5_python_I

January 9, 2025

1 Coding is Just Coding

Prepared by Andie Creel for the Into to Programming Workshop at YSE

2 Google Collab

Today, we will be using Google Collab to write Python code in a notebook environment.

Go to `https://colab.research.google.com/` and create a New Notebook. You do need a Google Account to do this. Once in this environment, we can write text (like this) and code (like the chunks below).

3 Goal for Today

Once you learn one coding language, you can read (and sometimes even write) in a lot of other coding languages. Today we're going to go through how to code what we did on the first day in Python (another popular coding language). The syntax is slightly different, but you can still read a lot of it. The point of this is to realize that if someone hands you a code script in a different language, you shouldn't freak out.

Let's run our first line of code in Python.

[1]:
```
print("Hello World")
```

```
Hello World
```

4 Basic Data Types

Let's revist our basic data types.

[2]:
```
# Run basic arithmatic
2 + 3
```

[2]:
```
5
```

[3]:
```
# variable assingment is done with an '=' sign, instead of the '<-' sign
a = 2
b = 3
a+b
```

```
[3]: 5
```

```
[4]: # Let's change the value of a, demonstating that it's a variable
     a = 6
     a+b
```

```
[4]: 9
```

Now, let's revisit our basic data types.

```
[5]: # Numeric -- integer: no decimal points
     myInt = 1
     print(myInt) # In python, we need to print anything we'd like to see at the end␣
      ↪of our notebook

     # Numeric -- floating point: decimal points
     myNum = 2.4
     myNum # if we hadn't used print, we would have seen the last line of the cell␣
      ↪only
```

```
1
```

```
[5]: 2.4
```

```
[6]: # character (string)
     myChar_a = 'a'
     print(myChar_a) # a flag with the print statement: it wont print the '' around␣
      ↪the character

     myChar_b = 'b'
     myChar_b
```

```
a
```

```
[6]: 'b'
```

```
[7]: # logical (Boolean): a true/false statement. Use parentheses to evaluate if␣
      ↪something is true or false
     myBool_1 = (3 < 4)
     print(myBool_1)

     myBool_2 = (3 > 4)
     print(myBool_2)
```

```
True
False
```

```
[8]: # we can till have "tricky" variables like in R
     trick_q = "1"
```

```
trick_q
```

[8]:
```
'1'
```

Now, with the print statement, we didn't see the ' ' symbol to remind us that `myChar_a` is a character data type. This is a feature of the `print()` and how it interacts with strings/character variables.

With `trick_q` we did see the ' ' again because we returned the variable at the end of the notebook cell (instead of using `print()`). Python's (and R's) ability to infer a variable's type without explicit type declarations is known as **dynamic typing**.

In dynamically typed languages like Python and R, the type of a variable is determined at runtime, and you do not need to declare the type explicitly when you create the variable. This allows for more flexibility in coding, as the same variable can hold different types of data at different times during execution. However, you as the programmer need to be aware of what data type you intend your variable to be and make sure Python or R has inferred the data type correctly.

5 Ways to store datatypes

You will need the `numpy` package. NumPy is the main package for scientific computing in python. It's already installed in Google Collab, so we just need to import it.

5.1 Vectors and Matrices

Notice that indexing in python starts at 0, rather than 1 (as it did in R). In python, we just use lists instead of vectors.

[9]:
```python
import numpy as np

# Lists can contain elements of different data types
myList_n = [1, 2, 3, 4, 5]
print(myList_n[0])

myList_s = ["str", "b", "c"]
print(myList_s[0])

myList_all = ["str", 1, True]
print(myList_all)
```

```
1
str
['str', 1, True]
```

[10]:
```python
# NumPy array (similar to R matrix): should contain elements of the same data␣
↪type
# In this case, we're creating a 2x5 matrix
# the . here works similar to %>% in dplyr
myMat_n = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(2, 5)
myMat_n
```

```
[10]: array([[ 1,  2,  3,  4,  5],
              [ 6,  7,  8,  9, 10]])
```

5.2 Lists

```
[11]: # Lists: Can contain elements of different data types, including other lists or␣
       ↪arrays
      myList = [2, "c", myMat_n]

      # Accessing the first element of the list
      myList[0]  # returns numeric (2 in this case)
```

```
[11]: 2
```

```
[12]: myList[1] # returns C
```

```
[12]: 'c'
```

```
[13]: myList[2] # returns the matrix
```

```
[13]: array([[ 1,  2,  3,  4,  5],
              [ 6,  7,  8,  9, 10]])
```

5.3 Dataframes

To work with data frames, we need to load the `pandas` package.

```
[14]: import pandas as pd

      # Create a DataFrame from the NumPy array
      myDF = pd.DataFrame(myMat_n)
      myDF
```

```
[14]:    0  1  2  3   4
      0  1  2  3  4   5
      1  6  7  8  9  10
```

```
[30]: # Print column names (initially they are just integer indices)
      print(myDF.columns)
```

```
Index(['num_hunts', 'temp', 'num_adults', 'num_cubs', 'distance_from_road',
       'total_lions'],
      dtype='object')
```

Unlike R, python automatically names unamed columns with integrers. But like R, these are not informative.

```
[16]: # Rename the columns
```

```
myDF.columns = ["num_hunts", "temp", "num_adults", "num_cubs",␣
 ↪"distance_from_road"]
myDF
```

[16]:

```
   num_hunts  temp  num_adults  num_cubs  distance_from_road
0          1     2           3         4                   5
1          6     7           8         9                  10
```

There are two ways too look at columns in Python. [] notation and then . notation. The bracket notation is the same as R (except the index starts at 0 instead of 1). The period notation is similar to the $ notation.

[17]:

```
# investigate one column with [] notation
myDF['num_adults']
```

[17]:

```
0    3
1    8
Name: num_adults, dtype: int64
```

[18]:

```
# investigate one column with . notation
myDF.num_adults
```

[18]:

```
0    3
1    8
Name: num_adults, dtype: int64
```

[19]:

```
# Create a new column (you can only create a column with the [] notation)
myDF['total_lions'] = myDF['num_adults'] + myDF['num_cubs']
myDF['total_lions']
```

[19]:

```
0     7
1    17
Name: total_lions, dtype: int64
```

5.3.1 Example

Let's build a DataFrame with multiple data types

[20]:

```
# Create the DataFrame
myPpl = pd.DataFrame({
    'name': ["Andie", "Bridger", "Scott"],
    'gender': ["Female", "non-binary", "Male"],
    'male': [True, False, False],
    'income_cat': ["middle", "poor", "rich"],
    'park_dist_mi': [1, 0.5, 0.1]
})

myPpl
```

[20]:
```
       name      gender   male income_cat  park_dist_mi
0     Andie      Female   True     middle           1.0
1   Bridger  non-binary  False       poor           0.5
2     Scott        Male  False       rich           0.1
```

Now that we have a dataframe, let's reference column and rows and cells in our multiple ways.

[21]:
```
# Accessing a column using the . notation
myPpl.name
```

[21]:
```
0      Andie
1    Bridger
2      Scott
Name: name, dtype: object
```

[22]:
```
# Accessing a column using the [] notation
myPpl['name']
```

[22]:
```
0      Andie
1    Bridger
2      Scott
Name: name, dtype: object
```

5.3.2 `.iloc[]` Pandas

In pandas, `.iloc` and `.loc` are essential tools for data manipulation and retrieval within DataFrames. They allow you to access and modify data in a DataFrame in different ways.

`.iloc[]` is primarily used for integer-location based indexing. It allows you to select rows and columns by their integer positions (i.e., their index numbers). This is useful when you know the exact positions of the rows and columns you want to access.

`.loc[]`can be used with index based locating and names.

[23]:
```
# Access the first row using .iloc[]
myPpl.iloc[0]
```

[23]:
```
name              Andie
gender           Female
male               True
income_cat       middle
park_dist_mi        1.0
Name: 0, dtype: object
```

We can also use `.iloc[]` to reference a single cell.

[24]:
```
# Access the first row of the 'name' column using .iloc[]
print(myPpl.iloc[0,0])

# Access the first row of the 'name' column using .loc[]
```

```
myPpl.loc[0, 'name']
```

```
Andie
```

```
[24]: 'Andie'
```

```
[25]: # Access the second row of the 'name' column using .loc[]
      myPpl.loc[1, 'name']
```

```
[25]: 'Bridger'
```

6 Functions

def stands for definition. The syntax for writing a function is different, and is a good example of how white space is important in python (notice that there are no parentheses).

Let's revisit our function that models the relationship between the probability of someone visiting a national park and the temperature (F). You know that people don't visit the park when it's very cold, nor when it's very hot. You model the relationship using the following quadratic equation

$$v = F/100 - (F/100)^2$$

where v is visits and F is the temperature.

We want to define a function that gets the temperature F and returns the predicted number of trips taken v.

```
[26]: def get_visits(F):
          v = F/100 - (F/100)**2
          return v

      print(get_visits(0))
      print(get_visits(25))
      print(get_visits(50))
      print(get_visits(75))
      print(get_visits(100))
```

```
0.0
0.1875
0.25
0.1875
0.0
```

We will come back to this after loops!

7 Loops

Loops are another example where you can read the code even if you don't know python. However, they have some diffrent syntax with the range function, specifically that the last value is excluded.

It’s also important to remember when iteracting over a dataframe, the first row or column is indexed with a 0.

[27]:
```
# Notice that 5 doesn't print
for i in range(0, 5):  # range(start, stop) in Python is inclusive of start and␣
 ↪exclusive of stop
    print(i)
```

```
0
1
2
3
4
```

7.1 Combining a loop with a function

[28]:
```
import numpy as np

# create the vector to iterate over
vec_Fs = np.linspace(0, 100, 11) # np.linspace(start, stop, num) creates a␣
 ↪vector of 'num' elements from 'start' to 'stop'
print(vec_Fs)


for T in vec_Fs:
    v = get_visits(T)
    print(v)
```

```
[  0.  10.  20.  30.  40.  50.  60.  70.  80.  90. 100.]
0.0
0.09
0.16
0.21
0.24
0.25
0.24
0.21000000000000002
0.15999999999999992
0.08999999999999997
0.0
```

8 If Else Statement

Let’s say we want to create a clasification to say if someone is close or far to a park. We can iterate over our dataframe and use an if-else statement.

The major thing we need to be aware with in Python is use of the `.loc` funciton, which let’s us reference cells with their column names and row index.

[29]:
```
# Create a new var for park distance
for i in range(0, len(myPpl)):
    if myPpl.loc[i, 'park_dist_mi'] > 0.75:
        myPpl.loc[i, 'park_dist_cat'] = 'far'
    else:
        myPpl.loc[i, 'park_dist_cat'] = 'close'

myPpl
```

[29]:
```
      name      gender   male income_cat  park_dist_mi park_dist_cat
0    Andie      Female   True     middle           1.0           far
1  Bridger  non-binary  False       poor           0.5         close
2    Scott        Male  False       rich           0.1         close
```