GIS Exercise 5
QGIS Python Introduction
Feb. 19, 2016

Introduction

Python is a computer language which is used in both ArcGIS and in QGIS. It allows you to automate repetitive tasks, create new "plugins" or "toolboxes", and create more sophisticated processing than possible in the stand-alone programs. Python itself is a free, open-software language available from a number of sources. Copies are built into ArcGIS and QGIS and you should use the latter for this exercise. If you do want to install a stand-alone version of Python for other use, Windows versions are available from <https://www.python.org/download/windows/> and most Linux systems come with some option for installing Python if that hasn't been done already as part of system installation.

To run Python from QGIS from the main menu select **Plugins/Python Console** and a panel will appear in the lower right. You can place the cursor on the top edge of that panel and drag it upwards to make it larger. The top of the panel shows a record of the commands you issued and their output, while the bottom contains a single line for you to enter new commands.

For most parts I give you explicit instructions on what to type, but for one of the later parts, you will need to look through the following documentation to learn how to slightly modify the commands you've used:

<https://docs.python.org/2.7/tutorial/index.html>.

In that tutorial sections 1 and 2 are introductory material not directly applicable to the following. Sections 3 (An informal introduction to Python) and Section 4 (More Control Flow Tools) contain the material relevant to this exercise.

Part 1. Simple Python statements and programs

Python can handle integers, floating point numbers, strings, and a variety of other data types. In most cases it will convert automatically from one type to another when needed, although in some cases you have to prompt it to make the conversion.

Duplicate the following calculations, just to see how Python can do simple operations: In the following **`type the text formatted like this`** except note that a **`#`** symbol tells Python the rest of the line is a comment -- so you don't need to type that part.

Just once, at the beginning, type the following cryptic line which will be explained later.

```
from __future__ import print_function
```

There are two underscores before and after **`future`**. Also, the Python Console may try to be helpful and automatically insert the text **`import`** on its own. If it does, skip typing **`import`** yourself. That is, don't have two **`import`**s on the line.

```
x = 2.0           # Declare x to be a floating point variable

y = 1             # Declare y to be an integer variable

z = 2*(x + y)           # Z will be defined as a floating point variable,
                  #  and the value of y will be automatically converted
                  #  to f.p. as the addition is performed

z5 = z*5;         #  This creates a variable called z5.
print (z,z5)      #  Print the above results


aaa="The value of z5 is "     # aaa is a string

print (aaa + str(z5))         # str converts the z5 number to a string,
                              # and + concatenates the two strings
```

Part 2: String, List and Array manipulation

You can access parts of a string using indices, surrounded by square brackets. The first character is index 0. A peculiarity of Python, which in the end does have advantages, is that the substring runs up to, BUT NOT INCLUDING, the final index.

Type the following and check you get the range of characters expected.

```
ix=4; fx = 2; lx=7;           # Set 3 integer index variables
                              # The ; lets you put separate statements
                              # on a single line

aaa[ix]                       # A single character
aaa[fx:lx]                    # A range of characters
aaa[fx:]                      # A range, running to the end
aaa[:lx]                      # A range, from start to last1_index

                              # Note in the above if you just enter
                              # a variable, the Python Console
                              # assumes you want to print it.
                              # That WON'T happen inside the editor
                              # we'll use shortly.
```

You can use the same syntax for accessing elements of a list or (as we'll see later), an array:

```
pets = ["dog", "cat", "goldfish"];  # A list of 3 possible pets

print ("The " +  pets[1] + " ate the " + pets[2])

nval = [3.1416, 2.7183, 1.4142]     #  list of three useful numbers

print("The constant e = ", nval[1]) # Here we just told it to print two
                                    # items, which get separated by a comma

nval[1] = 6.2832                    # You can change elements of a list
print (nval)                        # It turns out you can't change elements
                                    # of a string, but you can create new
                                    # strings out of pieces of old strings.
```

We'll see you use this same syntax for accessing elements in arrays of numbers.

Part 3: Libraries

While Python has a set of basic capabilities built in, more advanced capabilities (for example most math functions) are in libraries which need to be imported. You can import all the functions of a library with:

```
from library_name import *    # Just an example -- don't actually do this
```

but most of the time you import just part of a library, for example

```
from numpy import sin         # Just an example -- don't actually do this
```

or import the library with a "prefix" name that must be used in front of all the library functions.

Actually do the following, to import the numpy (numerical python) library:

```
import numpy as np          # Actually import numpy, with the np prefix

n = range(10)               # List of [0, 1, .... 9]  up to but not
print(n)                    #   including 10

x = np.linspace(0., 2*np.pi, 10)   # Array with 10 numbers from 0 to 2pi
print(x)

y = np.sin(x)               # Sine of the above.  Sin is a function
print(y)                    # in the numpy library.
```

We'll see later that this **xxx.yyy** notation is part of the "Object Oriented" capabilities of Python. In OO nomenclature the **sin** and **linspace** "functions" are "methods" of the numpy object class. And **pi** is a (constant) "variable" of the numpy class. That nomenclature will become clearer after a later example.

Part 4: Loops, If statements, and other control statements

First I'll explain how to efficiently enter the following code into QGIS, and as part of that I need to explain how Python defines "blocks" of instructions, for example the lines of code which lie inside a loop.

1) **QGIS Python Editor:** For anything more than a few short lines of code, rather than entering it line-by-line in the Python Console, you'll want to first enter that code into the QGIS mini-editor so you can correct any mistakes and also generate the consistent indentation which is needed as described below. If you click on the editor icon  at the top of the Python Console then QGIS opens a small editor panel where you can enter and edit text. You can save it to a file, import it back from a file, or execute it directly using the  icon. Explore the editor commands by letting the cursor hover over the various icons at the top of the editor window. Also, by dragging on the right spot in the "title" portion of the Python Console, you can actually detach it from the main QGIS window and position it where it has more space and won't interfere with QGIS data display. You can also drag it back to its original position in the QGIS window.

2) **Python defines blocks of code using indentation:** Unlike some languages such as Basic where a `FOR` statement ends with a `NEXT` and an `IF` statement ends with a `THEN`, Python just uses indentation to define the blocks. You need to consistently indent by the same number of spaces to mark off a block of lines after the `IF` or `FOR`. Unfortunately the Python Console window within QGIS isn't very smart about indenting. You need to manually add consistent spaces to the beginning of each line yourself. When you finally enter a blank line, that signals to QGIS that your are done with the indented block. QGIS does at least gives you a hint about what it expects, as it shows >>> when it thinks you are NOT within a block, and ... when you are in the middle of a block. However for anything more than a few lines of block code, use the editor. The editor is smart enough to usually guess right about what indentation you want, and if it guesses wrong you can add a tab or backspace to correct it.

In the following examples you will probably want to enter code into the editor then just use the  icon to run it.

The general structure of a for loop is as follows, where indentation controls which statements are executed inside the for loop:

```
for i in list :
   do_this
statement_after_for_loop
```

Run the following:

```
for i in n:
    print( i, x[i], y[i])     # Indent marks the loop code
print("Done with the loop")   # No indent signals the end of the loop
```

Note the above example with all three lines only works within the editor. Trying to enter the final non-indented line in the Console confuses it. If you really want to enter it in the console, leave off the last line. After the **`print( i ...)`** just enter a blank line to signal the Python Console you are done with the indented section. But that means you can't enter complex blocks in the Console with different levels of indentation. Use the editor for that.

"If" statements are similar:

```
if  condition_to_test :
        do_this_if_true:
else :
        do_this_if_false
statement_following_if_else
```

Run the following:

```
for i in n:
    if ( i == 5) :                        # == means "is equal"
        print("Skipping number 5")
        print(" to show I can.")
    else :
        print( i, x[i], y[i])
print("Done with the loop")
```

Part 5: Functions

We define functions or subroutines (there really isn't a difference in Python) with the following:

```
def test_1( arg1 ):
      print("Two times ", arg1, "equals ", 2*arg1)

                  # Note, enter the above into the editor then run it
                  # to define the function.  Then go back to the regular
                  # Python Console to actually test the function below.
                  # In the editor just typing a variable (or a function)
                  # does NOT cause it to print the output.
                  # You could FORCE that printing by replacing the
                  # following line by
                  # print(test_1(5))  .


test_1(5)         # Run it for arg1 = 5     Do this in the Console
```

If instead of printing the result we wanted to return the value , for example to assign it to some other variable,
we would use

```
def test_2( arg1 ):           # Enter these two lines in the editor
      return (2 * arg1)       # then click run to define the function.


tvalue = test_2(1.234)        # Enter this back in the console.
print(tvalue)

test_2(x)                     # It can also work on arrays


test_2("Bye")                 # See what happens with a string
```

Part 5: Plotting

Matplotlib is the library most commonly used for plotting. The following is just a very simple example. Try executing the following commands. The first line imports the core plotting functions, with the alias **plt** which we need to add before each command.

```
import matplotlib.pyplot as plt
plt.plot(x, y)                # Plot the x, sin(x) curve we created earlier
plt.show()                    # This makes the output window appear
plt.plot(x,y,'ko')            # Replot using small circle symbols (the "o")
                              # in the color black (the "k")
                              # The new symbols won't show yet.
plt.xlabel("Radians")         # Add some labels
plt.ylabel("Signal")
plt.title("Test plot")
plt.ylim(-1.2,1.2)            # Adjust the y range
plt.draw()                    # Refresh the displayed plot
```

DO NOT EXECUTE THE FOLLOWING AS IT CAUSES ESB1006 QGIS TO CRASH.
IT WORKS OK ON LINUX AND MIGHT WORK ON OTHER WINDOWS INSTALLATIONS.

```
plt.savefig("testplot.png")   # Save the plot as a file in some default dir.
```

Experiment with the tools in the plot window which let you zoom, pan, restore, and save the plot. Try experimenting with various plots, and with different colors such as "r" or "g" or "b", and different symbols such as "x" or "+".

There is extensive matplotlib documentation at **<http://matplotlib.org/>**. You can control almost all the characteristics of the plot, and can create multiple plots per page and complex three dimensional plots and images. A gallery of plots and example code is located at **<http://matplotlib.org/gallery.html>**.

Part 7: Your own function with loops and if statements

Use the commands above to create a function **`test_3(mx)`** which will loop through all numbers between 10 and mx and do the following: On each line it should print the number, then the square of that number, EXCEPT, for i=14 and i=16 where it should print the cube. My definition of the function takes just 6 lines of code. See the Python documentation reference above on how to modify **`range`** to run from some beginning value other than 1, in this case 10. Also look through the documentation to test for **`(i==a or i==b)`**, that is, a condition which is true if either is true. You could also create a "nested" **`if else`** structure if you wanted to avoid that "**`or`**".

To turn in:

When you have it working save the program using the built-in editor's "save file" function.

Run it for ***`test_3(20)`*** *then save the output. A simple way to save that output is to highlight it in the console with the mouse, then right click, select copy (to copy it to the clipboard), then paste it into a text file.*

Part 8: QGIS Python Programming
(and an introduction to Object Oriented Programs)

Open the Suquamish map from exercise 2. That data file is still on the web site. We'll write a short Python program to extract and print information about the ages in the GeochronPoints layer.

In the following you will see more of an introduction to object oriented programming. At this stage examples are probably more useful than a long abstract description. Very briefly, QGIS contains a hierarchy of software "objects" many of which correspond closely to recognizable parts of the collected data. For example there are "layer" objects which in turn contain "feature" objects which in turn contain "geometry" objects, etc. For each level or "class" of objects there are functions (called "methods" in OO nomenclature) to operate on the information contained in that object. For example a layer object has methods which control how the layer data is displayed, as well as methods which return information about the features the layer contains. A feature object has methods which return information about its location and the contents of its row in the attribute table.

For a given object **`obj`** and a given method **`mthd`** you call that method by typing **`obj.mthd()`** . Sometimes you need to put arguments in the **`()`** but even if none are needed you have to type the **`()`** to tell the system you are calling a method. Since **`obj1`** might have a **`mthd1()`** which returns an **`obj2`** which in turn has a **`mthd2()`** and so on, you sometimes see a chain of methods calls like: **`obj1.mthd1().mthd2().mthd3()`**. The following examples should make this more concrete.

To start we need to obtain the object which represents the Geochron Point layer. First make that the active layer by selecting it in the QGIS Layers Panel. (At the end of this exercise open its attribute table to compare to what we get below.)

Next, in the Python Console, to acquire the layer "object", type the following line.

```
layer = iface.activeLayer()  # Sets layer to be an "object" variable
                             # which represents the current active layer.
                             # Note iface is the "object" which represents
                             # the interface to QGIS and activeLayer()
                             # is it's method which returns
                             # the current active layer.


field_list = layer.fields()  # Get a list of the fields, that is,
                             #   the attribute table columns


for fld in field_list :      # Print a list of the field names
    print( fld.name() )

features = list( layer.getFeatures() )    # Get a list of all 6 features
                                  # (geochron points) in this layer.
                                  # I'll explain later why I used the
                                  # list() function to force this to be a
                                  # true list.  GetFeatures() is a method
                                  # which actually returned something more
                                  # complicated called an "iterator".


f = features[0]              # Set the variable f to the first feature.
                             # Python lets us retrieve the values from
                             # the attribute table using the field names
                             # as the index.
sname = f["GeochronPo"]      # Get the sample name
print(sname)                 # Print it


for f in features:                  # Collect and print the interesting info.
    sname   = f["GeochronPo"]       # for all the features.
    sage    = f["Age"]
    saunits = f["AgeUnits"]
    snotes  = f["Notes"]
    print( sname, sage, saunits, snotes)
```

They give actual ages in thousands of years for two samples, and give lower age limits (in the notes) for the other four samples.

The above print just used default formatting. We'll see later how to clean up formatting. If you replace the last line above with the following you'll have nicer looking output:

```
print("%6s %6.1f %5s %7s" % (sname, sage, saunits, snotes)).
```

To turn in:

Use the console right-click copy function to once again save in a text file the output of the above program.

Other commands could have been used to change attribute values, to modify the display based on the attributes, or do more complicated processing based on geometry information. For example we could have extracted the x,y position of the point (in the layer's CRS) using

```
x,y = f.geometry().asPoint()
```

Summary of what to turn in

By next Friday (Feb. 26) turn in a text file which contains:

1) The program you used to produce the list of squares and cubes.
2) The output of that program showing the cubes and squares..
3) The output of the above program listing the Geochron point information.