EXERCISE 3

Indexing, Conditionals and Functions

Introduction

This exercise introduces scripting concepts that aren't specific to any scripting language or geospatial tool but are necessary to understand to begin writing code.

Objectives

- Become familiar with basic concepts of scripting

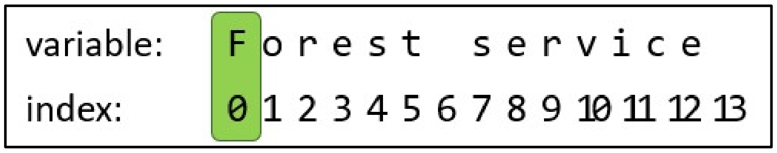
Table of Contents

Part 1: Indexing Variables

Sometimes you only need one part of a variable, especially if that variable is a list or string. Indexing is specifying a specific character in a variable.

A. Get one character from a variable

The characters (both letters and numbers) within a variable are numbered. This numbering starts at 0 so the first character will be the 0 character. This can be seen below:


variable: F o r e s t s e r v i c e
index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

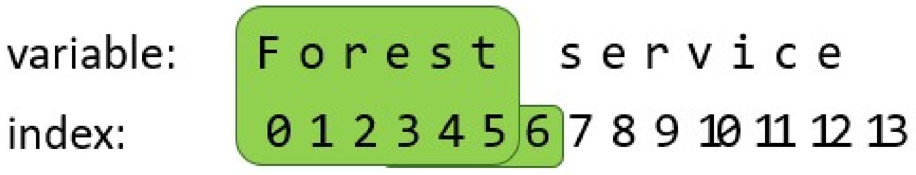
1. Create variable **FS** and have it equal to the string **Forest Service** then print the variable
2. Index the variable by the first character **F**

Language	Environment	Enter	Output
Python	IDLE or ArcPro	`print(FS[0])`	F
R	RStudio	`FS[0]`	F
JavaScript	Google Earth Engine	`print(FS.slice(0,1))`	F

3. Index the variable **FS** to get the letter **s**

B. Slice a variable

Simply, a slice is part of a variable. We take a slice by using the general syntax **variable[start:stop]** taking a slice of a variable or indexing a variable does not change the variable, rather it references the elements of the variable. Note:


variable: F o r e s t s e r v i c e
index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

1. Slice the variable **FS** to the word "Forest"

Language	Environment	Enter	Output
Python	IDLE or ArcPro	`print(FS[0:6])`	Forest
R	RStudio	`FS[0:6]`	Forest
JavaScript	Google Earth Engine	`print(FS.slice(0,6))`	Forest

Notice we had to use the index 6 to stop, not 5. That is because the start is inclusive and the stop is exclusive of the character.

2. Slice the variable **FS** to the word "service", you must figure out the start and stop index

Language	Environment	Enter	Output
Python	IDLE or ArcPro	`print(FS[start:stop])`	`service`
R	RStudio	`FS[start:stop]`	`service`
JavaScript	Google Earth Engine	`print(FS.slice(start,stop))`	`service`

3. You can also index lists and other variable types. Create the variable **Region** and have it equal to **[1,2,3,4,5,6,8,9]** now index the variable to the region you are located in.
4. What would be the output of the following slice? Write it down then test the code to find out!

Language	Environment	Enter	Expected Output

Python	IDLE or ArcPro	`tree = ‘spruce’`
		`print(tree[1:3])`	

R	RStudio	`tree <- ‘spruce’`
		`tree[1:3]`	

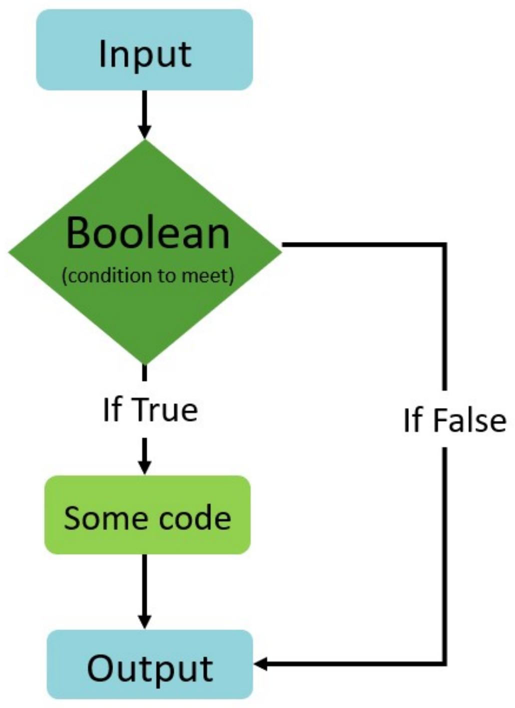
JavaScript	Google Earth Engine	**`var`** `tree = ee.String(‘tree’);`
		`print(FS.slice(start,stop))`	

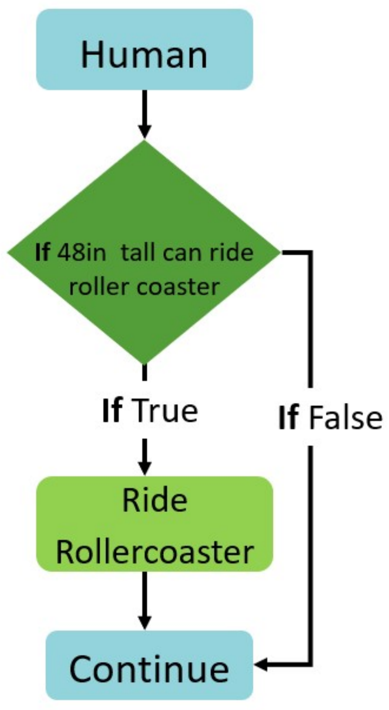
Part 2: Conditional Statements

Within a conditional statement are some lines of code that will only be executed if the condition you supply is met.

A. **If** statements

An **if** statement is the simplest conditional statement. The programmer will supply a condition that can be either true or false, often referred to as a Boolean. If the condition is met (condition = TRUE), then the code within the if statement is executed. Otherwise, nothing happens. This is visualized below:





1. Begin by creating several variables. Create **x** and set it equal to **5** and create **y** and set it equal to 7 using the syntax you learned in the previous exercise. If you need to, refer to the tables in exercise 2 for syntax.
2. Now begin to build a simple if statement. We're going to give the condition: **x** is less than **y**. If the condition is true, **y** will be divided by 2. As always, syntax will differ across languages. To see the differences, refer to the table below.

Language	Environment	Enter

Python	IDLE or ArcPy	`if x<y:`
		`        y=y/2`

R	RStudio	`if (x<y) {`
		`        y=y/2`
		`}`

JavaScript	Google Earth Engine	`y = ee.Algorithms.If(x.gt(y), y = y.divide(2), y = ee.Number(3));`

3. Because x *is* less than y (the condition is true), the if statement **will be** executed. To confirm that the y variable has been manipulated, **add** a print statement for the variable y.
4. After typing in the if statement and print statement, **Run the script**.
   i. Python: **Click** Run, then Run Module
   ii. R: **Highlight** the entire script and **Type** Ctrl+Enter or Ctrl+R
   iii. JavaScript: **Click** Run in the Code Editor

Language	Environment	Enter	Outputs	
Python 2	IDLE	`print(y)`	`3`	
Python 3	ArcPy	`print(y)`	`3.5`	
R	RStudio	`y`	`[1] 3.5`	
JavaScript	Google Earth Engine	`print(y)`	`3.5`	

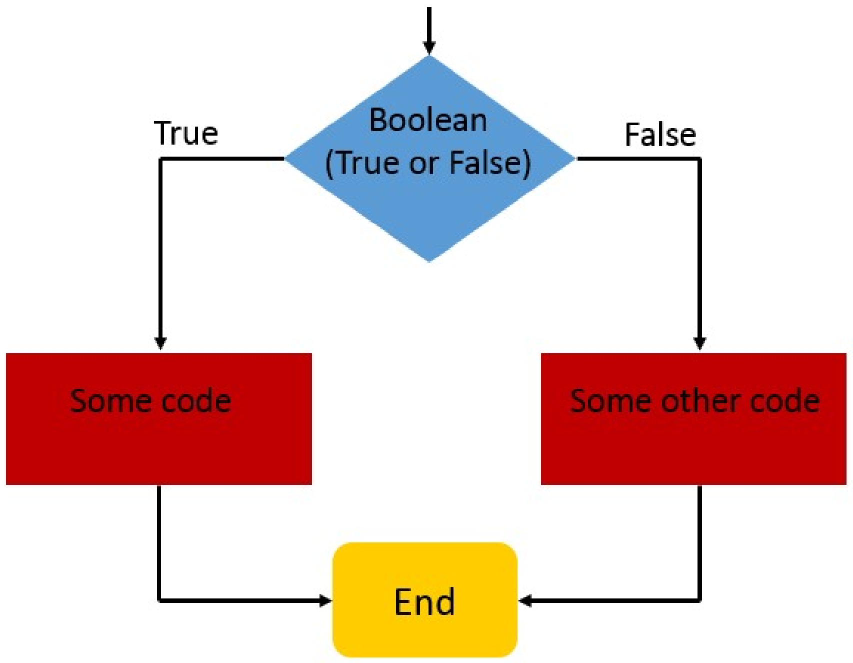
Note: *The output from Python is 3, rather than 3.5. Remember from exercise 2 that there are different types of numbers. In Python the variable y was set to 7, an integer, so math done on that variable is going to output an integer, and 3.5 is not an integer. To output 3.5 in python, the variable will need to be a floating point. If you set y equal to 7.0 in python and run the script again, it would have output 3.5*

5. To demonstrate the if statement working, **change** the variable y to 3. **Run the script**. The variable y will not have changed. This is because the condition (x<y) is no longer true, and therefore the if statement is not executed.

If statements can be very helpful in geospatial scripting to specify what you want the computer to do, and when. In exercise 1 you saw an example of an if statement making an elevation mask. Think of some other instances where having a conditional statement might be useful when writing a geospatial program.

B. **If Else** Statements

If statements are the simplest conditional statements. They execute code if a condition is true and do nothing if a condition is false. If else statements add an additional piece of code so that if a condition is true, some code will be executed, and if a condition is false, some other code will be executed. This can be visualized in the flowchart in figure 2.



1. To learn the structure of an if else statement you're going to begin by writing a simple one. In your script, **set** your variable x equal to 5 and y equal to 7.
2. The structure of an if else statement is very similar to a simple if statement, and the first part is identical. **Set** the "if" condition as x<y, just like in part A, and **set** the true code to y=y/2.
3. Now you need to add in the "else" code. The "else" code will be executed if x<y is false. In this example, the else code is going to be x=x*2. The syntax for else will be the same as for if, but it will still vary across languages. For a complete if else statement, see the table below.

Language	Environment	Enter

Python	IDLE or ArcPro	`if x<y:`
		`    y=y/2`
		`else:`
		`    x=x*2`

R	RStudio	`if (x<y) {`
		`    y=y/2`
		`}else{`
		`    x=x*2`
		`}`

JavaScript	Google Earth Engine	`var z = ee.Algorithms.If(x.gt(y), y.divide(2), x.multiply(2));`
		`print(z)`

4. Because x is less than y, the if statement will be executed. To confirm that the y variable has been manipulated, **issue** a print statement for the variable y by entering print(y). Then, **Run the Script**. Again, the computer should output the same results as it did in part A number 3, because x was less than y.

Note: *In JavaScript, print the variable z instead of x or y. The variable z will take the value of either y.divide(2) or x.multiply(2), depending on the evaluation of the conditional x.gt(y). Just like before, this is necessary because of the way that Earth Engine objects are computed.*

5. Now just as you did with the if statement in part A, **change** the variable y to 3, then **re-enter** the if else statement and **run** it again. Now, because x is no longer less than y, y will not be manipulated. Instead, the else code will be executed and x will be multiplied by 2.

These are simple examples, intended to show the structure of if and if else statements. A conditional statement that you write into a script will have the exact same structure, but you will create it to perform some geoprocessing task.

Part 3: Objects and Methods

Python, R, and JavaScript are languages that use objects. An **object** represents a collection of data which has specific properties and characteristics.

Methods are functions that are specific to objects.

Objects and methods can be quite confusing for a while so don’t worry if you don’t fully understand them.

A. Objects

1. Begin to understand objects is by examining a list or an array. Create the variable **x** and set it equal to the list **[12,7,99]**. If you need to remember the syntax of the language that you’re using, see Exercise 2 Part 1. *Remember to use ee.List if you're using JavaScript.*
2. Now think about ways that you can describe this variable. It has a length, min and max values, and an order. These are some of the object properties. The object has methods that can be used to examine or process the object.

B. Methods

You now have a List object. Let’s learn a few methods that you can use to pull information out of the object. The syntax of a method involves “calling” the method, then “passing” it arguments.

1. One property of a list object is length. To get the length use the length method, **len()**. Print the **len()** method to output the length

Language	Environment	Enter	Outputs
Python	IDLE or ArcPro	`print(len(x))`	`3`
R	RStudio	`length(x)`	`[1] 3`
JavaScript	Google Earth Engine	`print(x.length());`	`3`

This is an example of how methods can be used to get properties of an object. All methods follow the same basic structure: calling the method, then passing in the necessary arguments.

Note: If you're doing this exercise in **Python 2**, the method you're about to write uses the Python Module "Math".

What is a Module? A module is a code library. Many scripting languages such as Python and R support code libraries, where programmers can develop and share commonly used pieces of code as a special script or library file that can be referenced and loaded into the main program. These code libraries, called modules in python, extend the functionality of Python. These extra pieces of code in the modules are not available unless you specifically import the module. Similarly, R has code libraries called packages, rather than modules. JavaScript uses code libraries as well, but its math methods don't require additional modules to be imported. For more information on modules and packages, see the resources section below or the glossary

If you are working through this exercise in Python 2, you **must** type:

import math

in your script before you use math methods (best practice is to begin your script importing all the modules you will need, so type import math on the very first line of your script).

2. To find the maximum value of list **x** use the **maximum** method. Call **max()** and input list **x** as your argument. Print the method and run the script to see the output

Language	Environment	Enter	Outputs
Python	IDLE or ArcPro	`print(max(x))`	`99`
R	RStudio	`max(x)`	`[1] 99`
JavaScript	Google Earth Engine	`print(x.sort().get(-1));`	`99`

3. What if we wanted to order the list from smallest value to the largest value? We can use the **sort** method to order our list! We will save our ordered list to a new variable **y**

Language	**Environment**	**Enter**
Python	IDLE or ArcPro	`y = sorted(x)`
R	RStudio	`y <- sort(x)`
JavaScript	Google Earth Engine	`var y = x.sort();`

Print out variable **y** to see the ordered list. Suppose you had a set of remotely sensed imagery, or a table of field data that you needed to set in order by the date. The sort method may prove very useful.

4. Explore a variation of the sort method, sorting in reverse order. The sort method has an argument “reverse” that is by default equal to “false”, if we change that to “true” we will have a list in revered order.

Language	Environment	Enter
Python	IDLE or ArcPro	`y = sorted(y, reverse = True)`
R	RStudio	`y <- sort(y, decreasing = TRUE)`
JavaScript	Google Earth Engine	`var y = ee.List(y).sort().reverse();`

There are scores of useful methods that you can use in a script, but keep in mind they are specific to each object. Some useful methods are outlined in the glossary!

Much like words in a natural language, you will continue learning new, useful methods the more you program. Similarly, just like words in a natural language, some methods are more common and will become very familiar to you, others you may not learn until you need it for a specific purpose. It is more important to know where to look for new methods, than trying to memorize them all.

Part 4: Functions

Functions allow you to recall the same piece of code multiple times. Functions are ideal for repetitive tasks. Functions will make script maintenance easier if at any point you need to modify or update your script.

Up to this point, all the code that we have used has executed sequentially. That is, the computer reads through the code line by line and executes it as it goes along. Think of code that executes sequentially like a novel. You begin at page 1 and always read through line by line until the end of the book. Functions, however, can be thought of like an entry in an encyclopedia. If you needed to look up an entry, it would be extremely inefficient to read through the entire encyclopedia line by line until you reached the entry, you’re interested in. Instead, you just skip straight to the entry that you need. Similarly, a function is like a single entry. It stores all the information that you need, and can be looked up, or “called”, only when it must be used.

A. Writing a Function

We are going to calculate the area of a circle. Suppose you have a list of crop circle radii and need to calculate the total area of crop. This is a relatively simple equation so you could write it in each time, but why do the extra work if you don’t need to?

1. Create the variable **radius** and set it equal to **9**.

Language	Environment	Assign Variable Value
Python	IDLE or ArcPro	`radius=9`
R	RStudio	`radius=9 or radius <- 9`
JavaScript	Google Earth Engine	`var radius = ee.Number(9);`

Note: Before you move on writing your function, you must understand variable scope. Variables can exist in a global, and local scope. To this point, all the variables you have created are global variables. They exist "globally", meaning that no matter where you use them in your script, they exist, they can be used, and they can be changed. By contrast, a local variable does not exist everywhere in the script, rather they exist **only inside of a function**. In several steps you will create a local variable **A**. This means that outside the function, you will not be able to use or manipulate **A**. If you try to **print(A)**, you will get an error.

2. The basic structure of the function is below on the left, with our example of a circle area on the right:

```
<function name><arguments>              circleArea(radius)
      <body>                              area = pi*radius²
      <return>                            return(area)
```

The first line of the function must include the function name and the arguments (arguments are things that you must give to the function each time you run it). Name this function **circleArea** and make its arguments **r**

Language	Environment	Enter
Python	IDLE or ArcPro	`def circleArea(r):`
R	RStudio	`circleArea <- function(r){`
JavaScript	Google Earth Engine	`function circleArea(r){`

Note: In python, the first line of the function ends with a colon (**:**). In R and JavaScript the first line ends with an opening bracket (**{**). These brackets are supposed to enclose the function. When you go to a new line, RStudio and Google Earth Engine will probably add a closing bracket (**}**) for you. If they don't, make sure to add one yourself, otherwise the function will not work.

3. On the second line of the function **create** a new variable, **A**, and set it equal to the equation for the area of a circle (A=πr^2). These languages already know that pi=3.14159265… so you can simply type **pi** Look below for the syntax to create an exponent

Language	Environment	Enter
Python	IDLE or ArcPro	`A = math.pi*r**2`
R	RStudio	`A <- pi*r**2`
JavaScript	Google Earth Engine	`var A = ee.Number(Math.PI).multiply(r.pow(2));`

4. The last thing we need to include is a return. A return is what the function will give back when we call it in the script, in this case we want the function to give us the area of a circle, **A**

Language	Environment	Complete Function

Python	IDLE or ArcPro	`def circleArea(r):`
		`    A = math.pi*r**2`
		`    return A`

R	RStudio	`circleArea <- function(r){`
		`    A <- pi*r**2`
		`    return(A)`
		`}`

JavaScript	Google Earth Engine	`function circleArea(r){`
		`    var A =`
		`    ee.Number(Math.PI).multiply(r.pow(2));`
		`    return(A)`
		`}`

5. Let's use our function! Create variable **area** and set it equal to your function name, with the argument, **radius**

Language	Environment	Enter
Python	IDLE or ArcPro	`area = circleArea(radius)`
R	RStudio	`area <- circleArea(radius)`
JavaScript	Google Earth Engine	`var area = circleArea(radius);`

   Print the variable **area**. It will print the area of a circle with a radius of 9. Run the script

6. Recall from the earlier note the difference between global and local variables. **area** is a global variable. It exists in a wide scope, which means it can be manipulated outside the circle function, and we can print its value. The variable **A** that you created in the function is a local variable. It exists only inside the function, but nowhere else. Try printing **A** then running the script, what happens?

Functions should not be written sequentially in a script. It is best practice to write all the functions you need at the beginning of the script, then use them as you need them in the body of the script. After they're written, you won't ever need to write them again. This makes it easy to use equations many times, especially if the equation is a complicated one. It also means that once you've correctly programmed the code in a function, there isn't any chance that you'll accidentally write it incorrectly later in the script.

Part 5: Additional Resources

A. Conditional Statements

1. Python: If Else Statements
2. R
   i. How to use Statements in R
   ii. If Else Statements in R
3. Google Earth Engine: Conditionals in Google Earth Engine

B. Objects

1. Python: Classes and Objects in Python
2. R: Objects in R
3. JavaScript: GEE Objects and Methods Overview

C. Methods

1. Python: Python Methods, Functions, and Libraries
2. R: R Object System
3. JavaScript: Methods for Image Objects

D. Functions

1. Python: Software Carpentry Writing Functions
2. R: Software Carpentry Writing Functions in R
3. JavaScript: Functions and Mapping

Congratulations! You have completed this exercise. You have learned many of the skills necessary for writing your own script.

Google Earth Engine: Exercise 4

Python 2 with ArcGIS: Exercise 5

R : Exercise 6

Python 3 with ArcPro: Exercise 7