Session 01

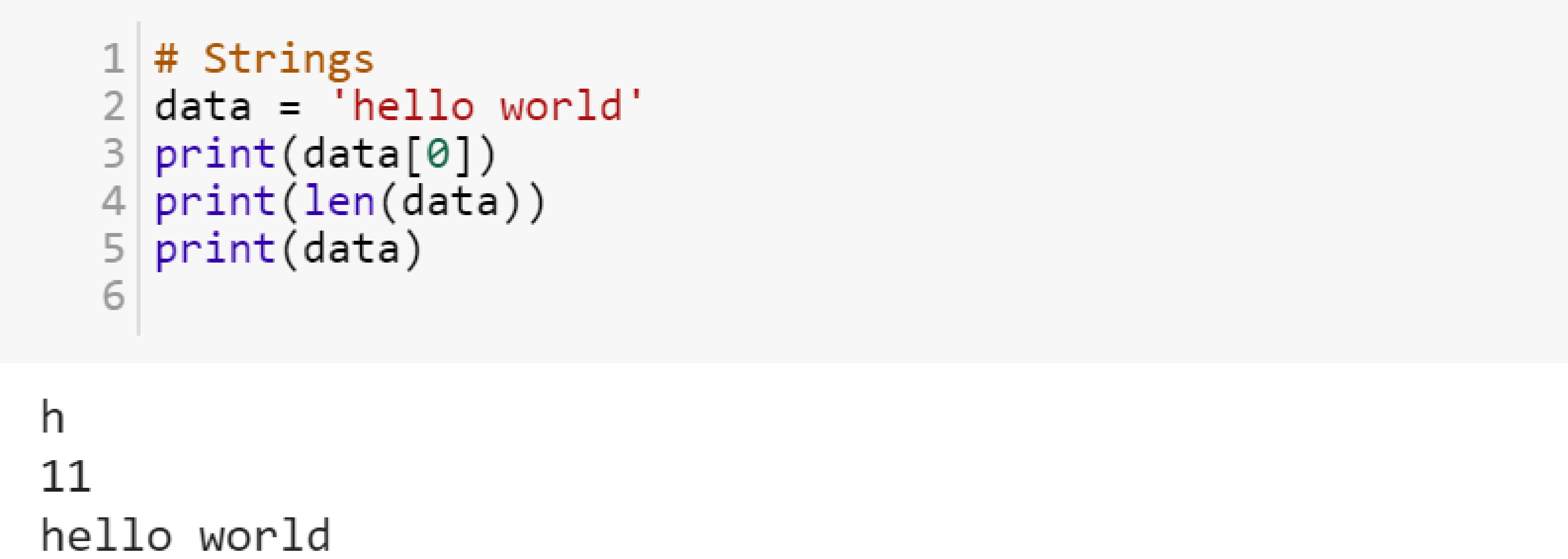
Python Crash course

Contents:

- Assignment.
- Flow Control. □
- Data Structures. □
- Functions.
- Numpy
- Pandas
- Matplotlib

Assignment

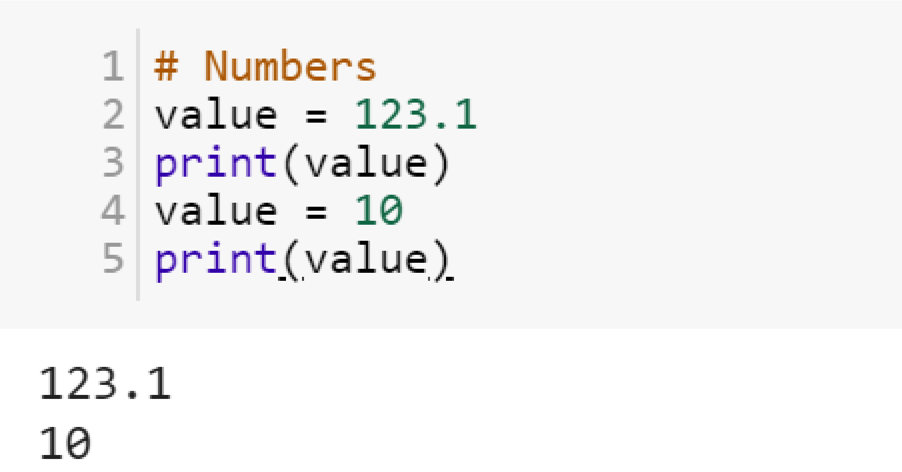
- Strings:



```python
# Strings
data = 'hello world'
print(data[0])
print(len(data))
print(data)

```

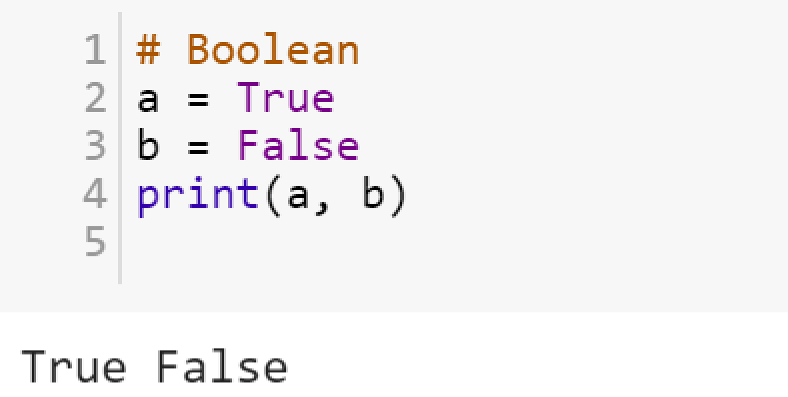
```
h
11
hello world
```

Numbers:



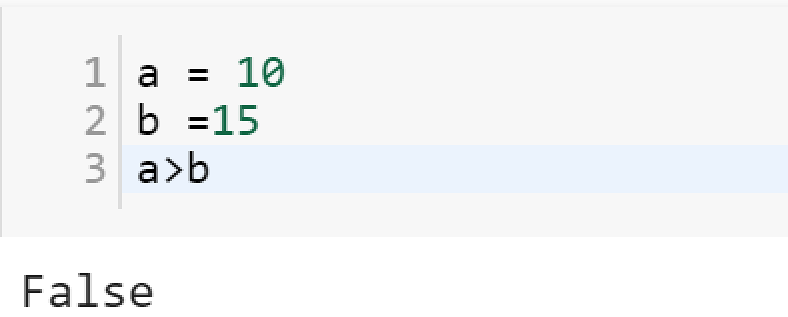
```python
# Numbers
value = 123.1
print(value)
value = 10
print(value)
```

```
123.1
10
```



```python
# Boolean
a = True
b = False
print(a, b)

```

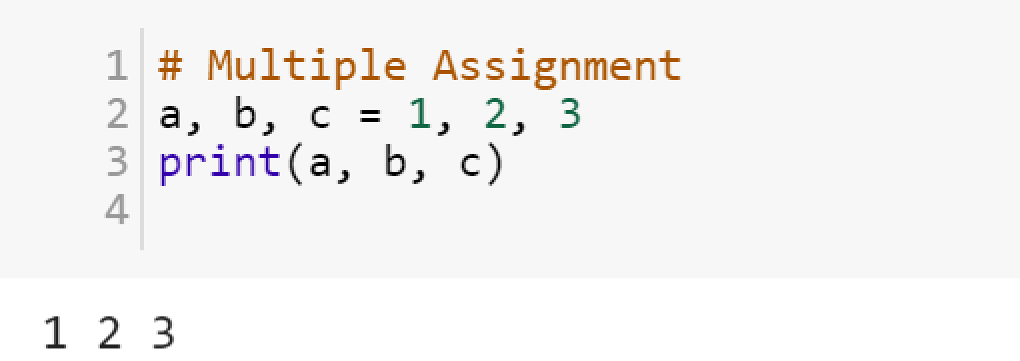
```
True False
```



```python
a = 10
b =15
a>b
```

```
False
```

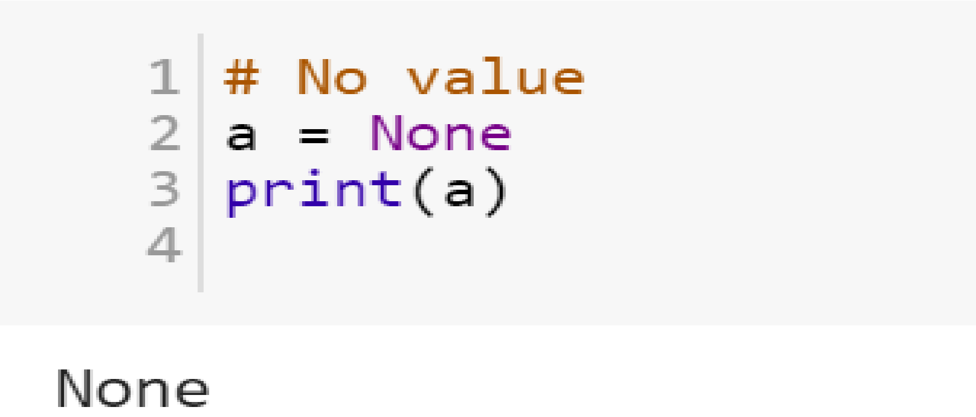
Multiple Assignment



```python
# Multiple Assignment
a, b, c = 1, 2, 3
print(a, b, c)

```

```
1 2 3
```

No Value

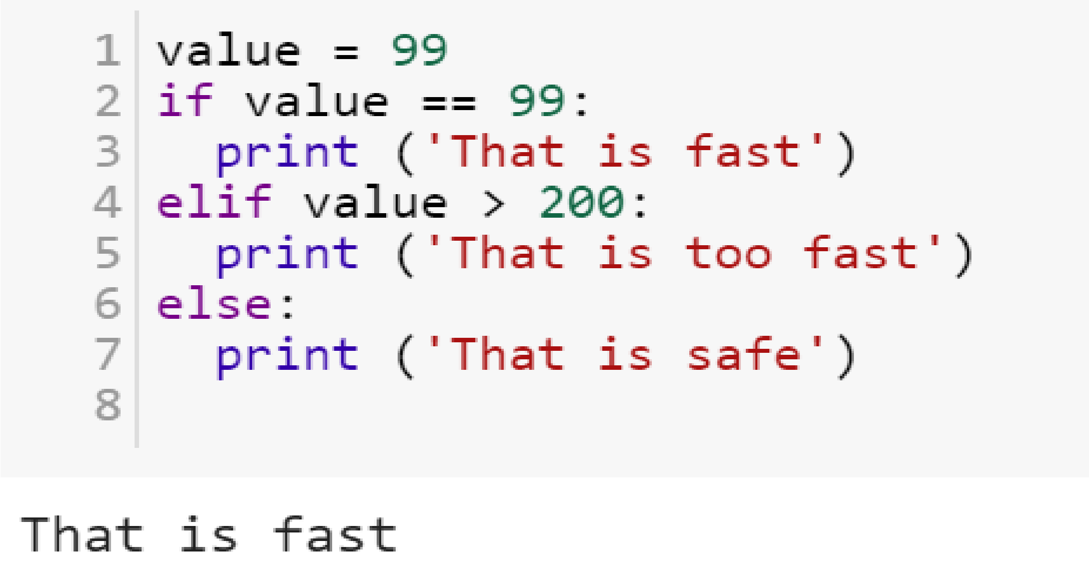


```python
# No value
a = None
print(a)

```

```
None
```

Flow control

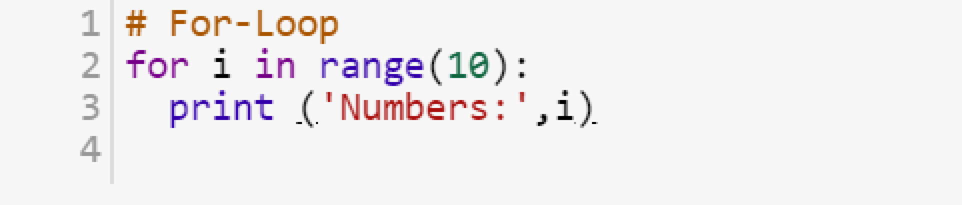
- If–Then–Else Conditional



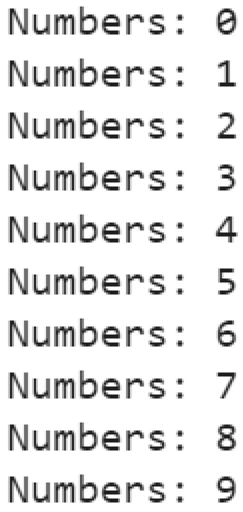
```python
value = 99
if value == 99:
  print ('That is fast')
elif value > 200:
  print ('That is too fast')
else:
  print ('That is safe')

```

```
That is fast
```

For–Loop

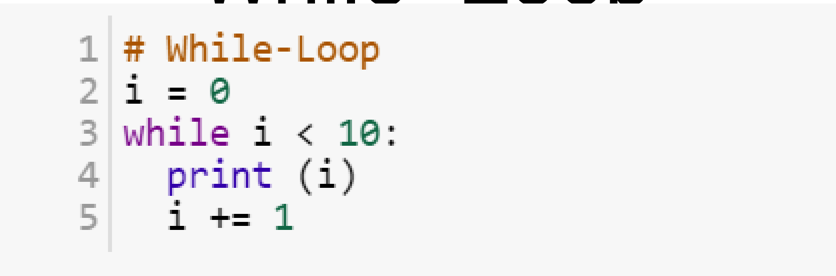


```python
# For-Loop
for i in range(10):
  print ('Numbers:',i)

```

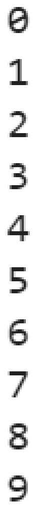


```
Numbers: 0
Numbers: 1
Numbers: 2
Numbers: 3
Numbers: 4
Numbers: 5
Numbers: 6
Numbers: 7
Numbers: 8
Numbers: 9
```

While–Loop



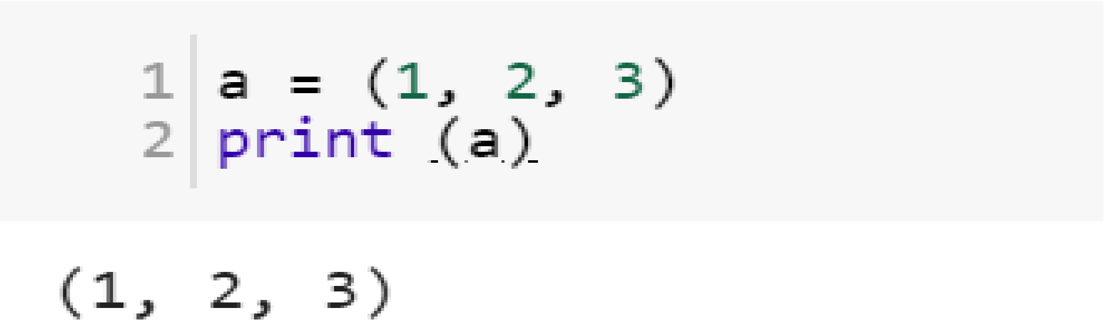
```python
# While-Loop
i = 0
while i < 10:
  print (i)
  i += 1
```



```
0
1
2
3
4
5
6
7
8
9
```

Data Structures

- There are three data structures in Python that you will find the most used and useful.
- They are tuples, lists and dictionaries.
- Tuples are read-only collections of items.

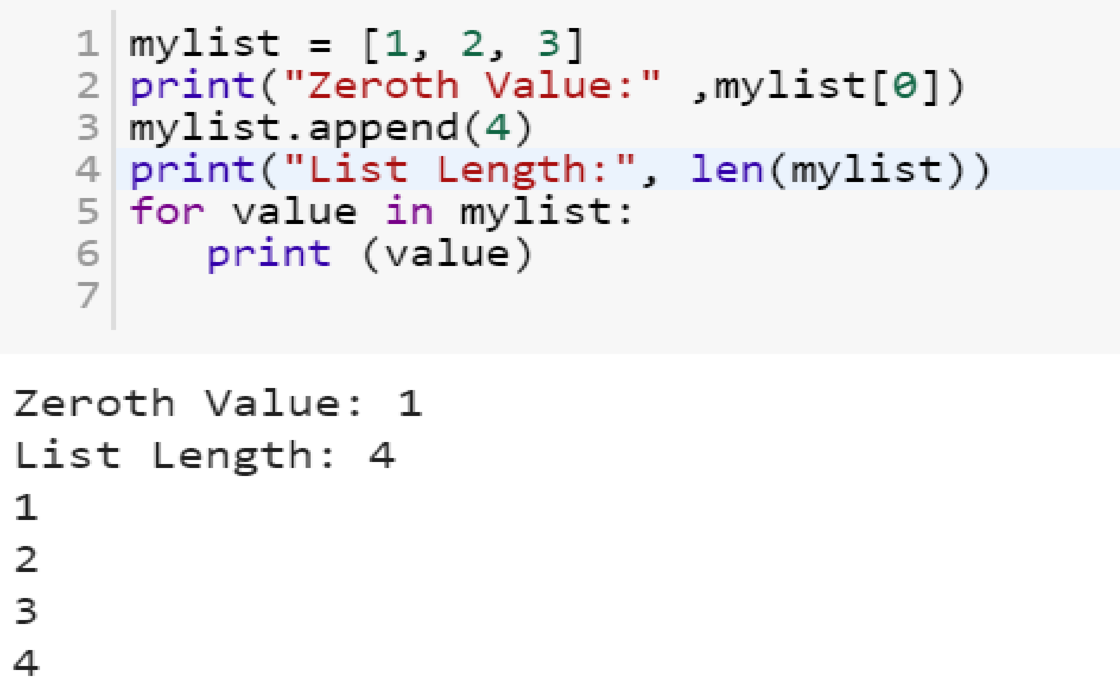


```
a = (1, 2, 3)
print (a)
```

```
(1, 2, 3)
```

List

- Lists use the square bracket notation and can be index using array notation.

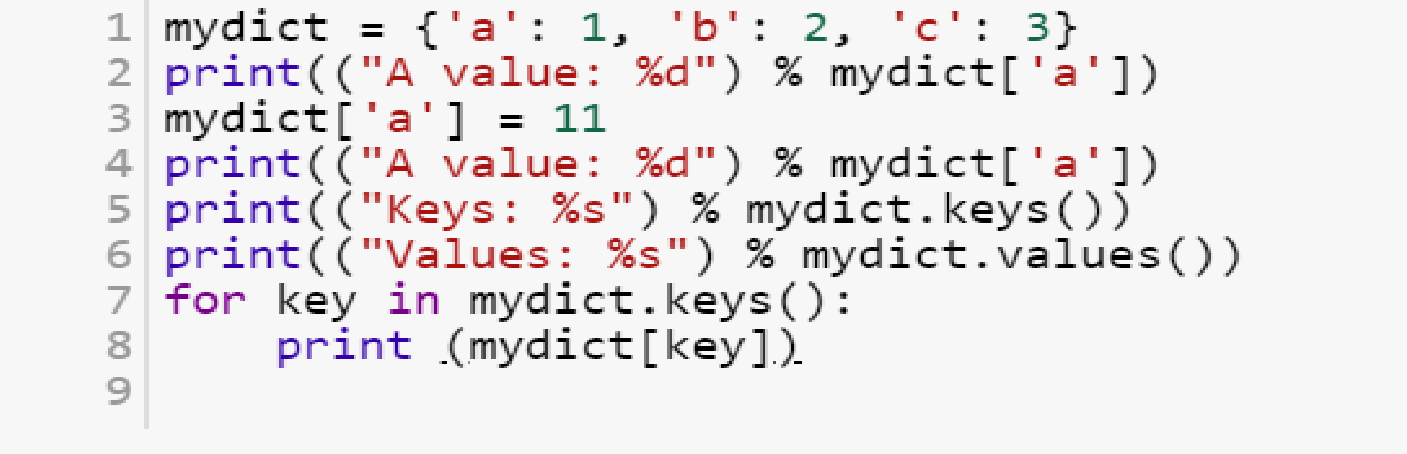


```python
mylist = [1, 2, 3]
print("Zeroth Value:" ,mylist[0])
mylist.append(4)
print("List Length:", len(mylist))
for value in mylist:
    print (value)

```

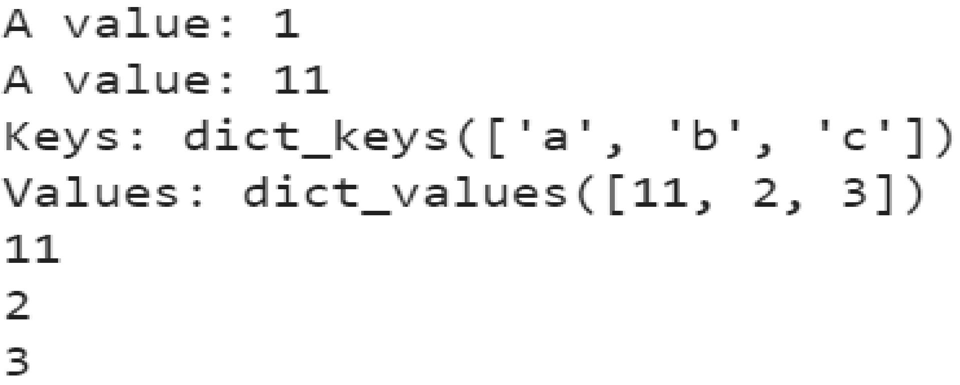
```
Zeroth Value: 1
List Length: 4
1
2
3
4
```

Dictionary

- Dictionaries are mappings of names to values, like key-value pairs.



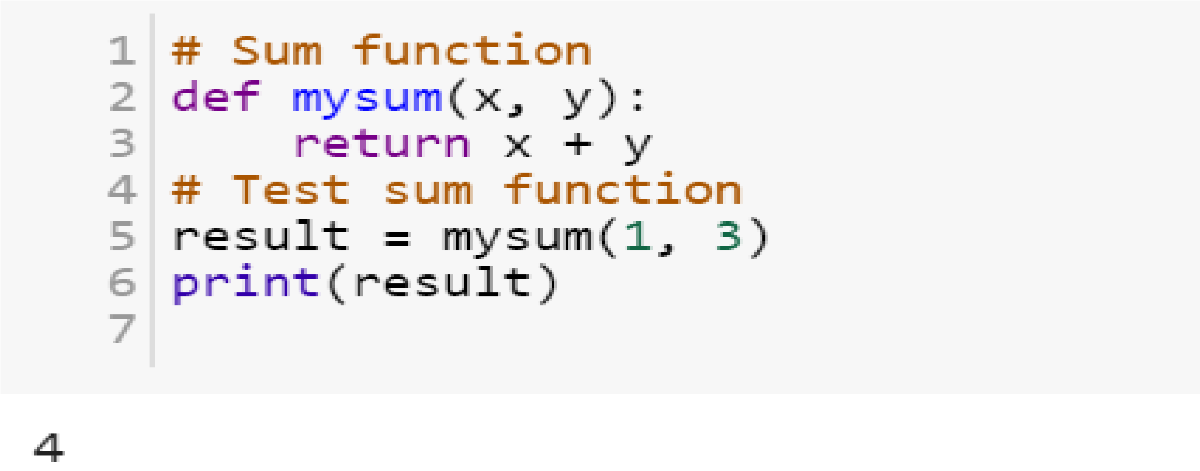
```python
mydict = {'a': 1, 'b': 2, 'c': 3}
print(("A value: %d") % mydict['a'])
mydict['a'] = 11
print(("A value: %d") % mydict['a'])
print(("Keys: %s") % mydict.keys())
print(("Values: %s") % mydict.values())
for key in mydict.keys():
    print (mydict[key])

```



```
A value: 1
A value: 11
Keys: dict_keys(['a', 'b', 'c'])
Values: dict_values([11, 2, 3])
11
2
3
```

Functions

- Functions The biggest gotcha with Python is the whitespace.
- The example below defines a new function to calculate the sum of two values and calls the function with two arguments.

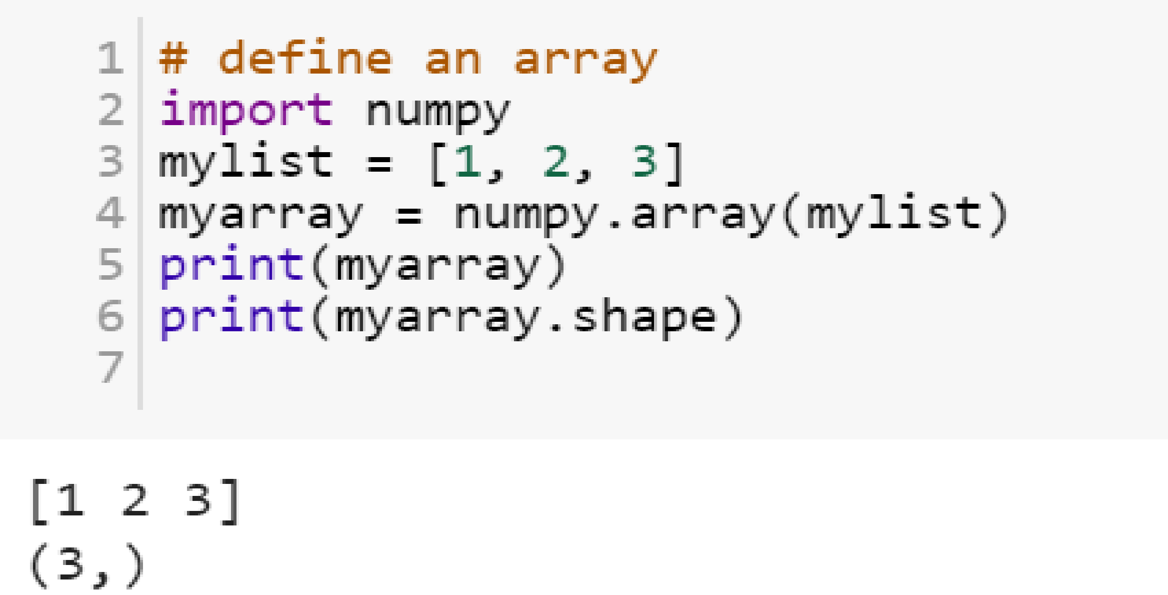


```python
# Sum function
def mysum(x, y):
    return x + y
# Test sum function
result = mysum(1, 3)
print(result)

```

```
4
```

NumPy Crash Course

- NumPy provides the foundation data structures and operations for SciPy.
- These are arrays (ndarrays) that are efficient to define and manipulate.

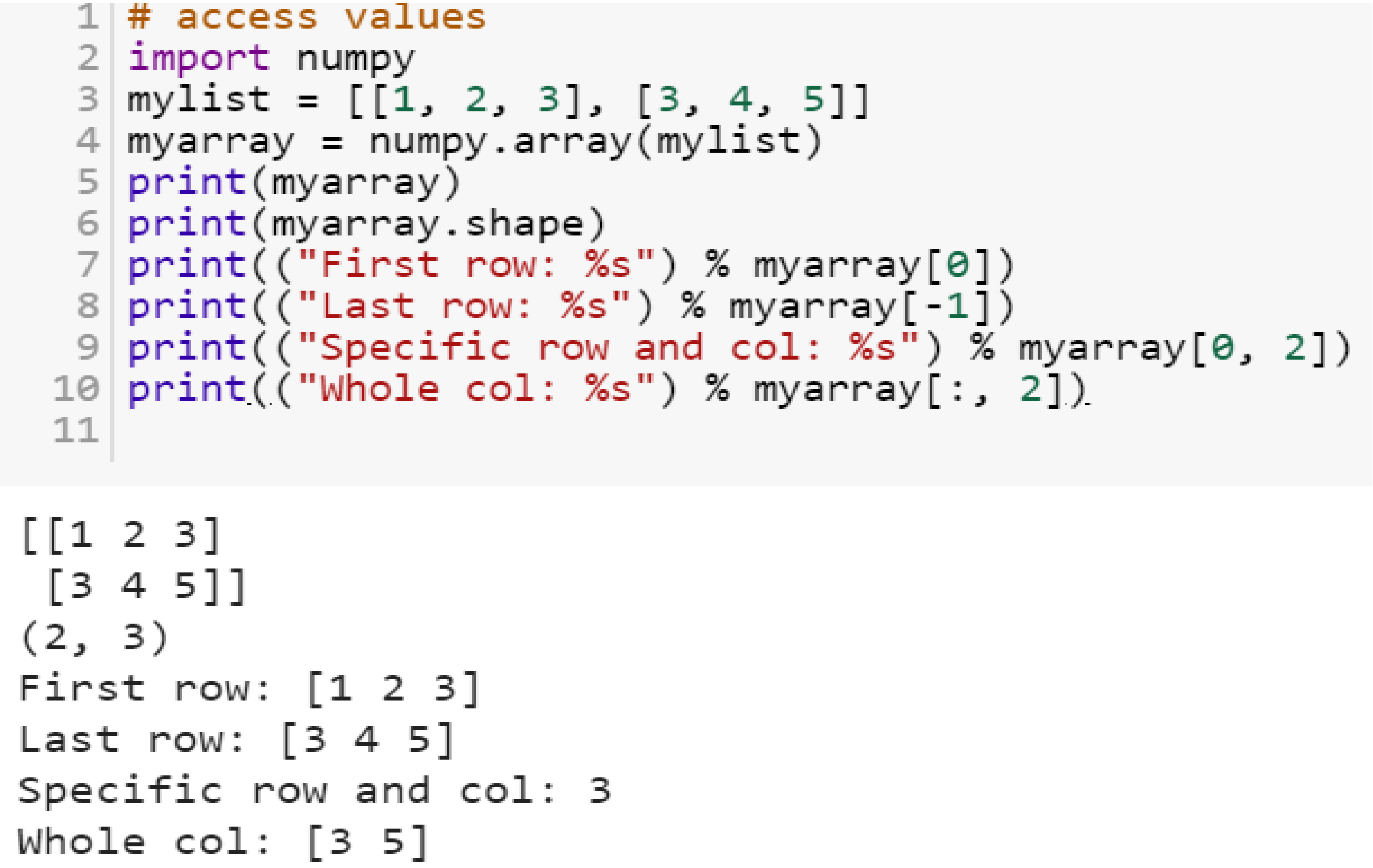


```python
# define an array
import numpy
mylist = [1, 2, 3]
myarray = numpy.array(mylist)
print(myarray)
print(myarray.shape)

```

```
[1 2 3]
(3,)
```

Access Data

- Array notation and ranges can be used to efficiently access data in a NumPy array.

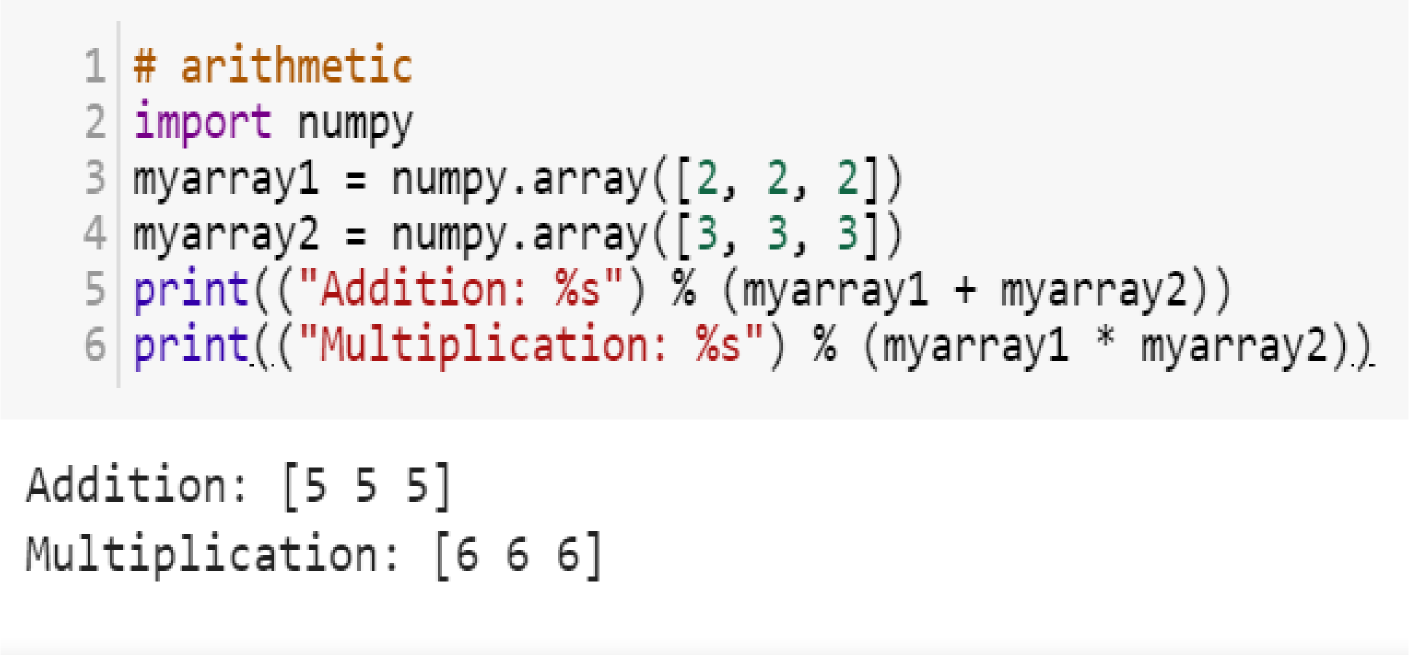


```python
# access values
import numpy
mylist = [[1, 2, 3], [3, 4, 5]]
myarray = numpy.array(mylist)
print(myarray)
print(myarray.shape)
print(("First row: %s") % myarray[0])
print(("Last row: %s") % myarray[-1])
print(("Specific row and col: %s") % myarray[0, 2])
print(("Whole col: %s") % myarray[:, 2])

```

```
[[1 2 3]
 [3 4 5]]
(2, 3)
First row: [1 2 3]
Last row: [3 4 5]
Specific row and col: 3
Whole col: [3 5]
```

Arithmetic

- NumPy arrays can be used directly in arithmetic.

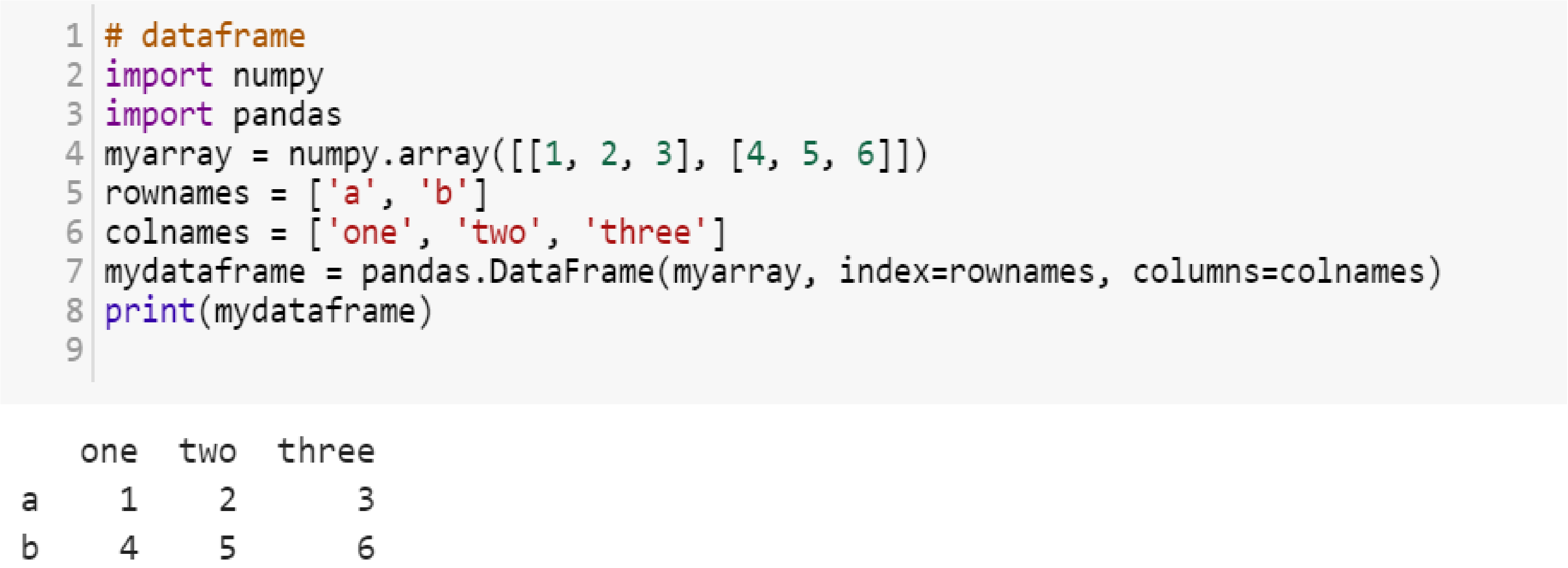


```python
# arithmetic
import numpy
myarray1 = numpy.array([2, 2, 2])
myarray2 = numpy.array([3, 3, 3])
print(("Addition: %s") % (myarray1 + myarray2))
print(("Multiplication: %s") % (myarray1 * myarray2))
```

```
Addition: [5 5 5]
Multiplication: [6 6 6]
```

DataFrames

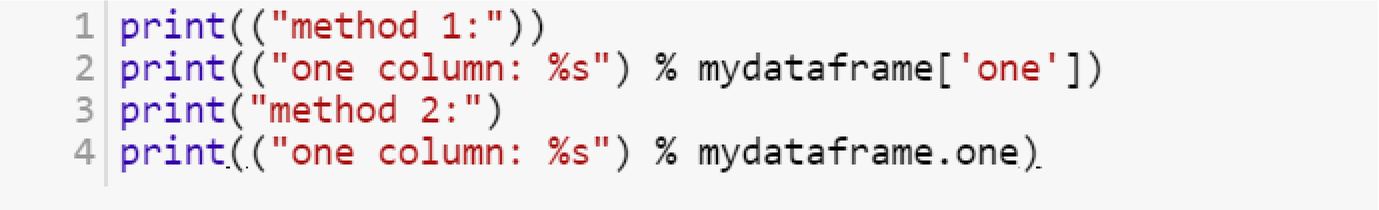
- A data frame is a multi-dimensional array where the rows and the columns can be labeled.



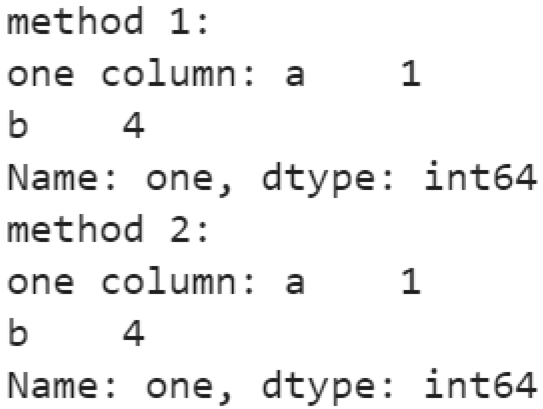
```python
# dataframe
import numpy
import pandas
myarray = numpy.array([[1, 2, 3], [4, 5, 6]])
rownames = ['a', 'b']
colnames = ['one', 'two', 'three']
mydataframe = pandas.DataFrame(myarray, index=rownames, columns=colnames)
print(mydataframe)

```

```
   one  two  three
a    1    2      3
b    4    5      6
```

Accessing DataFrame



```
print(("method 1:"))
print(("one column: %s") % mydataframe['one'])
print("method 2:")
print(("one column: %s") % mydataframe.one)
```

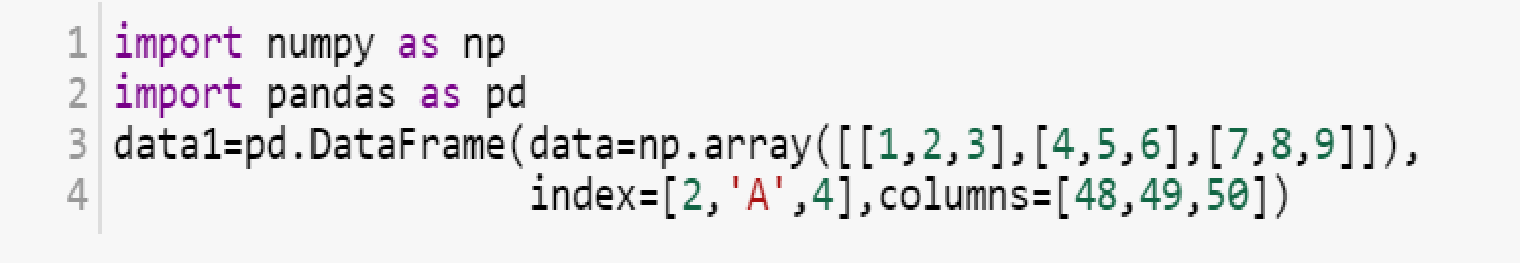


```
method 1:
one column: a    1
b    4
Name: one, dtype: int64
method 2:
one column: a    1
b    4
Name: one, dtype: int64
```

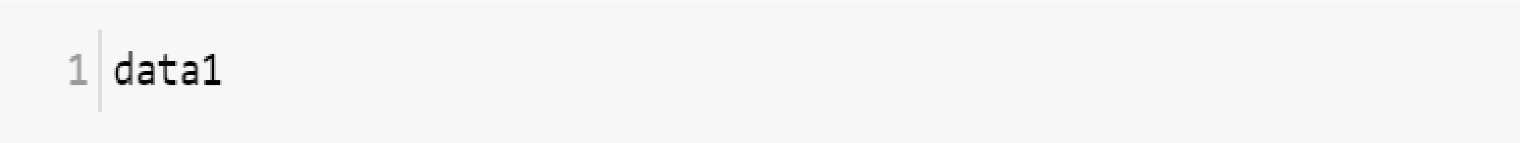
Accessing the data with index

- More on indexes
- .loc[ ] works on labels of your index.
- This means that if you give in loc[2], you look for the values of
- your DataFrame that have an index labeled 2.
- .iloc [ ] works on the positions in your index.
- This means that if you give in iloc[2], you look for the values of
- your DataFrame that are at index ’2`.
- .ix[ ] is a more complex case:
- when the index is integer-based, you pass a label to .ix[].

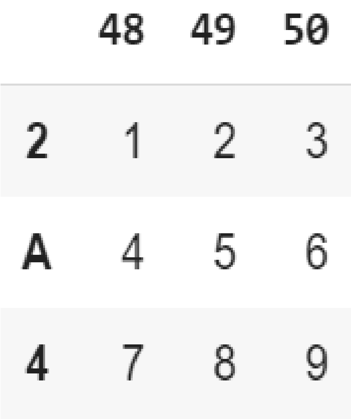
More on DataFrames



```python
import numpy as np
import pandas as pd
data1=pd.DataFrame(data=np.array([[1,2,3],[4,5,6],[7,8,9]]),
                   index=[2,'A',4],columns=[48,49,50])
```

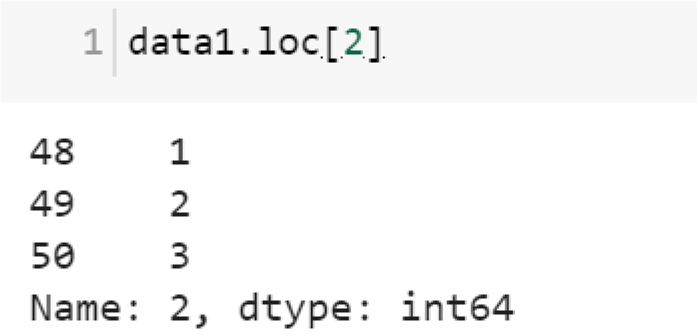


```python
data1
```



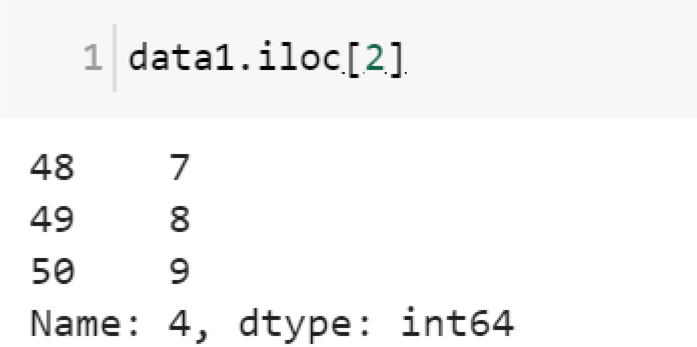
```
   48  49  50
2   1   2   3
A   4   5   6
4   7   8   9
```

Using index



```
data1.loc[2]
```

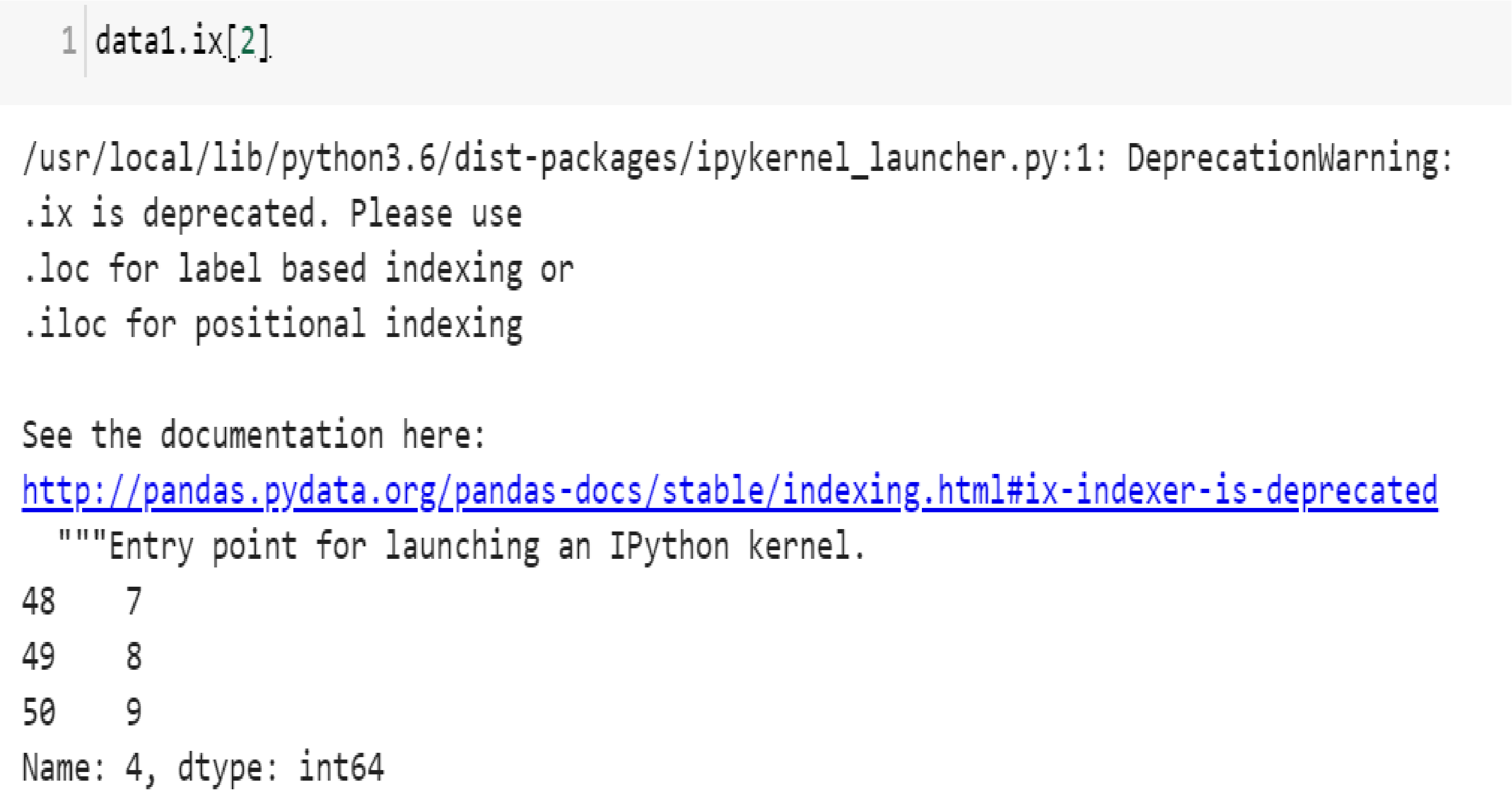
```
48    1
49    2
50    3
Name: 2, dtype: int64
```



```
data1.iloc[2]
```

```
48    7
49    8
50    9
Name: 4, dtype: int64
```

More on index



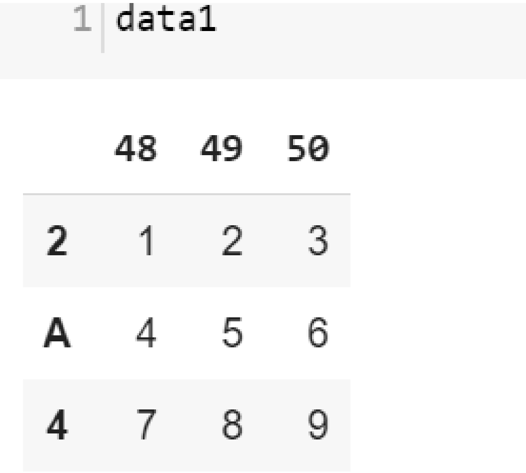
```
data1.ix[2]
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  """Entry point for launching an IPython kernel.
48    7
49    8
50    9
Name: 4, dtype: int64
```

Deleting a Column from Your DataFrame

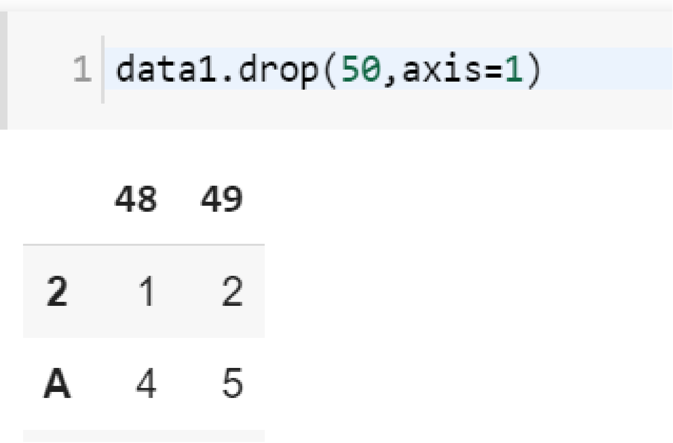
- To get rid of (a selection of) columns from your DataFrame, you can use the drop() method:
- The axis argument is either 0 when it indicates rows and 1 when it is used to drop columns.
- You can set inplace to True to delete the column without having to reassign.

Example



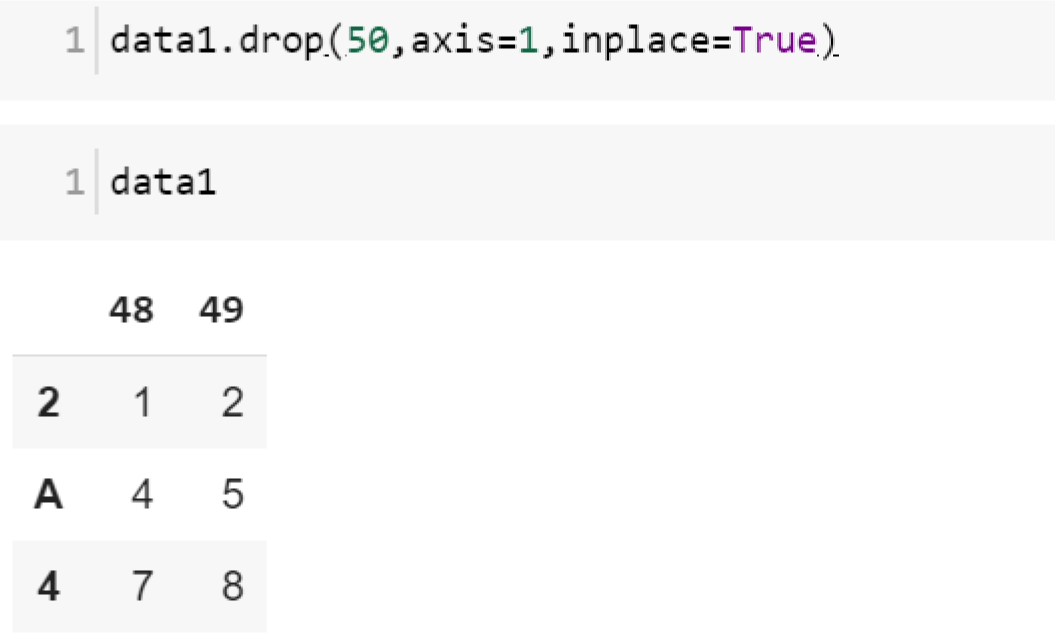
```
data1
```

	48	49	50
2	1	2	3
A	4	5	6
4	7	8	9



```
data1.drop(50,axis=1)
```

	48	49
2	1	2
A	4	5

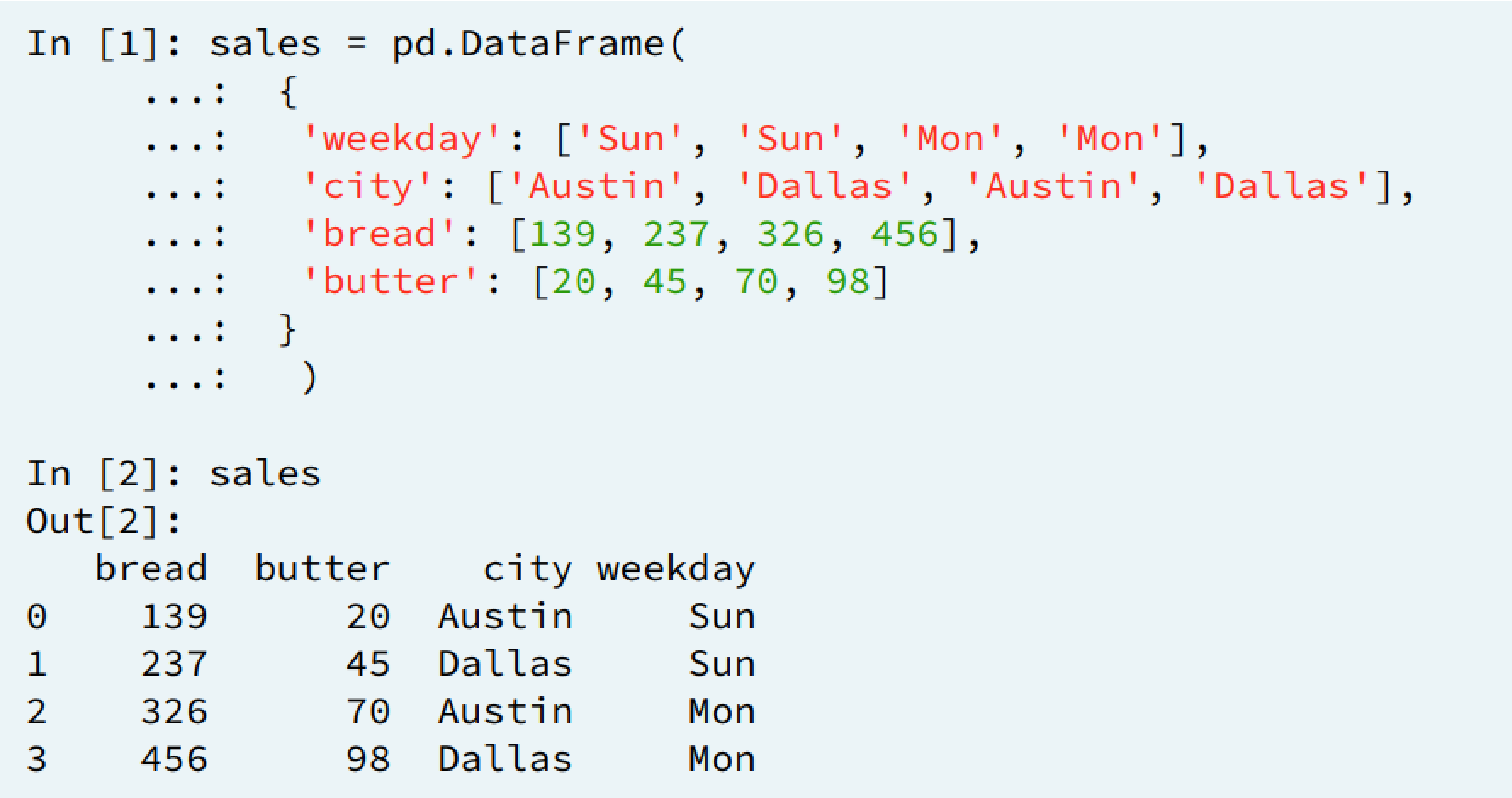


```
data1.drop(50,axis=1,inplace=True)
```

```
data1
```

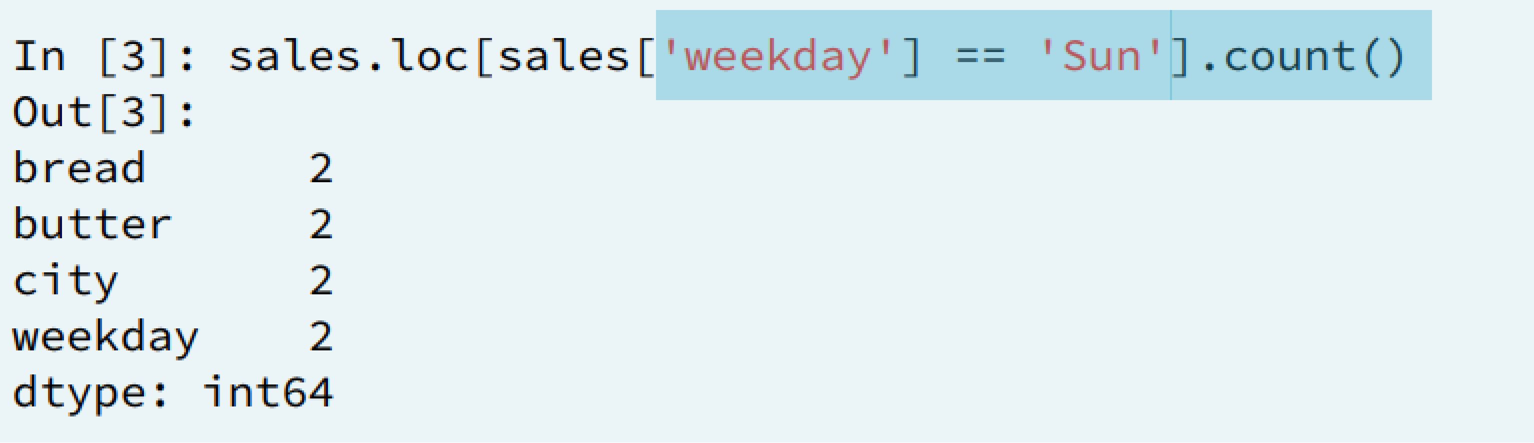
	48	49
2	1	2
A	4	5
4	7	8

Categorical and group by



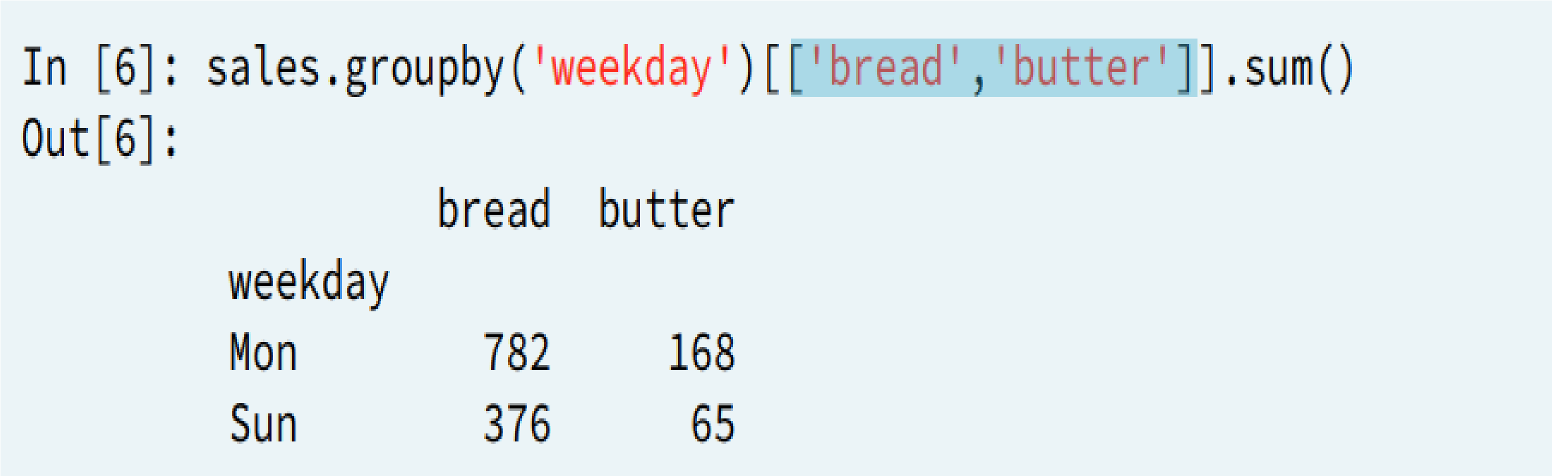
```
In [1]: sales = pd.DataFrame(
     ...:  {
     ...:   'weekday': ['Sun', 'Sun', 'Mon', 'Mon'],
     ...:   'city': ['Austin', 'Dallas', 'Austin', 'Dallas'],
     ...:   'bread': [139, 237, 326, 456],
     ...:   'butter': [20, 45, 70, 98]
     ...:  }
     ...:   )

In [2]: sales
Out[2]:
   bread  butter    city weekday
0    139      20  Austin     Sun
1    237      45  Dallas     Sun
2    326      70  Austin     Mon
3    456      98  Dallas     Mon
```

Boolean filter and count



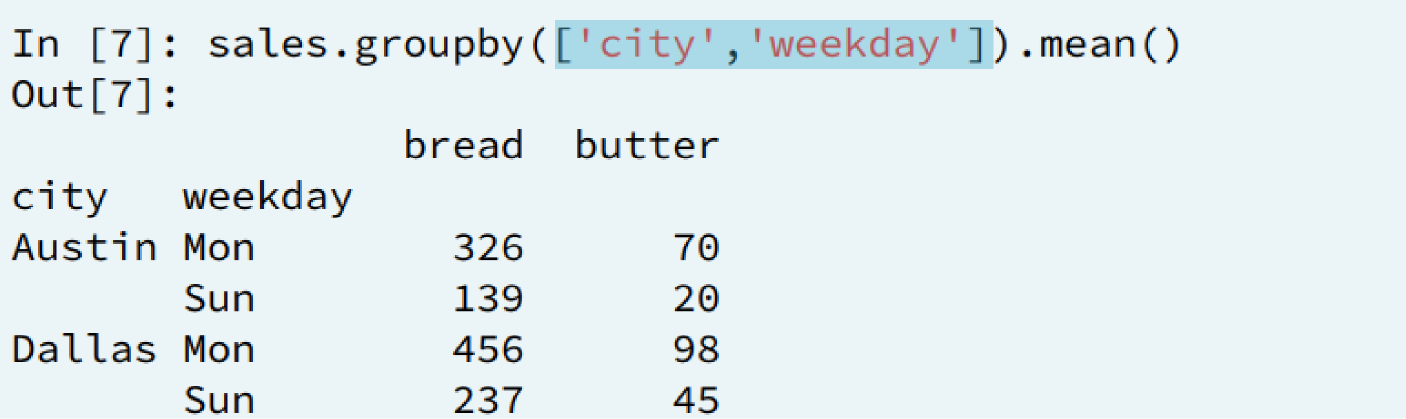
```
In [3]: sales.loc[sales['weekday'] == 'Sun'].count()
Out[3]:
bread      2
butter     2
city       2
weekday    2
dtype: int64
```

Group by and sum: multiple columns

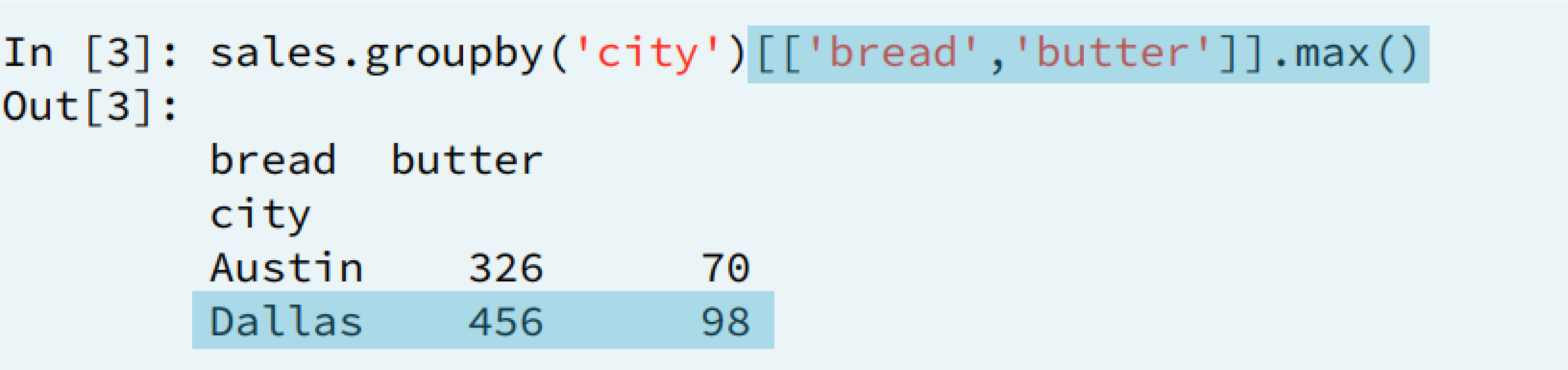


```
In [6]: sales.groupby('weekday')[['bread','butter']].sum()
Out[6]:
         bread  butter
weekday
Mon        782     168
Sun        376      65
```

Group by and mean: multi-level index



```
In [7]: sales.groupby(['city','weekday']).mean()
Out[7]:
                bread  butter
city   weekday
Austin Mon        326      70
       Sun        139      20
Dallas Mon        456      98
       Sun        237      45
```



```
In [3]: sales.groupby('city')[['bread','butter']].max()
Out[3]:
        bread  butter
        city
        Austin    326      70
        Dallas    456      98
```

- Thank you