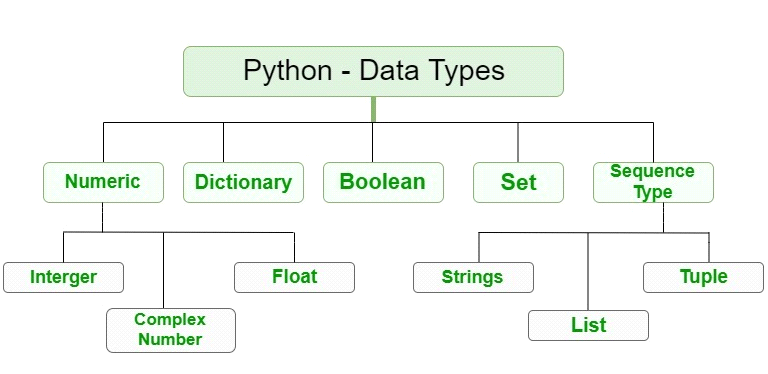
**Module 3 – Introduction to Numpy and Pandas**

**3.1 Python Data Types**

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. The type() function is used to determine the type of data type.

the standard or built-in data type of Python:

* Numeric
* Sequence Type
* Boolean
* Set
* Dictionary



**Numeric -** Numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex

* **Integers**– This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal)
* **Float**– This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point.
* **Complex Numbers** – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

**Sequence Type -**  sequence is the ordered collection of similar or different data types

* **Strings** are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote or triple quote
* [**Lists**](https://www.geeksforgeeks.org/python-list/)are just like the arrays, declared in other languages which is a ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.
* **[Tuple](https://www.geeksforgeeks.org/python-tuples/)** is also an ordered collection of Python objects. The only difference between tuple and list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by tuple class

**Boolean** - Data type with one of the two built-in values, True or False.True and False with capital ‘T’ and ‘F’ are valid booleans otherwise python will throw an error

**Set** -  [Set](https://www.geeksforgeeks.org/python-sets/) is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements

**Dictionary** - [Dictionary](https://www.geeksforgeeks.org/python-dictionary/) in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a ‘comma’.

**3.2 Basics of Numpy Arrays**

**What Is A Python Numpy Array?**

NumPy is a Python library that is the core library for scientific computing in Python. It contains a collection of tools and techniques that can be used to solve on a computer mathematical models of problems in Science and Engineering. The library’s name is short for “Numeric Python” or “Numerical Python”. One of these tools is a high-performance multidimensional array object that is a powerful data structure for efficient computation of arrays and matrices.

on a structural level, an array is basically nothing but pointers. It’s a combination of a memory address, a data type, a shape, and strides:

* The data pointer indicates the memory address of the first byte in the array,
* The data type or dtype pointer describes the kind of elements that are contained within the array,
* The shape indicates the shape of the array, and
* The strides are the number of bytes that should be skipped in memory to go to the next element. If your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.

**How To Make NumPy Arrays**

To make a numpy array, you can just use the np.array() function. All you need to do is pass a list to it, and optionally, you can also specify the data type of the data

>>># Import `numpy` as `np`

>>>import numpy as np

>>># Make the array `my\_array`

>>>my\_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

>>># Print `my\_array`

>>>print(my\_array)

**Output**

[[1 2 3 4]

[5 6 7 8]]

**How To Make An “Empty” NumPy Array**

“empty” arrays are used to create initial placeholders, You can initialize arrays with ones or zeros or constant or random values.

# Create an array of ones

np.ones((3,4))

Output

array([[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.]])

# Create an array with random values

>>>np.random.random((2,2))

Output

array([[ 0.43857224, 0.0596779 ],

[ 0.39804426, 0.73799541]])

# Create an empty array

np.empty((3,2))

Output

array([[ 0., 0.],

[ 0., 0.],

[ 0., 0.]])

**Note** – there are n number of such functions that can be used to specify the kind of values the array must contain.

**How To Load NumPy Arrays From Text**

Creating arrays with the help of initial placeholders or with some example data is an excellent way of getting started with numpy. But when you want to get started with data analysis, you’ll need to load data from text files, functions  such as loadtxt() can be used for this.

# This is your data in the text file

# Value1 Value2 Value3

# 0.2536 0.1008 0.3857

# 0.4839 0.4536 0.3561

# 0.1292 0.6875 0.5929

# 0.1781 0.3049 0.8928

# 0.6253 0.3486 0.8791

import numpy as np

File\_data=np.loadtxt('E:\data.txt',dtype=int)

print(File\_data)

**How To Save NumPy Arrays**

you can also save arrays to a file, if you want to save the array to a text file, you can use the savetxt() function to do this

import numpy as np

x = np.random.random((2,2))

np.savetxt('test.out', x, delimiter=',')

**Broadcasting Numpy arrays**

The term broadcasting refers to how numpy treats arrays with different Dimension during arithmetic operations which lead to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python as we know that Numpy implemented in C

Example -

import numpy as np

A = np.array([5, 7, 3, 1])

B = np.array([90, 50, 0, 30])

# array are compatible because of same Dimension

c = a \* b

print (c)

Output -

[[ 3 16 10]

[ 18 28 48]

[ 0 -10 55]]

Broadcasting Rules:

Broadcasting two arrays together follow these rules:

1. If the arrays don’t have the same rank then prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together iff they are compatible with all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the element-wise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

Broadcasting is also frequently used in displaying images based on two-dimensional functions. If we want to define a function z=f(x, y).

Example -

import numpy as np

import matplotlib.pyplot as plt

#To Demonstrate Numpy Broadcasting

A = np.array([[11, 22, 33], [10, 20, 30]])

print(A)

b = 4

print(b)

C = A + b

print(C)

# Computes x and y coordinates for

# points on sine and cosine curves

x = np.arange(0, 3 \* np.pi, 0.1)

y\_sin = np.sin(x)

y\_cos = np.cos(x)

# Plot the points using matplotlib

plt.plot(x, y\_sin)

plt.plot(x, y\_cos)

plt.xlabel('x axis label')

plt.ylabel('y axis label')

plt.title('Sine and Cosine')

plt.legend(['Sine', 'Cosine'])

plt.show()

Output -

[[11 22 33]

[10 20 30]]

4

[[15 26 37]

[14 24 34]]

**3.3 Computation on Numpy Arrays**

You can just use +, -, \*, / or % to add, subtract, multiply, divide or calculate the remainder of two (or more) arrays. However, a big part of why NumPy is so handy, is because it also has functions to do this. The equivalent functions of the operations that you have seen just now are, respectively, np.add(), np.subtract(), np.multiply(), np.divide() and np.remainder(). Below are some of the functions which can be used for arithmetic operations on arrays.

|  |  |
| --- | --- |
| **Function** | **Operation** |
| np.add() | Addition |
| np.subtract() | Subtraction |
| np.multiply() | Multiplication |
| np.divide() | Divide |
| np.remainder() | Reminder |
| np.exp() | Exponential |
| np.sqrt() | Squarteroot |
| np.sin() | Sine |
| np.cos() | Cosine |
| np.log() | natural logarithm |
| dot() | dot product |
| a.sum() | Array-wise sum |
| a.min() | Array-wise minimum value |
| b.max(axis=0) | Maximum value of an array row |
| b.cumsum(axis=1) | Cumulative sum of the elements |
| a.mean() | Mean |
| b.median() | Median |
| a.corrcoef() | Correlation coefficient |
| np.std(b) | Standard deviation |

Example

**Array X**

[[1 2 3]

[3 4 5]]

**Array Y**

[[1 2 3]

[3 4 5]]

# Add `x` and `y`

np.add(x,y)

# Subtract `x` and `y`

np.subtract(x,y)

**How To Subset, Slice, And Index Arrays**

Besides mathematical operations, you might also consider taking just a part of the original array (or the resulting array) or just some array elements to use in further analysis or other operations. In such case, you will need to subset, slice and/or index your arrays. These operations are very similar to when you perform them on Python lists.

To Subset the data, the index or indices needs to be passed through the square brackets

Example

>>>np.random.random((2,2))

Output

array([[ 0.43857224, 0.0596779 ],

[ 0.39804426, 0.73799541]])

print(np [1,1])

Output - 0.43857224

print(np [2,1])

Output - 0.39804426

To Slice the data, a colon(:) can be used

Example –

my\_2d\_array

[[1 2 3 4]

[5 6 7 8]]

# Select items at row 0 and 1, column 1

print(my\_2d\_array[0:2,1])

Output – [2 6]

**Array Manipulation - Searching,Sorting and Splitting of Arrays**

# Python program to demonstrate sorting in numpy

import numpy as np

import numpy as np2

a = np.array([[1, 4, 2],

[3, 4, 6],

[0, -1, 5]])

#array before sorting

print ("Array elements before sorting :\n",a)

# sorted array

print ("Array elements in sorted order:\n",

np.sort(a, axis = None))

i = np.where(a == 6)

print("i = {}".format(i))

# specify sort algorithm

print ("Column wise sort by applying merge-sort:\n",

np.sort(a, axis = 0, kind = 'mergesort'))

# looking for value 30 in arr and storing its index in i

#Splitting of Arrays

b = np.array([[3, 4, 5],

[6, 7, 8],

[9, 10, 11]])

print(b)

print(b[0:3,1])

**Output -**

('Array elements before sorting :\n', array([[ 1, 4, 2],

[ 3, 4, 6],

[ 0, -1, 5]]))

('Array elements in sorted order:\n', array([-1, 0, 1, 2, 3, 4, 4, 5, 6]))

i = (array([1]), array([2]))

('Column wise sort by applying merge-sort:\n', array([[ 0, -1, 2],

[ 1, 4, 5],

[ 3, 4, 6]]))

[[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

[ 4 7 10]

**3.4 Pandas – Introduction to Pandas data structure**

Pandas is a popular Python package for data science, and with good reason: it offers powerful, expressive and flexible data structures that make data manipulation and analysis easy, among many other things. The DataFrame is one of these structures.

DataFrames in Python come with the [Pandas](http://pandas.pydata.org/) library, and they are defined as two-dimensional labeled data structures with columns of potentially different types. A Dataframe is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. you could say that the Pandas DataFrame consists of three main components: the data, the index, and the columns.

* Firstly, the DataFrame can contain data that is:
* a Pandas DataFrame
* a Pandas Series: a one-dimensional labeled array capable of holding any data type with axis labels or index. An example of a Series object is one column from a DataFrame.
* a NumPy ndarray, which can be a record or structured
* a two-dimensional ndarray
* dictionaries of one-dimensional ndarray’s, lists, dictionaries or Series.
* Besides data, you can also specify the index and column names for your DataFrame. The index, on the one hand, indicates the difference in rows, while the column names indicate the difference in columns.

**3.5 Essential functionality**

**How To Create a Pandas DataFrame**

A basic DataFrame, which can be created is an Empty Dataframe. An Empty Dataframe is created just by calling a dataframe constructor. To make a data frame from a NumPy array, you can just pass it to the DataFrame() function in the data argument.

Example –

# import pandas as pd

import pandas as pd

# list of strings

lst = ['Geeks', 'For', 'Geeks', 'is',

'portal', 'for', 'Geeks']

# Calling DataFrame constructor on list

df = pd.DataFrame(lst)

print(df)

output:

0

0 Geeks

1 For

2 Geeks

3 is

4 portal

5 for

6 Geeks

# Python code demonstrate creating

# DataFrame from dict narray / lists

# By default addresses.

import pandas as pd

# initialise data of lists.

data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}

# Create DataFrame

df = pd.DataFrame(data)

# Print the output.

print(df)

output:

Age Name

0 20 Tom

1 21 nick

2 19 krish

3 18 jack

# importing pandas as pd

import pandas as pd

# dictionary of lists

dict = {'name':["aparna", "pankaj", "sudhir", "Geeku"],

'degree': ["MBA", "BCA", "M.Tech", "MBA"],

'score':[90, 40, 80, 98]}

df = pd.DataFrame(dict)

print(df)

Output -

degree name score

0 MBA aparna 90

1 BCA pankaj 40

2 M.Tech sudhir 80

3 MBA Geeku 98

**Fundamental DataFrame Operations**

**How To Select an Index or Column From a Pandas DataFrame**

selecting an index, column or value from your DataFrame similar to what you see in other languages (or packages!) that are used for data analysis.

let’s say you have a DataFrame like this one and you want to access the value that is at index 0, in column ‘A’.

A B C

0 1 2 3

1 4 5 6

2 7 8 9

Different ways to do this is

import pandas as pd

# initialise data of lists.

data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}

# Create DataFrame

df = pd.DataFrame(data)

# Print the output.

print(df.iloc[0][0])

print(df.loc[0]['Age'])

print(df.at[0,'Age'])

print(df.iat[0,0])

output:

20

20

20

20

To access rows and columns

Example

import pandas as pd

# initialise data of lists.

data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}

# Create DataFrame

df = pd.DataFrame(data)

# Print the output.

print(df.iloc[0])

print(df.loc[:,'Age'])

output:

Age 20

Name Tom

Name: 0, dtype: object

0 20

1 21

2 19

3 18

Name: Age, dtype: int64

you can either access the values by calling them by their label or by their position in the index or column.

**How To Add an Index, Row or Column to a Pandas DataFrame**

**Adding an Index**

When you create a DataFrame, you have the option to add input to the ‘index’ argument to make sure that you have the index that you desire. When you don’t specify this, your DataFrame will have, by default, a numerically valued index that starts with 0 and continues until the last row of your DataFrame. You can re-use one of your columns and make it your index by calling set\_index() on your DataFrame.

# Print out your DataFrame `df` to check it out

print(df)

# Set 'C' as the index of your DataFrame

df.set\_index('C')

output:

A B C

0 1 2 3

1 4 5 6

**Adding a Row**

.loc[] ,.iloc[] and .ix[] are some attributes used for this purpose.

* .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[] - when the index is integer-based, you pass a label to .ix[]. ix[2] then means that you’re looking in your DataFrame for values that have an index labeled 2 .if your index is not solely integer-based, ix will work with positions, just like .iloc[]

Example

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), index= [2, 'A', 4], columns=[48, 49, 50])

# Pass `2` to `loc`

print(df.loc[2])

# Pass `2` to `iloc`

print(df.iloc[2])

# Pass `2` to `ix`

print(df.ix[2])

output:

48 1

49 2

50 3

Name: 2, dtype: int32

48 7

49 8

50 9

Name: 4, dtype: int32

48 7

49 8

50 9

Name: 4, dtype: int32

**Adding a Column**

If you want your DataFrame to take a column as its index, you can assign it to the .index property

Example

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), columns=[48, 49, 50])

# Use `.index`

df['D'] = df.index

# Print `df`

print(df)

output:

48 49 50 D

0 1 2 3 0

1 4 5 6 1

2 7 8 9 2

**Resetting the Index of Your DataFrame**

You can reset  your index with .reset\_index()

Example

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), columns=[48, 49, 50])

# Check out the weird index of your dataframe

print(df)

# Use `reset\_index()` to reset the values

df\_reset = df.reset\_index(level=0, drop=True)

# Print `df\_reset`

print(df\_reset)

output:

48 49 50

0 1 2 3

1 4 5 6

2 7 8 9

48 49 50

0 1 2 3

1 4 5 6

2 7 8 9

**How to Delete Indices, Rows or Columns From a Pandas Data Frame**

**Deleting an Index from Your DataFrame**

A Dataframe will always contain an index, so you should re-consider deleting an index. You can rather reset the index of your DataFrame or remove the index name or remove duplicate index values.

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [40, 50, 60], [23, 35, 37]]),

index= [2.5, 12.6, 4.8, 4.8, 2.5],

columns=[48, 49, 50])

# Check out the DataFrame `df`

print(df)

# Drop the column with label '48'

df.drop(48, axis=1, inplace=True)

# Drop the column at position 1

df.drop(df.columns[[1]], axis=1)

print(df)

Output

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 7 8 9

4.8 40 50 60

2.5 23 35 37

49 50

2.5 2 3

12.6 5 6

4.8 8 9

4.8 50 60

2.5 35 37

**Deleting a Column from Your DataFrame**

To get rid of (a selection of) columns from your DataFrame, you can use the drop() method.

Example

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [40, 50, 60], [23, 35, 37]]),

index= [2.5, 12.6, 4.8, 4.8, 2.5],

columns=[48, 49, 50])

# Check out the DataFrame `df`

print(df)

# Drop the column with label '48'

df.drop(48, axis=1, inplace=True)

# Drop the column at position 1

df.drop(df.columns[[1]], axis=1)

print(df)

output:

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 7 8 9

4.8 40 50 60

2.5 23 35 37

49 50

2.5 2 3

12.6 5 6

4.8 8 9

4.8 50 60

2.5 35 37

**Removing a Row from Your DataFrame**

You can use the drop() method to remove rows, where you use the index property to specify the index of which rows you want to remove from your DataFrame:

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[4, 2, 3], [4, 5, 6], [4, 5, 9]]),

index= [11,22, 33],

columns=[48, 49,50])

# Check out the DataFrame `df`

print(df)

# Drop the duplicates in `df`

print(df.drop(df.index[1]))

output:

48 49 50

11 4 2 3

22 4 5 6

33 4 5 9

48 49 50

11 4 2 3

33 4 5 9

**How to Rename the Columns of a Pandas DataFrame**

To give the columns or your index values of your dataframe a different value, it’s best to use the .rename() method.

Example

import pandas as pd

import numpy as np

df = pd.DataFrame(data=np.array([[4, 2, 3], [4, 5, 6], [4, 5, 9]]),

index= [11,22, 33],

columns=[48, 49,50])

# Check out the DataFrame `df`

print(df)

# Define the new names of your columns

newcols = {

48: 'new\_column\_1',

49: 'new\_column\_2',

50: 'new\_column\_3'

}

# Use `rename()` to rename your columns

df.rename(columns=newcols, inplace=True)

# Rename your index

print(df)

output:

48 49 50

11 4 2 3

22 4 5 6

33 4 5 9

new\_column\_1 new\_column\_2 new\_column\_3

11 4 2 3

22 4 5 6

33 4 5 9

**3.6 Summarizing and Computing Descriptive Statistics**

A large number of methods collectively compute descriptive statistics and other related operations on DataFrame. Most of these are aggregations like sum(), mean(), but some of them, like sumsum(), produce an object of the same size.

Let us create a DataFrame and use this object throughout this chapter for all the operations.

**Example**

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df

**output**

Age Name Rating

0 25 Tom 4.23

1 26 James 3.24

2 25 Ricky 3.98

3 23 Vin 2.56

4 30 Steve 3.20

5 29 Smith 4.60

6 23 Jack 3.80

7 34 Lee 3.78

8 40 David 2.98

9 30 Gasper 4.80

10 51 Betina 4.10

11 46 Andres 3.65

**sum()**

Returns the sum of the values for the requested axis. By default, axis is index (axis=0)

#Create a DataFrame

df = pd.DataFrame(d)

print df.sum()

Output -

Age 382

Name TomJamesRickyVinSteveSmithJackLeeDavidGasperBe...

Rating 44.92

dtype: object

**mean()**

Returns the average value

#Create a DataFrame

df = pd.DataFrame(d)

print df.mean()

**Output**

Age 31.833333

Rating 3.743333

dtype: float64

**std()**

Returns the Bressel standard deviation of the numerical columns.

#Create a DataFrame

df = pd.DataFrame(d)

print df.std()

Output

Age 9.232682

Rating 0.661628

dtype: float64

Functions & Description

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Function** | **Description** |
| 1 | count() | Number of non-null observations |
| 2 | sum() | Sum of values |
| 3 | mean() | Mean of Values |
| 4 | median() | Median of Values |
| 5 | mode() | Mode of values |
| 6 | std() | Standard Deviation of the Values |
| 7 | min() | Minimum Value |
| 8 | max() | Maximum Value |
| 9 | abs() | Absolute Value |
| 10 | prod() | Product of Values |
| 11 | cumsum() | Cumulative Sum |
| 12 | cumprod() | Cumulative Product |

Summarizing Data

The describe() function computes a summary of statistics pertaining to the DataFrame columns.

#Create a DataFrame

df = pd.DataFrame(d)

print df.describe()

Output

Age Rating

count 12.000000 12.000000

mean 31.833333 3.743333

std 9.232682 0.661628

min 23.000000 2.560000

25% 25.000000 3.230000

50% 29.500000 3.790000

75% 35.500000 4.132500

max 51.000000 4.800000

This function gives the mean, std and IQR values. And, function excludes the character columns and given summary about numeric columns. 'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.

* object − Summarizes String columns
* number − Summarizes Numeric columns
* all − Summarizes all columns together (Should not pass it as a list value)

Example

#Create a DataFrame

df = pd.DataFrame(d)

print df.describe(include=['object'])

Output

Name

count 12

unique 12

top Ricky

freq 1