**Dictionaries and Intro to Pandas**

Pandas is a data manipulation library that is built on NumPy. The basic Pandas data structures are Series and DataFrames. Series are essentially one-dimensional arrays and DataFrames are two-dimensional arrays, but the row and/or columns can be labeled. Frequently DataFrames are created by reading in data from a csv (comma separated value) file or spreadsheet file, but they can also be created programmatically which is what we will focus on here. Python dictionaries are commonly used to create Series and DataFrames. In this section, we will first introduce dictionaries, and then in an introduction to Pandas, will use dictionaries to create Series and DataFrames.

Dictionaries

***Dictionaries*** are a Python data structure, but they are a ***mapping*** type, not a sequence. This is because indexing into a dictionary is accomplished using ***keys***, not integers. The type of a dictionary is **dict**.

Dictionaries are created by putting ***key-value pairs*** in curly braces. Keys are names specified as strings. The pair is specified as a key followed by a colon, followed by the value for that key, as in ‘key’:value. The following is an example of a dictionary that stores information about a particular hurricane.

*>>> bigone = {'name':'Bertha', 'year': 1952, 'category': 4}*

Values can be retrieved from a dictionary by giving the name of the dictionary variable and then the name of the key in square brackets.

*>>> bigone['year']*

1952

An error will result if the key does not exist in the dictionary.

*>>> bigone['size']*

KeyError: 'size'

An error will also occur if you try to index using an integer; values are only referenced using keys.

Dictionaries are a mutable type. Values can be modified using assignment statements.

*>>> bigone['category'] = 5*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5}

More key-value pairs can be added to a dictionary in the same way.

*>>> bigone['eyewidth'] = 400*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5, 'eyewidth': 400}

Key-value pairs can be deleted from a dictionary using the **del** command.

*>>> del bigone['eyewidth']*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5}

The **len** function returns the number of key-value pairs in the dictionary.

*>>> len(bigone)*

3

The **in** and **not in** operators can be used to find whether a key is in a dictionary or not.

*>>> 'year' in bigone*

True

*>>> 'sqarea' not in bigone*

True

The **list** function will return a list of all of the key names from a dictionary.

*>>> list(bigone)*

['name', 'year', 'category']

There are a number of methods that work with dictionaries.

The **get** method retrieves a value from a dictionary; it returns **None** by default if the key is not in the dictionary.

*>>> bigone.get('year')*

1952

If the key is not in the dictionary, the value **None** that is returned is not automatically shown, but can be printed, or tested in a selection statement.

*>>> print(bigone.get('size'))*

None

A value other than None can also be specified to be used if the key is not in the dictionary, e.g. a flag value of -999.

*>>> bigone.get('size', -999)*

-999

Note that this does not add a key-value pair to the dictionary.

The **get** method is therefore safer to use than just putting a key name in square brackets, since **get** will return a value if the key is not found, whereas using square brackets will result in an error if the key is not found.

The **pop** method deletes an item from a dictionary and returns its value. A default value to return can be specified in case the key is not in the dictionary; if the default is not specified and the key is not found in the dictionary, an error results.

*>>> delcat = bigone.pop('category')*

*>>> delcat*

5

*>>> bigone*

{'name': 'Bertha', 'year': 1952}

*>>> bigone.pop('size')*

KeyError: 'size'

*>>> bigone.pop('size', -999)*

-999

The **popitem** method deletes the last key-value pair in a dictionary, and returns this as a tuple.

*>>> k,v = bigone.popitem()*

*>>> print(k,v)*

year 1952

The **clear** method deletes all of the key-value pairs from a dictionary.

*>>> bigone.clear()*

*>>> bigone*

{}

There are also methods that can be used to iterate: **keys()**, **values()**, and **items()**, for the dictionary’s keys, values, and key-value pairs, respectively.

For example, the following call to the **keys** method returns the names of the keys. Technically, it returns a ***view object***, *dict\_keys*.

*>>> bigone = {'name':'Bertha', 'year': 1952, 'category': 4}*

*>>> bigone.keys()*

dict\_keys(['name', 'year', 'category'])

Normally, the view object would not be accessed or displayed, however. Instead, it would be typical to iterate over the result.

*for kname in bigone.keys():*

*print('The key is', kname)*

The key is name

The key is year

The key is category

To loop over the key-value pairs, the **items** method is used, and two iterator variables are created to store the keys and values individually.

*for k, v in bigone.items():*

*print('The value of',repr(k),'is',v)*

The value of 'name' is Bertha

The value of 'year' is 1952

The value of 'category' is 4

We have stored information about one hurricane in a dictionary. In order to store information on more than one hurricane, we could create a list of dictionaries.

*>>> hurr0 = {'name': 'Bertha', 'year': 1952, 'category': 4}*

*>>> hurr1 = {'name': 'Bob', 'year': 1990, 'category': 2}*

*>>> hurr2 = {'name': 'Camilla', 'year': 1960, 'category': 5}*

*>>> hurricanes = [hurr0, hurr1, hurr2]*

*>>> hurricanes[2] # show example*

{'name': 'Camilla', 'year': 1960, 'category': 5}

Once we have a list of dictionaries, we could iterate through the list, for example to print information on the more powerful hurricanes.

*print('The largest hurricanes were:')*

*for hurr in hurricanes:*

*if hurr['category'] >= 4:*

*print(repr(hurr['name']))*

The largest hurricanes were:

'Bertha'

'Camilla'

It is also possible for values in a dictionary to be data structures themselves. This has already been done, since strings are a sequence data type. Another possibility would be to have a list as a value. Note also in this example that each key-value pair is entered on a separate line, which makes it easier to read.

*icdessert = {*

*'flavor': 'malted',*

*'cone': True,*

*'nscoops': 2,*

*'addins': ['pecans', 'chocolate chips']*

*}*

*icdessert['addins']*

['pecans', 'chocolate chips']

Since the key 'addins' is a list, it can be indexed.

*>>> icdessert['addins'][0]*

'pecans'

We could use a **for** loop to print the elements from the list individually.

*print('Your ice cream add-ins are:')*

*for yummy in icdessert['addins']:*

*print(yummy.title(),'!!!')*

Your ice cream add-ins are:

Pecans !!!

Chocolate Chips !!!

For a dictionary comprehension, key-value pairs must be created.

*>>> cubedict = {i: i\*\*3 for i in range(5)}*

*>>> cubedict*

*{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}*

**Introduction to Pandas**

Since Pandas is built on NumPy, they are frequently imported together using the following:

*>>> import numpy as np*

*>>> import pandas as pd*

The rest of this section will assume that these have been imported.

**Series**

A Pandas Series is a one-dimensional array that can be created using a list or an array. For example,

>>> numseries = pd.Series([11, 33, 15, 2])

>>> numseries

0 11

1 33

2 15

3 2

dtype: int64

Notice that the indices are explicitly listed as part of the Series (which must be capitalized) and that the **dtype** is also stored in the variable and displayed.

The numbers can be accessed using **values**:

>>> numseries.values

array([11, 33, 15, 2])

You can see that it is stored as a NumPy array. You can index into the Series and slice it, just like in Python and NumPy.

>>> numseries[0]

11

>>> numseries[1:3]

1 33

2 15

dtype: int64

The indices can be accessed using **index**, although the result might not be what you would expect!

>>> numseries.index

RangeIndex(start=0, stop=4, step=1)

From this you can see that the indices are a range beginning at 0. Indices can also be specified.

>>> numlabels = pd.Series([11, 33, 15, 2], ['a','b','c','d'])

>>> numlabels

a 11

b 33

c 15

d 2

dtype: int64

The labels can then be used to index and slice into the Series.

>>> numlabels['c']

15

>>> numlabels['b':'d']

b 33

c 15

d 2

dtype: int64

Implicit integer indices can also be used to index and to slice into the Series. Notice, however, that using 'b':'d' returns 3 rows, whereas [1:3] only returns 2 (like Python and NumPy).

>>> numlabels[1:3]

b 33

c 15

dtype: int64

A Series can also be created using a dictionary.

>>> numdict = {'a':11, 'b':33, 'c':15, 'd':2}

>>> numlab = pd.Series(numdict)

>>> numlab

a 11

b 33

c 15

d 2

dtype: int64

**DataFrames**

A Pandas DataFrame looks like a two-dimensional array with labels for the rows and the columns. A DataFrame can be constructed from a single Series, for example from above:

>>> nl = pd.DataFrame(numlab,columns = ['Nums'])

>>> nl

|  | **Nums** |
| --- | --- |
| **a** | 11 |
| **b** | 33 |
| **c** | 15 |
| **d** | 2 |

DataFrames usually have multiple columns, however.

Let’s create a DataFrame with information on students by constructing multiple Series using dictionaries.

id\_dict = {'Joey': 123, 'Javier': 234, 'Juanita': 456, 'Jane': 678}

ids = pd.Series(id\_dict)

exam1\_dict = {'Joey': 95, 'Javier': 99, 'Juanita': 88, 'Jane': 100}

exam1s = pd.Series(exam1\_dict)

students = pd.DataFrame({'ID': ids, 'Exam1': exam1s})

students

|  | **ID** | **Exam1** |
| --- | --- | --- |
| **Joey** | 123 | 95 |
| **Javier** | 234 | 99 |
| **Juanita** | 456 | 88 |
| **Jane** | 678 | 100 |

The **index** attribute shows the indices, which are in an Index object that stores the row labels.

>>> students.index

Index(['Joey', 'Javier', 'Juanita', 'Jane'], dtype='object')

The **columns** attribute shows the indices, which are in an Index object that stores the column labels.

>>> students.columns

Index(['ID', 'Exam1'], dtype='object')

The **values** attribute returns the numbers in the two columns as a 2D array.

>>> students.values

array([[123, 95],

[234, 99],

[456, 88],

[678, 100]])

In order to access a column in the DataFrame, the column name can be specified in two ways:

>>> idcol = students['ID']

>>> idcol

Joey 123

Javier 234

Juanita 456

Jane 678

Name: ID, dtype: int64

>>> idcol = students.ID

>>> idcol

Joey 123

Javier 234

Juanita 456

Jane 678

Name: ID, dtype: int64

Note that this is a Series. Then, to access an individual Id from the Series, we can index using the implicit integer index or using the row label.

>>> idcol[0]

123

>>> idcol['Javier']

234

Of course, it is not necessary to extract the column first before indexing to get an individual value. These can be combined.

>>> students.ID[0]

123

>>> students['ID']['Jane']

678

In order to access a row in a DataFrame, the **iloc** method can be used with an implicit index, or the **loc** method can be used with a label.

>>> students.loc['Juanita']

ID 456

Exam1 88

Name: Juanita, dtype: int64

>>> students.iloc[2]

ID 456

Exam1 88

Name: Juanita, dtype: int64

Note that this is a Series.

There are several functions and methods that can be used to find the dimensions of a DataFrame. The **len** function can be used to find the number of rows, and by specifying .columns the number of columns.

>>> len(students)

4

>>> len(students.columns)

2

The **shape** method will return both dimensions as a tuple.

>>> students.shape

(4, 2)

A column can be added to the DataFrame as follows:

>>> students['Exam2'] = [100, 99, 98, 97]

>>> students

|  | **ID** | **Exam1** | **Exam2** |
| --- | --- | --- | --- |
| **Joey** | 123 | 95 | 100 |
| **Javier** | 234 | 99 | 99 |
| **Juanita** | 456 | 88 | 98 |
| **Jane** | 678 | 100 | 97 |

Column(s) can be deleted using the **drop** method. This must be assigned to the DataFrame object.

>>> students = students.drop(columns = 'ID')

>>> students

|  | **Exam1** | **Exam2** |
| --- | --- | --- |
| **Joey** | 95 | 100 |
| **Javier** | 99 | 99 |
| **Juanita** | 88 | 98 |
| **Jane** | 100 | 97 |

Statistical methods that can be used include mean(), sum(), min(), max(), std(), and var(). They can be used with the entire DataFrame, or an individual column.

>>> students.mean()

Exam1 95.5

Exam2 98.5

dtype: float64

If a DataFrame object is large, which it frequently will be if read from a file, the first 5 rows can be viewed using the **head**() method, and the last 5 rows can be viewed using the **tail**() method.