DSF24_PandasNotes!

February 3, 2025

1 Pandas Basics

Pandas is another important library in the data science world. It is the standard for manipulating multidimensional data in an efficient way using Python and has many powerful routines we will use to investigate and manipulate data sets.

1.0.1 Resources

- Pandas Documentation
- Python Data Science Handbook
- Pandas Basics Notes

1.1 What is Pandas?

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a `DataFrame`. `DataFrames` are essentially multidimensional arrays with attached row and column labels that can handle multiple data types and/or missing data. Pandas also defines `Series` which represents observations of a single variable. Not only is Pandas is a convenient medium for storing labeled data, it implements many powerful operations from database managers and spreadsheet programs.

Pandas was created to offer more versatile data structures that are straightforward to use for storing, manipulating and analyzing heterogeneous data: 1. Data is clearly organized in *variables* and *observations* 2. Each variable is permitted to have a different data type. 3. We can use *labels* to select observations, instead of having to use a linear numerical index as with NumPy. We could, for example, index a data set using National Insurance Numbers. 4. Pandas offers many convenient data aggregation and reduction routines that can be applied to subsets of data. For example, we can easily group observations by city and compute average incomes. 5. Pandas also offers many convenient data import / export functions that go beyond what's in NumPy.

Then shouldn't we be using pandas at all times, then? No! - For low-level tasks where performance is essential, use NumPy. - For homogenous data without any particular data structure, use NumPy. - On the other hand, if data is heterogeneous, needs to be imported from an external data source and cleaned or transformed before performing computations, use pandas.

1.2 Using pandas

Pandas has two main data structures: 1. `Series` represents observations of a single variable. 2. `DataFrame` is a container for several variables. You can think of each individual column of a `DataFrame` as a `Series`, and each row represents one observation.

The easiest way to create a `Series` or `DataFrame` is to create them from pre-existing data. To access pandas data structures and routines, we need to import them first. The near-universal convention is to make pandas available using the name `pd`:

[52]:
```
import pandas as pd
```

1.3 Pandas Series object

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

[53]:
```
data = pd.Series([0.25, 0.5, 0.75, 1])
data
```

[53]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

As we can see, the `Series` object is a combination of a sequence of values and a sequence of indicies, which we can access with the `values` and `index` attributes.

[54]:
```
data.values
```

[54]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

[55]:
```
data.index
```

[55]:
```
RangeIndex(start=0, stop=4, step=1)
```

The `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail later.

We can access elements in a `Series` the same way as regular Python lists with the familiar square-bracket notation.

[56]:
```
data[1]
```

[56]:
```
0.5
```

Slicing works the same as well.

[57]:
```
data[1:3]
```

[57]:
```
1    0.50
2    0.75
dtype: float64
```

So far, a `Series` object may seem basically interchangable with a NumPy array or a Python list.

The essential difference is the presence of the index: while the NumPy array has an *implicitly defined* integer index used to access the values, the pandas `Series` has an *explicitly defined* index associated with the values.

This explicit index definition gives the `Series` object additional capabilities. For example, the index doesn't need to be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an indicies:

[58]:
```
data = pd.Series([0.25, 0.5, 0.75, 1], index=['a', 'b', 'c', 'd'])
data
```

[58]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And we can access the elements the same way

[59]:
```
data['b']
```

[59]:
```
0.5
```

Or we can use non-contiguous or non-sequential indices

[60]:
```
data = pd.Series([0.25, 0.5, 0.75, 1], index=[47,31,24,1])
data
```

[60]:
```
47    0.25
31    0.50
24    0.75
1     1.00
dtype: float64
```

[61]:
```
data[24]
```

[61]:
```
0.75
```

In this way, you can think of a pandas `Series` a bit like a specialization of a Python dictionary. - A dictionary is a structure that maps arbitrary keys to a set of arbitrary values - A `Series` is a structure which maps typed keys to a set of typed values.

This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a pandas `Series` makes it much more efficient than Python dictionaries for certain operations.

The `Series`-as-dictionary analogy can be made even more clear by constructing a `Series` object directly from a Python dictionary:

[62]:
```
population_dict = {'California': 38332521,'Texas': 26448193,'New York':␣
↪19651127,'Florida': 19552860,'Illinois': 12882135}
```

```
pop = pd.Series(population_dict)
pop
```

[62]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

[63]:
```
pop['California']
```

[63]:
```
38332521
```

Though unlike a dictionary, the `Series` also supports array-style operations such as slicing

[64]:
```
pop['Texas':'Florida']
```

[64]:
```
Texas       26448193
New York    19651127
Florida     19552860
dtype: int64
```

1.4 Pandas DataFrame object

We can create a `DataFrame` from a NumPy array:

[65]:
```
import numpy as np

arr = np.random.randint(10, size=(10,3))
names = ['A', 'B', 'C']
df = pd.DataFrame(arr, columns=names)
df
```

[65]:

	A	B	C
0	8	9	3
1	9	4	4
2	8	2	0
3	9	0	8
4	5	6	7
5	1	0	6
6	8	5	2
7	7	1	3
8	6	3	9
9	0	4	4

We can also recreate our data table from earlier with multiple data types:

[66]:
```
names = ['Alice', 'Bob']
bdates = pd.to_datetime(['1985-01-01','1997-05-12'])
incomes = np.array([30000,np.nan])

df = pd.DataFrame({'Name':names, 'Birthdate':bdates, 'Incomes':incomes})
df
```

[66]:

	Name	Birthdate	Incomes
0	Alice	1985-01-01	30000.0
1	Bob	1997-05-12	NaN

If data types differ across columns, as in the above example, it is often convenient to create the `DataFrame` by passing a dictionary as an argument. Each key represents a column name and each corresponding value contains the data for that variable. This is also often eaiser than adding columns separately.

You can also create `DataFrames` from one of more `Series` objects.

[67]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

[67]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

We will primarily be populating our `DataFrames` by reading in `.csv` files as you will see in the next section. Pandas also has support for creating `DataFrames` from other standard formats like JSON and XML.

1.5 Viewing data

With large data sets, you hardly ever want to print the entire `DataFrame`. Pandas by default limits the amount of data shown. You can use the `head()` and `tail()` methods to explicitly display a specific number of rows from the top or the end of a `DataFrame`.

To illustrate, we use a data set of 23 UK universities that contains the following variables: - `Instititution`: Name of the institution - `Country`: Country/nation within the UK (England, Scotland, . . .) - `Founded`: Year in which university (or a predecessor institution) was founded - `Students`: Total number of students - `Staff`: Number of academic staff - `Admin`: Number of administrative staff - `Budget`: Budget in million pounds - `Russell`: Binary indicator whether university is a member of the Russell Group, an association of the UK's top research universities.

The data was compiled based on information from Wikipedia.

We read in the data stored in the file `universities.csv` like this.

[68]:
```
df = pd.read_csv('universities.csv',sep=';')
df
```

[68]:

	Institution	Country	Founded	Students	Staff	Admin	Budget	Russell
0	University of Glasgow	Scotland	1451	30805	2942.0	4003.0	626.5	1
1	University of Edinburgh	Scotland	1583	34275	4589.0	6107.0	1102.0	1
2	University of St Andrews	Scotland	1413	8984	1137.0	1576.0	251.2	0
3	University of Aberdeen	Scotland	1495	14775	1086.0	1489.0	219.5	0
4	University of Strathclyde	Scotland	1964	22640	NaN	3200.0	304.4	0
5	LSE	England	1895	11850	1725.0	2515.0	415.1	1
6	UCL	England	1826	41180	7700.0	5375.0	1451.1	1
7	University of Cambridge	England	1209	23247	7913.0	3615.0	2192.0	1
8	University of Oxford	England	1096	24515	7000.0	NaN	2450.0	1
9	University of Warwick	England	1965	27278	2610.0	4033.0	688.6	1
10	Imperial College London	England	1907	19115	4390.0	4075.0	1064.0	1
11	King's College London	England	1829	32895	5220.0	3485.0	902.0	1
12	University of Manchester	England	2004	40250	3849.0	NaN	1095.4	1
13	University of Bristol	England	1595	25955	3285.0	6199.0	642.7	1
14	University of Birmingham	England	1825	35445	4020.0	NaN	673.8	1
15	Queen Mary University of London	England	1785	20560				
16	University of York	England	1963	19470				
17	University of Nottingham	England	1798	30798				
18	University of Dundee	Scotland	1967	15915				
19	Cardiff University	Wales	1883	25898				
20	University of Stirling	Scotland	1967	9548				
21	Queen's University Belfast	Northern Ireland	1810	18438				
22	Swansea University	Wales	1920	20620				

```
15  3235.0  4620.0   459.5        1
16  1935.0  3091.0   331.4        1
17  3495.0     NaN   656.5        1
18  1410.0  1805.0   256.4        0
19  3330.0  5739.0   644.8        1
20     NaN  1872.0   113.3        0
21  2414.0  1489.0   369.2        1
22     NaN  3290.0     NaN        0
```

If we want to know what are columns are called, we can use the `columns` attribute.

[69]:
```
df.columns
```

[69]:
```
Index(['Institution', 'Country', 'Founded', 'Students', 'Staff', 'Admin',
       'Budget', 'Russell'],
      dtype='object')
```

This is very similiar to how we already know to read in files, except it is built to read in CSVs as `DataFrames`. Now we can take a look at the first and last rows of the file

[70]:
```
df.head(3)
```

[70]:
```
                Institution   Country  Founded  Students   Staff   Admin  \
0    University of Glasgow  Scotland     1451     30805  2942.0  4003.0
1  University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0

   Budget  Russell
0   626.5        1
1  1102.0        1
2   251.2        0
```

[71]:
```
df.tail(10)
```

[71]:
```
                       Institution           Country  Founded  Students  \
13           University of Bristol           England     1595     25955
14         University of Birmingham           England     1825     35445
15  Queen Mary University of London           England     1785     20560
16              University of York           England     1963     19470
17         University of Nottingham           England     1798     30798
18             University of Dundee          Scotland     1967     15915
19               Cardiff University             Wales     1883     25898
20           University of Stirling          Scotland     1967      9548
21       Queen's University Belfast  Northern Ireland     1810     18438
22               Swansea University             Wales     1920     20620

     Staff   Admin  Budget  Russell
13  3285.0  6199.0   642.7        1
```

```
14  4020.0     NaN   673.8        1
15  3235.0  4620.0   459.5        1
16  1935.0  3091.0   331.4        1
17  3495.0     NaN   656.5        1
18  1410.0  1805.0   256.4        0
19  3330.0  5739.0   644.8        1
20     NaN  1872.0   113.3        0
21  2414.0  1489.0   369.2        1
22     NaN  3290.0     NaN        0
```

To quickly compute some descriptive statistics for the *numerical* variables in the `DataFrame`, we use `describe()`.

[72]:
```
df.describe()
```

[72]:
```
           Founded      Students        Staff        Admin       Budget  \
count    23.000000     23.000000    20.000000    19.000000    22.000000
mean   1745.652174  24106.782609  3664.250000  3556.736842   768.609091
std     256.992149   9093.000735  2025.638038  1550.434342   608.234948
min    1096.000000   8984.000000  1086.000000  1489.000000   113.300000
25%    1589.000000  18776.500000  2294.250000  2193.500000   340.850000
50%    1826.000000  23247.000000  3307.500000  3485.000000   643.750000
75%    1941.500000  30801.500000  4439.750000  4347.500000  1023.500000
max    2004.000000  41180.000000  7913.000000  6199.000000  2450.000000

         Russell
count  23.000000
mean    0.739130
std     0.448978
min     0.000000
25%     0.500000
50%     1.000000
75%     1.000000
max     1.000000
```

Note that this automatically ignores the columns `Institution` and `Country` as they contain strings, and computing the mean, etc. of a string variable does not make sense.

To see low-level information about the data type used in each column, we call `info()`:

[73]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Institution  23 non-null     object
 1   Country      23 non-null     object
```

```
 2   Founded      23 non-null     int64
 3   Students     23 non-null     int64
 4   Staff        20 non-null     float64
 5   Admin        19 non-null     float64
 6   Budget       22 non-null     float64
 7   Russell      23 non-null     int64
dtypes: float64(3), int64(3), object(2)
memory usage: 1.6+ KB
```

Pandas automatically discards missing information in computations. For example, the number of academic staff is missing for several universities, so the number of non-null entries reported in the table above is less than 23, the overall sample size.

1.5.1 What questions do you want to answer?

Thinking about the dataset we just imported, what questions can we ask? What information would we want to know from this dataset? What questions can this dataset help us answer?

- What is the avg student to staff ratio
- Funding of Newest or oldest institution
- Avg year a Russell group college was founded vs non-Russell
- Vowels in name vs funding
- How many universities are in England, Scotland, Wales, Northern Ireland

1.6 Indexing

Pandas supports two types of indexing: 1. Indexing by position. This is basically identical to the indexing of other Python and NumPy options. 2. Indexing by label, i.e., by the values assigned to the row or column index. These labels need not be integers in increasing order, as is the case for NumPy. We will see how to assign labels below.

Pandas indexing is performed either by using brackets `[]`, or by using `.loc[]` for label indexing, or `.iloc[]` for positional indexing. Indexing via `[]` can be somewhat confusing: - specifying `df['name']` returns the column name as a Series object. - On the other hand, specifying a range such as `df[5:10]` returns the rows associated with the positions `5,. . . ,9.`

Let's use our example `DataFrame` to demonstrate different ways to index.

[74]:
```
df['Institution']
```

[74]:
```
0              University of Glasgow
1            University of Edinburgh
2          University of St Andrews
3            University of Aberdeen
4          University of Strathclyde
5                                LSE
6                                UCL
7            University of Cambridge
8               University of Oxford
9            University of Warwick
```

```
10            Imperial College London
11              King's College London
12           University of Manchester
13              University of Bristol
14           University of Birmingham
15    Queen Mary University of London
16                  University of York
17           University of Nottingham
18               University of Dundee
19                 Cardiff University
20             University of Stirling
21          Queen's University Belfast
22                  Swansea University
Name: Institution, dtype: object
```

[75]:
```
df[['Institution','Students']]
```

[75]:

	Institution	Students
0	University of Glasgow	30805
1	University of Edinburgh	34275
2	University of St Andrews	8984
3	University of Aberdeen	14775
4	University of Strathclyde	22640
5	LSE	11850
6	UCL	41180
7	University of Cambridge	23247
8	University of Oxford	24515
9	University of Warwick	27278
10	Imperial College London	19115
11	King's College London	32895
12	University of Manchester	40250
13	University of Bristol	25955
14	University of Birmingham	35445
15	Queen Mary University of London	20560
16	University of York	19470
17	University of Nottingham	30798
18	University of Dundee	15915
19	Cardiff University	25898
20	University of Stirling	9548
21	Queen's University Belfast	18438
22	Swansea University	20620

[76]:
```
df[1:4]
```

[76]:

	Institution	Country	Founded	Students	Staff	Admin	\
1	University of Edinburgh	Scotland	1583	34275	4589.0	6107.0	
2	University of St Andrews	Scotland	1413	8984	1137.0	1576.0	

```
3    University of Aberdeen  Scotland     1495     14775  1086.0  1489.0

   Budget  Russell
1  1102.0        1
2   251.2        0
3   219.5        0
```

Pandas follows the Python convention that indexes start at 0, and the endpoint of a slice is not included.

You can also create whole new columns using indices.

[77]:
```
df['CostPerStudent'] = df['Budget']/df['Students']
df
```

[77]:

	Institution	Country	Founded	Students	Staff	Admin	Budget	Russell	CostPerStudent
0	University of Glasgow	Scotland	1451	30805	2942.0	4003.0	626.5	1	0.020338
1	University of Edinburgh	Scotland	1583	34275	4589.0	6107.0	1102.0	1	0.032152
2	University of St Andrews	Scotland	1413	8984	1137.0	1576.0	251.2	0	0.027961
3	University of Aberdeen	Scotland	1495	14775	1086.0	1489.0	219.5	0	0.014856
4	University of Strathclyde	Scotland	1964	22640	NaN	3200.0	304.4	0	0.013445
5	LSE	England	1895	11850	1725.0	2515.0	415.1	1	0.035030
6	UCL	England	1826	41180	7700.0	5375.0	1451.1	1	0.035238
7	University of Cambridge	England	1209	23247					
8	University of Oxford	England	1096	24515					
9	University of Warwick	England	1965	27278					
10	Imperial College London	England	1907	19115					
11	King's College London	England	1829	32895					
12	University of Manchester	England	2004	40250					
13	University of Bristol	England	1595	25955					
14	University of Birmingham	England	1825	35445					
15	Queen Mary University of London	England	1785	20560					
16	University of York	England	1963	19470					
17	University of Nottingham	England	1798	30798					
18	University of Dundee	Scotland	1967	15915					
19	Cardiff University	Wales	1883	25898					
20	University of Stirling	Scotland	1967	9548					
21	Queen's University Belfast	Northern Ireland	1810	18438					
22	Swansea University	Wales	1920	20620					

```
7   7913.0  3615.0  2192.0        1        0.094292
8   7000.0     NaN  2450.0        1        0.099939
9   2610.0  4033.0   688.6        1        0.025244
10  4390.0  4075.0  1064.0        1        0.055663
11  5220.0  3485.0   902.0        1        0.027421
12  3849.0     NaN  1095.4        1        0.027215
13  3285.0  6199.0   642.7        1        0.024762
14  4020.0     NaN   673.8        1        0.019010
15  3235.0  4620.0   459.5        1        0.022349
16  1935.0  3091.0   331.4        1        0.017021
17  3495.0     NaN   656.5        1        0.021316
18  1410.0  1805.0   256.4        0        0.016111
19  3330.0  5739.0   644.8        1        0.024898
20     NaN  1872.0   113.3        0        0.011866
21  2414.0  1489.0   369.2        1        0.020024
22     NaN  3290.0     NaN        0             NaN
```

1.6.1 Manipulated indicies

Pandas uses *labels* to index and align data. These can be integer values starting at 0 with increments of 1 for each additional element, which is the default, but they need not be. The two main methods to manipulate indices are: - `set_index(keys=['column1', ...])`: uses the values of column1 and optionally additional columns as indices, discarding the current index. - `reset_index()`: resets the index to its default value, a sequence of increasing integers startingat 0. Both methods return a new `DataFrame` and leave the original `DataFrame` unchanged. If we want to change the existing `DataFrame`, we need to pass the argument `inplace=True`.

If we want to know what our indices currently are we can use the `index` attribute, just like `Series` data.

[78]:
```
df.index
```

[78]:
```
RangeIndex(start=0, stop=23, step=1)
```

As we can see, right now the indices of the `DataFrame` are the numbers 0, 1,..., 22

We can replace the row index and use the lowercase Roman characters `a, b, c,...` as labels instead of integers:

[79]:
```
import string
index = list(string.ascii_lowercase)
index = index[0:len(df)]

df['index'] = index
df.set_index(keys=['index'], inplace=True)
df
```

[79]:
```
                        Institution          Country  Founded  Students  \
index
```

a	University of Glasgow	Scotland	1451	30805
b	University of Edinburgh	Scotland	1583	34275
c	University of St Andrews	Scotland	1413	8984
d	University of Aberdeen	Scotland	1495	14775
e	University of Strathclyde	Scotland	1964	22640
f	LSE	England	1895	11850
g	UCL	England	1826	41180
h	University of Cambridge	England	1209	23247
i	University of Oxford	England	1096	24515
j	University of Warwick	England	1965	27278
k	Imperial College London	England	1907	19115
l	King's College London	England	1829	32895
m	University of Manchester	England	2004	40250
n	University of Bristol	England	1595	25955
o	University of Birmingham	England	1825	35445
p	Queen Mary University of London	England	1785	20560
q	University of York	England	1963	19470
r	University of Nottingham	England	1798	30798
s	University of Dundee	Scotland	1967	15915
t	Cardiff University	Wales	1883	25898
u	University of Stirling	Scotland	1967	9548
v	Queen's University Belfast	Northern Ireland	1810	18438
w	Swansea University	Wales	1920	20620

index	Staff	Admin	Budget	Russell	CostPerStudent
a	2942.0	4003.0	626.5	1	0.020338
b	4589.0	6107.0	1102.0	1	0.032152
c	1137.0	1576.0	251.2	0	0.027961
d	1086.0	1489.0	219.5	0	0.014856
e	NaN	3200.0	304.4	0	0.013445
f	1725.0	2515.0	415.1	1	0.035030
g	7700.0	5375.0	1451.1	1	0.035238
h	7913.0	3615.0	2192.0	1	0.094292
i	7000.0	NaN	2450.0	1	0.099939
j	2610.0	4033.0	688.6	1	0.025244
k	4390.0	4075.0	1064.0	1	0.055663
l	5220.0	3485.0	902.0	1	0.027421
m	3849.0	NaN	1095.4	1	0.027215
n	3285.0	6199.0	642.7	1	0.024762
o	4020.0	NaN	673.8	1	0.019010
p	3235.0	4620.0	459.5	1	0.022349
q	1935.0	3091.0	331.4	1	0.017021
r	3495.0	NaN	656.5	1	0.021316
s	1410.0	1805.0	256.4	0	0.016111
t	3330.0	5739.0	644.8	1	0.024898
u	NaN	1872.0	113.3	0	0.011866

```
v      2414.0  1489.0   369.2        1        0.020024
w         NaN  3290.0     NaN        0             NaN
```

Now that we changed our index from numbers to letters, we can get certain rows using our new indices.

[80]:
```
df['a':'c']
```

[80]:
```
                    Institution   Country  Founded  Students   Staff   Admin  \
index
a          University of Glasgow  Scotland     1451     30805  2942.0  4003.0
b       University of Edinburgh   Scotland     1583     34275  4589.0  6107.0
c      University of St Andrews   Scotland     1413      8984  1137.0  1576.0

       Budget  Russell  CostPerStudent
index
a       626.5        1        0.020338
b      1102.0        1        0.032152
c       251.2        0        0.027961
```

To add to the confusion, note that when specifying a range in terms of labels, the last element *is* included! Hence the row with index `c` in the above example is shown.

We can reset the index to its default integer values using the `reset_index()` method:

[81]:
```
df = df.reset_index(drop=True)
df
```

[81]:
```
                        Institution          Country  Founded  Students  \
0              University of Glasgow         Scotland     1451     30805
1            University of Edinburgh         Scotland     1583     34275
2           University of St Andrews         Scotland     1413      8984
3             University of Aberdeen         Scotland     1495     14775
4          University of Strathclyde         Scotland     1964     22640
5                                 LSE          England     1895     11850
6                                 UCL          England     1826     41180
7            University of Cambridge          England     1209     23247
8               University of Oxford          England     1096     24515
9              University of Warwick          England     1965     27278
10           Imperial College London          England     1907     19115
11             King's College London          England     1829     32895
12          University of Manchester          England     2004     40250
13             University of Bristol          England     1595     25955
14          University of Birmingham          England     1825     35445
15  Queen Mary University of London          England     1785     20560
16                University of York          England     1963     19470
17          University of Nottingham          England     1798     30798
18              University of Dundee         Scotland     1967     15915
19                Cardiff University            Wales     1883     25898
```

```
20           University of Stirling          Scotland     1967      9548
21       Queen's University Belfast  Northern Ireland     1810     18438
22                Swansea University             Wales     1920     20620

     Staff   Admin  Budget  Russell  CostPerStudent
0   2942.0  4003.0   626.5        1        0.020338
1   4589.0  6107.0  1102.0        1        0.032152
2   1137.0  1576.0   251.2        0        0.027961
3   1086.0  1489.0   219.5        0        0.014856
4      NaN  3200.0   304.4        0        0.013445
5   1725.0  2515.0   415.1        1        0.035030
6   7700.0  5375.0  1451.1        1        0.035238
7   7913.0  3615.0  2192.0        1        0.094292
8   7000.0     NaN  2450.0        1        0.099939
9   2610.0  4033.0   688.6        1        0.025244
10  4390.0  4075.0  1064.0        1        0.055663
11  5220.0  3485.0   902.0        1        0.027421
12  3849.0     NaN  1095.4        1        0.027215
13  3285.0  6199.0   642.7        1        0.024762
14  4020.0     NaN   673.8        1        0.019010
15  3235.0  4620.0   459.5        1        0.022349
16  1935.0  3091.0   331.4        1        0.017021
17  3495.0     NaN   656.5        1        0.021316
18  1410.0  1805.0   256.4        0        0.016111
19  3330.0  5739.0   644.8        1        0.024898
20     NaN  1872.0   113.3        0        0.011866
21  2414.0  1489.0   369.2        1        0.020024
22     NaN  3290.0     NaN        0             NaN
```

We usually keep the indices as numbers because we often have more than 26 records in our `DataFrame`(as the programmer you would have to decide what comes after `z`: `aa`? `A`?), but theoretically you could use any string you wanted as an index.

Pandas also has other specialty index types like `DatetimeIndex` and `TimedeltaIndex` that can be really useful for standardizing data with dates and times

Generally speaking, when you are talking about indexing you are refering to the columns of a DataFrame. When you are talking about slicing, you are refering to rows.

1.7 Selecting elements

To more clearly distinguish between selection by label and by position, pandas provides the `.loc[]` and `.iloc[]` methods of indexing. To make your intention obvious, you should therefore adhere to the following rules: 1. Use `df['name']` only to select columns and nothing else. 2. Use `.loc[]` to select by label. 3. Use `.iloc[]` to select by position.

To illustrate, using `.loc[]` indexes by label:

[82]:
```
df.loc[4:8, ['Institution','Students']]
```

[82]:

	Institution	Students
4	University of Strathclyde	22640
5	LSE	11850
6	UCL	41180
7	University of Cambridge	23247
8	University of Oxford	24515

With `.loc[]` we can even perform slicing on column names, which is not possible with the simpler `df[]` syntax:

[83]:
```
df.loc[4:8, 'Institution':'Founded']
```

[83]:

	Institution	Country	Founded
4	University of Strathclyde	Scotland	1964
5	LSE	England	1895
6	UCL	England	1826
7	University of Cambridge	England	1209
8	University of Oxford	England	1096

Somewhat surprisingly, we can include boolean statements in `.loc[]` even though these are clearly not labels. Here, we can use `.loc` to select all rows with the "Scotland" in the `Country` column.

[84]:
```
df.loc[df['Country'] == 'Scotland']
```

[84]:

	Institution	Country	Founded	Students	Staff	Admin	Budget	Russell	CostPerStudent
0	University of Glasgow	Scotland	1451	30805	2942.0	4003.0	626.5	1	0.020338
1	University of Edinburgh	Scotland	1583	34275	4589.0	6107.0	1102.0	1	0.032152
2	University of St Andrews	Scotland	1413	8984	1137.0	1576.0	251.2	0	0.027961
3	University of Aberdeen	Scotland	1495	14775	1086.0	1489.0	219.5	0	0.014856
4	University of Strathclyde	Scotland	1964	22640	NaN	3200.0	304.4	0	0.013445
18	University of Dundee	Scotland	1967	15915	1410.0	1805.0	256.4	0	0.016111
20	University of Stirling	Scotland	1967	9548	NaN	1872.0	113.3	0	0.011866

You can also use other more complicated boolean operators to splice data.

[85]:
```
df.loc[df['Founded'] > 1800]
df = df.sort_values(by='Founded', ascending=False)
df
```

[85]:

	Institution	Country	Founded	Students
12	University of Manchester	England	2004	40250
20	University of Stirling	Scotland	1967	9548
18	University of Dundee	Scotland	1967	15915
9	University of Warwick	England	1965	27278
4	University of Strathclyde	Scotland	1964	22640
16	University of York	England	1963	19470
22	Swansea University	Wales	1920	20620
10	Imperial College London	England	1907	19115
5	LSE	England	1895	11850
19	Cardiff University	Wales	1883	25898
11	King's College London	England	1829	32895
6	UCL	England	1826	41180
14	University of Birmingham	England	1825	35445
21	Queen's University Belfast	Northern Ireland	1810	18438
17	University of Nottingham	England	1798	30798
15	Queen Mary University of London	England	1785	20560
13	University of Bristol	England	1595	25955
1	University of Edinburgh	Scotland	1583	34275
3	University of Aberdeen	Scotland	1495	14775
0	University of Glasgow	Scotland	1451	30805
2	University of St Andrews	Scotland	1413	8984
7	University of Cambridge	England	1209	23247
8	University of Oxford	England	1096	24515

	Staff	Admin	Budget	Russell	CostPerStudent
12	3849.0	NaN	1095.4	1	0.027215
20	NaN	1872.0	113.3	0	0.011866
18	1410.0	1805.0	256.4	0	0.016111
9	2610.0	4033.0	688.6	1	0.025244
4	NaN	3200.0	304.4	0	0.013445
16	1935.0	3091.0	331.4	1	0.017021
22	NaN	3290.0	NaN	0	NaN
10	4390.0	4075.0	1064.0	1	0.055663
5	1725.0	2515.0	415.1	1	0.035030
19	3330.0	5739.0	644.8	1	0.024898
11	5220.0	3485.0	902.0	1	0.027421
6	7700.0	5375.0	1451.1	1	0.035238
14	4020.0	NaN	673.8	1	0.019010
21	2414.0	1489.0	369.2	1	0.020024
17	3495.0	NaN	656.5	1	0.021316
15	3235.0	4620.0	459.5	1	0.022349
13	3285.0	6199.0	642.7	1	0.024762
1	4589.0	6107.0	1102.0	1	0.032152
3	1086.0	1489.0	219.5	0	0.014856
0	2942.0	4003.0	626.5	1	0.020338
2	1137.0	1576.0	251.2	0	0.027961

```
7   7913.0  3615.0  2192.0        1        0.094292
8   7000.0     NaN  2450.0        1        0.099939
```

1.8 Selection by position

Conversely, if we want to select items exclusively by their position and ignore their labels, we use `.iloc[]`:

[86]:
```
df.iloc[0:4,0:2]
```

[86]:
```
                 Institution   Country
12  University of Manchester   England
20    University of Stirling  Scotland
18      University of Dundee  Scotland
9      University of Warwick   England
```

1.9 Common DataFrame routines

Using `DataFrame` we can compute many common statisical calculations and database refactoring routines.

1.9.1 Statistical operations

While `df.describe()` can be a great tool, sometimes we want to get more specific.

Methods such as `mean()` are by default applied column-wise to each column. The `numeric_only=True` argument is used to discard all non-numeric columns (depending on the version of pandas, `mean()` will issue a warning otherwise).

One big advantage over NumPy is that missing values (represented by `np.nan`) are automatically ignored:

[87]:
```
df.mean(numeric_only=True)
```

[87]:
```
Founded            1745.652174
Students          24106.782609
Staff              3664.250000
Admin              3556.736842
Budget              768.609091
Russell               0.739130
CostPerStudent        0.031189
dtype: float64
```

[88]:
```
df['Staff'].mean()
```

[88]:
```
3664.25
```

1.9.2 Aggregation routines

Applying aggregation functions to the entire `DataFrame` is similar to what we can do with NumPy. The added flexibility of pandas becomes obvious once we want to apply these functions to subsets of data, i.e., groups, which we can define based on values or index labels.

For example, we can easily group our universities by country:

[89]:
```
groups = df.groupby('Country')
```

Here `groups` is a special pandas objects which can subsequently be used to process group-specific data. To compute the group-wise averages, we can simply run

[90]:
```
groups.mean(numeric_only=True)
```

[90]:
```
                      Founded      Students         Staff         Admin  \
Country
England           1745.923077  27119.846154  4336.692308  4112.000000
Northern Ireland  1810.000000  18438.000000  2414.000000  1489.000000
Scotland          1691.428571  19563.142857  2232.800000  2864.571429
Wales             1901.500000  23259.000000  3330.000000  4514.500000

                       Budget   Russell  CostPerStudent
Country
England           1001.700000  1.000000        0.038808
Northern Ireland   369.200000  1.000000        0.020024
Scotland           410.471429  0.285714        0.019533
Wales              644.800000  0.500000        0.024898
```

Groups suppot column indexing: if we want to only compute the total number of students for each country in our sample, we can do this:

[91]:
```
groups['Students'].sum()
```

[91]:
```
Country
England             352558
Northern Ireland     18438
Scotland            136942
Wales                46518
Name: Students, dtype: int64
```

There are numerous routines to aggregate grouped data, for example: - `mean()`, `sum()`: averages and sums over numerical items within groups. - `std()`, `var()`: within-group std. dev. and variances - `size()`: group sizes - `first()`, `last()`: first and last elements in each group - `min()`, `max()`: minimum and maximum elements within a group

[92]:
```
groups.size()
```

[92]:
```
Country
England             13
```

```
Northern Ireland     1
Scotland             7
Wales                2
dtype: int64
```

[93]:
```
groups.first()
```

[93]:
```
                                Institution  Founded  Students   Staff  \
Country
England             University of Manchester     2004     40250  3849.0
Northern Ireland  Queen's University Belfast     1810     18438  2414.0
Scotland              University of Stirling     1967      9548  1410.0
Wales                     Swansea University     1920     20620  3330.0

                   Admin  Budget  Russell  CostPerStudent
Country
England           4033.0  1095.4        1        0.027215
Northern Ireland  1489.0   369.2        1        0.020024
Scotland          1872.0   113.3        0        0.011866
Wales             3290.0   644.8        0        0.024898
```

1.10 Visualization

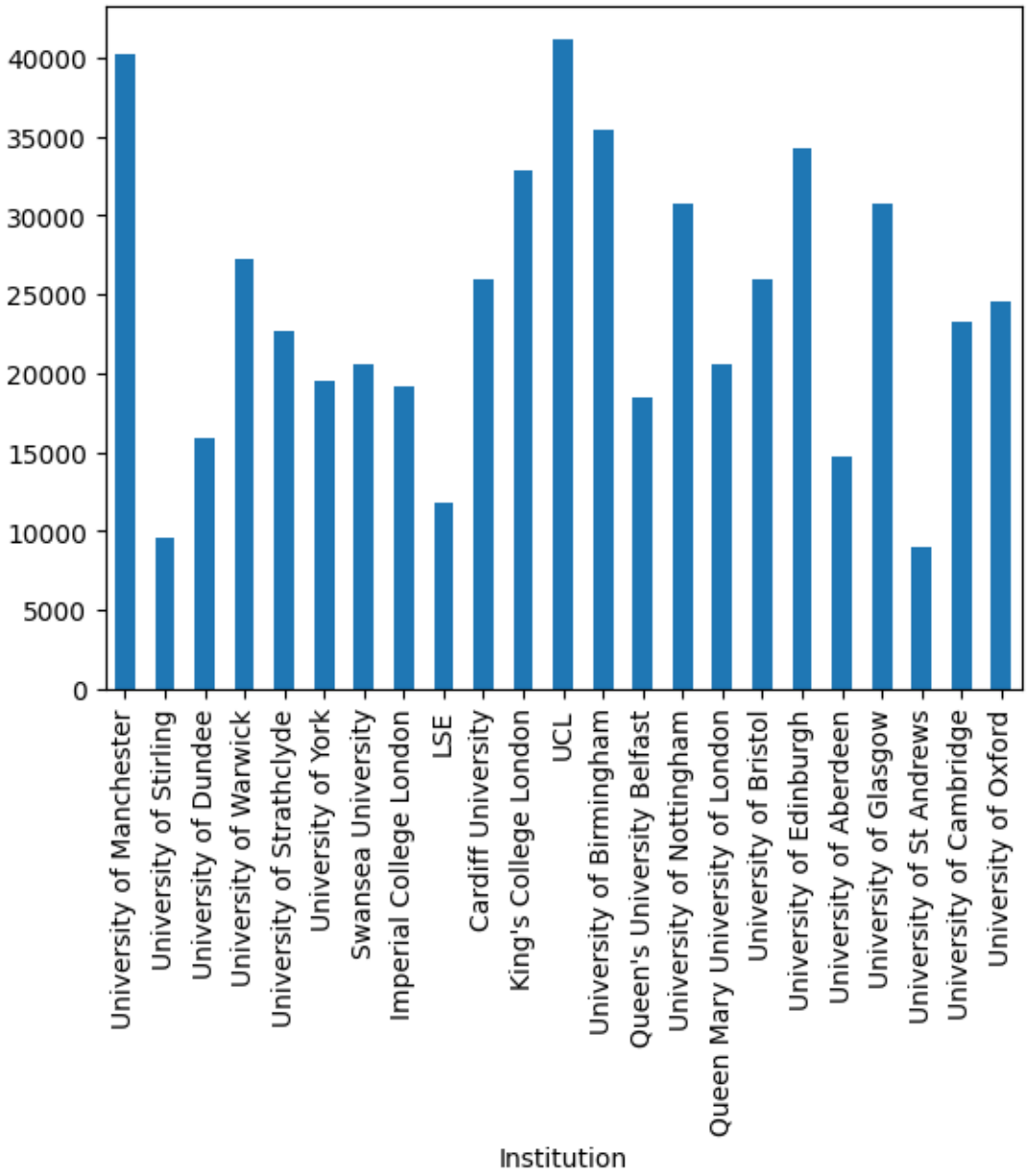
We have used the `Matplotlib` library in the past to display images as plots. Pandas itself implements some convenience wrappers around Matplotlib plotting routines which allow us to quickly inspect data stored in `DataFrames`. Alternatively, we can extract the numerical data and pass it to Matplotlib's routines manually.

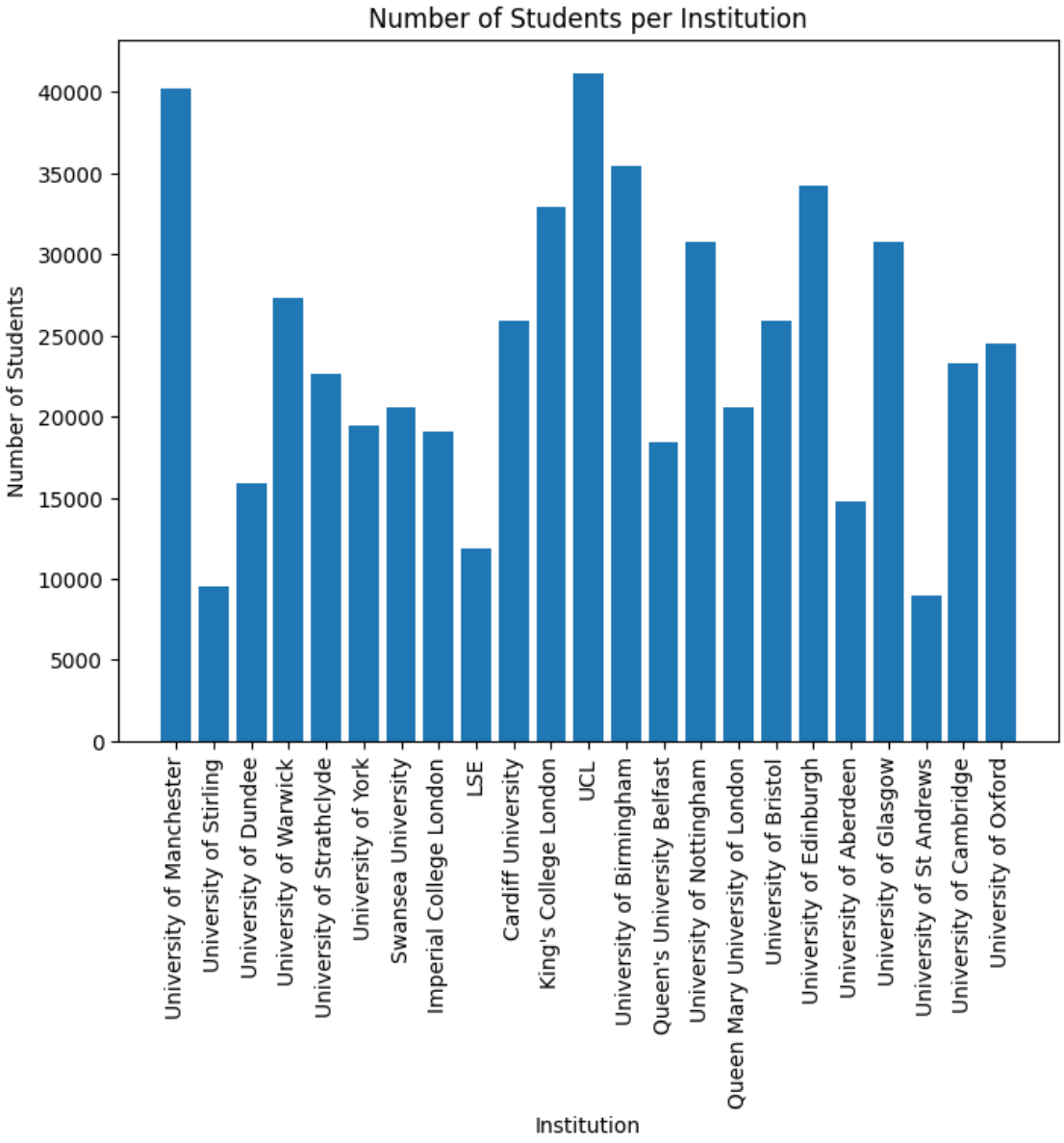
There are lot of different plots we can make, so let's go through some examples now.

To plot each institutions' student numbers as a bar chart:

[94]:
```
df2 = df.set_index(keys = ['Institution'])
df2['Students'].plot(kind='bar')

import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
plt.bar(df2.index, df2['Students'])
plt.xlabel('Institution')
plt.xticks(rotation=90) # Rotate x-axis labels vertically
plt.ylabel('Number of Students')
plt.title('Number of Students per Institution')
plt.show()
```


40000
35000
30000
25000
20000
15000
10000
5000
0
University of Manchester
University of Stirling
University of Dundee
University of Warwick
University of Strathclyde
University of York
Swansea University
Imperial College London
LSE
Cardiff University
King's College London
UCL
University of Birmingham
Queen's University Belfast
University of Nottingham
Queen Mary University of London
University of Bristol
University of Edinburgh
University of Aberdeen
University of Glasgow
University of St Andrews
University of Cambridge
University of Oxford
Institution



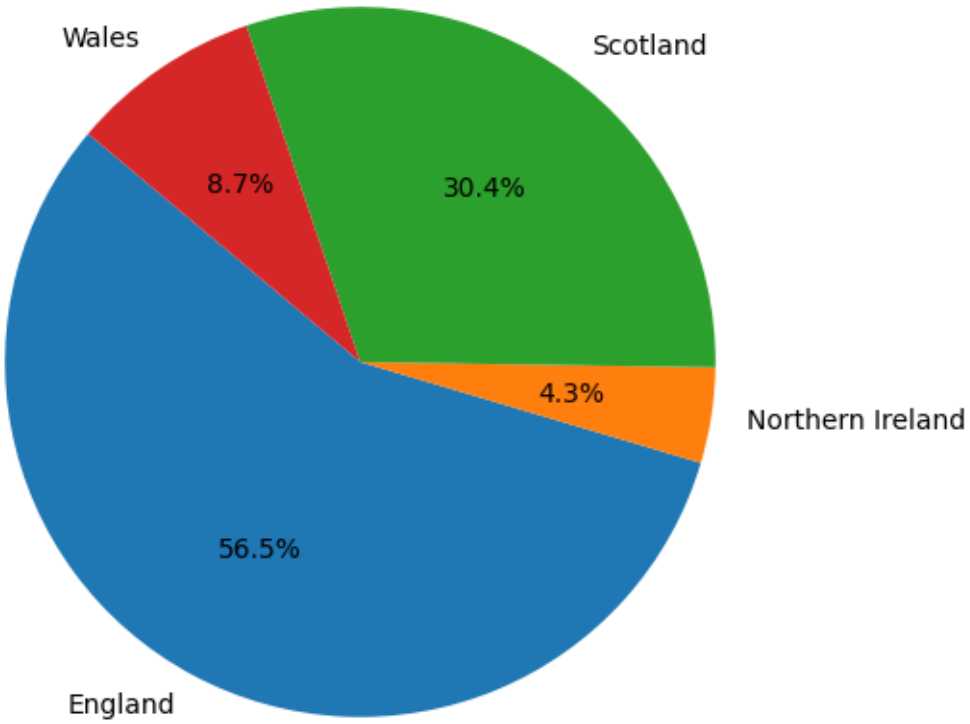
Or we can use groups to get a snapshot of different statistics.

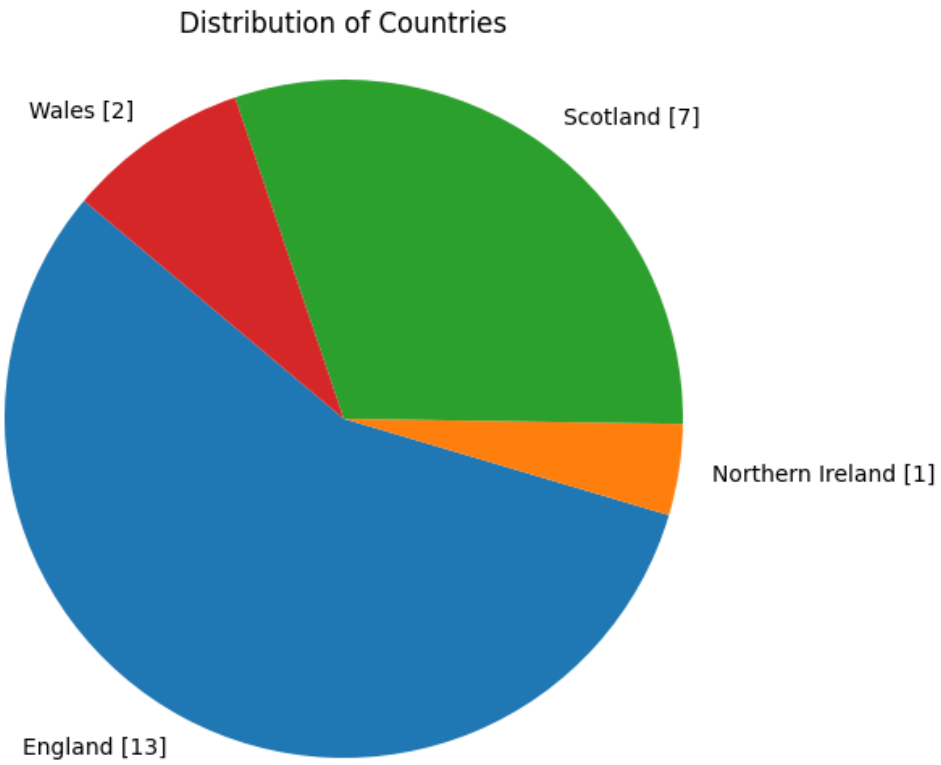
Make a pie chart to show the number of universities in each country.

[95]:
```
groups = df.groupby('Country').size()
groups.plot(kind='pie', autopct='%1.1f%%', startangle=140, figsize=(8,6))

## If you want it a little bit prettier and show the numerical counts instead,␣
 ↪you will likely want to use matplotlib
grouped = df.groupby('Country').size()
plt.figure(figsize=(8, 6))
```

```
plt.pie(grouped, labels=[f'{c} [{grouped[c]}]' for c in grouped.index],␣
  ↪autopct='', startangle=140)
plt.axis('equal')
plt.title('Distribution of Countries')
plt.show()
```





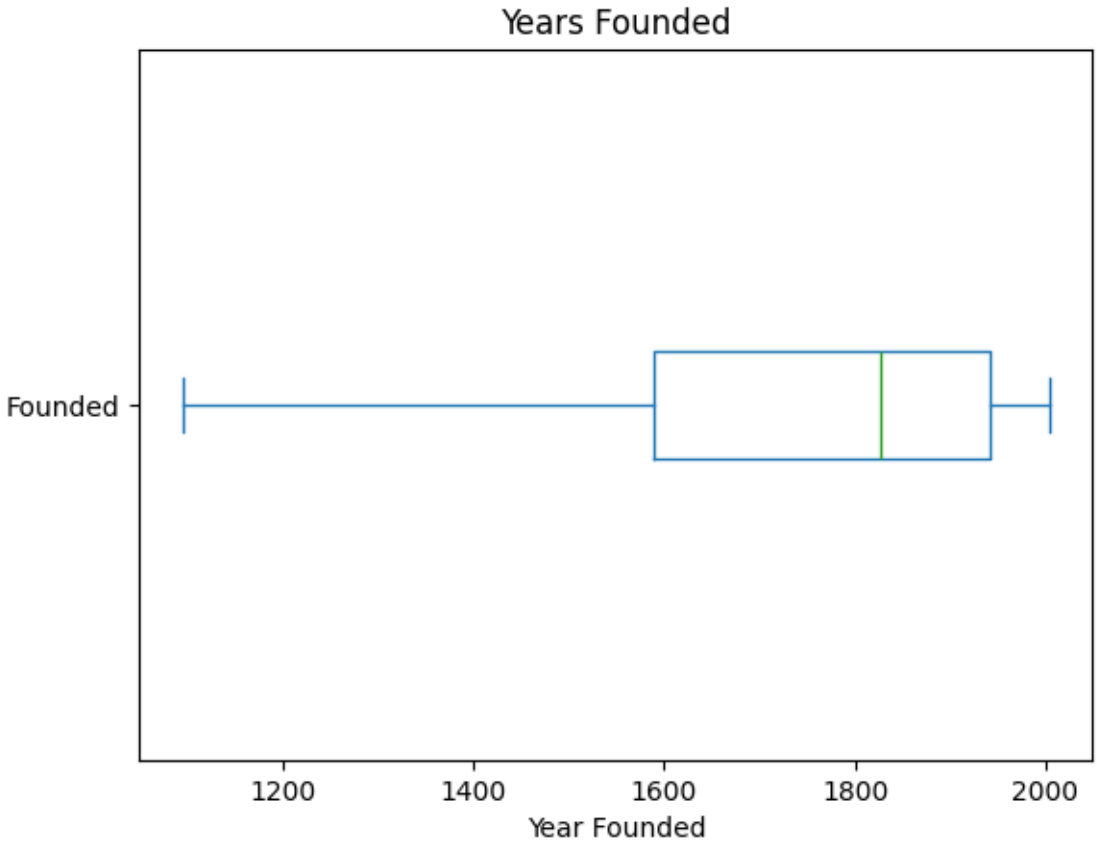
Since displaying the counts is difficult, maybe we want a historgram instead?

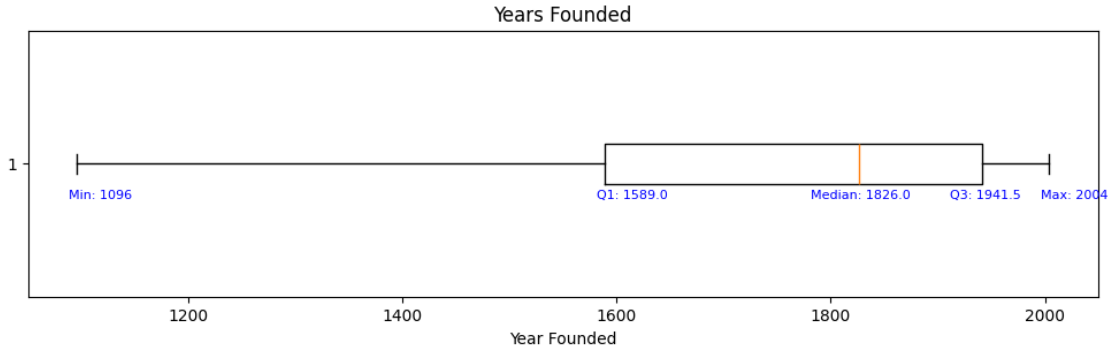
What if I wanted to make a box and whisker plot of the years each institution was founded?

[96]:
```
df['Founded'].plot(kind='box', vert=False, ylabel='', xlabel='Year Founded',␣
 ↪title='Years Founded' )
# Again using Matplotlib
plt.figure(figsize=(12, 3))
plt.boxplot(df['Founded'], vert=False)
plt.xlabel('Year Founded')
plt.title('Years Founded')

# Calculate quartiles and median
quartiles = df['Founded'].quantile([0.25, 0.5, 0.75])
# Add labels for quartiles and endpoints
plt.annotate(f'Q1: {quartiles[0.25]}', xy=(quartiles[0.25], 1), xytext=(-5,␣
 ↪-22),
             textcoords='offset points', fontsize=8, color='blue')
plt.annotate(f'Median: {quartiles[0.5]}', xy=(quartiles[0.5], 1), xytext=(-30,␣
 ↪-22),
             textcoords='offset points', fontsize=8, color='blue')
```

```
plt.annotate(f'Q3: {quartiles[0.75]}', xy=(quartiles[0.75], 1), xytext=(-20,␣
↪-22),
             textcoords='offset points', fontsize=8, color='blue')
plt.annotate(f'Min: {df["Founded"].min()}', xy=(df["Founded"].min(), 1),␣
↪xytext=(-5, -22),
             textcoords='offset points', fontsize=8, color='blue')
plt.annotate(f'Max: {df["Founded"].max()}', xy=(df["Founded"].max(), 1),␣
↪xytext=(-5, -22),
             textcoords='offset points', fontsize=8, color='blue')
plt.show()
```





What are must have features of graphs?

1. Titles
2. Axis labels
3. UNITS!!!!