1_input_output

May 28, 2024

1 Input and output

File as PDF

1.1 Keyboard input

Python has a function called *input* for getting input from the user and assigning it a variable name.

```
Tell me a number:  12
```

```
 str
```

The value contains the keyboard input as expected, but it is a string. We want to use a number and not a string, so we need to convert it from a string to a number.

```
 12
```

1.2 Screen output

Screen output is possible by using the `print` command. The argument of the print function can be of different type.

1.2.1 str.format() Formatting

You can format your output by modifying the string given to the `print` function by `str.format()`, The `str` contains text that is written to be the screen, as well as certain format specifiers contained in curly braces {}. The format function contains the list of variables that are to be printed.

```
 ***
How
How are you my friend?
 1. How are you my friend?
 2. How are you my friend?
 3. How How are you my friend? - How are you my friend?
 4. How       are you my friend?
 ***
34 942885
 6. 34 942885
 7.       34     942885
 ***
```

```
 8. -3.000
 9. -3.000
10.   -3.000
 11.   -3.000 11.   -3.000
 ***
12. 3.142e-14
13.  3.142e-14
14.      0.000
 ***
15. 12345678901234567890
16. are you my friend?--      34, 3.142e-14
```

1.2.2 %-Formatting

A very similar formatting can be achieved with the `%`operator.

```
Hello, Frank.
```

1.2.3 f-Strings

Formatted string literals are the string literals that start with an `f` at the beginning and use curly braces {} to enclose the expressions that will be replaced with other values.

```
I'm here for the 3. time and this Python Lecture is awesome!
```

```
You just have to sent me    100.000 Euros.
```

1.3 File input/output

File input and output is one of the most important features. We will have a look at reading and writing of text files with numpy and pandas. Python itself also allows you to open files and the file object provides the methods read, write and close.

1.3.1 File I/O with NumPy

Most of the time we want import numbers from text files. So direct connection to NumPy seems useful and we will study that first.

Reading data from a text file Often you would like to analyze data that you have stored in a text file. Consider, for example, the data file below for an experiment measuring the free fall of a mass.

```
Data for falling mass experiment
Date: 16-Aug-2013
Data taken by Frank and Ralf
data point  time (sec)  height (mm) uncertainty (mm)
0       0.0     180     3.5
1       0.5     182     4.5
2       1.0     178     4.0
3       1.5     165     5.5
4       2.0     160     2.5
```

```
5       2.5     148     3.0
6       3.0     136     2.5
```

Suppose that the name of the text file is **MyData.txt**. Then we can read the data into four different arrays with the following NumPy statement:

If you don't want to read in all the columns of data, you can specify which columns to read in using the usecols key word. For example, the call

```
'/home'
```

reads in only columns 1 and 2; columns 0 and 3 are skipped.

Writing data to a text file There are plenty of ways to write data to a data file in Python. We will stick to one very simple one that's suitable for writing data files in text format. It uses the NumPy savetxt routine, which is the counterpart of the loadtxt routine introduced in the previous section. The general form of the routine is

```
savetxt(filename, array, fmt="%0.18e", delimiter=" ", newline="\n", header="", footer="", comme
```

We illustrate savetext below with a script that first creates four arrays by reading in the data file MyData.txt, as discussed in the previous section, and then writes that same data set to another file MyDataOut.txt.

```
array([1., 2., 3., 4., 5., 6.])
```

```
[(1.0, 0.5, 182.0, 4.5),
 (2.0, 1.0, 178.0, 4.0),
 (3.0, 1.5, 165.0, 5.5),
 (4.0, 2.0, 160.0, 2.5),
 (5.0, 2.5, 148.0, 3.0),
 (6.0, 3.0, 136.0, 2.5)]
```

```
# This is a header
#  # ; time (sec); height (m); error (m)
     1.00000;     0.50000;   182.00000;     4.50000
     2.00000;     1.00000;   178.00000;     4.00000
     3.00000;     1.50000;   165.00000;     5.50000
     4.00000;     2.00000;   160.00000;     2.50000
     5.00000;     2.50000;   148.00000;     3.00000
     6.00000;     3.00000;   136.00000;     2.50000
```

1.3.2 File I/O with Pandas

Pandas is a software library written for the Python programming language. It is used for data manipulation and analysis. It provides special data structures and operations for the manipulation of numerical tables and time series and builds on top of numpy.

- Easy handling of missing data
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets

The data formats provided by the pandas module are used by several other modules, such as the trackpy which is a moduly for feature tracking and analysis in image series.

Short intro to Pandas Pandas provides two data structures

- Series
- Data Frames

A **Series** is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

```
array([ 0.16753316,  0.60115104,  0.29874055, -0.41653506, -0.13600342,
       -0.20579562,  1.0869474 ])
```

```
0.1675331648744497
```

```
array([ 1.67528219,  0.36104717, -1.07891124,  0.5284189 ,  0.95976588,
        0.15718688,  1.42820191])
```

There is a whole lot of functionality built into pandas data types. You may of course also obtain the same functionality using numpy commands, but you may find the pandas abbrevations very useful.

```
min    -0.416535
max     1.086947
sum     1.396038
mean    0.199434
dtype: float64
```

A **DataFrame** is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). The example below shows how such a DataFrame can be generated from the scratch. In addition to the data supplied to the DataFrame method, an index column is generated when creating a DataFrame. As in the case of *Series* there is a whole lot of functionality integrated into the DataFrame data type which you may explore on the website.

	column 1	column 2	columns 3	column 4	column 5
11	7	4	5	2	8
12	4	6	5	8	7
13	0	7	8	7	0
14	5	4	1	8	8
15	2	0	9	9	9

Due to the labelling of the columns, each column may be accessed by its column label. Labeling by names improves readability considerably.

```
0      0
1      5
2      2
```

```
3     7
4     7
5     5
6     6
7     6
8     6
9     0
10    9
11    6
12    3
13    6
14    7
15    7
Name: x, dtype: int64
```

If you don't like this format, you can always return to a simple numpy array with the *as_matrix()* method.

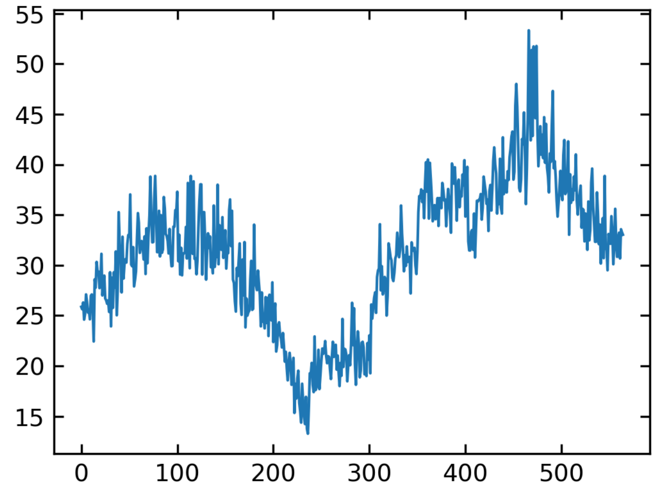
```
array([[5, 9, 9, 5, 5],
       [5, 0, 4, 7, 5],
       [0, 9, 3, 8, 1],
       [1, 4, 2, 7, 5],
       [7, 5, 4, 6, 5]])
```

Reading CSV data with Pandas DataFrames may also be populated by text files such as comma separated value files (short *.csv*). These files contain data in text format but also a column label, which can be read by the pandas method *read_csv()*. You can find an example below, which reads the data from the dust sensor on my balcony from April, 11th. You see the different columns, where **P1** and **P2** correspond to the **PM10** and **PM2.5** dust values in $\mu g/m^3$.

```
   sensor_id sensor_type  location     lat    lon            timestamp     P1  \
0      12253      SDS011      6189  52.527  13.39  2018-04-11T00:01:58  25.87
1      12253      SDS011      6189  52.527  13.39  2018-04-11T00:04:24  25.63
2      12253      SDS011      6189  52.527  13.39  2018-04-11T00:06:55  26.30
3      12253      SDS011      6189  52.527  13.39  2018-04-11T00:09:23  24.60
4      12253      SDS011      6189  52.527  13.39  2018-04-11T00:11:51  25.17

   durP1  ratioP1     P2  durP2  ratioP2
0    NaN      NaN  19.37    NaN      NaN
1    NaN      NaN  20.53    NaN      NaN
2    NaN      NaN  22.00    NaN      NaN
3    NaN      NaN  20.30    NaN      NaN
4    NaN      NaN  20.23    NaN      NaN
```

```
<Axes: >
```




```
0      1.335570
1      1.248417
2      1.195455
3      1.211823
4      1.244192
          …
560    1.172962
561    1.116491
562    1.186219
563    1.170243
564    1.280233
Length: 565, dtype: float64
```

```
<Axes: >
```