Objective: In Data Science and many other fields of engineering, many times it is requirement to read a file and then perform operations on it. This worksheet provides a quick hands-on for few ways of doing it. There are two text files (*planets_small.txt* and *planets_big.txt*) provided along with this worksheet. They will be utilised to demonstrate the functionality.

- First the file *planets_small.txt* file will be read. This file is about 9 planets and their 3 attributes Mass, Diameter and DayLength arranged in two dimensional array of size 3 x 9. The values are space separated.

planets_small - Notepad

File Edit Format View Help

	MERCURY	VENUS	EARTH	MARS	JUPITER	SATURN	URANUS	NEPTUNE	PLUTO
Mass	0.330	4.87	5.97	0.642	1898	568	86.8	102	0.0146
Diameter	57.9	108.2	149.6	227.9	778.6	1433.5	2872.5	4495.1	5906.4
DayLength	4222.6	2802.0	24.0	24.7	9.9	10.7	17.2	16.1	153.3

- ***loadtxt ()*** function will be used to read the text file. The file name along with the complete path needs to be provided within the double quotes. Also please note that the Unix style forward slash (/) is to be used in the path. Few important arguments of this function are:
  - ***skiprows***: the row that has to be skipped while reading. In this file, the first row is for the planet names and that can be skipped.
  - ***usecols:*** It needs to be a tuple of columns which need to be read. For the given file it has to be (1, 2, 3, 4, 5, 6, 7, 8, 9) leaving the first column (0^{th}) that is the name of the attributes.
  - ***delimiter***: Default delimiter is space. Other delimiter can be specified for separating the element values while reading. This argument is useful in reading an Excel file (by saving an Excel file as CSV and then reading with delimiter as comma)

```
planets = np.loadtxt("C:/Users/BITS-PC/Desktop/ISM/Python Code/Datasets/planets_small.txt",
                     skiprows = 1,
                     usecols = (1, 2, 3, 4, 5, 6, 7, 8, 9))
```

- The file is read in an array called *planets* of size 3 x 9.

```
planets
```

```
array([[3.3000e-01, 4.8700e+00, 5.9700e+00, 6.4200e-01, 1.8980e+03,
        5.6800e+02, 8.6800e+01, 1.0200e+02, 1.4600e-02],
       [5.7900e+01, 1.0820e+02, 1.4960e+02, 2.2790e+02, 7.7860e+02,
        1.4335e+03, 2.8725e+03, 4.4951e+03, 5.9064e+03],
       [4.2226e+03, 2.8020e+03, 2.4000e+01, 2.4700e+01, 9.9000e+00,
        1.0700e+01, 1.7200e+01, 1.6100e+01, 1.5330e+02]])
```

- The dimension can also be verified as 3 x 9.

```
planets.shape
```

```
(3, 9)
```

- Now the other file *planets_big.txt* will be read through NumPy. It contains the data for 10 planets for 20 different attributes. Few attribute values are not available and mentioned as *Unknown*. This is a very common situation in Data Science. You can open the file in a text editor and have a look.
- If the same mechanism as described above is used to read the data, it throws an error that *could not convert string to float: 'Unknown'.*
- A different function ***genfromtxt ()*** can be used for the purpose. It takes similar arguments. In place of *skiprows*, the argument is *skip_header* in this function.

```
planets = np.genfromtxt("C:/Users/BITS-PC/Desktop/ISM/Python Code/Datasets/planets_big.txt",
                        skip_header = 1,
                        usecols = (1, 2, 3, 4, 5, 6, 7, 8, 9))
```

- The shape of the returned array is 20 x 9.

```
planets.shape
```

```
(20, 9)
```

- If the content of the array planets is printed, it can be noticed that all attribute values *Unknown* are replaced by *nan*, that mean Not a Number.
- A function ***isnan ()*** can be used over the returned array. Wherever *nan* is present, it writes a boolean value True there. All other attributes values will be False.

```
np.isnan(planets)
```

```
False,  False,  False,  False,  False],
False,  False,  False,  False,  False],
False,   True,   True,   True,   True],
```

- A function ***nan_to_num()*** can be used to replace the *nan* with some numbers. The following example replaces *nan* to -1.

```
newPlanets = np.nan_to_num(planets, nan = -1)
```

- The read and transformed file can also be saved in human readable text format using ***savetxt()*** function. This function takes the file name, the array that needs to be saved and a delimiter among few other arguments. The complete path of the file can be specified.

```
np.savetxt("newArray.txt", newPlanets, delimiter = ',')
```

- NumPy also supports storing and retrieval of multiple arrays through one file in a NumPy specific format. Let us say, we have two arrays of random numbers of different sizes. They are stored in a single file using the ***savez ()*** function. The created file will have *.npz* file extension.
- If there is only a single array, the function ***save()*** can be used that creates a file with extension *.npy*

```
arr1 = np.random.rand(1000, 10)
arr2 = np.random.rand(10, 1000)

np.savez ("manyArrays", arr1, arr2)
```

- You can notice a file is created with the name *manyArrays.npz* that stores arr1 and arr2 in a NumPy specific format.
- ***load ()*** function can be used to load the files back to NumPy. Notice that returned array has a type as NpzFile.

```
arrays = np.load ("manyArrays.npz")

type(arrays)

numpy.lib.npyio.NpzFile

arrays.files

['arr_0', 'arr_1']
```

- Observe that the attribute ***files*** provides the indices of the arrays stored in the *.npz* file. It can be accessed in the following way: ***arrays ['arr_0']*** to access the arrays back.
- While saving many arrays in a single file, a different function ***savez_compressed()*** can be used which compress the file and saves some disk size.

```
np.savez_compressed ("manyArrays_comp", arr1, arr2)
```

- The file size can be verified as shown below:

```
160,506 manyArrays.npz
151,347 manyArrays_comp.npz
```

Exercise:

The compression does not seem to be impressive in the shown example above. Create an array of dimension 10^4 x 10^4 of all zeros. Save two files for this array: uncompressed and compressed. What is your opinion now about the compression that is achieved?