**EGM722 - Week 1 Practical: Intro to Python**

**Overview**

As we discussed in this week's lecture, programming is a powerful tool that allows us to do complicated calculations and analysis, visualize data, and automate workflows to ensure consistency, accuracy, and reproducability in our research. In this practical, you will learn and practice basic commands in python and import modules to complete more complicated tasks. In the weeks to come, you will learn to work with different GIS datasets such as vector (e.g., shapefiles) or raster files using python libraries.

**Objectives**

* Learn and gain experience with some of the basic elements of python and programming
* Learn how to use the python command line interface
* Practice planning out a script

**Data provided**

In the data\_files folder, you should have the following:

* GPSPoints.txt
* Glaciers.shp (and associated files)

**1. The python interpreter**

Before we get started, it's important that we check which verison of python we're running. To do this, we can use the **sys** module. If you run the following cell, you should see something like this print out:

3.9.16 | packaged by conda-forge | (main, Feb 1 2023, 21:39:03) [GCC 11.3.0]

If you see a version of python other than 3.x, we'll need to switch your working environment.

In [1]:

**import** sys

print(sys**.**version)

3.11.0 | packaged by conda-forge | (main, Oct 25 2022, 06:12:32) [MSC v.1929 64 bit (AMD64)]

Now that we know we're using the right version of python, run the following code snippet:

In [2]:

print("Hello, World!")

Hello, World!

The print() **function** allows us to print messages and information to the screen, or to a file (more on this later), but it doesn't allow us to save the messages that we display.

How can we edit the code above to store our message as a variable? In the cell below, type a line of code that will define a **variable**, myString, that stores our message (Hello, World!). Then, print the message to the screen using the **variable** you've just defined.

In [3]:

**help(print)**

Help on built-in function print in module builtins:

print(\*args, sep=' ', end='\n', file=None, flush=False)

Prints the values to a stream, or to sys.stdout by default.

sep

string inserted between values, default a space.

end

string appended after the last value, default a newline.

file

a file-like object (stream); defaults to the current sys.stdout.

flush

whether to forcibly flush the stream.

Often, you will want to know how to use a particular function. To get help, we can use the built-in help() function. For example, to get more information on how to use the print() function, we could type the following at the prompt:

>>> help**(**print**)**

Help on built-in **function** print **in** module builtins:

Print**(…)**

file: a file-like object **(**stream**)**; defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Go ahead and try this now:

In [4]:

**help(print)**

Help on built-in function print in module builtins:

print(\*args, sep=' ', end='\n', file=None, flush=False)

Prints the values to a stream, or to sys.stdout by default.

sep

string inserted between values, default a space.

end

string appended after the last value, default a newline.

file

a file-like object (stream); defaults to the current sys.stdout.

flush

whether to forcibly flush the stream.

In jupyter/ipython, you can also use the ? operator, which will open the **docstring** in a new panel at the bottom of the window:

In [194]:

print**?**

This is a lot of information for now, but if you want to know how to use a particular function, method, or class, you can find that help here. A warning, however: some python packages are better-documented than others (which is why we should always provide thorough documentation when writing our own code, right?)

**2. Variables**

We have already seen one example of a **variable**, myString, above. Remember that in programming, a **variable** is name that represents or refers to a value. **Variables** store temporary information that can be manipulated or changed as we type commands or run scripts.

In practice, variables can refer to almost anything. As the chosen name suggests, myString is a **string**, or text. Variables can also be **int\_\_egers, \_\_float\_\_ing point numbers (floats or decimal numbers), \_\_list\_\_s, \_\_tuple**, \_\_dict\_\_ionaries, entire files, and many more possibilities.

As we covered in this week's lecture, in python, variable names can consist of letters, digits, or underscores, but they **cannot** begin with a digit. If you try to name a **variable** using an illegal name, you will get a SyntaxError:

>>> 3var = "this won't work"

Cell In[5], line 1

3var = 'this won't work'

^

SyntaxError: invalid syntax

To confirm this, try it for yourself below:

In [195]:

3var **=** "this won't work"

**Cell In[195], line 1**

**3var = "this won't work"**

**^**

**SyntaxError:** invalid decimal literal

**3. Numeric operations**

A large part of what we will use python for is the manipulation of numeric data. Thus, it is a good idea for us to understand how python treats numeric data. One by one, type the following expressions into the cell below. Before running the cell, be sure to think about what you expect the result to be. Does the result you see match your expectation? Why or why not?

* 19 + 32 # the + operator does addition
* 19 - 87 # the - operator does subtraction
* 19 \* 12 # the \* operator does multiplication
* 1 / 3 # the / operator does division
* 1 // 3 # the // operator does floor division
* 10 % 3 # the % operator does modular arithmetic
* 2 ^ 4*#Sets each bit to 1 if only one of two bits is 1 (2^4)*
* 2 \*\* 4 # the \*\* operator does exponentiation

Can you explain what the second-to-last operator (the ^ symbol) is doing?

In [196]:

2**/**4

**Out[196]:**

**0.5**

**4. String variables and operations**

We have already worked with one example of a **str**ing variable, myString.

As noted in the lecture, we can easily access parts of a string by using the desired index inside square brackets [ ]. Remember that the index has to be an **int**eger value:

In [6]:

myString[0]

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

Cell **In[6], line 1**

**----> 1** myString[0]

**NameError**: name 'myString' is not defined

If we use a **float**ing point value, it raises a TypeError:

In [204]:

myString[0.0] *# slice indices have to be integers, not floats!*

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

Cell **In[204], line 2**

1 myString = [1,2,3,4,5,6,7]

**----> 2** myString[0.0] # slice indices have to be integers, not floats!

**TypeError**: list indices must be integers or slices, not float

As an additional example, to get the 3rd character in myString, we would type myString[2] at the prompt and press ENTER:

In [203]:

myString **=** [1,2,3,4,5,6,7]

myString[2] *# get the 3rd character in myString*

Out[203]:

**3**

Why does this give us the third character from myString? Well, remember that the first element of a **str** (or any sequence; more on that later) has an index of 0.

To access the last element of a **str** (or a sequence), we could count up all of the elements of the **str** and subtract one (remember that we start counting at 0, not 1), but python gives us an easier way: **negative indexing**.

Thus, to get the last element of myString, we can type myString[-1]. To get the second-to-last element, we could type myString[-2], and so on.

In [202]:

myString **=** [1,2,3,4,5,6,7]

myString[**-**1] *# get the last character in myString*

Out[202]:

**7**

If we want to access more than one element of the string, we can use multiple indices, with the basic form of:

**>>>** sliced **=** myString[first:last]

This will select the letters of the string starting at index first up to, **but not including**, last.

This is also called **slicing**. What does the command myString[1:5] return?

In [201]:

myString **=** [1,2,3,4,5,6,7]

myString[1:5]

Out[201]:

**[2, 3, 4, 5]**

In [12]:

fruits **=** ["Apple", "Banana", "Melon", "Grapes", "Raspberries"]

print(fruits)

**['Apple', 'Banana', 'Melon', 'Grapes', 'Raspberries']**

If we want to find an element in a string, we can use the helpfully-named built-in function (or method) find(). For example, typing myString.find(’W’) will return the index of the letter ’W’. What happens if the given letter (or substring) isn't found in the string? Remember to use the help() function if you get stuck.

In [198]:

myString**=** "Hello World"

myString**.**find('W') *# find the index of the character W in myString*

Out[198]:

**6**

Finally, although we can’t subtract or divide strings, we do have two operators at our disposal: + (concatenation) and \* (repeated concatenation).

Before running the cell below, what do you expect will be stored in each variable below? Does the result match what you expected?

* newString = "Hello" + "World!"
* repString = "Hello" \* 5

In [9]:

newString **=** "Hello" **+** "World!"

repString **=** "Hello" **\*** 5

print('newString is: ', newString)

print('repString is: ', repString)

**newString is: HelloWorld!**

**repString is: HelloHelloHelloHelloHello**

**5. Lists**

**list**s are an incredibly powerful and versatile data type we can use in python to store a sequence of values.

Any other data type can be inserted into a **list**, including other **list**s. Run the following cell to see how we can create a new **list** object:

In [25]:

fruits **=** ["Apple", "Banana", "Melon", "Grapes", "Raspberries"]

print(fruits[2:4])

**['Melon', 'Grapes']**

Like with **str** objects, we can access and manipulate **list** objects using indexing and slicing techniques, in much the same way.

Can you write a command below to print() 'Grapes' by using the corresponding index from the **list**?

In [206]:

fruits **=** ["Apple", "Banana", "Melon", "Grapes", "Raspberries"]

print(fruits[3]) *#insert the correct command inside the ()*

**Grapes**

If we want to access more than one element of a list, we can slice the list, using the same syntax as with the myString examples above.

What do you think will print if you type print fruits[2:-1] in the cell below?

What about print(fruits[2:-1][0])? print(fruits[2:-1][0][4])? Try it and see!

**NB!** while indexing a list returns the value of a single element, a list slice is itself a list. This difference is subtle, but important to remember.

In [15]:

print(fruits[2:**-**1]) *# what does this operation return? first the thirds melon and one below rasberies = grapes*

print(fruits[2:**-**1][0]) *# what about this one? nula znamena ze se vybere prvni hodnota z predchoziho -asi*

print(fruits[2:**-**1][0][4]) *# and this one? after the above vybrat 4 z predchoziho asi*

**['Melon', 'Grapes']**

**Melon**

**N**

**6. Classes, functions and methods**

In programming, a **function** is essentially a short program that we can use to perform a specific action.

Functions take in **parameters** in the form of **arguments**, and (often, but not always) return a result, or otherwise perform an action.

Parameters can be **positional** (in other words, the order they are given matters), or they can be **keyword** (i.e., you specify the argument with the parameter name, in the form parameter=value).

Python has a number of built-in functions for us to use. For example, instead of typing 2 \*\* 8 earlier, we could instead have typed **pow(2,8)**:

In [30]:

print('using the \*\* operator: {}'**.**format(2**\*\***8))

print('using the pow() function: {}'**.**format(pow(2, 8)))

**using the \*\* operator: 256**

**using the pow() function: 256**

In [207]:

In [14]:

fruits **=** ["Apple", "Banana", "Melon", "Grapes", "Raspberries"]

print(fruits[3])

**Grapes**

In [14]:

Here, we are calling the function pow() and supplying the **positional** arguments 2 and 8. The result returned is the same, 256 (or 28), but the approach used is different.

If you want to see a list of **built-in** functions and classes in python, you can type print(dir(\_\_builtins\_\_)) (note the two underscores on either side of **builtins**):

**>>>** print(dir(\_\_builtins\_\_))

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

While it may not be completely clear at first what each of these things are, remember that we can use the help() **function** to get more information.

For example, one very useful built-in **class** is range ([documentation](https://docs.python.org/3/library/stdtypes.html#range)).

To create a new **range** object, we call it like we would a function:

range(stop)

range([start,] stop [,step])

"Under the hood", so to speak, remember that this is actually calling the **\_\_init\_\_()** method of the **class**, which is the **function** that python uses to *initialize*, or create, a new object.

Note that **range()** takes between one and three arguments:

* range(stop) creates a **range** object that will "count" from 0 up to (**but not including**) stop, incrementing by 1.
* range(start, stop) creates a **range** object that will "count" from start up to (**but not including**) stop, incrementing by 1.
* range(start, stop, step) creates a **range** object that will "count" from start to (**but not including**) stop, incrementing by step.

To pass multiple parameters to a function, we separate each parameter by a comma.

In the cell below, write a statement that returns a list of numbers counting from a start of 10 to 0 (inclusive).

In [34]:

print('using the \*\* operator: {}'**.**format(2**\*\***8))

print('using the pow() function: {}'**.**format(pow(2, 8)))

**using the \*\* operator: 256**

**using the pow() function: 256**

In [209]:

**for** ii **in** range(11): *# modify this to print out a list of numbers 10, 9, 8, ... 0.*

print('using the range() function: {}'**.**format(range(10,0,1)))

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

**using the range() function: range(10, 0)**

A **method** is a type of **function** that acts directly on an object.

In general, methods are called just like functions - the general syntax is object.method(arguments).

For example, **str** objects have a **method**, str.count(), which counts the number of times a character (or substring) occurs in the **str**.

If you type topic = "Geographic Information Systems" into the interpreter, what would you expect the result of topic.count("i") to be? What about topic.count("s")?

In [65]:

topic **=** "Geographic Information Systems"

print(topic**.**count("i"))

print(topic**.**count("s"))

**2**

**2**

Another powerful **str** method is str.split(), which returns a **list** of the given **str**, split into substrings based on the delimeter provided as an argument:

**>>>** help(str**.**split)

split(self, **/**, sep**=None**, maxsplit**=-**1)

Return a list of the words **in** the string, using sep **as** the delimiter string**.**

sep

The delimiter according which to split the string**.**

**None** (the default value) means split according to any whitespace,

**and** discard empty strings **from** the result**.**

maxsplit

Maximum number of splits to do**.**

**-**1 (the default value) means no limit**.**

From this, we can see that if we call topic.split() without any arguments at all, it will split topic based on any whitespace and discard any *empty* strings. That is, if we have multiple spaces in our string, it will treat those as a single space:

In [210]:

singlespace **=** 'Geographic Information Systems'

multispace **=** 'Geographic Information Systems'

print(singlespace**.**split())

print(multispace**.**split())

**['Geographic', 'Information', 'Systems']**

**['Geographic', 'Information', 'Systems']**

If we want to specify a single space character (' '), though, the result will change:

In [211]:

singlespace**.**split(' ')

multispace**.**split(' ')

**Out[211]:**

**['Geographic', '', '', 'Information', '', 'Systems']**

Using str.split() and an additional method to change all of the letters in the **str** to *lower*\ -case, can you get the following result in the cell below?

**>>>** topic **=** 'Remote Sensing and Geographic Information Systems'

**>>>** *# something else goes here*

**>>>** print(topic**.**split('s'))

**['Remote Sen', 'ing and Geographic Information Sy', 'tem', '']**

In [68]:

topic **=** 'Remote Sensing and Geographic Information Systems'

topic**=**topic**.**lower() *# something else goes here - changing for lower case*

print(topic**.**split('s'))

**['remote ', 'en', 'ing and geographic information ', 'y', 'tem', '']**

**6.1 Defining our own functions**

Often, we will want to define our own **function**s. Using functions has many benefits, including:

* improving readability,
* eliminating repetitive code,
* allowing for easier debugging of a program,
* and even allowing us to re-use code in other scripts/programs.

Defining a **function** in python is quite easy.

We begin the definition with a def **statement** that includes the function name and all parameters (this first line is called the **header**). The header must end with a colon (:):

**def** cat\_twice(str1, str2):

The **body** of the function (i.e., the set of instructions that make up the function) are *indented* - like other forms of flow control in python, once the interpreter sees a non-indented line, it marks the end of the function:

**def** cat\_twice(str1, str2):

cat **=** str1 **+** str2

print(cat) *# this is part of the function*

print(cat) *# this is part of the function*

*# this is no longer part of the function*

Let's say that we want to define a function to **concatenate** (add together) two strings, and print the resulting string two times:

In [212]:

**def** cat\_twice(str1, str2):

cat **=** str1 **+** str2

print(cat)

print(cat)

That's it. Simple, right? Now, we can use our function by calling it like any other function:

In [213]:

cat\_twice('bing tiddle ', 'tiddle bing')

**bing tiddle tiddle bing**

**bing tiddle tiddle bing**

Notice that our function doesn't return anything, however - it just prints to the screen. Functions like this that don't return any a value are called **void** functions. To see what happens if we try to assign the result of such a function, run the cell below:

In [71]:

result **=** cat\_twice('bing tiddle ', 'tiddle bing')

print(result)

**bing tiddle tiddle bing**

**bing tiddle tiddle bing**

**None**

If we want to return something from a function, we use a return **statement**, followed by the variable(s) that we want to return:

**def** cat\_twice\_return(str1, str2):

cat **=** str1 **+** str2

print(cat)

print(cat)

**return** cat

Using what you have learned about functions, define your own function below to split a string on a given character (letter), regardless of whether that character in the original string is upper or lowercase:

In [214]:

**def** special\_split(string, sep):

string**.**lower()**.**split('s')

topic **=** 'Remote Sensing and Geographic Information Systems'

my\_split **=** special\_split(topic, 's')

print (my\_split)

**return** my\_split

None

**Cell In[214], line 6**

**return my\_split**

**^**

**SyntaxError:** 'return' outside function

**7. Controlling Flow**

Some of the most important uses that we'll have for programming are repeating tasks and executing different code based on some condition. For example, we might want to loop through a list of files and run a series of commands on each file, or apply an analysis only if the right conditions are met.

In python, we can use the while, for, and if operators to control the flow of our programs. For example, given a number, we might want to check whether the value is positive, negative, or zero, and perform a different action based on which condition is True:

In [149]:

**def** pos\_neg\_zero(x):

**if** x **>** 0:

print('{} is a positive number'**.**format(x))

**elif** x **<** 0:

print('{} is a negative number'**.**format(x))

**else**:

print('{} is zero'**.**format(x))

pos\_neg\_zero(x**=-**6)

-6 is a negative number

Here, we take in a number, x, and execute code based on whether x is positive, negative, or zero.

Note that in the following:

print('{} is a positive number'**.**format(x))

we are using the str.format() method ([documentation](https://docs.python.org/3/tutorial/inputoutput.html#the-string-format-method)) to insert the value of x into the str where the curly brackets ({ }) are, before printing it to the screen:

**>>>** x **=** 2

**>>>** print('{} is a positive number'**.**format(x))

2 **is** a positive number

We can use this to insert as many values into our str as we like, so long as there are the same number of { } available:

**>>>** y **=** 3

**>>>** print('{} is larger than {}`.format(y, x))

3 **is** larger than 2

Note also that the order matters - the first argument to str.format() goes into the first {}, the second argument into the second {}, and so on. We'll use this a bit more later to print() nicely-formatted numeric results, as well.

Anyway, back to if/elif/else statements.

Like the header of a function, an if **statement** has to be terminated with a colon (:).

If we have multiple options, we can use an elif **statement**. There isn't a limit to the number of elif statements we can use, but note that the order matters - once a condition is evaluated as True, the indented code is executed and the whole block is exited.

For this reason, an else **statement** is optional, but it must always be last (since it automatically evaluates as True).

In the cell below, write a function to compare two numbers, x and y, and print() a statement based on the comparison (including the possibility that they are equal).

In [148]:

**def** which\_is\_greater(x,y):

**if** x **>** y:

print('{} is larger than {}'**.**format(x,y))

**elif** x **==** y:

print('{} is equal to {}'**.**format(x,y))

**else**:

print('{} is smaller than {}'**.**format(x,y))

which\_is\_greater(x**=**"2", y**=**"5")

2 is smaller than 5

In addition to conditional flow, we might also want to repeat actions. For example, we can write a simple function that counts down to some event, then announces the arrival of that event. We could define this function using a while loop, making sure to update a variable in each step:

In [153]:

**def** countdown(n):

**while** n **>** 0:

print(n)

n **-=** 1 *# note that this is the same as n = n - 1*

print("Blastoff!")

**countdown(n=6)**

**6**

**5**

**4**

**3**

**2**

**1**

**Blastoff!**

Note the importance of updating the variable that we are testing in the loop. If we remove the n -= 1 line, our function will never stop running (an **infinite loop**).

while loops are useful for actions without a pre-defined number of repetitions. We could just as easily re-define countdown() using a for loop, using something else we've seen before:

**def** countdown\_for(n):

**for** ii **in** range(n, 0, **-**1):

print(ii)

print("Blastoff!")

This version uses range to iterate from n to 1 in increments of -1, printing the value of i each time - that is, we leave n unchanged.

We can also use the break and continue statements to **break** out of a loop, or to **continue** to the next step of a loop:

**def** break\_example(n):

*# prints values from n to 1, then Blastoff!*

**while** **True**: *# here, the loop will always run*

*# unless we reach a condition*

*# that breaks out of it:*

**if** n **<=** 0:

**break**

print(n)

n **-=** 1

print("Blastoff!")

**def** continue\_example(n):

*# given an integer, n, prints the values from 0 to n that are even.*

**for** x **in** range(n):

**if** x **%** 2 **==** 1:

**continue**

print('{} is even'**.**format(x))

In the cell below, write a function to print the even values from 1 to n, unless the value is divisible by 3 or 4.

In [161]:

**def** evenNoThreesorFours(n):

*# given an integer n, prints values from 1 to n that are even, unless they are divisible by 3 or 4.*

**for** x **in** range(1, n):

**if** x **%** 2 **==** 1:

**continue**

print('{} is even'**.**format(x))

evenNoThreesorFours(n**=**10)

**2 is even**

**4 is even**

**6 is even**

**8 is even**

(*Hint: you can use the % operator to determine whether one number is divisble by another*).

In [215]:

**def** evenNoThreesorFours(n):

*# given an integer n, prints values from 1 to n that are even, unless they are divisible by 3 or 4.*

**for** x **in** range(1, n):

**if** x **%** 2 **==** 1:

**if** **not** x **%** 3:

**if** **not** x **%** 4:

**continue**

print('{} is even and not divisible by 3 or 4'**.**format(x))

evenNoThreesorFours(n**=**50)

1 is even and not divisible by 3 or 4

2 is even and not divisible by 3 or 4

3 is even and not divisible by 3 or 4

4 is even and not divisible by 3 or 4

5 is even and not divisible by 3 or 4

6 is even and not divisible by 3 or 4

7 is even and not divisible by 3 or 4

8 is even and not divisible by 3 or 4

9 is even and not divisible by 3 or 4

10 is even and not divisible by 3 or 4

11 is even and not divisible by 3 or 4

12 is even and not divisible by 3 or 4

13 is even and not divisible by 3 or 4

14 is even and not divisible by 3 or 4

15 is even and not divisible by 3 or 4

16 is even and not divisible by 3 or 4

17 is even and not divisible by 3 or 4

18 is even and not divisible by 3 or 4

19 is even and not divisible by 3 or 4

**8. Importing modules**

Modules provide a convenient way to package functions and object classes, and load these items when needed. This also means that we only end up loading the functionality that we need, which helps save on memory and other resources.

We have already imported one such module, the sys module. Another useful module to use is the math module, which provides much more than the built-in operators we explored earlier. Run the following cell:

In [180]:

**import** math

**from** math **import** pi, floor

print('math.pi is equal to: {}'**.**format(math**.**pi))

print('pi is equal to: {}'**.**format(pi))

h1 **=** math**.**floor(10.19)

print('math.floor(10.19) is equal to: {}'**.**format(h1))

h2 **=** floor(10.19)

print('floor(10.19) is equal to: {}'**.**format(h2))

math.pi is equal to: 3.141592653589793

pi is equal to: 3.141592653589793

math.floor(10.19) is equal to: 10

floor(10.19) is equal to: 10

There are a few things to pay attention to here. First, notice that the syntax for calling functions from a module is the same as calling a method: module.function(arguments).

You should also notice that we can **import** a whole module (import math) or **import** an attribute, class, or function from a module (from math import pi).

When we specifically name the things we want to import, we only have access to those things - importing pi from math does not also import floor - hence, the error message.

It is also possible to **import** all of the functions and classes from a given module (from math import \*), but this is not really recommended - why do you think this is?

Modify the cell above to **import** both floor and pi from math, and run the cell again. Remember that to **import** multiple things from a single module, you can separate them by commas:

**from** math **import** sin, cos, tan

Notice also that **math.pi** is not a **method**, but an **attribute**. You can check this for yourself by typing math.pi() into the interpreter and seeing the result.

**9. Working with shapefiles**

When using vector data in this course, we will primarily work with geopandas ([documentation](http://geopandas.org/)), "an open source project to make working with geospatial data in python easier."

To work with the provided shapefile (data\_files/Glaciers.shp), we first have to **import** geopandas and load the data using geopandas.read\_file():

In [181]:

**import** geopandas **as** gpd

glacier\_data **=** gpd**.**read\_file('data\_files/Glaciers.shp')

Note that in the cell above, we're using an *alias* when we **import** geopandas:

**import** geopandas **as** gpd

This means that instead of having to write geopandas every time we want to use a **method** or **class** from the geopandas package, we instead type gpd. You will most likely see this syntax a lot - it's mostly used to make the code easier to read (or out of laziness).

**9.1 a note on filepaths**

On Windows computers, filepaths are separated using \. For example, on my Windows machine, this notebook file has the following path:

C:\Users\bob\egm722\Week1\Practical1.ipynb

This is a problem in python, because \ is a protected character -- specifically, it's either used as a line continuation to split a string over multiple lines:

**>>>** mystring **=** 'this is a string that is split ' **+** \

**...** 'over multiple lines'

**>>>** print(mystring)

this **is** a string that **is** split over multiple lines

Inside of a string, \ *escapes* the next character - effectively, it makes the interpreter change how it processes it:

**>>>** not\_escaped **=** 'this isn't going to work'

File "<stdin>", line 1

not\_escaped **=** 'this isn't going to work'

**^**

SyntaxError: invalid syntax

In the example above, the single quote in "isn't" actually ends the **str** - because mashing a string ('this isn') and something else (t going to work') doesn't work, python raises a SyntaxError.

But, we can use \ to make python see that the single quote in the middle of the **str** should be treated as part of the **str**:

**>>>** escaped **=** 'this isn\'t going to cause a problem.'

**>>>** print(escaped)

this isn't going to cause a problem.

What this means is that when we're working with Windows paths in python, we have to do something competely different. When writing paths as **str** objects, there are three main options:

1. replace \ with /: C:/Users/bob/egm722/Week1/Practical1.ipynb
2. *escape* the \ chracter: C:\\Users\\bob\\egm722\\Week1\\Practical1.ipynb
3. use a **raw string literal** by appending r before the start of the **str**: r'C:\Users\bob\egm722\Week1\Practical1.ipynb'

Option 1. has the advantage of working on multiple platforms. As an example, the *relative* path:

'data\_files/Glaciers.shp'

will work on Windows, MacOS, and linux systems. Later in this module, we will see how we can use the os module to work with filepaths; if we're using python 3.4 or newer, the [pathlib](https://docs.python.org/3/library/pathlib.html) module provides an even nicer way of working with filepaths.

**9.2 working with GeoDataFrames**

Back to working with shapefile data.

In [182]:

print(glacier\_data**.**head())

RGIID GLIMSID RGIFLAG BGNDATE ENDDATE CENLON \

0 RGI40-08.01398 G006462E59966N 000 20030809 20060916 6.46207

1 RGI40-08.01414 G006415E60121N 000 20030809 20060916 6.41519

2 RGI40-08.01415 G006422E60101N 000 20030809 20060916 6.42158

3 RGI40-08.01416 G006291E60040N 010 20030809 20060916 6.29127

4 RGI40-08.01417 G006256E60024N 010 20030809 20060916 6.25622

CENLAT O1REGION O2REGION AREA ZMIN ZMAX ZMED SLOPE ASPECT LMAX \

0 59.9663 8 1 0.115 1239 1352 1296 23.5 90 224

1 60.1211 8 1 0.176 1291 1385 1335 12.9 0 532

2 60.1011 8 1 4.352 1173 1578 1473 12.3 315 2814

3 60.0398 8 1 2.764 1114 1683 1483 13.4 0 2749

4 60.0240 8 1 2.974 1049 1691 1498 14.2 315 2937

GLACTYPE NAME geometry

0 9099 NaN POLYGON ((6.45968 59.96954, 6.45965 59.96929, ...

1 9099 NaN POLYGON ((6.41323 60.12366, 6.41318 60.12315, ...

2 9099 NaN POLYGON ((6.42682 60.11614, 6.42679 60.11589, ...

3 9099 NaN POLYGON ((6.30861 60.04641, 6.30862 60.04637, ...

4 9099 NaN POLYGON ((6.25752 60.03359, 6.25792 60.03346, ...

Note that the data are stored in a table (a **GeoDataFrame**), much like the attribute table in ArcMap. One small difference is the additional column, geometry, which stores the geometry for each feature (in this case, a polygon).

One thing that we might be interested in, is the number of features stored in our dataset. Within a script, the best way to do this is by using the shape of the **GeoDataFrame**, which prints out the size of the **GeoDataFrame** in (rows, columns):

In [183]:

rows, cols **=** glacier\_data**.**shape *# get the number of rows, columns in the table*

print('Number of features: {}'**.**format(rows))

**Number of features: 62**

We can also perform calculations on the data, get statistics, etc. Many of these are built-in methods that can be called on an individual column.

We can access individual columns in two ways. We'll stick with the "normal" way for now, but we'll revisit the other method later on in another lab. The normal way is by indexing with the column name (i.e., dataframe['column\_name']).

If we want to see what columns are available in the data table, we can look at the columns **attribute** of the table:

In [184]:

glacier\_data**.**columns

Out[184]:

Index(['RGIID', 'GLIMSID', 'RGIFLAG', 'BGNDATE', 'ENDDATE', 'CENLON', 'CENLAT',

'O1REGION', 'O2REGION', 'AREA', 'ZMIN', 'ZMAX', 'ZMED', 'SLOPE',

'ASPECT', 'LMAX', 'GLACTYPE', 'NAME', 'geometry'],

dtype='object')

There are quite a few columns here, and most of them aren't so important for now. We might be interested in working with the **AREA** column, though, to find out more about the size of glaciers in our study region. For example, to find the average glacier area in our dataset, we could type the following:

In [185]:

glacier\_data['AREA']**.**mean()

Out[185]:

3.252387096774194

This gives us the average area in square kilometers (the units of the column). What if, for some unfathomable reason, we wanted the glacier areas to be stored in square feet? We could do this by updating the column directly:

In [ ]:

glacier\_data['AREA'] **=** glacier\_data['AREA'] **\*** 1e6 **\*** 3.2808399 **\*** 3.2808399 *# km2 x m2/km2 X ft/m x ft/m*

glacier\_data['AREA']**.**mean()

But, just in case we didn't want to lose the sensible units, we could also have created a new column:

In [186]:

glacier\_data['AREA\_SQKM'] **=** glacier\_data['AREA'] **/** 1e6 **/** 3.2808399 **/** 3.2808399 *# the reverse of the above*

glacier\_data['AREA\_SQKM']**.**mean()

**Out[186]:**

**3.021566476285405e-07**

**10. Example - converting text data into shapefiles**

Another useful application we'll make use of is pandas ([documentation](https://pandas.pydata.org/)), a powerful data analysis package that provides the basis for geopandas (as you perhaps guessed by the name). In this example, we'll take a Comma Separated Value (**CSV**) file representing data about lakes from around the world, and convert this into a shapefile that we can load into ArcGIS, QGIS, or other GIS software.

First, let's import the necessary modules and load the data:

In [187]:

**import** pandas **as** pd

**import** geopandas **as** gpd

**from** shapely.geometry **import** Point

**df = pd.read\_csv('data\_files/GPSPoints.txt')**

Like we did with the geopandas data above, let's have a look at the DataFrame we've just loaded:

In [188]:

df**.**head()

Out[188]:

|  | **area** | **lon** | **lat** | **id** | **level** | **max\_depth** | **name** | **parent\_id** | **sibling\_id** | **source** | **date** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 371000 | 50.7974 | 41.7510 | 145424 | 2 | 1025.0 | Caspian Sea | 0 | 180487 | WDBII | 2000-03-17 |
| **1** | 59600 | -82.2559 | 44.9583 | 145425 | 2 | 229.0 | Lake Huron | 2 | 180488 | WDBII | 2009-11-15 |
| **2** | 31722 | 107.7016 | 53.4145 | 145429 | 2 | 1642.0 | Lake Baikal | 0 | 180492 | WDBII | 2013-10-02 |
| **3** | 31153 | -120.9532 | 66.0093 | 145430 | 2 | 446.0 | Great Bear Lake | 2 | 180493 | WDBII | 2012-02-02 |
| **4** | 27200 | -113.7628 | 61.7776 | 145431 | 2 | 614.0 | Great Slave Lake | 2 | 180494 | WDBII | 2006-08-20 |

Here, we see a number of lake names, with accompanying data such as the lake area (presumably in square kilometers?), the maximum depth (missing for some lakes), Lat/Lon information, and so on. Remember that a GeoDataFrame can hold lots of different data, but it needs a column that specifies the geometry of a feature. Given that we only have point information (a single Lat/Lon coordinate) for each lake, it makes sense to create a Point object for each feature using that point. We can do this by first using the python built-in **zip**, then the **apply** method of the DataFrame to create a point object from the list of coordinates.

In [189]:

df['geometry'] **=** list(zip(df['lon'], df['lat'])) *# zip is an iterator, so we use list to create*

*# something that pandas can use.*

df['geometry'] **=** df['geometry']**.**apply(Point) *# using the 'apply' method of the dataframe,*

*# turn the coordinates column*

*# into points (instead of a tuple of lat, lon coordinates).*

*# NB: Point takes (x, y) coordinates*

Let's look at the DataFrame again. We should have a geometry column, with the lat/lon coordinates for each feature:

In [190]:

df

Out[190]:

|  | **area** | **lon** | **lat** | **id** | **level** | **max\_depth** | **name** | **parent\_id** | **sibling\_id** | **source** | **date** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 371000 | 50.7974 | 41.7510 | 145424 | 2 | 1025.0 | Caspian Sea | 0 | 180487 | WDBII | 2000-03-17 | POINT (50.7974 41.751) |
| **1** | 59600 | -82.2559 | 44.9583 | 145425 | 2 | 229.0 | Lake Huron | 2 | 180488 | WDBII | 2009-11-15 | POINT (-82.2559 44.9583) |
| **2** | 31722 | 107.7016 | 53.4145 | 145429 | 2 | 1642.0 | Lake Baikal | 0 | 180492 | WDBII | 2013-10-02 | POINT (107.7016 53.4145) |
| **3** | 31153 | -120.9532 | 66.0093 | 145430 | 2 | 446.0 | Great Bear Lake | 2 | 180493 | WDBII | 2012-02-02 | POINT (-120.9532 66.0093) |
| **4** | 27200 | -113.7628 | 61.7776 | 145431 | 2 | 614.0 | Great Slave Lake | 2 | 180494 | WDBII | 2006-08-20 | POINT (-113.7628 61.7776) |
| **5** | 29600 | 34.5098 | -11.9643 | 145432 | 2 | 706.0 | Lake Malawi | 1 | 180495 | WDBII | 2010-04-05 | POINT (34.5098 -11.9643) |
| **6** | 25667 | -81.2358 | 42.1555 | 145433 | 2 | 64.0 | Lake Erie | 2 | 180496 | WDBII | 2013-10-28 | POINT (-81.2358 42.1555) |
| **7** | 24514 | -97.8114 | 52.6031 | 145434 | 2 | 36.0 | Lake Winnipeg | 2 | 180497 | WDBII | 2011-11-10 | POINT (-97.8114 52.6031) |
| **8** | 19000 | -77.7592 | 43.6671 | 145435 | 2 | 244.0 | Lake Ontario | 2 | 180498 | WDBII | 2007-02-15 | POINT (-77.7592 43.6671) |
| **9** | 17891 | 31.4610 | 60.8466 | 145436 | 2 | 230.0 | Lake Ladoga | 0 | 180499 | WDBII | 2003-04-15 | POINT (31.461 60.8466) |
| **10** | 16996 | 75.6321 | 46.2842 | 145437 | 2 | 26.0 | Lake Balkash | 0 | 180500 | WDBII | 2009-10-31 | POINT (75.6321 46.2842) |
| **11** | 1350 | 14.0564 | 13.1940 | 145438 | 2 | 11.0 | Lake Chad | 1 | 180501 | WDBII | 2004-02-12 | POINT (14.0564 13.194) |
| **12** | 4380 | 28.4338 | 62.0452 | 145439 | 2 | 82.0 | Storsaimen | 0 | 180502 | WDBII | 2009-02-06 | POINT (28.4338 62.0452) |
| **13** | 9894 | 35.3822 | 61.7853 | 145440 | 2 | 127.0 | Lake Onega | 0 | 180503 | WDBII | 2009-08-18 | POINT (35.3822 61.7853) |
| **14** | 8264 | -85.3904 | 11.5322 | 145442 | 2 | 26.0 | Lake Nicaragua | 2 | 180505 | WDBII | 2000-01-04 | POINT (-85.3904 11.5322) |
| **15** | 7850 | -109.3507 | 59.1734 | 145443 | 2 | 124.0 | Lake Athabasca | 2 | 180506 | WDBII | 2009-04-18 | POINT (-109.3507 59.1734) |
| **16** | 8372 | -69.3648 | -15.8825 | 145444 | 2 | 281.0 | Lake Titicaca | 3 | 180507 | WDBII | 2008-03-17 | POINT (-69.3648 -15.8825) |
| **17** | 6405 | 36.1790 | 3.5880 | 145445 | 2 | 109.0 | Lake Turkana | 1 | 180508 | WDBII | 2011-01-17 | POINT (36.179 3.588) |
| **18** | 5650 | -102.3784 | 57.2870 | 145446 | 2 | 219.0 | Reinsjøen | 2 | 180509 | WDBII | 2013-07-22 | POINT (-102.3784 57.287) |
| **19** | 6236 | 77.2663 | 42.4381 | 145447 | 2 | 668.0 | Issyk-Kul | 0 | 6236 | WDBII | 2005-06-09 | POINT (77.2663 42.4381) |
| **20** | 5650 | 13.2708 | 58.9076 | 145448 | 2 | 106.0 | Vänern | 0 | 180511 | WDBII | 2002-02-25 | POINT (13.2708 58.9076) |
| **21** | 82100 | -87.7567 | 47.5367 | 145425 | 2 | 406.0 | Lake Superior | 2 | 180488 | WDBII | 1999-08-12 | POINT (-87.7567 47.5367) |
| **22** | 58000 | -86.7612 | 44.0135 | 145425 | 2 | 281.0 | Lake Michigan | 2 | 180488 | WDBII | 2004-10-03 | POINT (-86.7612 44.0135) |
| **23** | 5370 | -100.1435 | 52.5566 | 145451 | 2 | 12.0 | Lake Winnipegosis | 2 | 180514 | WDBII | 2014-03-03 | POINT (-100.1435 52.5566) |
| **24** | 4624 | -98.6423 | 50.9195 | 145455 | 2 | 7.0 | Lake Manitoba | 2 | 180518 | WDBII | 2012-07-20 | POINT (-98.6423 50.9195) |
| **25** | 4848 | -88.5262 | 49.8306 | 145456 | 2 | 165.0 | Lake Nipigon | 2 | 180519 | WDBII | 2014-05-25 | POINT (-88.5262 49.8306) |
| **26** | 4560 | 102.0923 | 74.5772 | 145458 | 2 | 26.0 | Lake Taymyr | 0 | 180521 | WDBII | 2013-06-08 | POINT (102.0923 74.5772) |
| **27** | 4168 | -94.7027 | 49.2872 | 145459 | 2 | 64.0 | Lake of the Woods | 2 | 180522 | WDBII | 2006-10-02 | POINT (-94.7027 49.2872) |
| **28** | 1320 | -100.1255 | 53.5496 | 145461 | 2 | 10.0 | Cedar Lake | 2 | 180524 | WDBII | 2003-01-27 | POINT (-100.1255 53.5496) |
| **29** | 1070 | 25.9600 | 62.1873 | 145466 | 2 | 96.0 | Päijänne | 0 | 180529 | WDBII | 2006-04-27 | POINT (25.96 62.1873) |
| **30** | 3833 | -101.4030 | 63.1110 | 145470 | 2 | NaN | Dubawnt Lake | 2 | 180533 | WDBII | 2010-11-22 | POINT (-101.403 63.111) |
| **31** | 3555 | 27.5429 | 58.5428 | 145471 | 2 | 15.0 | Lake Peipus | 0 | 180534 | WDBII | 2001-04-14 | POINT (27.5429 58.5428) |
| **32** | 3115 | -71.2279 | 64.9084 | 145476 | 2 | NaN | Lake Amadjuak | 11 | 180539 | WDBII | 2016-11-05 | POINT (-71.2279 64.9084) |
| **33** | 68800 | 32.9078 | -1.1007 | 145426 | 2 | 83.0 | Lake Victoria | 1 | 180489 | WDBII | 2004-12-30 | POINT (32.9078 -1.1007) |

We could simply create a GeoDataFrame from this DataFrame, but let's first remove a few extra columns from the table, and change the units of the area column to be in square meters:

In [191]:

**del** df['lat'], df['lon'] *# we don't really need these, since they're in the 'geometry' column now*

df['area'] *# convert the area column to square meters here*

Out[191]:

0 371000

1 59600

2 31722

3 31153

4 27200

5 29600

6 25667

7 24514

8 19000

9 17891

10 16996

11 1350

12 4380

13 9894

14 8264

15 7850

16 8372

17 6405

18 5650

19 6236

20 5650

21 82100

22 58000

23 5370

24 4624

25 4848

26 4560

27 4168

28 1320

29 1070

30 3833

31 3555

32 3115

33 68800

Name: area, dtype: int64

Great. Now we can create a new GeoDataFrame from the DataFrame. We'll also be sure to set the spatial reference information, so that our GIS software knows what reference frame our data use. For this, we'll use the EPSG code representing WGS84 Lat/Lon, 4326. EPSG codes are a concise way to refer to a given reference system - more information about them (and a comprehensive list of codes) can be found [here](http://spatialreference.org/).

In [192]:

gdf **=** gpd**.**GeoDataFrame(df)

gdf**.**set\_crs("EPSG:4326", inplace**=True**) *# this sets the coordinate reference system to epsg:4326, wgs84 lat/lon*

Out[192]:

|  | **area** | **id** | **level** | **max\_depth** | **name** | **parent\_id** | **sibling\_id** | **source** | **date** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 371000 | 145424 | 2 | 1025.0 | Caspian Sea | 0 | 180487 | WDBII | 2000-03-17 | POINT (50.79740 41.75100) |
| **1** | 59600 | 145425 | 2 | 229.0 | Lake Huron | 2 | 180488 | WDBII | 2009-11-15 | POINT (-82.25590 44.95830) |
| **2** | 31722 | 145429 | 2 | 1642.0 | Lake Baikal | 0 | 180492 | WDBII | 2013-10-02 | POINT (107.70160 53.41450) |
| **3** | 31153 | 145430 | 2 | 446.0 | Great Bear Lake | 2 | 180493 | WDBII | 2012-02-02 | POINT (-120.95320 66.00930) |
| **4** | 27200 | 145431 | 2 | 614.0 | Great Slave Lake | 2 | 180494 | WDBII | 2006-08-20 | POINT (-113.76280 61.77760) |
| **5** | 29600 | 145432 | 2 | 706.0 | Lake Malawi | 1 | 180495 | WDBII | 2010-04-05 | POINT (34.50980 -11.96430) |
| **6** | 25667 | 145433 | 2 | 64.0 | Lake Erie | 2 | 180496 | WDBII | 2013-10-28 | POINT (-81.23580 42.15550) |
| **7** | 24514 | 145434 | 2 | 36.0 | Lake Winnipeg | 2 | 180497 | WDBII | 2011-11-10 | POINT (-97.81140 52.60310) |
| **8** | 19000 | 145435 | 2 | 244.0 | Lake Ontario | 2 | 180498 | WDBII | 2007-02-15 | POINT (-77.75920 43.66710) |
| **9** | 17891 | 145436 | 2 | 230.0 | Lake Ladoga | 0 | 180499 | WDBII | 2003-04-15 | POINT (31.46100 60.84660) |
| **10** | 16996 | 145437 | 2 | 26.0 | Lake Balkash | 0 | 180500 | WDBII | 2009-10-31 | POINT (75.63210 46.28420) |
| **11** | 1350 | 145438 | 2 | 11.0 | Lake Chad | 1 | 180501 | WDBII | 2004-02-12 | POINT (14.05640 13.19400) |
| **12** | 4380 | 145439 | 2 | 82.0 | Storsaimen | 0 | 180502 | WDBII | 2009-02-06 | POINT (28.43380 62.04520) |
| **13** | 9894 | 145440 | 2 | 127.0 | Lake Onega | 0 | 180503 | WDBII | 2009-08-18 | POINT (35.38220 61.78530) |
| **14** | 8264 | 145442 | 2 | 26.0 | Lake Nicaragua | 2 | 180505 | WDBII | 2000-01-04 | POINT (-85.39040 11.53220) |
| **15** | 7850 | 145443 | 2 | 124.0 | Lake Athabasca | 2 | 180506 | WDBII | 2009-04-18 | POINT (-109.35070 59.17340) |
| **16** | 8372 | 145444 | 2 | 281.0 | Lake Titicaca | 3 | 180507 | WDBII | 2008-03-17 | POINT (-69.36480 -15.88250) |
| **17** | 6405 | 145445 | 2 | 109.0 | Lake Turkana | 1 | 180508 | WDBII | 2011-01-17 | POINT (36.17900 3.58800) |
| **18** | 5650 | 145446 | 2 | 219.0 | Reinsjøen | 2 | 180509 | WDBII | 2013-07-22 | POINT (-102.37840 57.28700) |
| **19** | 6236 | 145447 | 2 | 668.0 | Issyk-Kul | 0 | 6236 | WDBII | 2005-06-09 | POINT (77.26630 42.43810) |
| **20** | 5650 | 145448 | 2 | 106.0 | Vänern | 0 | 180511 | WDBII | 2002-02-25 | POINT (13.27080 58.90760) |
| **21** | 82100 | 145425 | 2 | 406.0 | Lake Superior | 2 | 180488 | WDBII | 1999-08-12 | POINT (-87.75670 47.53670) |
| **22** | 58000 | 145425 | 2 | 281.0 | Lake Michigan | 2 | 180488 | WDBII | 2004-10-03 | POINT (-86.76120 44.01350) |
| **23** | 5370 | 145451 | 2 | 12.0 | Lake Winnipegosis | 2 | 180514 | WDBII | 2014-03-03 | POINT (-100.14350 52.55660) |
| **24** | 4624 | 145455 | 2 | 7.0 | Lake Manitoba | 2 | 180518 | WDBII | 2012-07-20 | POINT (-98.64230 50.91950) |
| **25** | 4848 | 145456 | 2 | 165.0 | Lake Nipigon | 2 | 180519 | WDBII | 2014-05-25 | POINT (-88.52620 49.83060) |
| **26** | 4560 | 145458 | 2 | 26.0 | Lake Taymyr | 0 | 180521 | WDBII | 2013-06-08 | POINT (102.09230 74.57720) |
| **27** | 4168 | 145459 | 2 | 64.0 | Lake of the Woods | 2 | 180522 | WDBII | 2006-10-02 | POINT (-94.70270 49.28720) |
| **28** | 1320 | 145461 | 2 | 10.0 | Cedar Lake | 2 | 180524 | WDBII | 2003-01-27 | POINT (-100.12550 53.54960) |
| **29** | 1070 | 145466 | 2 | 96.0 | Päijänne | 0 | 180529 | WDBII | 2006-04-27 | POINT (25.96000 62.18730) |
| **30** | 3833 | 145470 | 2 | NaN | Dubawnt Lake | 2 | 180533 | WDBII | 2010-11-22 | POINT (-101.40300 63.11100) |
| **31** | 3555 | 145471 | 2 | 15.0 | Lake Peipus | 0 | 180534 | WDBII | 2001-04-14 | POINT (27.54290 58.54280) |
| **32** | 3115 | 145476 | 2 | NaN | Lake Amadjuak | 11 | 180539 | WDBII | 2016-11-05 | POINT (-71.22790 64.90840) |
| **33** | 68800 | 145426 | 2 | 83.0 | Lake Victoria | 1 | 180489 | WDBII | 2004-12-30 | POINT (32.90780 -1.10070) |

Let's take another look at the GeoDataFrame, then we'll write it to a shapefile and load it into ArcGIS. Note that it looks mostly the same as the DataFrame does - it really is just an extension of the pandas DataFrame.

In [193]:

print(gdf)

gdf**.**to\_file('lake\_points.shp')

area id level max\_depth name parent\_id \

0 371000 145424 2 1025.0 Caspian Sea 0

1 59600 145425 2 229.0 Lake Huron 2

2 31722 145429 2 1642.0 Lake Baikal 0

3 31153 145430 2 446.0 Great Bear Lake 2

4 27200 145431 2 614.0 Great Slave Lake 2

5 29600 145432 2 706.0 Lake Malawi 1

6 25667 145433 2 64.0 Lake Erie 2

7 24514 145434 2 36.0 Lake Winnipeg 2

8 19000 145435 2 244.0 Lake Ontario 2

9 17891 145436 2 230.0 Lake Ladoga 0

10 16996 145437 2 26.0 Lake Balkash 0

11 1350 145438 2 11.0 Lake Chad 1

12 4380 145439 2 82.0 Storsaimen 0

13 9894 145440 2 127.0 Lake Onega 0

14 8264 145442 2 26.0 Lake Nicaragua 2

15 7850 145443 2 124.0 Lake Athabasca 2

16 8372 145444 2 281.0 Lake Titicaca 3

17 6405 145445 2 109.0 Lake Turkana 1

18 5650 145446 2 219.0 Reinsjøen 2

19 6236 145447 2 668.0 Issyk-Kul 0

20 5650 145448 2 106.0 Vänern 0

21 82100 145425 2 406.0 Lake Superior 2

22 58000 145425 2 281.0 Lake Michigan 2

23 5370 145451 2 12.0 Lake Winnipegosis 2

24 4624 145455 2 7.0 Lake Manitoba 2

25 4848 145456 2 165.0 Lake Nipigon 2

26 4560 145458 2 26.0 Lake Taymyr 0

27 4168 145459 2 64.0 Lake of the Woods 2

28 1320 145461 2 10.0 Cedar Lake 2

29 1070 145466 2 96.0 Päijänne 0

30 3833 145470 2 NaN Dubawnt Lake 2

31 3555 145471 2 15.0 Lake Peipus 0

32 3115 145476 2 NaN Lake Amadjuak 11

33 68800 145426 2 83.0 Lake Victoria 1

sibling\_id source date geometry

0 180487 WDBII 2000-03-17 POINT (50.79740 41.75100)

1 180488 WDBII 2009-11-15 POINT (-82.25590 44.95830)

2 180492 WDBII 2013-10-02 POINT (107.70160 53.41450)

3 180493 WDBII 2012-02-02 POINT (-120.95320 66.00930)

4 180494 WDBII 2006-08-20 POINT (-113.76280 61.77760)

5 180495 WDBII 2010-04-05 POINT (34.50980 -11.96430)

6 180496 WDBII 2013-10-28 POINT (-81.23580 42.15550)

7 180497 WDBII 2011-11-10 POINT (-97.81140 52.60310)

8 180498 WDBII 2007-02-15 POINT (-77.75920 43.66710)

9 180499 WDBII 2003-04-15 POINT (31.46100 60.84660)

10 180500 WDBII 2009-10-31 POINT (75.63210 46.28420)

11 180501 WDBII 2004-02-12 POINT (14.05640 13.19400)

12 180502 WDBII 2009-02-06 POINT (28.43380 62.04520)

13 180503 WDBII 2009-08-18 POINT (35.38220 61.78530)

14 180505 WDBII 2000-01-04 POINT (-85.39040 11.53220)

15 180506 WDBII 2009-04-18 POINT (-109.35070 59.17340)

16 180507 WDBII 2008-03-17 POINT (-69.36480 -15.88250)

17 180508 WDBII 2011-01-17 POINT (36.17900 3.58800)

18 180509 WDBII 2013-07-22 POINT (-102.37840 57.28700)

19 6236 WDBII 2005-06-09 POINT (77.26630 42.43810)

20 180511 WDBII 2002-02-25 POINT (13.27080 58.90760)

21 180488 WDBII 1999-08-12 POINT (-87.75670 47.53670)

22 180488 WDBII 2004-10-03 POINT (-86.76120 44.01350)

23 180514 WDBII 2014-03-03 POINT (-100.14350 52.55660)

24 180518 WDBII 2012-07-20 POINT (-98.64230 50.91950)

25 180519 WDBII 2014-05-25 POINT (-88.52620 49.83060)

26 180521 WDBII 2013-06-08 POINT (102.09230 74.57720)

27 180522 WDBII 2006-10-02 POINT (-94.70270 49.28720)

28 180524 WDBII 2003-01-27 POINT (-100.12550 53.54960)

29 180529 WDBII 2006-04-27 POINT (25.96000 62.18730)

30 180533 WDBII 2010-11-22 POINT (-101.40300 63.11100)

31 180534 WDBII 2001-04-14 POINT (27.54290 58.54280)

32 180539 WDBII 2016-11-05 POINT (-71.22790 64.90840)

33 180489 WDBII 2004-12-30 POINT (32.90780 -1.10070)

Load your shapefile into a GIS software package such as ArcGIS Pro or QGIS - does everything look right? If not, go back through the steps and see if you can find where you went wrong.

Can you make a map that looks like this one below in your GIS software of choice? Give it a try!

By the end of the module, you should be able to put together a map like this using python.

Obviously, this is only a beginning, but hopefully you've gotten a taste for what we can do with python. Feel free to explore more, read through some documentation, and ask questions.