see the icon as it should appear on a Microsoft Windows desktop, for a source code file named `test.py` which was written to compute series and parallel impedance for an inductor and a capacitor:

Once we double-click on this icon, an interpreter window opens up to display the results of the executed program:



Note the "Enter anything to quit!" line at the end of the output, created by a `print()` instruction followed immediately by an `input()` instruction awaiting user keyboard input before proceeding. The existence of that final `input()` instruction is why this window remains visible for us to view all that happened before it. Without that final *input()* instruction, the interpreter window would simply close on its own immediately following the lines of printed output, which means we would never have an opportunity to actually see that output – all we would see is a "flash" on the screen as the interpreter window opens up and then immediately closes!

Shown below is a visual for option number 2. Here we have started up a Windows command-line environment where we first navigate to the Desktop where our Python source code file `test.py` resides, using the command-line instruction `cd Desktop` to get there. After that we type `python test.py` to run our Python code:



This happens to be the exact same impedance-calculating code as the prior example (option number 1), and here that final `print()` instruction saying "Enter anything to quit" and that final `input()` instruction are both unnecessary. As before, the program waits for us to make a keyboard entry before finally terminating, but unlike the spontaneous window that opened up when we double-clicked the icon and immediately closed upon program termination, the Windows command-line environment remains to show us our program's output even after our program stops running.

So, there are two easy options for running Python programs on your personal computer. The first is extremely easy to run but requires an extra line or two of code added to your programs so that the computer will pause before closing out the window used to show what your program did. The second is a little clumsier to run but requires no extra lines of code for you to add to your source.

1.2 Example: simple uses of the print() function

Code listing:

```
x = 11
y = 3

print ("X =", x)
print ("Y =", y)
print ("")
print ("The sum of", x, "and", y, "is", x + y)
print ("The difference of", x, "and", y, "is", x - y)
print ("The product of", x, "and", y, "is", x * y)
print ("The quotient of", x, "and", y, "is", x / y)
print ("The quotient of", y, "and", x, "is", '{:.5}'.format(y / x))
print (x, "raised to the", y, "power is", x ** y)
print (x, "modulo", y, "is", x % y)
```

Output of program when executed using `python3`:

```
X = 11
Y = 3

The sum of 11 and 3 is 14
The difference of 11 and 3 is 8
The product of 11 and 3 is 33
The quotient of 11 and 3 is 3.6666666666666665
The quotient of 3 and 11 is 0.27273
11 raised to the 3 power is 1331
11 modulo 3 is 2
```

1.3 Example: using input() for numerical user entries

Code listing:

```
print("Enter a floating-point value for X: ", end='')
x = float(input ())
print("Enter a floating-point value for Y: ", end='')
y = float(input ())

print ("X =", x)
print ("Y =", y)
print ("")
print ("The sum of", x, "and", y, "is", x + y)
print ("The difference of", x, "and", y, "is", x - y)
print ("The product of", x, "and", y, "is", x * y)
print ("The quotient of", x, "and", y, "is", x / y)
print ("The quotient of", y, "and", x, "is", '{:.5}'.format(y / x))
print (x, "raised to the", y, "power is", x ** y)
print (x, "modulo", y, "is", x % y)
```

The `input()` instruction in Python casts every entry as a *string* of text characters. Since we wish `x` and `y` to be mathematical variables rather than text, we use the `float()` function to immediately convert each user-input string into floating-point values.

By default every Python `print()` instruction terminates with a linefeed character, so in order to maintain user input on the same line as the prompting text we use the `end=` parameter to tell both `print()` instructions to add nothing at all to the standard output after printing the line of text. Then, when the `input()` instruction executes, it does so on the same line.

Output of program when executed using `python3`:

```
Enter a floating-point value for X: 3
Enter a floating-point value for Y: 4
X = 3.0
Y = 4.0

The sum of 3.0 and 4.0 is 7.0
The difference of 3.0 and 4.0 is -1.0
The product of 3.0 and 4.0 is 12.0
The quotient of 3.0 and 4.0 is 0.75
The quotient of 4.0 and 3.0 is 1.3333
3.0 raised to the 4.0 power is 81.0
3.0 modulo 4.0 is 3.0
```

1.4 Example: importing libraries

Like most other programming languages, Python is *extensible* which means it's possible to write code in the Python language that may be conveniently included within other Python programs to extend capabilities for other programmers. A prime example of such a *code library* is the `math` library containing advanced arithmetic functions such as exponential, trigonometric, etc.

Consider the following example program displaying the sine of the angle 45 degrees:

Code listing:

```
from math import *

print("The sine of 45 degrees is", sin(radians(45)))
```

Output of program when executed using `python3`:

```
The sine of 45 degrees is 0.7071067811865475
```

The line of code `from math import *` instructs the Python interpreter to import all functions contained within the standard Python `math` library. This give us access to such functions as `sin()` and `radians()`. However, it is also possible to do this instead:

```
import math

print("The sine of 45 degrees is", math.sin(math.radians(45)))
```

This still enables us to use the `sin()` and `radians()`, but to call those functions we must preface each of their names with `math` (separated by a period symbol) so that Python knows in which library to find them. This latter method is obviously less convenient, given the necessity to write `math.sin()` instead of just `sin()`, but it carries with it the advantage of allowing us to write our own custom functions that will not conflict with the ones built into the `math` library. For example, it would be possible for us to write our own `sin()` function accepting an argument in units of degrees instead of radians, and still have `math.sin()` accept angles in radians.

1.5 Example: for() and while() loops

Code listing:

```
from math import *

for t in range (0,6):
  print("Time =", t, "seconds", end="")
  print("  ", end="")
  print("Decaying value =", 100 * exp(-t), "%")
```

This `for()` loop takes the variable `t` and increments it from 0 to 5 as an integer[1] variable. In many languages such as C, C++, Java, etc. a loop containing multiple instructions would group those instructions together to form a "block" of code by means of curly-brace symbols ({ and }). However, in the Python programming language we use indenting rather than brace symbols. Indented code is typical for most programming languages but only as an optional means to enhance code readability. In Python indenting is *mandatory* for creating blocks of code.

Any number of spaces or tabs may be used to declare one code block, but that number must then be consistent for that block. Here I chose to use *two* spaces of indentation for the three `print()` instructions within the `for()` loop.

Output of program when executed using `python3`:

```
Time = 0 seconds  Decaying value = 100.0 %
Time = 1 seconds  Decaying value = 36.787944117144235 %
Time = 2 seconds  Decaying value = 13.53352832366127 %
Time = 3 seconds  Decaying value = 4.978706836786395 %
Time = 4 seconds  Decaying value = 1.8315638888734178 %
Time = 5 seconds  Decaying value = 0.6737946999085467 %
```

[1] Python's `range()` function is compatible only with integer variables, and not with floating-point variables or any other types.

Here we see a different version of the program where `t` increments by two each iteration rather than by one:

Code listing:

```
from math import *

for t in range (0,7,2):
  print("Time =", t, "seconds", end="")
  print("  ", end="")
  print("Decaying value =", 100 * exp(-t), "%")
```

Output of program when executed using `python3`:

```
Time = 0 seconds  Decaying value = 100.0 %
Time = 2 seconds  Decaying value = 13.53352832366127 %
Time = 4 seconds  Decaying value = 1.8315638888734178 %
Time = 6 seconds  Decaying value = 0.24787521766663584 %
```

If we wish to treat `x` as a floating-point variable so we may increment it by non-whole-numbered values, we cannot use the `range()` function which is integer-only. Instead, we will use a `while()` loop allowing us to use any standard conditional we might choose to determine how many iterations the loop will complete before finishing:

Code listing:

```
from math import *

t = 0.0

while t <= 5.0:
  print("Time =", t, "seconds", end="")
  print("  ", end="")
  print("Decaying value =", 100 * exp(-t), "%")
  t = t + 0.5
```

Note how the initial value for `t` is specified as 0.0 rather than simply as 0. Within the `while()` instruction's conditional, we also specify a value with what appears to be a gratiutous decimal point (5.0 rather than just 5). These are our way of instructing Python that we wish `t` to be a floating-point value. Python is a *dynamically typed* language which means it assumes no particular type for its variables until run-time when those variables become initialized, operated on, etc. This stands in stark contrast to languages such as C and C++ which are *statically typed* and require variables to be declared as particular types before being used in the code.

Output of program when executed using `python3`:

```
Time = 0.0 seconds  Decaying value = 100.0 %
Time = 0.5 seconds  Decaying value = 60.653065971263345 %
Time = 1.0 seconds  Decaying value = 36.787944117144235 %
Time = 1.5 seconds  Decaying value = 22.313016014842983 %
Time = 2.0 seconds  Decaying value = 13.53352832366127 %
Time = 2.5 seconds  Decaying value = 8.20849986238988 %
Time = 3.0 seconds  Decaying value = 4.978706836786395 %
Time = 3.5 seconds  Decaying value = 3.0197383422318502 %
Time = 4.0 seconds  Decaying value = 1.8315638888734178 %
Time = 4.5 seconds  Decaying value = 1.1108996538242306 %
Time = 5.0 seconds  Decaying value = 0.6737946999085467 %
```

Here is another example of a Python program utilizing a `for()` loop:

Code listing:

```
print("Enter the number of paralleled resistors: ", end="")
n = int(input())

Rtotal = 0.0

for count in range (0,n):
  print("Resistor #", count + 1, "value in Ohms = ", end="")
  R = float(input())
  Rtotal = Rtotal + (1 / R)

print("These", n, "resistors in parallel make", 1 / Rtotal, "Ohms")
```

Output of program when executed using `python3`:

```
Enter the number of paralleled resistors: 4
Resistor # 1 value in Ohms = 1500
Resistor # 2 value in Ohms = 1200
Resistor # 3 value in Ohms = 680
Resistor # 4 value in Ohms = 2200
These 4 resistors in parallel make 291.9594067135051 Ohms
```

1.6 Example: complex-number calculations

Code listing:

```
from math import *
from cmath import *

f = 60.0 ; L = 100e-3 ; C = 2.7e-6 ; R = 1.5e3

Zl = complex(0,2*pi*f*L)
Zc = complex(0,-1/(2*pi*f*C))

print("Zl =", abs(Zl), "Ohms @", degrees(phase(Zl)), "degrees")
print("Zc =", abs(Zc), "Ohms @", degrees(phase(Zc)), "degrees")
print("Series Ztotal =", abs(R+Zl+Zc), "Ohms @", degrees(phase(R+Zl+Zc)),
      "degrees")
print("Parallel Ztotal =", abs(1/(1/R+1/Zl+1/Zc)), "Ohms @",
       degrees(phase(1/(1/R+1/Zl+1/Zc))), "degrees")
```

Note the importation of two code libraries in this Python program: `math` and `cmath`. The `math` library is necessary for the constant `pi` and for functions such as `degrees()`. The `cmath` library is necessary the `phase()` function which is unique to complex-number arithmetic.

Output of program when executed using `python3`:

```
Zl = 37.69911184307752 Ohms @ 90.0 degrees
Zc = 982.4379203203417 Ohms @ -90.0 degrees
Series Ztotal = 1772.718651180452 Ohms @ -32.2037850758358 degrees
Parallel Ztotal = 39.19008481628548 Ohms @ 88.50287867064141 degrees
```

Note some features of this code listing useful for fitting as much code as possible into as small a screen space (page space) as possible. First, note the use of semicolons to separate individual instructions on one line of code. Here, the semicolons perform the same purpose as they do in languages such as C or C++, namely to indicate the end of an instruction prior to the beginning of another. Python does not need semicolons when instructions are placed in their own distinct lines, but if you wish to cram multiple instructions into a single line on the page they are essential. Second, note how two of the `print()` instructions are broken into two lines of text each so they do not run off the printed page here.

1.7 Example: generating .csv output

Code listing:

```
from math import *
from cmath import *

f = 100.0 ; L = 100e-3 ; C = 2.7e-6 ; R = 1.5e3 ; fmax = 1e3

print("Freq , Zseries , Zparallel")

while f <= fmax:
  Zl = complex(0,2*pi*f*L)
  Zc = complex(0,-1/(2*pi*f*C))
  print(f, ",", abs(R+Zl+Zc), ",", abs(1/(1/R+1/Zl+1/Zc)))
  f = f + 100.0
```

Output of program when executed using `python3`:

```
Freq , Zseries , Zparallel
100.0 , 1589.761021005475 , 70.25109187407097
200.0 , 1509.4978890423038 , 216.76684379650098
300.0 , 1500.0212906692964 , 1427.105076177392
400.0 , 1503.598363961192 , 346.614606251871
500.0 , 1512.785716281622 , 187.23184708642194
600.0 , 1525.6801997713828 , 132.35134475497486
700.0 , 1541.577543267316 , 103.89939950541053
800.0 , 1560.1336353329743 , 86.19638901083295
900.0 , 1581.1359259645624 , 73.98527407843896
1000.0 , 1604.4266157514717 , 64.98781723832091
```

This comma-separated data may be copied from the interpreter window and pasted into a spreadsheet or other mathematical visualizing software to be graphed. Alternatively, if running the Python program from a command-line environment, you may use redirection to send that text output to a file of your own preference. Below we see the command-line text that could be used to invoke the Python interpreter to run the Python source-code file `test.py` and redirecting the output to a new file named `data.csv`:

```
python test.py > data.csv
```

1.8 Example: Python functions

Code listing:

```
import math

def sin(angle):
  return math.sin(math.radians(angle))

print("The sine of pi/4 radians is", math.sin(math.pi/4))
print("The sine of 45 degrees is", sin(45))
```

Our custom function is named `sin()`, and since we imported the `math` library by saying `import math` rather than saying `from math import *` it means we can create our own sine function that is distinct from the sine function within the `math` library. Our own sine function is simply `sin()` expecting an argument in the unit of degrees, whereas the `math` library's version of the sine function (`math.sin()`) naturally expects its argument in the unit of radians.

Output of program when executed using `python3`:

```
The sine of pi/4 radians is 0.7071067811865475
The sine of 45 degrees is 0.7071067811865475
```

Interestingly, when functions are used within Python, their definitions *must* precede their use. For example, the following version of the same program which locates the custom `sin()` function *after* its first use will not work:

```
import math

print("The sine of pi/4 radians is", math.sin(math.pi/4))
print("The sine of 45 degrees is", sin(45))

# This placement of the sin() function definition is TOO LATE!!

def sin(angle):
  return math.sin(math.radians(angle))
```

1.9 Example: using lists

Python supports multiple types of data arrays: *lists*, *dictionaries*, *tuples*, and *sets*. In this example we will see the use of *lists* which are the most similar of those four to arrays used in C and C++, used to randomly pick between three different types of components as well as display their values and units:

```
import random as rn
rn.seed()  # "Seeds" the random number generator with the system time value

types = ["resistor", "capacitor", "inductor"]
units = ["kiloOhm", "microFarad", "milliHenry"]

while 1:
  n = rn.randint(0,len(types) - 1)
  print(rn.randint(0,10000) * 0.01, end=' ')
  print(units[n], end=' ')
  print(types[n])
  input()  # Waits for Enter keystroke before looping again
```

In this program we define two lists, one named `types` and another named `units`. Within the `while` loop we generate a random integer value (`n`) which is then used as the *index* to select a particular element stored within each of the two lists. Note how the same index is used to select the component type and the respective unit of measurement, as those two things should go together. The value stored in `n` ranges from 0 upwards to one less than the length of the `types` list, as a three-element list requires its index value to be either 0, 1, or 2.

When run, we get a sequence of random values for components, together with matching component descriptions and units of measurement:

```
57.83 microFarad capacitor

52.27 milliHenry inductor

50.29 microFarad capacitor

44.93 milliHenry inductor

17.01 milliHenry inductor

45.56 kiloOhm resistor
```

1.10 Example: creating and using Python objects

Python is an *object-oriented* programming language like C++ (and unlike C), where we not only can create custom functions, but custom *objects* which may contain *properties* (variables and constants within an object) and associated *methods* (functions within an object).

Here is a very simple program where a *class* of objects is defined for resistors, each object within that class containing nominal and tolerance values as properties as well as high-value and low-value calculation functions as methods. Once this class is defined, objects may be created according to that class template:

Code listing:

```
class Resistor:
  def __init__(R, x, y):
    R.nominal = x
    R.tolerance = y

  def highval(R):
    return R.nominal + (R.nominal * (R.tolerance / 100.0))

  def lowval(R):
    return R.nominal - (R.nominal * (R.tolerance / 100.0))

r1 = Resistor(1.5e3, 5)
r2 = Resistor(1.0e3, 1)
r3 = Resistor(2.2e3, 2)

print("Resistor R1 has a nominal value of", r1.nominal, "Ohms")
print("but may be as low as", r1.lowval(), "Ohms")
print("or as high as", r1.highval(), "Ohms")
print(" ")

print("Resistor R2 has a nominal value of", r2.nominal, "Ohms")
print("but may be as low as", r2.lowval(), "Ohms")
print("or as high as", r2.highval(), "Ohms")
print(" ")

print("Resistor R3 has a nominal value of", r3.nominal, "Ohms")
print("but may be as low as", r3.lowval(), "Ohms")
print("or as high as", r3.highval(), "Ohms")
```

Output of program when executed using `python3`:

```
Resistor R1 has a nominal value of 1500.0 Ohms
but may be as low as 1425.0 Ohms
or as high as 1575.0 Ohms

Resistor R2 has a nominal value of 1000.0 Ohms
but may be as low as 990.0 Ohms
or as high as 1010.0 Ohms

Resistor R3 has a nominal value of 2200.0 Ohms
but may be as low as 2156.0 Ohms
or as high as 2244.0 Ohms
```

In this program, `r1`, `r2`, and `r3` are actually *objects* containing all the properties and methods defined within the `Resistor` class template. Those properties and methods are accessed for each of the three resistor objects by means of the period (`.`) delimiter, such that `r1.nominal` is a property of the `r1` object, and `r1.lowval()` is a method of the `r1` object.

Just like variables, objects in Python may be re-defined at any point within the program's execution. Consider this example program, which defines object `r1` but then later re-defines its properties:

Code listing:

```
class Resistor:
  def __init__(R, x, y):
    R.nominal = x
    R.tolerance = y

  def highval(R):
    return R.nominal + (R.nominal * (R.tolerance / 100.0))

  def lowval(R):
    return R.nominal - (R.nominal * (R.tolerance / 100.0))

r1 = Resistor(1.5e3, 5)

print("Resistor R1 has a nominal value of", r1.nominal, "Ohms")
print("but may be as low as", r1.lowval(), "Ohms")
print("or as high as", r1.highval(), "Ohms")
print(" ")

r1.nominal = 10e3
r1.tolerance = 0.1

print("Resistor R1 has a nominal value of", r1.nominal, "Ohms")
print("but may be as low as", r1.lowval(), "Ohms")
print("or as high as", r1.highval(), "Ohms")
```

Output of program when executed using `python3`:

```
Resistor R1 has a nominal value of 1500.0 Ohms
but may be as low as 1425.0 Ohms
or as high as 1575.0 Ohms

Resistor R1 has a nominal value of 10000.0 Ohms
but may be as low as 9990.0 Ohms
or as high as 10010.0 Ohms
```

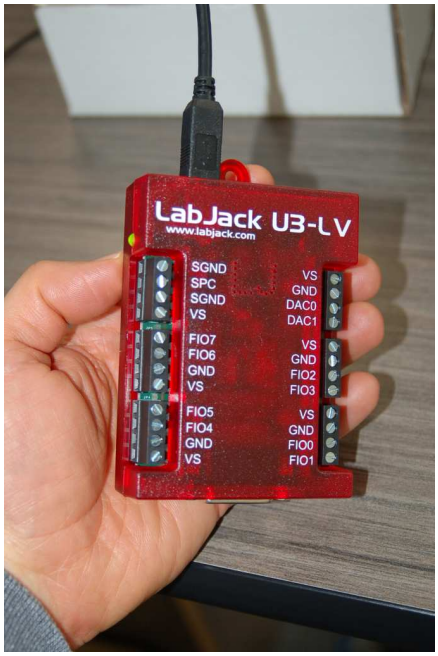
1.11 Example:

Code listing:

Output of program when executed using `python3`:

1.12 Example: using Python to control a LabJack model U3 DAQ

A popular manufacturer of low-cost data acquisition (DAQ) hardware is *LabJack*, with their model U3 DAQ being a good entry-level device. This particular model interfaces with a personal computer via a USB cable and is also powered by the computer's USB port 5 Volt DC source:



Several "Flexible I/O" ports (FIO0 through FIO7) are provided which may be configured for either discrete ("digital") or analog input usage. A Python package called `LabJackPython` provides a cross-platform Python programming language library with built-in functions and methods enabling low-level control of LabJack U3, U6, UE9, and U12 DAQ devices. A free software package provided by LabJack called `UD Library Installer` provides multiple applications[2] and software drivers for quick and easy configuration and control of LabJack devices with little or no programming required. The USB drivers contained in LabJack's `UD Library` are necessary in order to use the `LabJackPython` Python package, so you will need to perform *two* software installations: the `UD Library Installer` from LabJack, and the `LabJackPython` using the Python `pip` installation utility operated from the computer's command line.

Follow the instructions from LabJack on how to install this software on your computer before attempting to run the Python examples shown in this section. The Windows command-line instruction I used to install `LabJackPython` package on my computer for these demos is as follows:

```
py -m pip install LabJackPython
```

[2] Among these applications are `Kipling`, `LJControlPanel`, and `LJStreamUD`.

From a Python interpreter shell, you may enter the following commands to control the LED visible on the outside of the U3 unit:

```
>>> import u3
>>> d = u3.U3()
>>> d.toggleLED()
```

The first line (`import u3`) instructs Python to import the U3 device library previously installed on your personal computer. The next line constructs a new object named `d` tied to the first U3 device the computer finds plugged in to its USB port. Note that this simple discovery technique only works when you have *one* LabJack device plugged in at a time! The third and final instruction toggles the binary state of the DAQ's single external LED. Executing that line more than once continues to toggle the LED, first off, then on, then off again, etc. This is a simple and effective test to check that your computer is actually able to communicate with the DAQ via Python commands.

This next test uses the Python interpreter to manually read the analog voltage applied to input FIO0:

```
>>> import u3
>>> d = u3.U3()
>>> d.configIO(FIOAnalog = 0x01)
>>> d.getAIN(0)
```

The `configIO()` method[3] sets the functionality of the FIO inputs using a hexadecimal value whose eight bits relate to FIO0 through FIO7, respectively. In this case we are configuring FIO0 to be an analog input and the rest digital, since `0x01` is equal to a binary value of `0b 0000 0001`. If we had wished to configure the first three FIO inputs to be analog, we would have used the hexadecimal value `0x07` which is binary `0b 0000 0111`. It is important we specify the analog nature of any FIOs before we attempt to read their analog voltage values, otherwise the next instruction will return an error message!

The `getAIN(0)` method then reads the analog voltage applied to FIO0 and returns a floating-point value scaled in actual Volts according to the calibration data stored within the U3 DAQ's memory. Incidentally, if you wish to view this calibration data, you may do so using the following command:

```
>>> d.getCalibrationData()
```

[3] In object-oriented programming languages such as Python, a *method* is a function attached to an object. In this case, the object is `d` which is the particular U3 DAQ we're communicating with, and `configIO()` is one of the methods associated with that object.

An extremely simple Python program using the `toggleLED()` method is shown below, also using Python's `time` library to create a two-second pause in the `while` loop's execution:

```
import u3
import time
d = u3.U3()
while (1):
  d.toggleLED()
  time.sleep(2)
```

Writing this short program using a text editor program, then saving it to a filename ending with the extension `.py` makes it immediately recognizable to the computer's operating system as a Python source code file. Invoking that source file, either by double-clicking on its icon or by specifying it by name at a command-line interface (e.g. `py myfile.py`), will cause Python to execute these instructions.

A simple Python program reading the first three FIO analog voltage inputs (and re-reading them with every press of the "Enter" key on the controlling computer) is shown here ready to save to a file ending in `.py` and executed by double-clicking on the Windows icon for that file:

```
import u3
d = u3.U3()
d.configIO(FIOAnalog = 0x07) # Sets first three FIOs to analog mode

while (1):
  a = d.getAIN(0)  # Reads FIO0 as analog input and stores in "a"
  b = d.getAIN(1)  # Reads FIO0 as analog input and stores in "b"
  c = d.getAIN(2)  # Reads FIO0 as analog input and stores in "c"
  print("FIO0 =", a, "Volts")
  print("FIO1 =", b, "Volts")
  print("FIO2 =", c, "Volts")
  input()  # Pauses until user presses Enter
```

An example of this program's output when run is shown here, pressing the "Enter" key three times to get three readings of the analog inputs:

```
FIO0 = 0.438432256 Volts
FIO1 = 0.37528848000000004 Volts
FIO2 = 0.35026924800000003 Volts

FIO0 = 0.41698720000000006 Volts
FIO1 = 0.36397025600000005 Volts
FIO2 = 0.34490798400000006 Volts

FIO0 = 0.41758289600000004 Volts
FIO1 = 0.36397025600000005 Volts
FIO2 = 0.344312288 Volts
```

A more detailed and sophisticated way to read analog inputs on the model U3 DAQ makes use of the `getFeedback()` method. This method reads the raw analogto-digital converter (ADC) count value as an integer number.

```
import u3
d = u3.U3()
d.configIO(FIOAnalog = 0x07) # Sets first three FIOs to analog mode

while (1):
  a_raw, = d.getFeedback(u3.AIN(0))  # Reads FIO0 counts and stores in "a_raw"
  b_raw, = d.getFeedback(u3.AIN(1))  # Reads FIO1 counts and stores in "b_raw"
  c_raw, = d.getFeedback(u3.AIN(2))  # Reads FIO2 counts and stores in "c_raw"
  print("FIO0 =", a_raw, "counts")
  print("FIO1 =", b_raw, "counts")
  print("FIO2 =", c_raw, "counts")
  input()  # Pauses until user presses Enter
```

Note the comma symbols prior to the = assignment used to send the results of the `getFeedback()` method to the variables `a_raw`, `b_raw`, and `c_raw`. This is because the `getFeedback()` method returns a *list*[4] rather than a single variable. The list in this case happens to contain only one entry, and placing a comma before the "equals" assignment operator extracts that one entry's value as a regular integer value. If not for the commas, variables `a_raw`, `b_raw`, and `c_raw` would all become lists themselves.

An example of this program's output when run is shown here, pressing the "Enter" key three times to get three readings of the analog inputs:

```
FIO0 = 12048 counts
FIO1 = 10064 counts
FIO2 = 9520 counts

FIO0 = 11584 counts
FIO1 = 9808 counts
FIO2 = 9408 counts

FIO0 = 11568 counts
FIO1 = 9904 counts
FIO2 = 9440 counts
```

[4]Lists in Python are similar to arrays or structures in C or C++, namely collections of variables under a single name for easy access.

LabJackPython also supports discrete, or digital, I/O control for any FIO channels in digital mode rather than analog. For example, we may run the following commands manually from a Python interpreter to read the status of FIO6:

```
>>> import u3
>>> d = u3.U3()
>>> d.configIO(FIOAnalog = 0x01)
>>> d.getFeedback(u3.BitDirWrite(6,0))   # Sets the direction of FIO6 as input
>>> d.getFIOState(6)   # Reads the digital logic state of FIO6
```

In this example, the method `configIO(FIOAnalog = 0x01)` setting FIO0 to analog input mode is really just ensuring all the other FIO channels are set to digital. Next we have the `getFeedback(u3.BitDirWrite(6,0))` instruction setting (writing) the direction bit to a 0 value for FIO6 which configures that FIO to be an input rather than an output. The `getFIOState()` method reads the logical state of input FIO6 and displays it as either a 1 or a 0 depending on what the FIO6 input terminal is connected to.

Next, we will examine the Python instructions needed to configure FIO6 as a digital output rather than a digital input, and then setting its output value to be 1 and 0 in turn:

```
>>> import u3
>>> d = u3.U3()
>>> d.configIO(FIOAnalog = 0x01)
>>> d.getFeedback(u3.BitDirWrite(6,1))   # Sets the direction of FIO6 as output
>>> d.setFIOState(6, 1)   # Writes the digital logic state of FIO6 to a 1 value
>>> d.setFIOState(6, 0)   # Writes the digital logic state of FIO6 to a 0 value
```

Note that in all these examples it is not necessary to re-execute the `import u3` or `d = u3.U3()` or `d.configIO(FIOAnalog = 0x01)` instructions so long as they have already been executed during the same Python interpreter session. They are shown in these test examples assuming a fresh start of Python each time.

LabJack model U3 DAQs also contain 32-bit hardware counters which may be read via Python. These counters may be configured to read the digital (high/low) status of any FIO input terminal, incrementing by one every time the logic level transitions from a "high" state to a "low" state (i.e. a *negative-edge* clocked counter). The following Python program demonstrates how to read a counter inside of a model U3 DAQ:

```
import u3
d = u3.U3()

# Enables Counter 0 and sets FIO0-FIO3 to analog mode (0x0F = 0b 0000 1111)
d.configIO(EnableCounter0 = True, FIOAnalog = 0x0F)

while (1):
  count, = d.getFeedback(u3.Counter0(Reset = False))
  print("Counter value =", count)
  count, = d.getFeedback(u3.Counter0(Reset = True))
  input()  # Pauses until user presses Enter
```

Note how the first `getFeedback()` method does not reset the counter, but the second one does. This causes the counter's value to be re-set to zero just prior to the program waiting for the user's "Enter" keystroke. If no negative-edge pulses are detected before pressing "Enter", the next displayed counter value will be zero. However, if the input terminal receives some pulses prior to the "Enter" keystroke then the next value printed to the console after pressing "Enter" will register the number of pulses accumulated.

An example of this program's output when run is shown here, pressing the "Enter" key three times to get display four readings of the counter:

```
Counter value = 0

Counter value = 318

Counter value = 0

Counter value = 771
```

While it would be possible to write a Python program to repeatedly read the digital status of any FIO terminal and increment a Python variable to count how many times that digital signal has pulsed, this strategy would miss pulses whose frequency exceeded the Python program's loop execution rate. The beauty of a hardware counter is that the DAQ keeps its own count of pulses regardless of whether that counter value is read by a computer or not. Thus, when the Python program reads the counter's value, it is sure to read the true value of pulses captured in real time.

The model U3's counter is assigned to any FIO terminal other than FIO0 through FIO3 by means of an *offset* value which defaults to four (4). So, by default, FIO4 is the terminal driving the enabled counter. However, if we wish to assign the counter to sense logic states at a different terminal, all we have to do is change the offset value. The following program shows how this is done within the `configIO()` method, in this particular case setting that offset value to five (5) which makes FIO5 the terminal where any negative pulse edges will increment the counter:

```
import u3
d = u3.U3()

# Enables Counter 0 and sets FIO0-FIO3 to analog mode (0x0F = 0b 0000 1111)
d.configIO(EnableCounter0 = True, TimerCounterPinOffset = 5, FIOAnalog = 0x0F)

while (1):
  count, = d.getFeedback(u3.Counter0(Reset = False))
  print("Counter value =", count)
  count, = d.getFeedback(u3.Counter0(Reset = True))
  input()  # Pauses until user presses Enter
```

LabJack model U3 DAQs support up to *two* hardware counters, called counter 0 and counter 1. In the following program[5] we see both counters being read, with the offset value set to five (5) which means counter 0 is reading the digital status of FIO5 and counter 1 is reading the digital status of FIO6:

```
import u3
d = u3.U3()

# Enables Counters 0 & 1 and sets FIO0-FIO3 to analog mode (0x0F = 0b 0000 1111)
d.configIO(EnableCounter0 = True, EnableCounter1 = True,
           TimerCounterPinOffset = 5, FIOAnalog = 0x0F)

while (1):
  count0, = d.getFeedback(u3.Counter0(Reset = False))
  count1, = d.getFeedback(u3.Counter1(Reset = False))
  print("Counter 0 value =", count0, "  Counter 1 value =", count1)
  count0, = d.getFeedback(u3.Counter0(Reset = True))
  input()  # Pauses until user presses Enter
```

This program resets the value of counter 0 before every "Enter" keystroke from the user, but allows counter 1 to continue accumulating.

[5] Here the `configIO()` method's line is broken into two simply for readability on the printed page of this document.

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.

- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.

- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.

- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long technical career, education beats training every time.

- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.

- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.

- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.

- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and to dialogue with the industry representatives at those meetings.
- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.
- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.
- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for helping students build professional habits.
- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.
- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable**. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

B.4 Design of these learning modules

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[4] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[5] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[6] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

2 July 2025 – added another method to run a Python file, suggested by Jacob Stormes.

19 August 2024 – corrected a minor spelling error.

25 May 2024 – added another Case Tutorial section, this one showing a use of lists in a manner similar to C/C++ arrays.

12 February 2024 – modified the "Example: simple uses of the print() function" and "Example: using input() for numerical user entries" sections of the Case Tutorial chapter to include lines of code demonstrating how to format the number of decimals shown for a floating-point value.

1 January 2024 – added content to the Case Tutorial chapter.

31 December 2023 – document first created, with nothing but a Case Tutorial chapter.

Index

Adding quantities to a qualitative problem, 34
Annotating diagrams, 33

Checking for exceptions, 34
Checking your work, 34
Code, computer, 43
Compiled language, 4
Compiler, 4

DAQ, 23
Data acquisition unit (DAQ), 23
Dimensional analysis, 33

Edwards, Tim, 44

Graph values to solve a problem, 34

How to teach with these modules, 41
Hwang, Andrew D., 45

Identify given data, 33
Identify relevant principles, 33
Intermediate results, 33
Interpreted language, 4
Interpreter, 4
Inverted instruction, 41

Knuth, Donald, 44

LabJack, 23
Lamport, Leslie, 44
Limiting cases, 34
Linker, 4

Method, object, 19
Moolenaar, Bram, 43

Object, Python, 19
Open-source, 43

Problem-solving: annotate diagrams, 33
Problem-solving: check for exceptions, 34
Problem-solving: checking work, 34
Problem-solving: dimensional analysis, 33
Problem-solving: graph values, 34
Problem-solving: identify given data, 33
Problem-solving: identify relevant principles, 33
Problem-solving: interpret intermediate results, 33
Problem-solving: limiting cases, 34
Problem-solving: qualitative to quantitative, 34
Problem-solving: quantitative to qualitative, 34
Problem-solving: reductio ad absurdum, 34
Problem-solving: simplify the system, 33
Problem-solving: thought experiment, 33
Problem-solving: track units of measurement, 33
Problem-solving: visually represent the system, 33
Problem-solving: work in reverse, 34
Property, object, 19

Qualitatively approaching a quantitative problem, 34

Reductio ad absurdum, 34, 40, 41

Simplifying a system, 33
Socrates, 40
Socratic dialogue, 41
Stallman, Richard, 43

Thought experiment, 33
Torvalds, Linus, 43

Units of measurement, 33

Visualizing a system, 33

Work in reverse to solve a problem, 34
WYSIWYG, 43, 44