Episode 10
When Things Go Wrong

One of the most important skills we must acquire to be a competent programmer is correcting mistakes, or *debugging*. You will probably find that you spend more time debugging than writing the code in the first place. And in order to debug, you must test that your program is working correctly. Testing must be considered from the beginning, before you even start writing your own programs, and should be systematic.

As an example, suppose you have some data in a file and you wish to plot it. The file is VaGoldfinch.txt which you can download from this site. Unlike our previous example, this one has no year information, but we know from other sources that the time span is 1966-2015.

We can read it in with a NumPy function called loadtxt:

```
import numpy as np
import matplotlib.pyplot as plt
input_file="VaGoldfinch.txt"
obs=np.loadtxt(input_file,delimiter=',')
```

We need to set up the independent variable so we create an array of the years:

```
years=np.arange(1966,2015)
plt.plot(years,obs)
```

We run this and…it doesn't work. The size of the two arrays isn't the same. What did we forget? We forgot the Python rule about the upper bound of arange. We fix it with:

```
years=np.arange(1966,2016)
```

Now it works. However, the data are noisy and we'd like to smooth it. We'll use a very simple algorithm; each point will be replaced by the average of it and its two neighbors. We add a loop to do this computation:

```
for i in range(len(obs)):
    obs[i]=(obs[i+1]+obs[i]+obs[i-1])/3.
```

Another failure; an index exception. We forgot that Python always starts indices at 0, so that the range gives us integers from 0 to 49; but when `i=49`, `i+1` is `50`, which is out of bounds for our array. We can correct this error easily:

```
for i in range(len(obs)-1):
    obs[i]=(obs[i+1]+obs[i]+obs[i-1])/3.
```

This runs but the plot looks strange; the first value seems far too large. Now we need to start looking at the values so we can figure out what the program thinks it is supposed to be doing.

This is an important aspect of debugging! Just as you may not see typos in an essay you write no matter how many times you proofread it, you may have a difficult time finding your own bugs just by reading your code. So we add a print statement

```
for i in range(len(obs)-1):
    print i,obs[i],obs[i-1],obs[i+1]
    obs[i]=(obs[i+1]+obs[i]+obs[i-1])/3
```

This prints a lot of numbers to our console, so we must scroll back to find the first few values. We look and find the first value is 614, far too large compared to what it should be. But why? What happened? Looking back to the last numbers in the printout, we realize that we used the last value in an average for the first. Why did that happen? Looking more closely at our loop, we realize that when `i=0`, `i-1` is `-1`. That is a legal index in Python, but it's not what we want, it's the shortcut for the last element in the list. Now we change our loop to

```
for i in range(1,len(obs)-1):
    obs[i]=(obs[i+1]+obs[i]+obs[i-1])/3.
```

This time the plot looks correct. Neither the zeroth element nor the last element is changed at all, which is acceptable for our purposes here. If we had wished to provide some special handling for elements `0` and `len(obs)`, we would have had to add those outside our loop.

Let's try a more challenging example. Download the file DoW_buggy.py and the document Day_of_the_Week.pdf. The document describes an algorithm to find the day of the week for any date in the Gregorian calendar between the years 1400 and 2599. The algorithm is straightforward but has many steps, and also requires that we remind ourselves how to obtain a remainder from a division. In Python we use the modulo operator, represented by `%`; thus `7%3` is `1`. Our code hard-codes in days of the week rather than obtaining input from the user, so to change the date you will have to edit the source file. Our first date is 30 May 2016. The code computes this to be a Thursday when it should have been a Monday. We don't know where to start, so we'll start at the beginning.

Set apart either a cell (using `#%%` before and after the lines to be isolated) or selection, starting from above the *lookup table* for the month. (A cell is preferable but sometimes Mac OSX versions of Spyder don't handle it correctly.) From the Run menu, choose Run ell or Run selection or current line. Now go to the Variable Explorer tab and make sure we have values for day, month, and year. Those are set if you selected the appropriate lines, so we can run the cell or selection.

Watching the Variable Explorer pane as it runs shows that the variable D has the correct value, but M has the value 4. We are off by one on the month. That's because we number the months 1 to 12, but the corresponding indices of our lookup table run 0 to 11, so we must make a change to the indexing for the month. We change

```
M = months[month]
```

to

```
M = months[month-1]
```

Rerun the cell with that correction. Now M is correct. We remove the cell or selection and rerun the entire program. This time we get the right answer.

Are we then done? We have only tested the code for one date. We must test more thoroughly, and we must especially test "corner cases," situations where some unusual conditions may apply. For this code, we should test at least one date in each month, and we should test dates in years that are and are not leap years, taking special note of the rule for leap years in centuries. Century years must be divisible by 400, not 4, to be a leap year, so this is an example of a corner case.

Add the following code at the bottom of your script:

```
print "\n\nTesting first of each month"
day  = 1
month = 1
while month < 13:
    print "The day of the week is", DoW(day, month, year)
    month += 1
```

What do you get when you run it? Now we have to check with an independent calendar whether our results are correct. We are still in 2016 so we can use a calendar of that year.

We find that our code says that January 1, 2016 was a Saturday, when it was actually a Friday. We are also off by one day for February 1, but March, April, and May are correct. When we think about it, we realize that only days after February 29, 2016 will be affected by the leap year. We forgot to implement that part of the algorithm. Add the following code to fix this bug, right before it determines the value of C:

```
leap_year = (century_leap_year) or \
            (year%4==0 and year%100 > 0)
    if leap_year and month<3:
         L -= 1
```

Now it looks correct for the entire year…or does it? September 1, 2016 is reported to be a Friday rather than a Thursday. Another off-by-one error, it appears. Since it's dependent on the month, we need to look at the month computation. We double-check the values in our

lookup table months and discover a typo; we have a 6 rather than a 5, in the fourth position. Correcting that makes our days correct.

We need to test other days, however. Try at least the following:

```
February 14, 2000
February 14, 1900
July 4, 1971
July 4, 1776
```

It's easy to find day of the week calculators online, but test against two of them to make sure all the methods agree. You can try your own birth date as well.

Exceptions

You've seen several ways a code can fail. The code can run but fail to obtain correct results due to logic errors. The interpreter can stop due to some severe error, such as attempting to access an array element that does not exist. We may also try to perform a mathematically illegal operation, such as dividing by zero. When the interpreter detects an illegal condition, it throws an *exception*, outputs an error message, and stops.

You've seen the interpreter's message for at least some exceptions, such as an array-bounds error:

```
IndexError: index 50 is out of bounds for axis 0 with size 50
```

If we anticipate that something might go wrong, we can add our own exception-handling code. We do this with a *try/except* clause.

Go back to the goldfinch data code from the beginning of this episode. If we know we might have an index error, we can change the code in the main loop to

```
for i in range(1,len(obs)):
    try:
        obs[i]=(obs[i+1]+obs[i]+obs[i-1])/3.
    except IndexError:
        print "Out of bounds"
```

With this we have handled the exception, and the interpreter will not stop when an index error occurs. Therefore, it is safe to change the upper bound up back to len(obs). This will not protect us from an error that does not result in an illegal operation, however, so the lower bound of the range still has to be 1.

The `try` looks for the named exception. If it does not occur, the code continues as usual. If it does occur, the interpreter looks for the `except` block and executes the instructions there. Every `try` must have a corresponding `except`. You may use a tuple of exceptions (in parentheses) following `except`.

The interpreter includes a large number of built-in exceptions. You can write try blocks for specific named exceptions, or if you just want to handle any exception you may use a plain try:

```
try:
    output=open(output_file,'w')
except:
    print "Unable to open output file"
```

If the open fails for any reason, a message will be printed.

Below is a list of the most commonly-encountered named exceptions in Python:

```
EOFError
IOError
KeyboardInterrupt
IndexError
KeyError
NameError
NotImplementedError
TypeError
ValueError
ZeroDivisionError
```

You can add an `else` clause after the except for code to be executed if the try succeeds.

```
try:
    infile.open()
except:
    print "Unable to open file."
    exit
else:
    data=infile.readlines()
    infile.close()
```

One other clause you can add to a try block is finally. The code block following finally is run even if exceptions are thrown, including exceptions you did not catch. In the following example, we use finally because we should close the file regardless of whether any kind of exception occurred.

```
try:
    infile.open()
except:
    print "Unable to open file."
else:
    data=infile.readlines()
finally:
    infile.close()
```

When reading files, there is a particular form of exception handling you can use, the `with as` clause. This takes care of exceptions and also closes the file when the clause is terminated.

```
with open(datafile,'rb+') as infile:
    infile.readlines()
```

With/as is comparable to the try/except/else/finally construct shown above it, in fewer lines.

Once you are more proficient in Python there are additional actions you can take. You can raise an exception. You can write your own exceptions. A good introduction is at https://wiki.python.org/moin/HandlingExceptions.

Using Git and Github

As you develop your programming skills, you will want to learn to use version control. A version control system will track changes and do many other management tasks for you. If you need to undo a change, version control enables you to roll back to an older version. It is also very helpful when more than one programmer is working on a project. There are several such systems but git is very popular, and is the one we recommend.

To use git, first create an account at github.com. If you are new to git, we recommend a graphical user interface. Several are available, but we will use Github Desktop as our example.

Download Github Desktop from desktop.github.com and follow the procedure for your operating system to install it. For the Mac, unzip the file and drag the icon into the Applications folder. The application is cross-platform and looks very similar on Mac OSX and Windows, but our illustrations will be from Windows. When you first run Github Desktop, it will ask for your Github account information. You can then close the application.

If you wish to store your programs from this video series in Github, log in to Github. Click the green New Repository button and create a new repository of your own named Python_Programs. Clicking the Clone or Download button, clone your new repository to your local computer. Initially it will be empty. As you create new files that you wish to check in to your repository, log in to Github, select your Python_Programs repository, and upload files from

the local directory to the repository. You can drag and drop files or browse. Uploading the file adds it to your repository, but you must enter a comment and *commit* the file before it will be tracked by git. Then in Github Desktop, click Sync to download the new file to your local (cloned) repository. When you make changes to existing files in your local repository, the top bar will show the number of uncommitted changes. Click that tab and it will bring up the commit menu. You can then *push* your changes to Github through the menu.

Github Desktop is not an editor; you will create and change your files in an editor like Spyder's, then upload or commit/push them to your repository. The purpose of git is to keep track of changes. Storing your files on Github also provides a backup of your work. There are many advantages to learning version control; a useful introduction using Github Desktop is available at
http://programminghistorian.org/lessons/getting-started-with-github-desktop