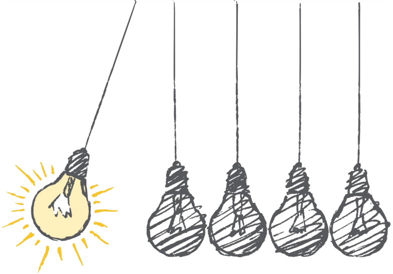
Introduction to Python for Stata Users



Luis Eduardo San Martin

Development Impact Evaluation (DIME)
The World Bank

Overview

Introduction

Getting started

Python variables

Python basic data types

Python basic syntax

Annex

Introduction

Introduction

- This session will introduce you to the basics of Python
- In the end we will apply this to a web scraping exercise
- After this session, you’ll be able to write and review **basic** Python code
- This session does not include how to use datasets in Python – instead it will focus on the fundamental building blocks to everything in Python, data types

Introduction - Python for Stata users

- There are many great Python courses available for free on the internet – so why is DIME Analytics making yet another one?
- This session makes two assumptions not common among the courses already available:
  - We assume that you will use Python for research and not computer science
  - We assume that you are coming from a Stata background
- Many concepts will be explained by referencing concepts in Stata

Introduction - Why Python if I already use Stata?

- Versatility: you can solve almost any programming task with Python:
  - Web scraping, text analysis, web applications to retrieve data, machine learning
- Much bigger user base
- Python is open source and free to use!
- Since it's open source it is easier to run everywhere – for example on big data servers

However, a big part of the user base does not do research or data science, and libraries for some less frequently used statistical operation have not yet been developed

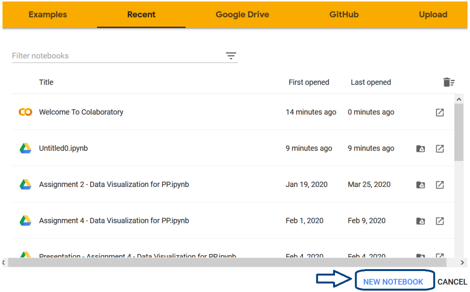
Getting started

Getting started

- We'll use Google Colab for this session: https://colab.research.google.com
- Colab is similar to a Google doc for coding, and it runs Python by default

Getting started - Colab

- Go to https://colab.research.google.com
- Click on `NEW NOTEBOOK` if you're already logged in, or go to `File > New notebook` if you're not



Title	First opened	Last opened
Welcome To Colaboratory	14 minutes ago	0 minutes ago
Untitled0.ipynb	9 minutes ago	9 minutes ago
Assignment 2 - Data Visualization for PP.ipynb	Jan 19, 2020	Mar 25, 2020
Assignment 4 - Data Visualization for PP.ipynb	Feb 1, 2020	Feb 9, 2020
Presentation - Assignment 4 - Data Visualization for PP.ipynb	Feb 4, 2020	Feb 4, 2020

Figure 1: Do this if you're already logged in

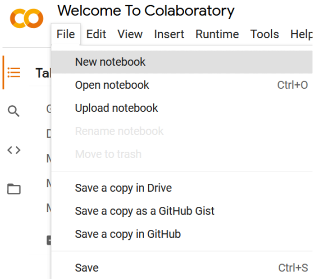
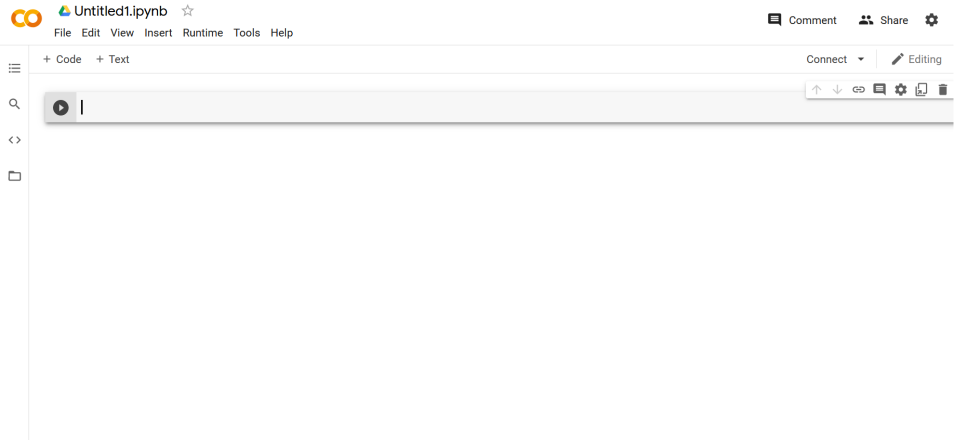


Figure 2: Do this if you're not – you'll be prompted to log in

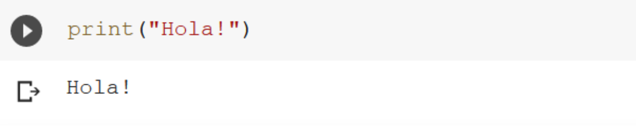
Getting started - Colab

You should end up with something like this in your browser:



Getting started - Colab

- Colab organizes code in blocks – each block is like its own script
- To run the code in a block, click the ▶ symbol or press `Ctrl` + `Enter`

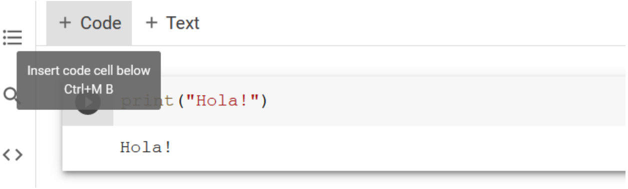


```
print("Hola!")
```

```
Hola!
```

Getting started - Colab

- Click on `+ Code` to add new blocks of code



```
print("Hola!")
```

```
Hola!
```

Important: Code blocks are a feature specific to Colab. Most Python distributions don't have this feature

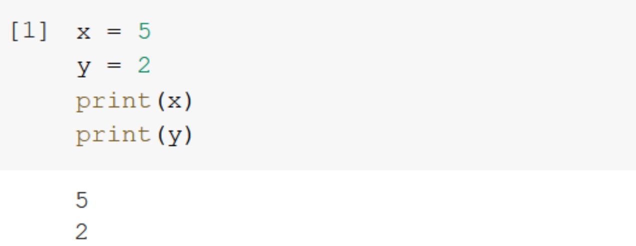
Python variables

Python variables

- In Stata, variables are columns of a dataframe
- In Python, variables are everything that we define with a name to be referenced – more similar to Stata's locals or globals (macros)
- Nonetheless, while macros in Stata are ”nice to have” and useful, variables in Python are the building block of everything and you cannot write code without them
- Variables are also broader than locals, globals or columns in Stata; for example, functions and datasets can be a variable in Python

Python variables

Just like in Stata, in Python we use the = operator to create variables

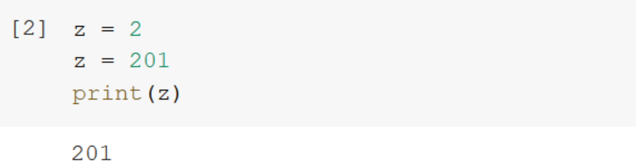


```
[1]  x = 5
     y = 2
     print(x)
     print(y)
```

```
     5
     2
```

Python variables

This also works when we’re trying to replace an existing variable



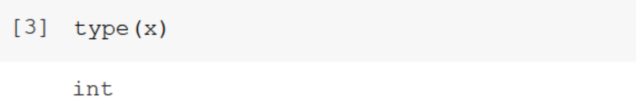
```
[2]  z = 2
     z = 201
     print(z)
```

```
201
```

Python basic data types

Python basic data types

- Every Python variable has a data type



```
[3]  type(x)

     int
```

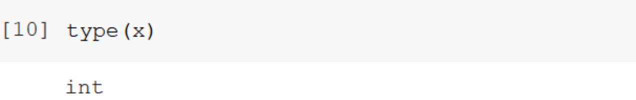
- Today we will cover the most basic data types: int/float (numbers), strings, booleans and lists
- Variables in Python do more than just store data. They provide operations related to their data type, for example: add and remove item from a list, make a string upper case, etc.

Python - more on data types

- Python has thousands of other data types
- This is because users can build their own data types based on the built-in types – you will frequently use such data types (and we'll use some of them later today)
- For example, a dataset in Python is a variable from the `pandas dataset` type, a custom data type implemented by the Python community
- All of these custom data types store data and provide built-in functionality specially implemented for the intended context

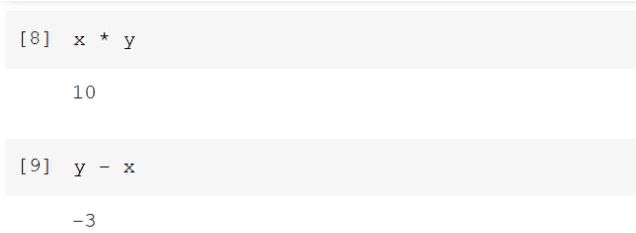
Python basic data types - `int`

The `x` and `y` variables we just defined have the data type `int`



```
[10]  type(x)

      int
```

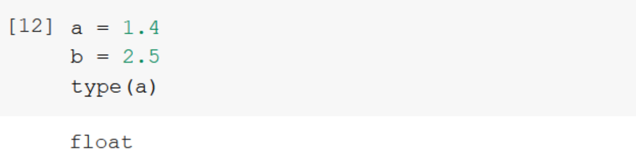
`int` variables are integer numbers. We can do mathematical operations with them



```
[8]  x * y

     10

[9]  y - x

     -3
```

Python basic data types - `float`

`float` variables, on the other hand, represent real numbers – we can do mathematical operations with floats as well

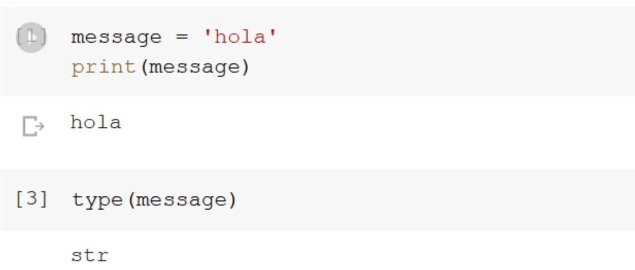


```
[12] a = 1.4
     b = 2.5
     type(a)

     float
```

- Python is what's called ”*dynamically typed*”, which means that you do not need to indicate what data type you want
- It detects when a variable is an integer, floating point (decimal number), text, etc. as long as it is a built-in data type.

Python basic data types - `str`

`str` variables are strings with text



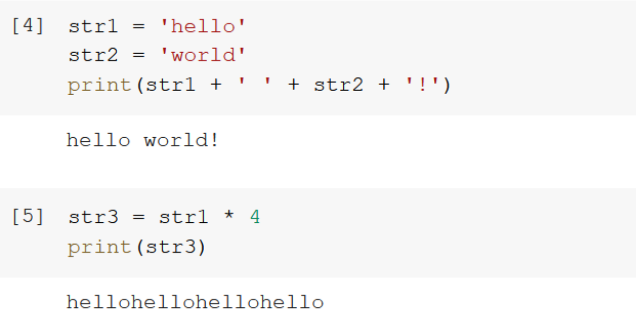
```
[1]  message = 'hola'
     print(message)

     hola
```

```
[3]  type(message)

     str
```

Note: A variable can be used across code blocks – this is common in all notebook styled python interfaces, like Colab

Python basic data types - `str`

Python allows two types of ”mathematical” operations with `str`: + and ∗

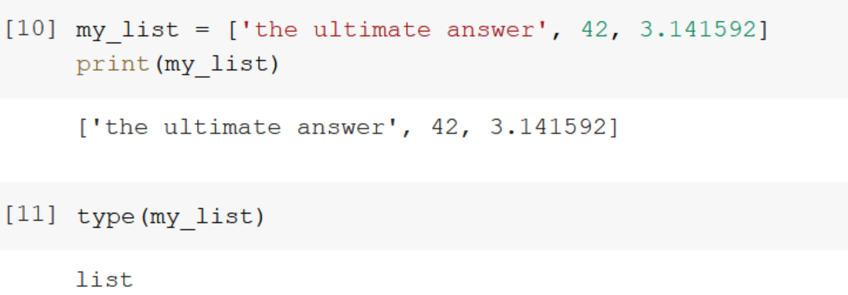


```
[4]  str1 = 'hello'
     str2 = 'world'
     print(str1 + ' ' + str2 + '!')

     hello world!
```

```
[5]  str3 = str1 * 4
     print(str3)

     hellohellohellohello
```

Python basic data types - Lists

- A list is a variable that groups other variables
- Lists can have different data types in them at the same time. They can even include other lists!
- Lists are defined enclosed in brackets and separating its values with commas

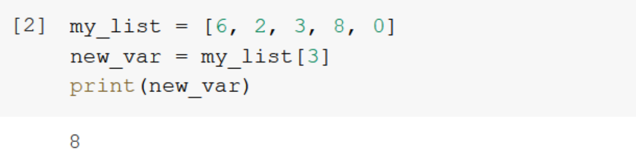


```
[10] my_list = ['the ultimate answer', 42, 3.141592]
     print(my_list)

     ['the ultimate answer', 42, 3.141592]
```

```
[11] type(my_list)

     list
```

Python basic data types - Lists

We can index lists

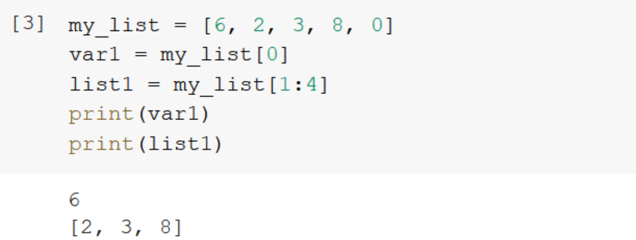


```
[2]  my_list = [6, 2, 3, 8, 0]
     new_var = my_list[3]
     print(new_var)

     8
```

Important: Python starts indexing at zero, not at one

Python basic data types - Lists

We can subset lists



```
[3]  my_list = [6, 2, 3, 8, 0]
     var1 = my_list[0]
     list1 = my_list[1:4]
     print(var1)
     print(list1)
```

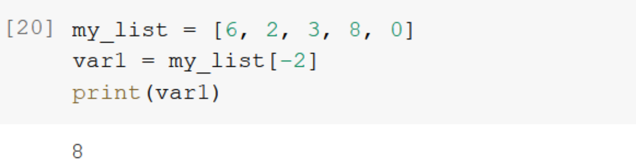
```
6
[2, 3, 8]
```

Important: When subsetting a list with `[a:b]`, Python will include the element at position `a` **but will exlude the one at position** `b`

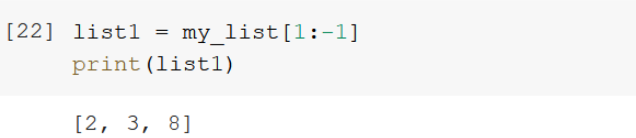
hence `my_list[1:4]` returns the elements at positions 1, 2, 3

Python basic data types - Lists

We can also use negative indices: they represent the elements of a list starting by the end



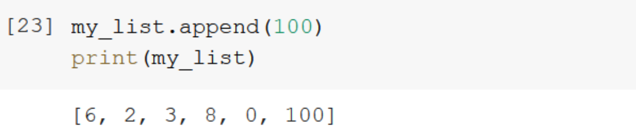
```
[20] my_list = [6, 2, 3, 8, 0]
     var1 = my_list[-2]
     print(var1)

     8
```



```
[22] list1 = my_list[1:-1]
     print(list1)

     [2, 3, 8]
```

Python basic data types - Lists

To add new elements to existing lists, we use `.append()`

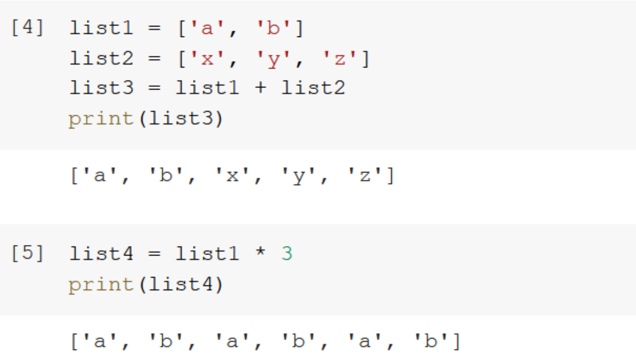


```
[23] my_list.append(100)
     print(my_list)

     [6, 2, 3, 8, 0, 100]
```

Note that this will modify our list variable in-place – it's not necessary to define the result as a new variable with = when we use `.append()`

Python basic data types - Lists

We can use the + and ∗ operators with lists

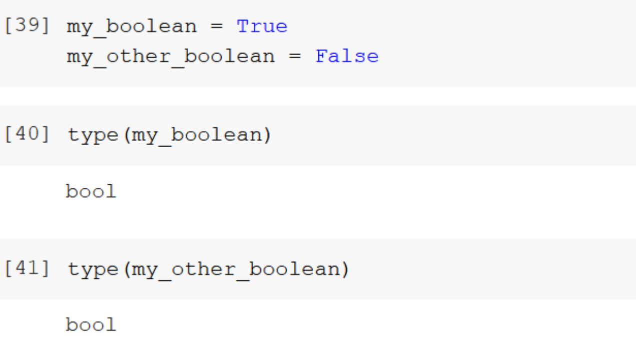


```
[4]  list1 = ['a', 'b']
     list2 = ['x', 'y', 'z']
     list3 = list1 + list2
     print(list3)

     ['a', 'b', 'x', 'y', 'z']
```

```
[5]  list4 = list1 * 3
     print(list4)

     ['a', 'b', 'a', 'b', 'a', 'b']
```

Python basic data types - Booleans

Booleans (`bool`) are variables representing boolean values – either `True` or `False`



```
[39]  my_boolean = True
      my_other_boolean = False
```

```
[40]  type(my_boolean)
```

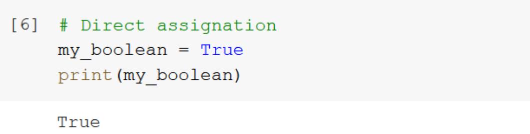
```
      bool
```

```
[41]  type(my_other_boolean)
```

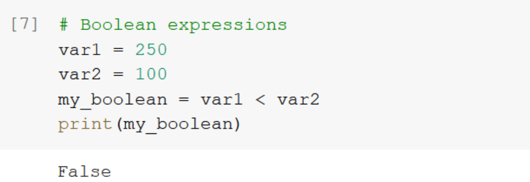
```
      bool
```

Python basic data types - Booleans

- We can create booleans by direct assignation or with boolean expressions
- When using direct assignation, Python recognizes booleans when they are written without quotes and with the first character in uppercase and the rest in lowercase



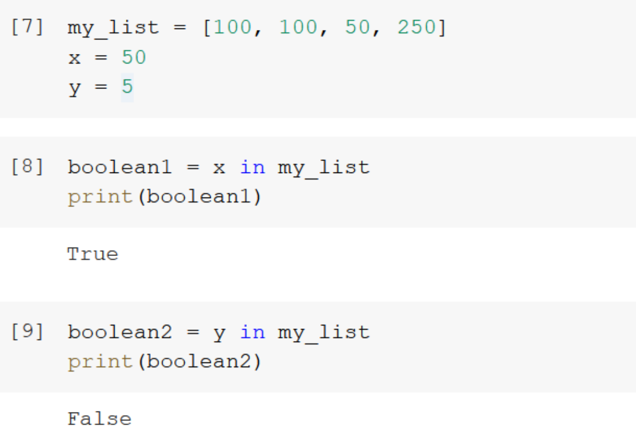
```
[6]  # Direct assignation
     my_boolean = True
     print(my_boolean)

     True
```



```
[7]  # Boolean expressions
     var1 = 250
     var2 = 100
     my_boolean = var1 < var2
     print(my_boolean)

     False
```

Python basic data types - Booleans

Some operators for boolean expressions are ==, >, >=, <, <=, and `in` (to check if an element is part of a list)



```
[7]  my_list = [100, 100, 50, 250]
     x = 50
     y = 5
```

```
[8]  boolean1 = x in my_list
     print(boolean1)
```

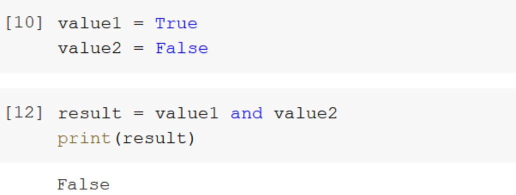
```
True
```

```
[9]  boolean2 = y in my_list
     print(boolean2)
```

```
False
```

Python basic data types - Booleans

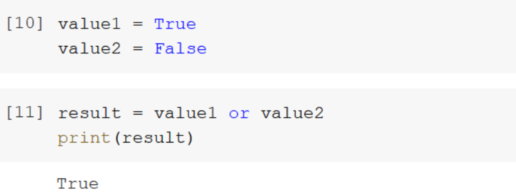
We can do logical operations with booleans using `and`, `or`



```
[10] value1 = True
     value2 = False
```

```
[12] result = value1 and value2
     print(result)
```

```
     False
```



```
[10] value1 = True
     value2 = False
```

```
[11] result = value1 or value2
     print(result)
```

```
     True
```

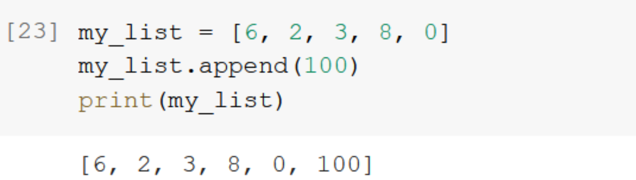
Python basic data types

- Until now, we've reviewed what Python variables and basic data types are
- Importantly, these are the building blocks of everything you do in Python
- It is simply impossible to do perform any task if you do not know how to work with the basic data types first

Python basic syntax

Basic syntax - Attributes

- Attributes are very often used when programming in Python
- They do one of two things:
  1. Attributes transform a variable in-place
     - For example: `.append()`, an attribute of list variables



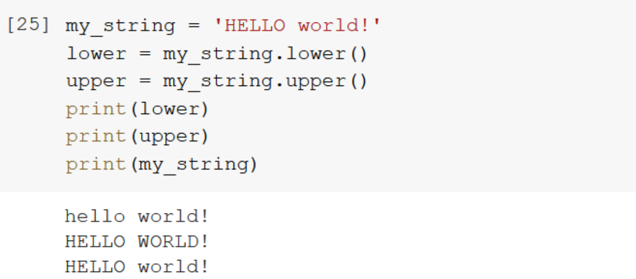
```
[23] my_list = [6, 2, 3, 8, 0]
     my_list.append(100)
     print(my_list)
```

```
     [6, 2, 3, 8, 0, 100]
```

Basic syntax - Attributes

2. Other attributes, by contrast, return a transformation of a variable without modifying the original

- For example: `.lower()` and `upper()`, attributes of string variables



```
[25] my_string = 'HELLO world!'
     lower = my_string.lower()
     upper = my_string.upper()
     print(lower)
     print(upper)
     print(my_string)
```

```
     hello world!
     HELLO WORLD!
     HELLO world!
```

Basic syntax - Attributes

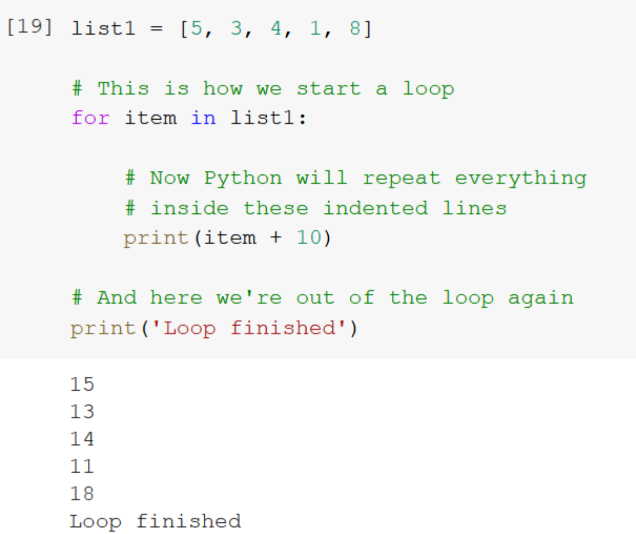
- Each data type has specific attributes. They relate to the built-in functionalities each data type has
- The syntax of attributes is *almost* always:

  ```
  VARIABLE_NAME.ATTRIBUTE_NAME(INPUTS_IF_ANY)
  ```

Basic syntax - Looping

- Many data types in Python belong to a group called iterables – variables you can loop through
- Lists are the most commonly used iterable: if we put a list in a loop, Python will loop through every one of its elements
- `int` and `float` are examples of non-iterable data types

Basic syntax - Looping



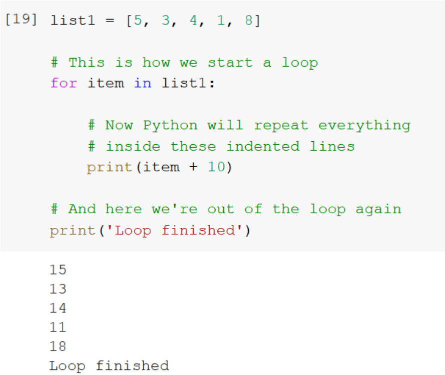
```
[19] list1 = [5, 3, 4, 1, 8]

     # This is how we start a loop
     for item in list1:

         # Now Python will repeat everything
         # inside these indented lines
         print(item + 10)

     # And here we're out of the loop again
     print('Loop finished')
```

```
     15
     13
     14
     11
     18
     Loop finished
```

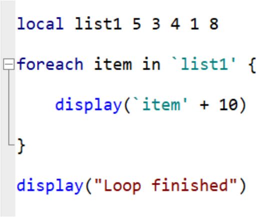
Basic syntax - Looping

Important:

Python knows what is inside the loop and where it ends with an indentation space – it works similar to the { } symbols you use to open and close a loop in Stata



```
[19] list1 = [5, 3, 4, 1, 8]

     # This is how we start a loop
     for item in list1:

         # Now Python will repeat everything
         # inside these indented lines
         print(item + 10)

     # And here we're out of the loop again
     print('Loop finished')

     15
     13
     14
     11
     18
     Loop finished
```

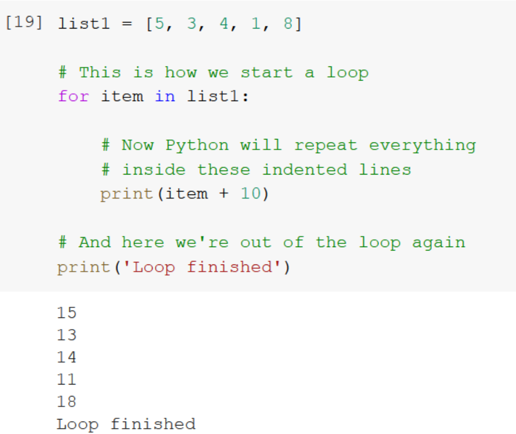


```
local list1 5 3 4 1 8

foreach item in `list1' {

    display(`item' + 10)

}

display("Loop finished")
```

Basic syntax - Looping

Important:

- Indentation can have two or four spaces depending on your Python interface. In any case, you can also press the `tab` key to create indented space
- If you ever run the script of a colleague who uses different indentation, Python will automatically know the correct one. All that matters is that indentation is consistent within the same script

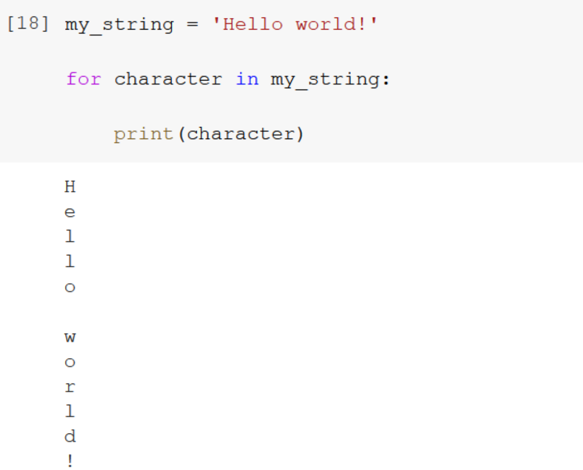


```
[19] list1 = [5, 3, 4, 1, 8]

     # This is how we start a loop
     for item in list1:

         # Now Python will repeat everything
         # inside these indented lines
         print(item + 10)

     # And here we're out of the loop again
     print('Loop finished')
```

```
15
13
14
11
18
Loop finished
```

Basic syntax - Looping

Strings are also iterables: Python loops through every character with them



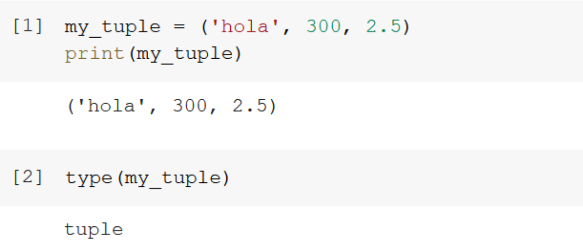
```
[18] my_string = 'Hello world!'

     for character in my_string:

         print(character)
```

```
H
e
l
l
o
 
w
o
r
l
d
!
```

Annex

Python basic data types - Tuples

Tuples are lists of variables. They are defined in parentheses and separate their elements by commas.

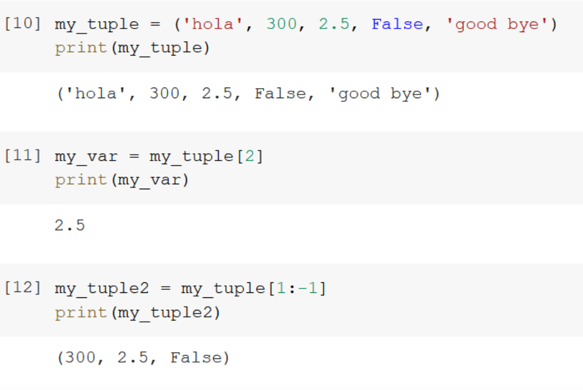


```
[1]  my_tuple = ('hola', 300, 2.5)
     print(my_tuple)

     ('hola', 300, 2.5)
```

```
[2]  type(my_tuple)

     tuple
```

Python basic data types - Tuples

Tuples are very similar to lists in that both use indices and subsets



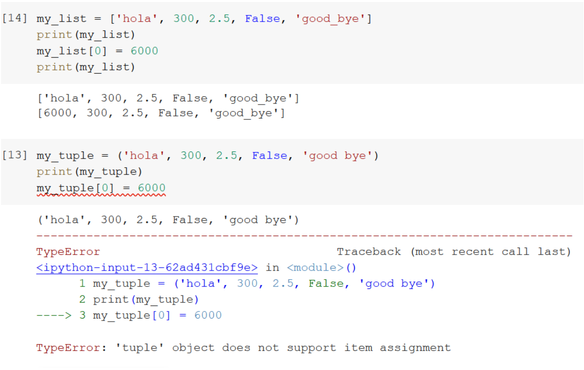
```
[10] my_tuple = ('hola', 300, 2.5, False, 'good bye')
     print(my_tuple)

     ('hola', 300, 2.5, False, 'good bye')
```

```
[11] my_var = my_tuple[2]
     print(my_var)

     2.5
```

```
[12] my_tuple2 = my_tuple[1:-1]
     print(my_tuple2)

     (300, 2.5, False)
```

Python basic data types - Tuples

The crucial difference between them is that tuples are inmutable: once defined, we can’t add new elements to them or replace the existing ones

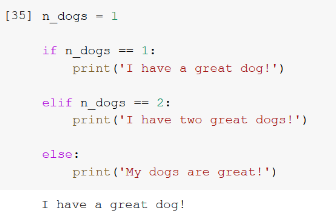


```
[14] my_list = ['hola', 300, 2.5, False, 'good_bye']
     print(my_list)
     my_list[0] = 6000
     print(my_list)

     ['hola', 300, 2.5, False, 'good_bye']
     [6000, 300, 2.5, False, 'good_bye']
```

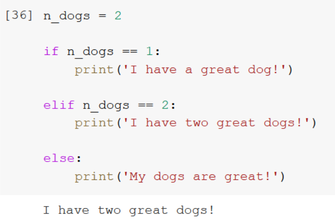
```
[13] my_tuple = ('hola', 300, 2.5, False, 'good bye')
     print(my_tuple)
     my_tuple[0] = 6000

     ('hola', 300, 2.5, False, 'good bye')
     ---------------------------------------------------------------------------
     TypeError                                 Traceback (most recent call last)
     <ipython-input-13-62ad431cbf9e> in <module>()
           1 my_tuple = ('hola', 300, 2.5, False, 'good bye')
           2 print(my_tuple)
     ----> 3 my_tuple[0] = 6000

     TypeError: 'tuple' object does not support item assignment
```

Basic syntax - Conditional expressions

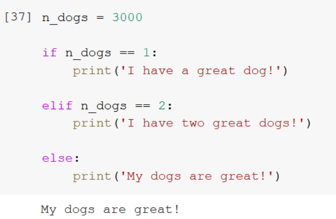
Conditional expressions: `if`, `elif`, and `else` are used to define conditional operations. They also use idented space



```
[35] n_dogs = 1

     if n_dogs == 1:
         print('I have a great dog!')

     elif n_dogs == 2:
         print('I have two great dogs!')

     else:
         print('My dogs are great!')

     I have a great dog!
```



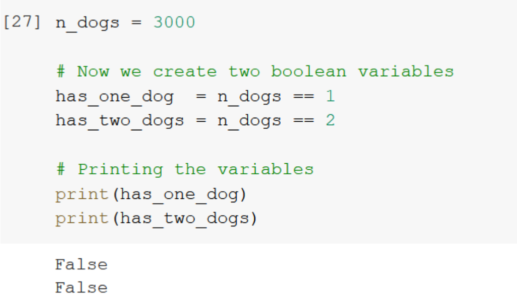
```
[36] n_dogs = 2

     if n_dogs == 1:
         print('I have a great dog!')

     elif n_dogs == 2:
         print('I have two great dogs!')

     else:
         print('My dogs are great!')

     I have two great dogs!
```



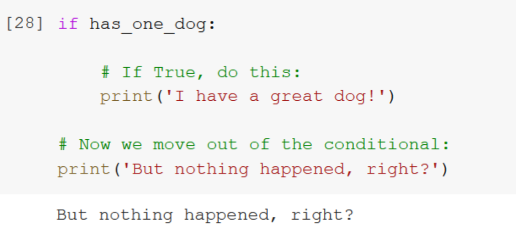
```
[37] n_dogs = 3000

     if n_dogs == 1:
         print('I have a great dog!')

     elif n_dogs == 2:
         print('I have two great dogs!')

     else:
         print('My dogs are great!')

     My dogs are great!
```

Basic syntax - Conditional expressions

- Instead of a boolean expression we can use a boolean value with `if` or `elif`
- `if` doesn't necessarily need to be used with `elif` of with `else`, we can use it alone



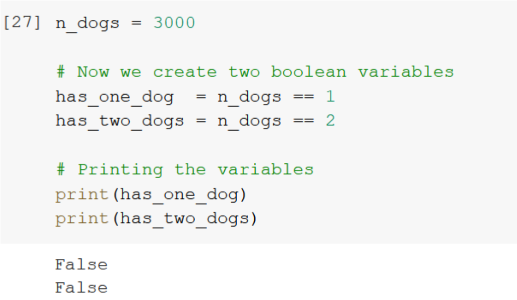
```
[27] n_dogs = 3000

     # Now we create two boolean variables
     has_one_dog  = n_dogs == 1
     has_two_dogs = n_dogs == 2

     # Printing the variables
     print(has_one_dog)
     print(has_two_dogs)

     False
     False
```



```
[28] if has_one_dog:

         # If True, do this:
         print('I have a great dog!')

     # Now we move out of the conditional:
     print('But nothing happened, right?')

     But nothing happened, right?
```

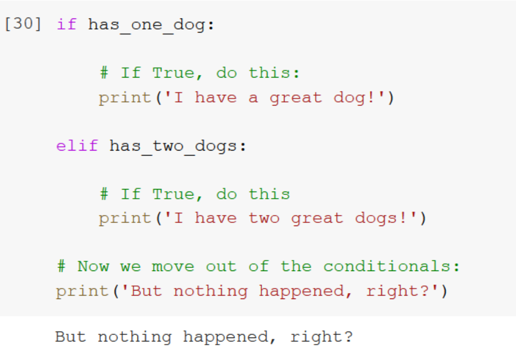
Basic syntax - Conditional expressions

We can also use `if` and `elif` without `else`



```
[27] n_dogs = 3000

     # Now we create two boolean variables
     has_one_dog  = n_dogs == 1
     has_two_dogs = n_dogs == 2

     # Printing the variables
     print(has_one_dog)
     print(has_two_dogs)
```

```
     False
     False
```

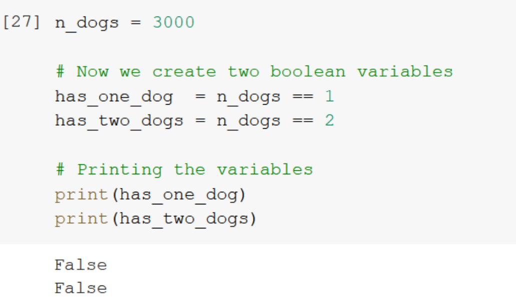


```
[30] if has_one_dog:

         # If True, do this:
         print('I have a great dog!')

     elif has_two_dogs:

         # If True, do this
         print('I have two great dogs!')

     # Now we move out of the conditionals:
     print('But nothing happened, right?')
```

```
     But nothing happened, right?
```

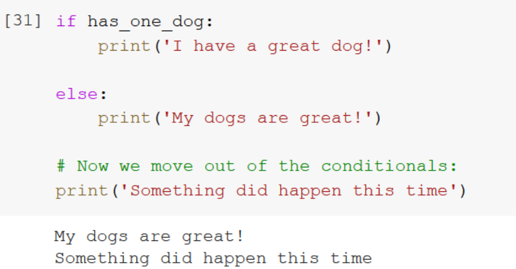
Basic syntax - Conditional expressions

And we can use `if` and `else` without `elif`



```
[27] n_dogs = 3000

     # Now we create two boolean variables
     has_one_dog  = n_dogs == 1
     has_two_dogs = n_dogs == 2

     # Printing the variables
     print(has_one_dog)
     print(has_two_dogs)
```

```
     False
     False
```

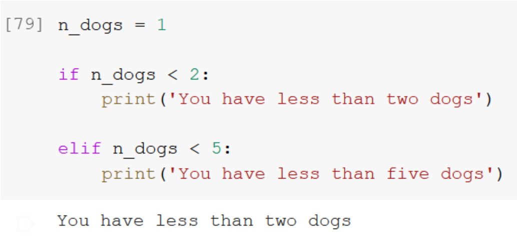


```
[31] if has_one_dog:
         print('I have a great dog!')

     else:
         print('My dogs are great!')

     # Now we move out of the conditionals:
     print('Something did happen this time')
```

```
     My dogs are great!
     Something did happen this time
```

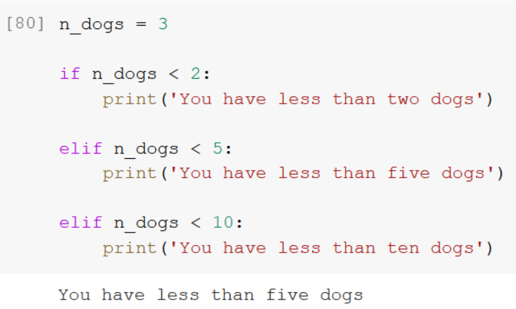
Basic syntax - Conditional expressions

- If a boolean expression returned `True` for conditions in both `if` and `elif`, only the operations under `if` would be executed
- If more than one boolean expression under several `elif` conditions were to return `True`, only the operations under the first `elif` condition evaluated to `True` would be executed



```
[79] n_dogs = 1

     if n_dogs < 2:
         print('You have less than two dogs')

     elif n_dogs < 5:
         print('You have less than five dogs')
```

```
     You have less than two dogs
```



```
[80] n_dogs = 3

     if n_dogs < 2:
         print('You have less than two dogs')

     elif n_dogs < 5:
         print('You have less than five dogs')

     elif n_dogs < 10:
         print('You have less than ten dogs')
```

```
     You have less than five dogs
```