MFRE Data Analytics Workshop Series

Workshop 3: Python I

Krisha Lim
University of British Columbia | Master of Food and Resource Economics
November 8, 2021

Workshop preparation

☑ Make sure you have a Gmail account

☑ Upload and open the `Workshop3_Student.ipynb` file in Google Colab

Overview

Python I

- Google Colab
- Variable Types - String, Integer, Float, Boolean
- Lists, Dictionaries, Functions

Python II

- Importing libraries in Python
- Importing files to Google Colab
- Working with data and Python packages

Learning Outcomes

- Explain the value of learning Python
- Demonstrate how to work with Google Colab
- Define lists and dictionaries as it relates to Python
- Access and manipulate data in lists and dictionaries

Motivation

Why Python?

- Free
- Open-source
- Has a large community
- Rich ecosystem of third-party packages
- Widely used in industry and academia

Python and R

Python

- General-purpose, object oriented programming
- Can more easily import SQL tables and other data from web
- Well suited for machine learning at scale and integration with web applications

R

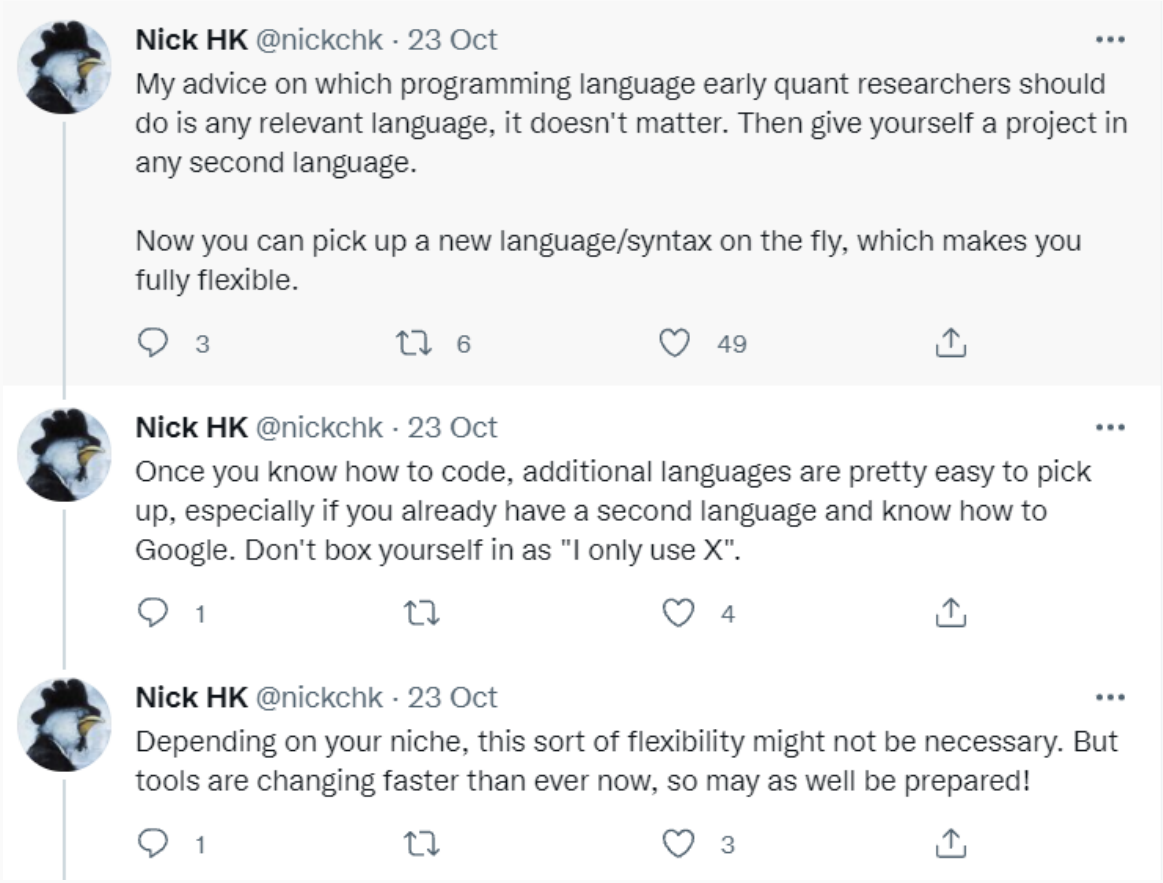
- Optimized for statistical analysis and data visualization
- Rich libraries for data cleaning, creating visualizations, training and evaluating machine learning and deep learning algorithms
- Strength of MFRE professors and team

[1] IBM

Python and R

> Increasingly, the question isn't which to choose, but how to make the best use of both programming languages for your specific use cases.
>
> --- IBM

Why learn one more language?



Nick HK @nickchk · 23 Oct

My advice on which programming language early quant researchers should do is any relevant language, it doesn't matter. Then give yourself a project in any second language.

Now you can pick up a new language/syntax on the fly, which makes you fully flexible.

3 6 49

Nick HK @nickchk · 23 Oct

Once you know how to code, additional languages are pretty easy to pick up, especially if you already have a second language and know how to Google. Don't box yourself in as "I only use X".

1 4

Nick HK @nickchk · 23 Oct

Depending on your niche, this sort of flexibility might not be necessary. But tools are changing faster than ever now, so may as well be prepared!

1 3

Python I

Google Colab

For this workshop and the Analytics course next semester, we will use Google Colab.

- All you need is a working Gmail account
- No need to install any software
- Works on Windows and MacOS
- Can easily share codes with colleagues

Keyboard shortcuts

- `Ctrl + Enter` to run
- `Ctrl + M + B` to insert code chunk
- `Ctrl + /` to comment out a code chunk (+ `Cmd` for Mac)

Participation

You are expected to fill in the Google Colab file.

At the end of the workshop, share this file to my UBC email.

Share -> Change from "Restricted" to "Anyone with the link" -> Share with my UBC email

Try it out

```
2 + 2
```

```
## 4
```

```
print("Hello World")
```

```
## Hello World
```

Strings, integers, and floats

In Python, the assignment operator is `=`.

To review the value of a variable, we can type the name of the variable into the interpreter and press Enter/Return.

However, to display output in a script, we have to use the `print` function.

```
text = "Data Carpentry"
number = 42
pi_value = 3.1415

text
```

```
## 'Data Carpentry'
```

```
print(number)
```

```
## 42
```

Strings, integers, and floats

Everything in Python has a type. To get the type of something, we can pass it to the built-in function `type`

- **Strings** hold sequences of characters, which can be letters, numbers, punctuation, etc.
- **Integers** are positive or negaitve whole numbers with no decimal points
- **Floats** are real numbers and can be written with a decimal point

```
type(text)
```

```
## <class 'str'>
```

```
type(number)
```

```
## <class 'int'>
```

```
type(pi_value)
```

```
## <class 'float'>
```

Operators

We can perform mathematical calculations in Python using basic operators `+, -, /, *, %`

```
2 + 2 # Addition
```

```
## 4
```

```
6 * 7 # Multiplication
```

```
## 42
```

```
2 ** 16 # Power
```

```
## 65536
```

```
13 % 5 # Modulo
```

```
## 3
```

Logic Operators

We can also use comparison and logic operators: `<`, `>`, `=`, `≠`, `⩽`, `⩾` and statements of identity such as `and`, `or`, `not`. The data type returned is called **boolean**.

```
3 > 4
```

```
## False
```

```
True and True
```

```
## True
```

```
True or False
```

```
## True
```

```
True and False
```

```
## False
```

Lists

Lists are the most fundamental data structure in Python. It is used to hold an ordered sequence of elements.

We build a list using **square brackets**.

```
country = ["canada", "usa", "china", "japan"]
gdp = [44100, 55700, 16200, 39300]
carbon = [15.3, 16.6, 7.06, 9.14]

country
```

```
## ['canada', 'usa', 'china', 'japan']
```

```
type(country)
```

```
## <class 'list'>
```

```
len(country)
```

```
## 4
```

Lists

Lists can also contain different data types. Lists can also contain lists.

```
example1 = ["canada", 15.3, "usa", 16.6, "china", 7.06, "japan", 9.14]

example2 = [["canada", 15.3],
            ["usa", 16.6],
            ["china", 7.06],
            ["japan", 9.14]]

print(example1)
```

```
## ['canada', 15.3, 'usa', 16.6, 'china', 7.06, 'japan', 9.14]
```

```
print(example2)
```

```
## [['canada', 15.3], ['usa', 16.6], ['china', 7.06], ['japan', 9.14]]
```

```
print("length of example2:", len(example2))
```

```
## length of example2: 4
```

Lists

Each element in a list can be accessed by an index. **Note that Python indexes start with 0 instead of 1.**

```
example1 = ["canada", 15.3, "usa", 16.6, "china", 7.06, "japan", 9.14]

example2 = [["canada", 15.3],
            ["usa", 16.6],
            ["china", 7.06],
            ["japan", 9.14]]

print("China's carbon emissions is", example1[5])
```

```
## China's carbon emissions is 7.06
```

```
print("Japan's carbon emissions is", example2[3][1])
```

```
## Japan's carbon emissions is 9.14
```

Lists

We can also slice lists, or select multiple elements from a list. In general, where it starts is included, where it ends is excluded, i.e., `[start_index:end_index]`.

```
example1 = ["canada", 15.3, "usa", 16.6, "china", 7.06, "japan", 9.14]

print(example1[2:6])
```

```
## ['usa', 16.6, 'china', 7.06]
```

```
print(example1[:4])
```

```
## ['canada', 15.3, 'usa', 16.6]
```

```
print(example1[4:])
```

```
## ['china', 7.06, 'japan', 9.14]
```

How do you print China's and Japan's emission only using data from `example2`?

Lists

To print the index of a certain value in a list, we can use the `.index` function

```
print("The index of 7.06 is", example1.index(7.06))
```

```
## The index of 7.06 is 5
```

Lists

We can also change the value of the elements in a list. For example, we can change the value of Canada's emission from 15.3 to 16.3.

```
example1 = ["canada", 15.3, "usa", 16.6, "china", 7.06, "japan", 9.14]

example1[1] = 16.3

print(example1)
```

```
## ['canada', 16.3, 'usa', 16.6, 'china', 7.06, 'japan', 9.14]
```

Lists

If we want to replace the values in Canada's entries with South Korea's emissions:

```
example1 = ["canada", 15.3, "usa", 16.6, "china", 7.06, "japan", 9.14]

example1[0:2] = ["south korea", 12.9]

print(example1)
```

```
## ['south korea', 12.9, 'usa', 16.6, 'china', 7.06, 'japan', 9.14]
```

Lists

If we want to add Canada to the list again:

```
example1_can = example1 + ["canada", 15.3]

print(example1_can)
```

```
## ['south korea', 12.9, 'usa', 16.6, 'china', 7.06, 'japan', 9.14, 'canada', 15.3]
```

Lists

We can delete elements using the `del` function. For example, we can delete South Korea from our list.

```
example1_can_append = ["south korea", 12.9, "usa", 16.6,
                       "china", 7.06, "japan", 9.14, "canada", 15.3]

del(example1_can_append[0:2])
print(example1_can_append)
```

```
## ['usa', 16.6, 'china', 7.06, 'japan', 9.14, 'canada', 15.3]
```

Lists

We can also take the sum of elements in a list using the `sum` function.

```
carbon = [15.3, 16.6, 7.06, 9.14]

print(sum(carbon))
```

```
## 48.1
```

Lists

We can use a `for` loop to access the elements in a list one at a time:

```
carbon = [15.3, 16.6, 7.06, 9.14]

for emissions in carbon:
  print(emissions)
```

```
## 15.3
## 16.6
## 7.06
## 9.14
```

Dictionaries

Another important data type is called the **dictionary**. It is a container that holds pairs of objects - keys and values.

Let's say you want to track the carbon emissions of each of these four countries.

```
country = ['canada', 'usa', 'china', 'japan']
carbon = [15.3, 16.6, 7.06, 9.14]

usa_index = country.index('usa')
print(usa_index)
```

```
## 1
```

```
print('usa emissions: ', carbon[usa_index])
```

```
## usa emissions:  16.6
```

Dictionaries

This method, however, can be time consuming especially if you have a very big dictionary.

Dictionaries allow us to connect each country directly to its emissions.

To create a dictionary, we use **curly brackets**.

Inside these curly brackets are a bunch of **key:value** pairs, where keys are countries and values are carbon emissions.

Dictionaries

Dictionaries work a lot like lists, except that you index them with **keys** instead of an index number.

To access a value of a given key, simply type in `data['key']`.

```
emissions = {
  'canada': 15.3,
  'usa':16.6,
  'china':7.06,
  'japan':9.14
}

emissions['china']
```

```
## 7.06
```

Dictionaries

For the lookup to work properly, keys in the dictionary must be unique.

In this example, the last key:value pair for China was stored in the dictionary.

```
emissions = {
  'canada': 15.3,
  'usa':16.6,
  'china':7.06,
  'japan':9.14,
  'china':15
}


print(emissions)
```

```
## {'canada': 15.3, 'usa': 16.6, 'china': 15, 'japan': 9.14}
```

Dictionaries

To add an item to a dictionary, we assign a value to a new key.

```
emissions['south korea'] = 12.9

print(emissions)
```

```
## {'canada': 15.3, 'usa': 16.6, 'china': 15, 'japan': 9.14, 'south korea': 12.9}
```

We can also update the value of an existing key

```
emissions['south korea'] = 15

print(emissions)
```

```
## {'canada': 15.3, 'usa': 16.6, 'china': 15, 'japan': 9.14, 'south korea': 15}
```

We use the `del` function to delete a key:value pair.

```
del(emissions['south korea'])

print(emissions)
```

```
## {'canada': 15.3, 'usa': 16.6, 'china': 15, 'japan': 9.14}
```

Dictionaries

Dictionaries can contain key:value pairs where the values are again dictionaries, just like lists can also contain lists.

In this example, the keys are still country names, but the values are GDP and carbon emissions.

```
data = {
  'canada': {'gdp':44100,
             'carbon':15.3},
  'usa': {'gdp':55700,
          'carbon':16.6}
}

# To print Canada's emissions
print(data['canada']['carbon'])
```

```
## 15.3
```

Dictionaries

To add China's GDP and emissions to the `data` dictionary we just created

```
data = {
  'canada': {'gdp':44100,
             'carbon':15.3},
  'usa': {'gdp':55700,
          'carbon':16.6}
}

china_data = {'gdp':16200, 'carbon':7.06}

#Add China to data
data['china'] = china_data
```

Dictionaries

Using `for` loops with dictionaries is a little more complicated.

1) Method 1

```
data = {'canada': {'gdp': 44100, 'carbon': 15.3},
        'usa': {'gdp': 55700, 'carbon': 16.6},
        'china': {'gdp': 16200, 'carbon': 7.06}}

for key, value in data.items():
  print(key, "→", value)
```

```
## canada → {'gdp': 44100, 'carbon': 15.3}
## usa → {'gdp': 55700, 'carbon': 16.6}
## china → {'gdp': 16200, 'carbon': 7.06}
```

Dictionaries

2) Method 2

```
data = {'canada': {'gdp': 44100, 'carbon': 15.3},
        'usa': {'gdp': 55700, 'carbon': 16.6},
        'china': {'gdp': 16200, 'carbon': 7.06}}

for key in data.keys():
  print(key, "→", data[key])
```

```
## canada → {'gdp': 44100, 'carbon': 15.3}
## usa → {'gdp': 55700, 'carbon': 16.6}
## china → {'gdp': 16200, 'carbon': 7.06}
```

Functions

A **function** is a code chunk that runs when is called. You can pass inputs called **parameters** or **arguments** into a function and get a result back.

Defining a section of code as a function in Python is done using the `def` keyword.

Functions

In the example below, we have a function called `my_function` that takes one argument `fname`. When `my_function` is called, we pass along a first name, which is used inside the function to print the student name.

```
def my_function(fname):
  print("Hello " + fname + ". Welcome to class today!")

my_function("Krisha")
```

```
## Hello Krisha. Welcome to class today!
```

```
my_function("Janelle")
```

```
## Hello Janelle. Welcome to class today!
```

[1] W3 Schools Tutorial

Functions

By default, a function must be called with the correct number of arguments. Meaning, if your function expects 2 arguments, you have to call the function with 2 arguments, not more and not less.

```
def my_function(fname, lname):
  print("Hello " + fname + " " + lname + "!")

my_function("Krisha", "Lim")
```

```
## Hello Krisha Lim!
```

Try running this code. What do you get?

```
my_function("Janelle")
```

[1] W3 Schools Tutorial

Functions

Here is an example of a function named `add_function` that takes two arguments and returns their sum:

```
def add_function(a, b):
  result = a + b
  print(result)

add_function(20, 22)
```

```
## 42
```

[1] W3 Schools Tutorial

Case Study

Kim has been asked to use Python to analyze the following dataset.

	Jan		Feb		Mar	
Products	Total Quantity	Quantity Rejected	Total Quantity	Quantity Rejected	Total Quantity	Quantity Rejected
Oranges	3000	1400	6000	3500	12000	6700
Bananas	500	700	3000	1750	15000	3350

Specifically, her tasks are:

- Create one or more Python "dictionaries" for all data in the above table (Hint: You can use nested dictionaries for fruits and quantity/rejected, and use lists to hold the values)
- Print fruit dictionary
- Print the sum of all "Orange" Total Quantity and sum of all "Orange" Quantity Rejected
- Calculate and print the rejection rate
- Calculate and print the maximum of "Orange" Total Quantity (Hint: use `max()` function)

Case Study - Suggested Solutions

- Create fruit dictionary

```
fruit = {'oranges': {'quantity': [3000, 6000, 12000],
  'rejected': [1400, 3500, 6700]},
  'bananas': {'quantity': [500, 3000, 15000],
    'rejected': [700, 1750, 3350]}}
```

- Print fruit dictionary

```
print(fruit)
```

Case Study - Suggested Solutions

- Print the sum of all "Orange" Total Quantity and sum of all "Orange" Quantity Rejected

```
print("Quantity = ", sum(fruit['oranges']['quantity']))
```

```
## Quantity =  21000
```

```
print("Rejected = ", sum(fruit['oranges']['rejected']))
```

```
## Rejected =  11600
```

Case Study - Suggested Solutions

- Calculate and print the rejection rate

```
orange_totalqty = sum(fruit['oranges']['quantity'])
orange_totalrej = sum(fruit['oranges']['rejected'])
orange_rejrate = orange_totalrej / orange_totalqty

# use round() to round to 2 decimal places only
print("Orange rejection rate = ", round(orange_rejrate, 2))
```

```
## Orange rejection rate =  0.55
```

Case Study - Suggested Solutions

- Calculate and print the maximum of "Orange" Total Quantity (Hint: use `max()` function)

```
fruit = {'oranges': {'quantity': [3000, 6000, 12000],
  'rejected': [1400, 3500, 6700]},
  'bananas': {'quantity': [500, 3000, 15000],
    'rejected': [700, 1750, 3350]}}

print("Orange max qty =", max(fruit['oranges']['quantity']))
```

```
## Orange max qty = 12000
```

Reminder

- Share your Colab file to my UBC email.

Share -> Change from "Restricted" to "Anyone with the link" -> Share with my UBC email

Recap

- Explain the value of learning Python
- Demonstrate how to work with Google Colab
- Define lists and dictionaries as it relates to Python
- Access and manipulate data in lists and dictionaries

Python II

- Next Monday, November 15
- Topics:
  - Importing libraries in Python
  - Importing files to Google Colab
  - Working with data and Python packages

References

- Data Analysis and Visualization in Python for Ecologists by Data Carpentry