Intermediate Python

Matplotlib

1. Data visualisation
    - very important in data analysis
    - use to explore data and report insight to others

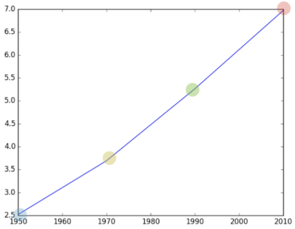
2. Matplotlib
    - import the sub-package of Matplotlib in IDE

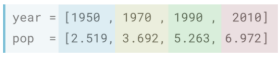
    ```
    import matplotlib.pyplot as plt
    ```

    - types of plot

    1. line plot
        - used when have time scale along the horizontal axis

        ```
        import matplotlib.pyplot as plt
        year = [1950, 1970, 1990, 2010]
        pop = [2.519, 3.692, 5.263, 6.972]
        #plt.plot(horizontal axis, vertical axis)
        plt.plot(year, pop)
        #need to call to show the plot
        plt.show()
        ```

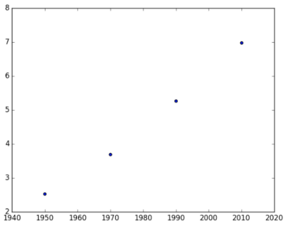




```
year = [1950 , 1970 , 1990 ,  2010]
pop  = [2.519, 3.692, 5.263, 6.972]
```

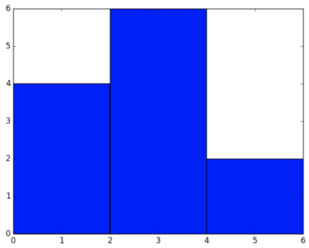
    2. scatter plot
        - plot the individual data without joining them with line
        - better than line plot
        - is known as more honest data plot because can see the individual data clearly
        - used when want to access the correlation (相互关系) between 2 variables

        ```
        import matplotlib.pyplot as plt
        year = [1950, 1970, 1990, 2010]
        pop = [2.519, 3.692, 5.263, 6.972]
        plt.scatter(year, pop)
        plt.show()
        ```



3. histogram
    - useful when exploring data
    - can get idea about the distribution of variables
    - bin(default = 10) = each bin has equal width
    - plt.clf() cleans the plot again so you can start afresh

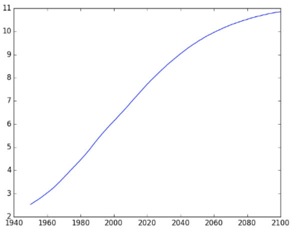
```
import matplotlib.pyplot as plt

values = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]
plt.hist(values, bins=3)
plt.show()
```



3. Customisation

- the customisation depends on the data and story told by the user
- Exp: basic line plot

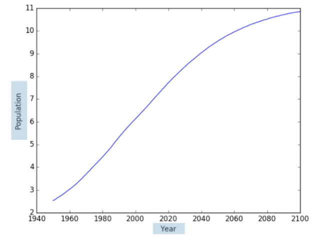
```
import matplotlib.pyplot as plt

year = [1950, 1951, 1952, ..., 2100]
pop = [2.538, 2.57, 2.62, ..., 10.85]
plt.plot(year, pop)
plt.show()
```



Before customisation

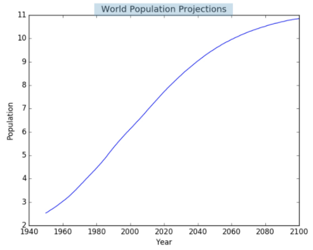
- types of customisation:
    1. axis label

```
import matplotlib.pyplot as plt

year = [1950, 1951, 1952, ..., 2100]
pop = [2.538, 2.57, 2.62, ..., 10.85]
plt.plot(year, pop)
plt.xlable('Year')
plt.ylable('Population')
plt.show()
```



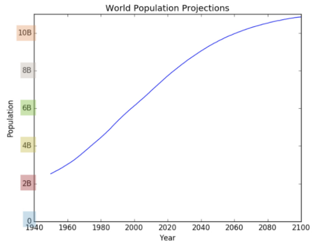
2. title

```
import matplotlib.pyplot as plt

year = [1950, 1951, 1952, ..., 2100]
pop = [2.538, 2.57, 2.62, ..., 10.85]
plt.plot(year, pop)
plt.xlable('Year')
plt.ylable('Population')
plt.title('World Population Projections')
plt.show()
```



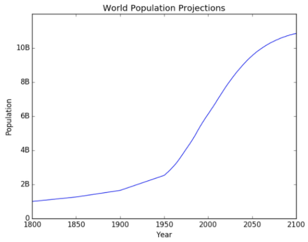
3. ticks

```
import matplotlib.pyplot as plt

year = [1950, 1951, 1952, ..., 2100]
pop = [2.538, 2.57, 2.62, ..., 10.85]
plt.plot(year, pop)
plt.xlable('Year')
plt.ylable('Population')
plt.title('World Population Projections')
plt.yticks([0, 2, 4, 6, 8, 10] #scale , [0B, 2B, 4B, 6B, 8B, 10B]) #B stand for billions
plt.show()
```



4. add historical data

```
import matplotlib.pyplot as plt

year = [1950, 1951, 1952, ..., 2100]
pop = [2.538, 2.57, 2.62, ..., 10.85]

#add more data (append data)
year = [1800, 1850, 1900] + year
pop = [1.0, 1.262, 1.650] + pop

plt.plot(year, pop)
plt.xlable('Year')
plt.ylable('Population')
plt.title('World Population Projections')
plt.yticks([0, 2, 4, 6, 8, 10] #scale , [0B, 2B, 4B, 6B, 8B, 10B]) #B stand for billions
plt.show()
```



After the customisation

Dictionaries & Pandas

1. Dictionary
   - can only contain unique key
   - the key is immutable object
   - add new key and value in to the dictionary that already created

```
world = {"afghanistan":30.55, "albania":2.77, "algeria":39.21}

world['sealand'] = 0.000027
print(world)

#Output:
#{'afghanistan': 30.55, 'albania': 2.81,'algeria': 39.21, 'sealand': 2.7e-05}

#to check whether the data is inside the dictionary
print("sealand"in worldTrue)
```

   - update the value

```
world['sealand'] = 0.000028
print(world)

#Output:
#{'afghanistan': 30.55, 'albania': 2.81,'algeria': 39.21, 'sealand': 2.8e-05}
```

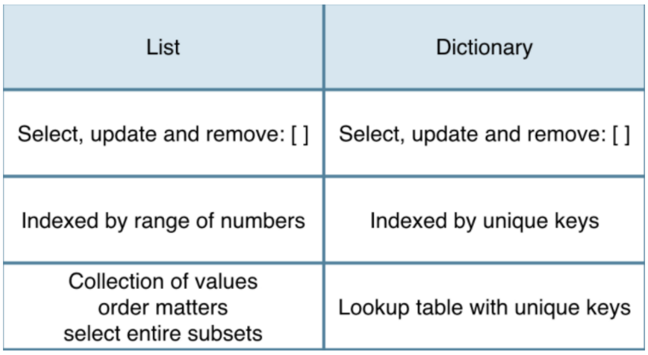
   - delete the key

```
del(world['sealand'])
print(world)

#Output:
#{'afghanistan': 30.55, 'albania': 2.81,'algeria': 39.21}
```

   - create a dictionary with sub-dictionary inside

```
europe = { 'spain': { 'capital':'madrid', 'population':46.77 },
           'france': { 'capital':'paris', 'population':66.03 },
           'germany': { 'capital':'berlin', 'population':80.62 },
           'norway': { 'capital':'oslo', 'population':5.084 } }

#search the population of spain
europe['spain']['population']

#Output:
#46.77
```

2. List vs. Dictionary

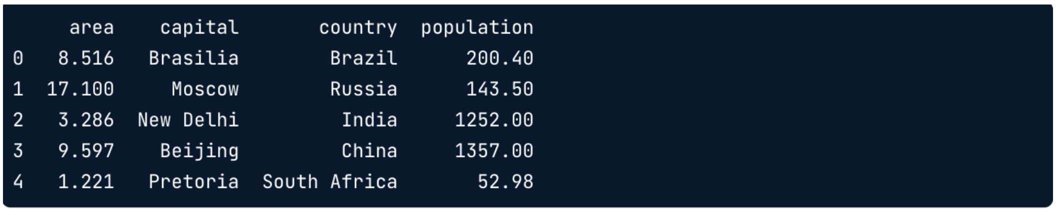


List	Dictionary
Select, update and remove: [ ]	Select, update and remove: [ ]
Indexed by range of numbers	Indexed by unique keys
Collection of values order matters select entire subsets	Lookup table with unique keys

3. Pandas

- high-level data manipulation tool
- built on NumPy
- store the tabular data in an object which called DataFrame
- can store different types ion data
- ways to build DataFrame
    1. DataFrame from dictionary
        - keys (column table)
        - values (data, column by column)

```
dict = { "country":["Brazil", "Russia", "India", "China", "South Africa"],
        "capital":["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"],
        "area":[8.516, 17.10, 3.286, 9.597, 1.221],
        "population":[200.4, 143.5, 1252, 1357, 52.98] }

import pandas as pd

brics = pd.DataFrame(dict)
print(brics)
```

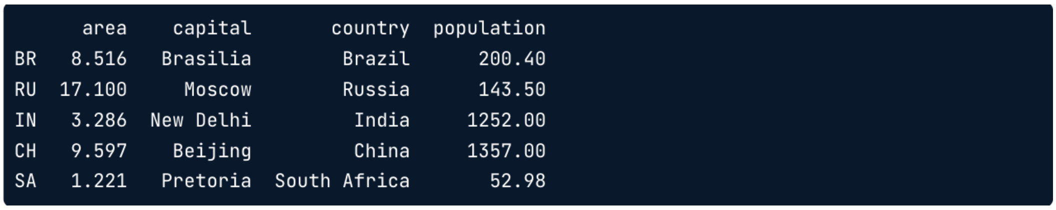


```
     area    capital       country  population
0   8.516   Brasilia        Brazil      200.40
1  17.100     Moscow        Russia      143.50
2   3.286  New Delhi         India     1252.00
3   9.597    Beijing         China     1357.00
4   1.221   Pretoria  South Africa       52.98
```

it will generate the index automatically

```
#able to channge index like shown as above
brics.index = ['BR', 'RU', 'IN', 'CH', 'SA']
```

```
print(brics)
```



```
      area    capital       country  population
BR   8.516   Brasilia        Brazil      200.40
RU  17.100     Moscow        Russia      143.50
IN   3.286  New Delhi         India     1252.00
CH   9.597    Beijing         China     1357.00
SA   1.221   Pretoria  South Africa       52.98
```

2. DataFrame from CSV file

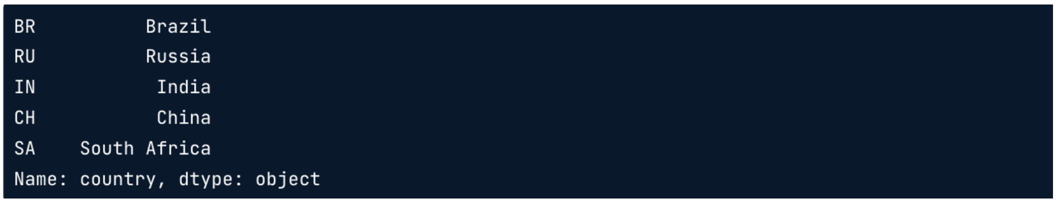
```
#index_col is used to set the index column to follow the data given in csv file
brics = pd.read_csv("path/to/brics.csv", index_col = 0)
```

- index and select data

1. select one of the column []

    - wrong example

```
#print the row column and row label
print(brics['country'])
```

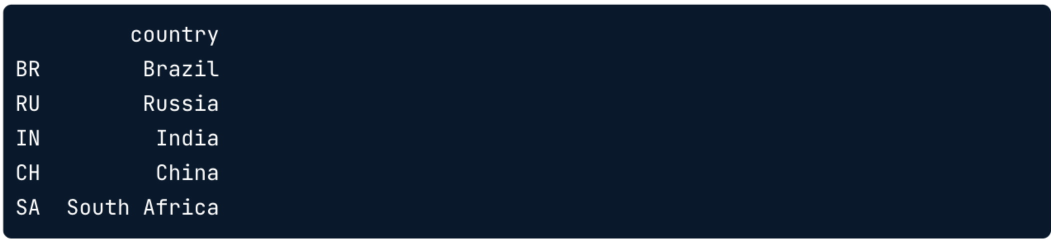


```
BR          Brazil
RU          Russia
IN           India
CH           China
SA    South Africa
Name: country, dtype: object
```

it is not dealing with the DataFrame

- select the column and remain the data in data frame by using double square brackets

```
print(brics[['country]])
```



```
         country
BR        Brazil
RU        Russia
IN         India
CH         China
SA  South Africa
```

2. select the row [] by using slicing

```
print(brics[1:4])
```

> 💡 There are limitations when using square brackets because it works similarly like 2DNumPy which can be solved by using pandas toolbox → loc (label-based) and iloc (integer position-based)

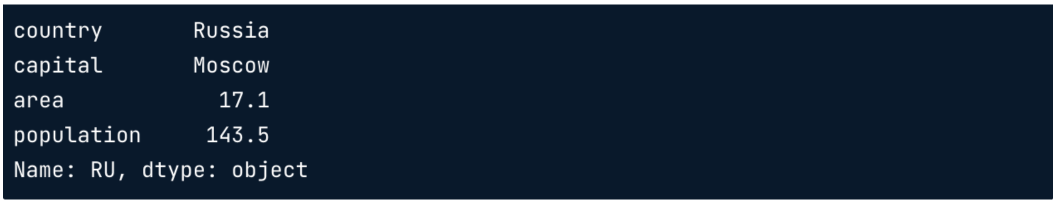
- another ways to select data instead of []

1. loc (local-based)

    - specify rows and columns based on their row and column labels

- select the data from row

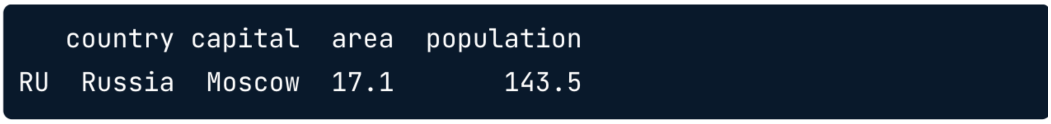
```
print(brics.loc['Russia'])
```



```
country       Russia
capital       Moscow
area            17.1
population     143.5
Name: RU, dtype: object
```

get the data in pandas series

```
print(brics.loc[['RU']])
```

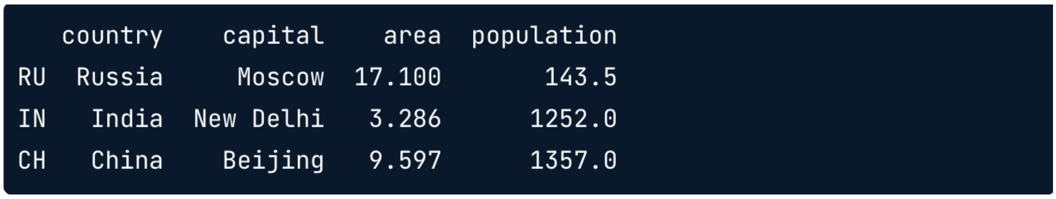


```
   country capital  area  population
RU  Russia  Moscow  17.1       143.5
```

get the data in DataFrame

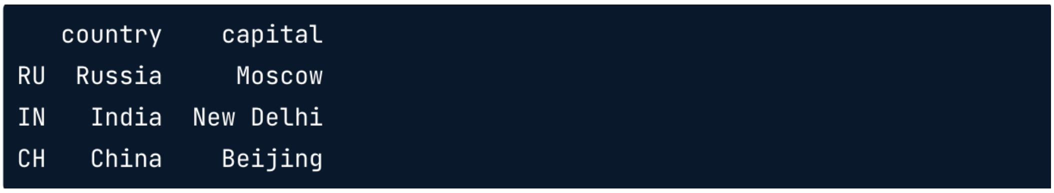
- select multiple row at the same time

```
print(brics.loc[['RU', 'IN', 'CH']])
```



```
   country    capital   area  population
RU  Russia     Moscow  17.100       143.5
IN   India  New Delhi   3.286      1252.0
CH   China    Beijing   9.597      1357.0
```

- select the specify row and column

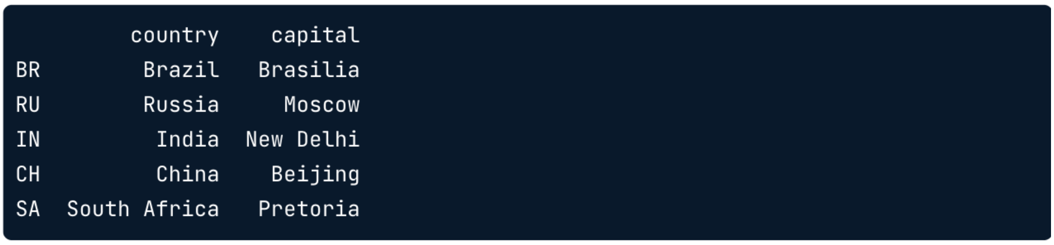


```
   country    capital
RU  Russia     Moscow
IN   India  New Delhi
CH   China    Beijing
```

```
print(brics.loc[['RU', 'IN', 'CH'], ['country', 'capital']])
```

- select all rows with specify columns

```
print(brics.loc[:, ['country', 'capital']])
```

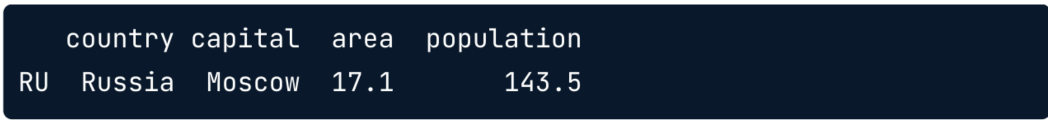


```
         country    capital
BR        Brazil   Brasilia
RU        Russia     Moscow
IN         India  New Delhi
CH         China    Beijing
SA  South Africa   Pretoria
```

💡 Using loc, the user can access the column and row by specifying the name of the column or row. Besides, can also select row and column at the same time

2. iloc (integer position-based)
    - specify rows and columns by their integer index
    - select the row by using index number

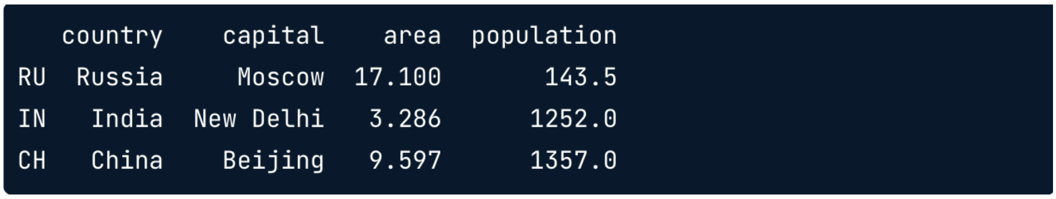
    ```
    print(brics.loc[[1]])
    ```

    ```
       country capital  area  population
    RU  Russia  Moscow  17.1       143.5
    ```

    - select multiple row

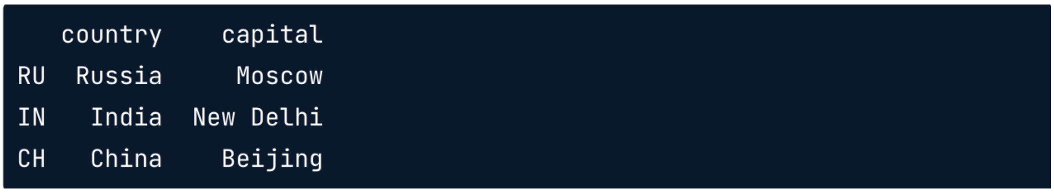
    ```
    print(brics.loc][1, 2, 3]])
    ```

    ```
       country    capital    area  population
    RU  Russia     Moscow  17.100       143.5
    IN   India  New Delhi   3.286      1252.0
    CH   China    Beijing   9.597      1357.0
    ```

    - select specify row and column

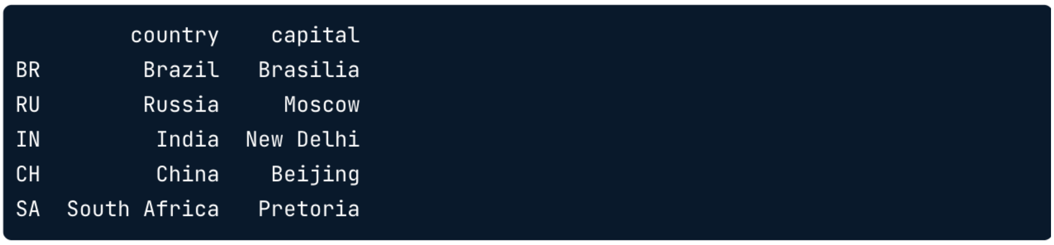
    ```
    print(brics.loc[[1, 2, 3], [0, 1]])
    ```

    ```
       country    capital
    RU  Russia     Moscow
    IN   India  New Delhi
    CH   China    Beijing
    ```

    - select all rows with specify columns

    ```
    print(brics.loc[:, [0, 1]])
    ```

    ```
             country    capital
    BR        Brazil   Brasilia
    RU        Russia     Moscow
    IN         India  New Delhi
    CH         China    Beijing
    SA  South Africa   Pretoria
    ```

Logic, Control Flow and Filtering

1. Comparison operators
    1. Numeric operators

        - <

          ```
          'carl' < 'chris'

          #Output:
          #because it measure by uisng the length of the string
          #determines the relationship based on alphabetical order
          True
          ```

        - >
        - ==
        - < =
        - > =
        - ! =

        > 💡 always compare the objects from the same type

    2. Boolean operators
        1. used for normal Python code
            - and
            - or
            - not
        2. used for NumPy code
            - logical_and()

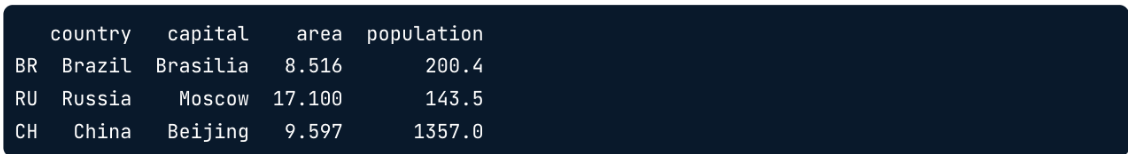
              ```
              print(np.logical_and(bmi > 21, bmi < 22))

              #Output:
              #array([True, False, True, False, True], dtype = bool)

              print(bmi[np.logical_and(bmi > 21, bmi < 22)])

              #Output:
              #array([21.852, 21,75, 21.441])
              ```

            - logical_or()
            - logical_not()
    3. conditional statements (if, elif, else)

2. filtering pandas DataFrame
    - Exp: select the country where the areas is greater than 8 million km2

      ```
      import pandas as pd

      brics = pd.read_csv("path/to/brics.csv", index_col = 0)

      #Step 1: Get the column
      print(brics['area'])
      #or
      print(brics.loc[:, 'area'])
      #or
      print(brics.iloc[:, 2])

      #Step 2: Compare
      print(brics['area'] > 8)

      #Step 3: subset DataFrame
      print(brics[is_huge])
      ```

```
#easy method 1
is_huge = brics['area'] > 8
brics[is_huge]

#easy method 2
print(brics[brics['area'] > 8]
```

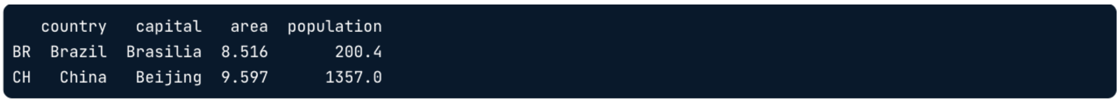


```
    country    capital    area  population
BR   Brazil   Brasilia   8.516       200.4
RU   Russia     Moscow  17.100       143.5
CH    China    Beijing   9.597      1357.0
```

Step 3: subset DataFrame

- Exp: select the country where the areas is between 8 and 10 million km2

```
import numpy as np

print(brics[np.logical_and(brics['area'] > 18, brics['area'] < 10)])
```



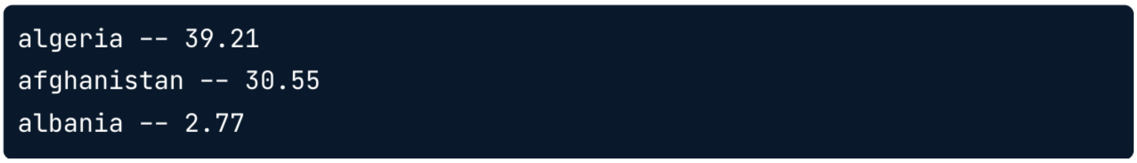
```
   country   capital   area  population
BR  Brazil  Brasilia  8.516       200.4
CH   China   Beijing  9.597      1357.0
```

Loops

1. Types of loops
    1. while loop
        - a repeated if statement as long as the condition is true
    2. for loop

2. Loops data structures
    1. for loop in dictionary

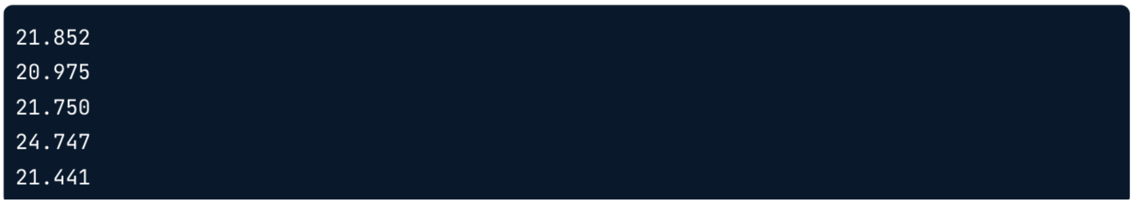
```
world = { "afghanistan":30.55,"albania":2.77, "algeria":39.21 }

#.items() allow the user to print the key value pairs without any error
for key, value in world.items() :
  print(key + " -- " + str(value))
```



```
algeria -- 39.21
afghanistan -- 30.55
albania -- 2.77
```

    2. for loop for NumPy array

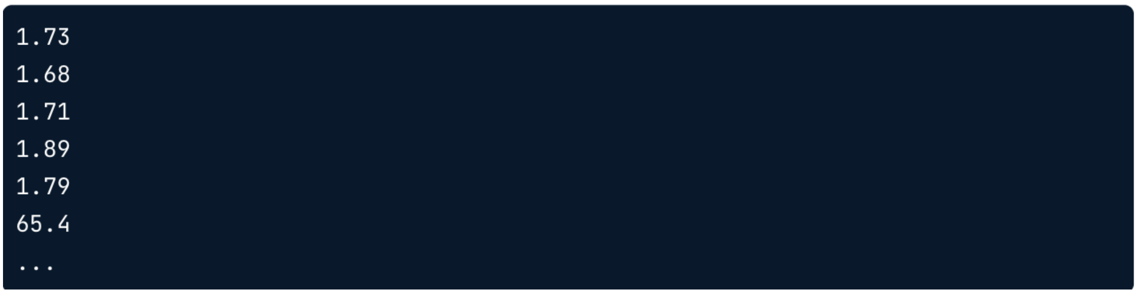
```
import numpy as np

np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
bmi = np_weight / np_height ** 2

for val in bmi :
  print(val)
```



```
21.852
20.975
21.750
24.747
21.441
```

3. for loop for 2D NumPy array

```
import numpy as np

np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
meas = np.array([np_height, np_weight])

#.nditer() is used to print each element in the array
for val in np.nditer(meas) :
  print(val)
```

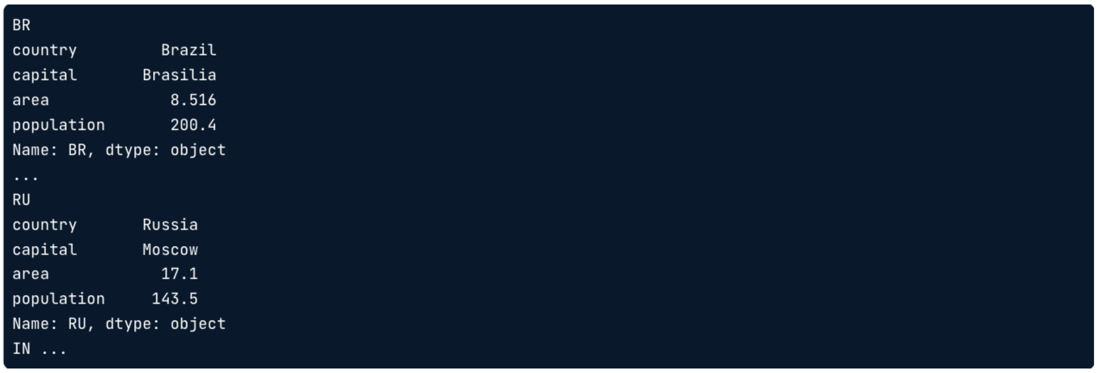


```
1.73
1.68
1.71
1.89
1.79
65.4
...
```

4. for loop for Pandas DataFrame

- print all data frame

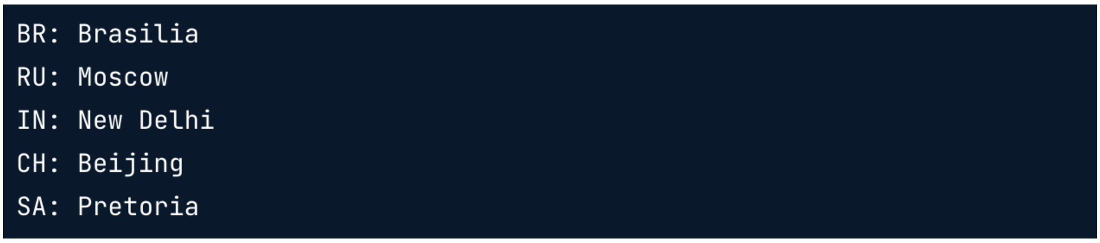
```
import pandas as pd

brics = pd.read_csv("path/to/brics.csv", index_col = 0)

#.itterrows() looks at DataFrame and on each iterates generate 2 pieces of data which is label of the row and data in the
for lab, row in brics.iterrows():
  print(lab)
  print(row)

#selective print
for lab, row in brics.iterrows():
  print(lab + ": " + row["capital"])
```



```
BR
country         Brazil
capital       Brasilia
area             8.516
population       200.4
Name: BR, dtype: object
...
RU
country       Russia
capital       Moscow
area            17.1
population     143.5
Name: RU, dtype: object
IN ...
```

- selective print

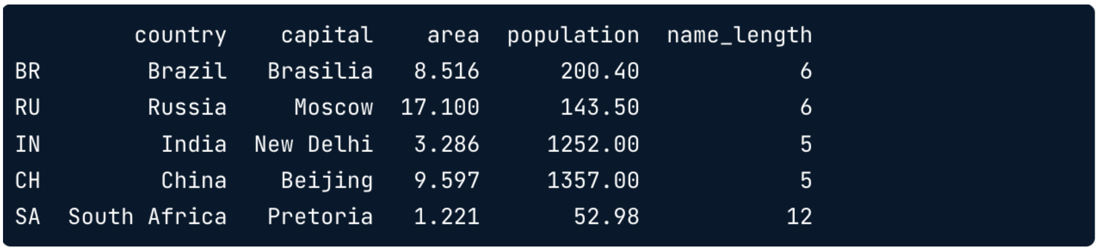
```python
import pandas as pd

brics = pd.read_csv("path/to/brics.csv", index_col = 0)

#selective print
for lab, row in brics.iterrows():
  print(lab + ": " + row["capital"])
```



```
BR: Brasilia
RU: Moscow
IN: New Delhi
CH: Beijing
SA: Pretoria
```

- add column

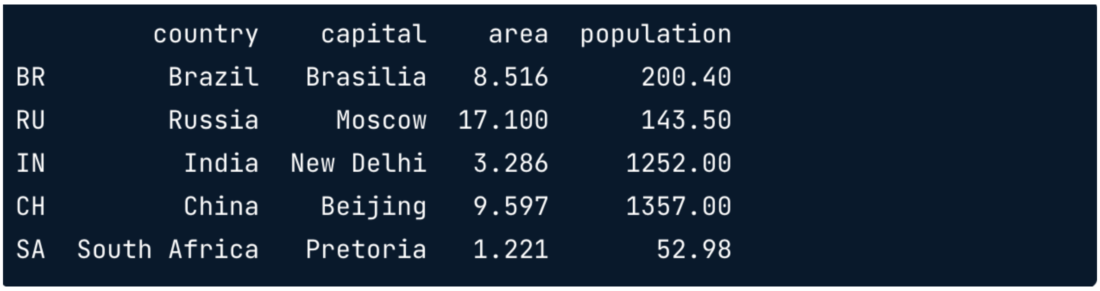
```python
import pandas as pd

brics = pd.read_csv("path/to/brics.csv", index_col = 0)

for lab, row in brics.iterrows():
  #creating series on every iteration
  brics.loc[lab, "name_length"] = len(row["country"])
  print(brics)

#only suitable for small data frame, because it will cause problem if implement on the big data frame
```



```
         country    capital    area  population  name_length
BR        Brazil   Brasilia   8.516      200.40            6
RU        Russia     Moscow  17.100      143.50            6
IN         India  New Delhi   3.286     1252.00            5
CH         China    Beijing   9.597     1357.00            5
SA  South Africa   Pretoria   1.221       52.98           12
```

- add column and calculate whole data frame by using apply()

```python
import pandas as pd

brics = pd.read_csv("path/to/brics.csv", index_col = 0)
#produce a new array and store in name_length
brics["name_length"] = brics["country"].apply(len)
print(brics)
```



```
         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98
```

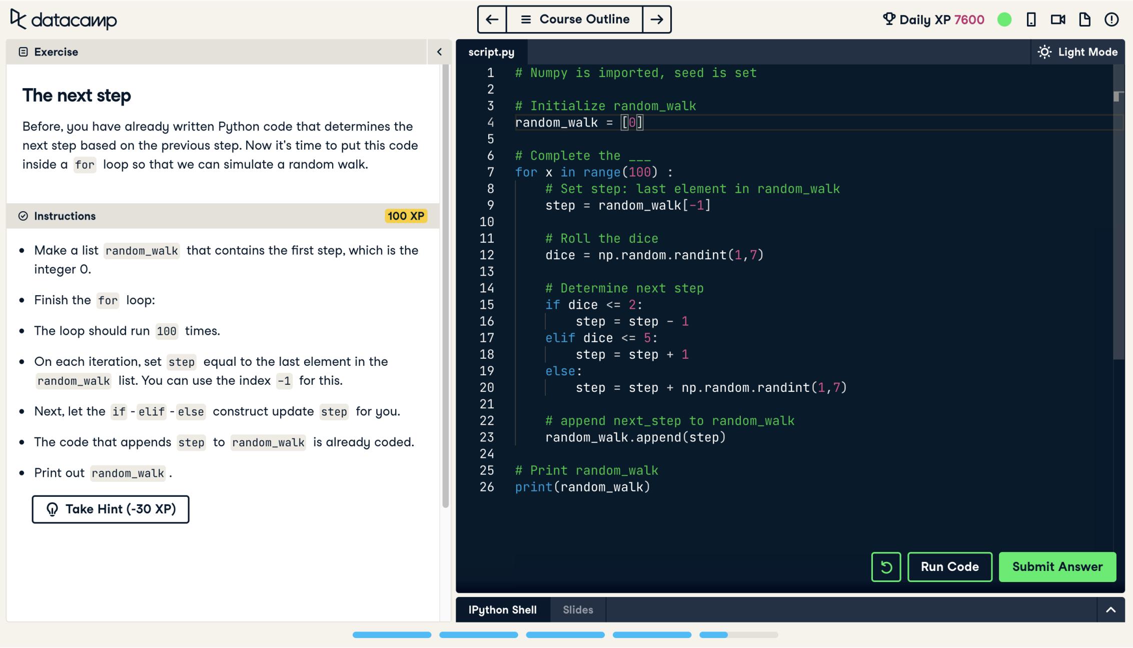
Case Study: Hacker Statistics

1. random generators

```
import numpy as np

#pseudo-random number
np.random.rand()

#.seed() sets the random seed, so that your results are reproducible between simulations
#with the .seed() is reset every time, same set of numbers will appear every time, if not different numbers appear with every invc
np.random.seed(123)
```

2. random walk

- process of taking successive steps in a randomised fashion
- implemented by using for loop
- Exp: An elementary example of a random walk is the random walk on the integer number line, which starts at 0 and at each step moves +1 or -1 with equal probability.

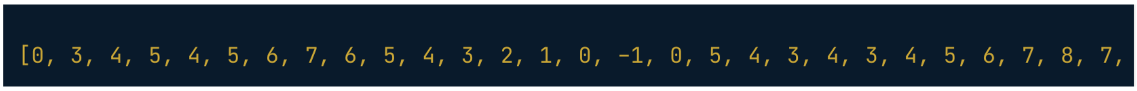


The next step

Before, you have already written Python code that determines the next step based on the previous step. Now it's time to put this code inside a `for` loop so that we can simulate a random walk.

- Make a list `random_walk` that contains the first step, which is the integer 0.
- Finish the `for` loop:
- The loop should run `100` times.
- On each iteration, set `step` equal to the last element in the `random_walk` list. You can use the index `-1` for this.
- Next, let the `if`-`elif`-`else` construct update `step` for you.
- The code that appends `step` to `random_walk` is already coded.
- Print out `random_walk`.

```
# Numpy is imported, seed is set

# Initialize random_walk
random_walk = [0]

# Complete the ___
for x in range(100) :
    # Set step: last element in random_walk
    step = random_walk[-1]

    # Roll the dice
    dice = np.random.randint(1,7)

    # Determine next step
    if dice <= 2:
        step = step - 1
    elif dice <= 5:
        step = step + 1
    else:
        step = step + np.random.randint(1,7)

    # append next_step to random_walk
    random_walk.append(step)

# Print random_walk
print(random_walk)
```



```
[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, -1, 0, 5, 4, 3, 4, 3, 4, 5, 6, 7, 8, 7,
```

3. Distribution

- can be visualise by using histogram