```
In [30]: titanic.groupby("sex").agg(["min", "max"])
```

Out[30]:

	pclass		survived		name		age		sibsp		cabin		embarked	
	min	max	min	max	min	max	min	max	min	max	min	max	min	max
sex														
female	1	3	0	1	Abbott, Mrs. Stanton (Rosa Hunt)	del Carlo, Mrs. Sebastiano (Argenia Genovesi)	0.1667	76.0	0	8	?	G6	?	S
male	1	3	0	1	Abbing, Mr. Anthony	van Melkebeke, Mr. Philemon	0.3333	80.0	0	8	?	T	C	S

2 rows × 26 columns

```
In [31]: titanic.groupby("sex").agg({"age": ["min", "max"], "pclass": [illegible]})
```

Out[31]:

	age		pclass
	min	max	mean
sex			
female	0.1667	76.0	2.154506
male	0.3333	80.0	2.372479

```
In [32]: carstocks.groupby("Symbol").agg({"Open": "mean", "Close": "m[illegible]lume": ["mean"
```

Out[32]:

	Open	Close	Volume	
	mean	mean	mean	sum
Symbol				
GM	61.937693	62.164615	2.025259e+07	263283700
LCID	48.761538	49.829231	1.081098e+08	1405427200
RIVN	127.710000	127.523077	5.252395e+07	682811400

```
397, 449, 508], 55.0: [79, 154, 158, 186, 210, 246, 308, 441],
0: [233, 264, 266, 267], 57.0: [276, 310, 330, 472, 490], 58.0:
4, 192, 215], 59.0: [42, 561, 736], 60.0: [43, 116, 123, 304,
5: [1225], 61.0: [252, 287, 300, 487, 1068], 62.0: [279, 284, 321,
0: [6, 286, 455, 1261], 64.0: [78, 83, 115, 217, 303], 65.0:
0: [594], 67.0: [285], 70.0: [81, 506], 70.5: [727], 71.0: [9,
5], 76.0: [61], 80.0: [14]}
```

In [17]:
```
df.groupby("age").first()
```

Out[17]:

age	pclass	survived	sex
0.1667	3	1	female
0.3333	3	0	male
0.4167	3	1	male
0.6667	2	1	male
0.7500	3	1	female
...	...	...	...
70.5000	3	0	male
71.0000	1	0	male
74.0000	3	0	male
76.0000	1	1	female
80.0000	1	1	male

98 rows × 3 columns

```
In [33]: def range(s):
             return s.max() - s.min()

         titanic.groupby("pclass")["age"].agg(["min", "max", range])
```

Out[33]:

pclass	min	max	range
1	0.9167	80.0	79.0833
2	0.6667	70.0	69.3333
3	0.1667	74.0	73.8333

```
In [34]: titanic["age"].size - titanic["age"].count()
```

```
Out[34]: 263
```

```
In [35]: def count_nulls(s):
             return s.size - s.count()

         titanic.groupby("pclass")["age"].agg(count_nulls)
```

```
Out[35]: pclass
         1     39.0
         2     16.0
         3    208.0
         Name: age, dtype: float64
```

```
In [36]: carstocks.groupby("Symbol").agg({"Open": ["min", "max"],"Close": ["min", "max"], })
```

Out[36]:

	Open		Close	
	min	max	min	max
Symbol				
GM	57.849998	64.330002	59.270000	64.610001
LCID	42.299999	56.200001	40.750000	55.520000
RIVN	106.750000	163.800003	100.730003	172.009995

```
In [37]: carstocks.groupby("Symbol").agg(
             min_open=("Open", "min"),
             max_open=("Open", "max"),
             min_close=("Close", "min"),
             max_close=("Close", "max")
         )
```

Out[37]:

Symbol	min_open	max_open	min_close	max_close
GM	57.849998	64.330002	59.270000	64.610001
LCID	42.299999	56.200001	40.750000	55.520000
RIVN	106.750000	163.800003	100.730003	172.009995

```
    "Close": ["min", "max"],
})
```

Out[38]:

	Symbol	Open		Close	
		min	max	min	max
0	GM	57.849998	64.330002	59.270000	64.610001
1	LCID	42.299999	56.200001	40.750000	55.520000
2	RIVN	106.750000	163.800003	100.730003	172.009995

In [39]:
```
carstocks.groupby("Symbol").agg({
    "Open": ["min", "max"],
    "Close": ["min", "max"],
})
```

Out[39]:

	Open		Close	
	min	max	min	max
Symbol				
GM	57.849998	64.330002	59.270000	64.610001
LCID	42.299999	56.200001	40.750000	55.520000
RIVN	106.750000	163.800003	100.730003	172.009995

Grouping By Multiple Columns & Multi Indexes

In [40]:
```
titanic.groupby("sex")["age"].mean()
```

Out[40]:
```
sex
female    28.687071
male      30.585233
Name: age, dtype: float64
```

In [41]:
```
titanic.groupby(["sex", "pclass", "survived"])["age"].mean()
```

Out[41]:
```
sex     pclass  survived
female  1       0           35.200000
                1           37.109375
        2       0           34.090909
                1           26.711051
        3       0           23.418750
                1           20.814815
male    1       0           43.658163
                1           36.168240
        2       0           33.092593
                1           17.449274
        3       0           26.679598
                1           22.436441
Name: age, dtype: float64
```

```
...groupby("Symbol", as_index=False).agg(
```

```
In [42]: titanic.groupby(["sex", "pclass"]).mean()
```

Out[42]:

sex	pclass	survived	age	sibsp	parch
female	1	0.965278	37.037594	0.555556	0.472222
	2	0.886792	27.499191	0.500000	0.650943
	3	0.490741	22.185307	0.791667	0.731481
male	1	0.340782	41.029250	0.340782	0.279330
	2	0.146199	30.815401	0.327485	0.192982
	3	0.152130	25.962273	0.470588	0.255578

```
In [43]: titanic.head()
```

Out[43]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	bo
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.55	C22 C26	S	
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.55	C22 C26	S	
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.55	C22 C26	S	
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.55	C22 C26	S	

In [20]:
```
gbo["age"].mean()
```

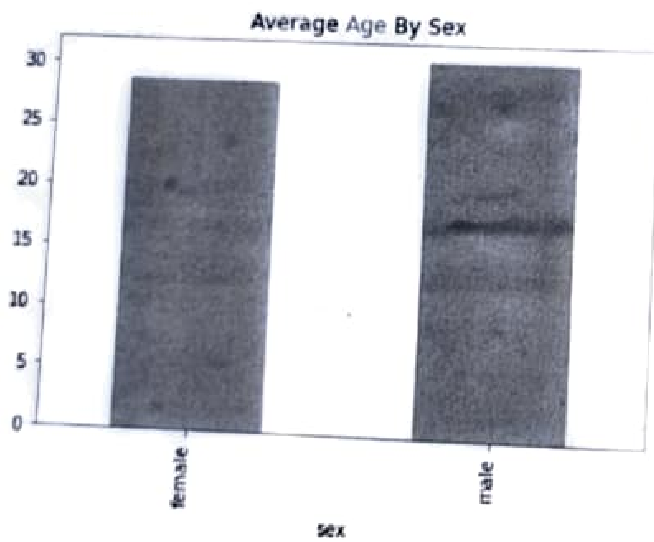
Out[20]:
```
sex
female    28.687071
male      30.585233
Name: age, dtype: float64
```

In [21]:
```
gbo["age"].max()
```

Out[21]:
```
sex
female    76.0
male      80.0
Name: age, dtype: float64
```

In [22]:
```
gbo["age"].mean().plot(kind="bar", title="Average Age By Sex")
```

Out[22]: `<AxesSubplot:title={'center':'Average Age By Sex'}, xlabel='sex'>`



In [23]:
```
titanic.groupby("pclass")["age"].mean()
```

Out[23]:
```
pclass
1    39.159918
2    29.506705
3    24.816367
Name: age, dtype: float64
```

In [24]:
```
titanic.groupby("sex")["pclass"].mean()
```

Out[24]:
```
sex
female    2.154506
male      2.372479
Name: pclass, dtype: float64
```

```
In [25]: titanic.groupby("sex").mean()
```

Out[25]:

sex	pclass	survived	age	sibsp	parch
female	2.154506	0.727468	28.687071	0.652361	0.633047
male	2.372479	0.190985	30.585233	0.413998	0.247924

```
In [26]: titanic.groupby("sex").median()
```

Out[26]:

sex	pclass	survived	age	sibsp	parch
female	2	1	27.0	0	0
male	3	0	28.0	0	0

```
In [27]: carstocks.groupby("Symbol")["High"].max()
```

```
Out[27]: Symbol
GM        65.180000
LCID      57.750000
RIVN     179.470001
Name: High, dtype: float64
```

Agg

```
In [28]: titanic.groupby("sex")["age"].agg("min")
```

```
Out[28]: sex
female    0.1667
male      0.3333
Name: age, dtype: float64
```

```
In [29]: titanic.groupby("sex")["age"].agg(["min", "max", "mean", "median"])
```

Out[29]:

sex	min	max	mean	median
female	0.1667	76.0	28.687071	27.0
male	0.3333	80.0	30.585233	28.0

```
gbo["age"].mean()
```

In [18]: `gbo.get_group("male")`

Out[18]:

	pclass	survived	sex	age
1	1	1	male	0.9167
3	1	0	male	30.0000
5	1	1	male	48.0000
7	1	0	male	39.0000
9	1	0	male	71.0000
...	...	...	...	...
1302	3	0	male	NaN
1303	3	0	male	NaN
1306	3	0	male	26.5000
1307	3	0	male	27.0000
1308	3	0	male	29.0000

843 rows × 4 columns

In [19]:
```
for name, group in gbo:
    print(name)
    print("----------")
    print(group)
```

```
female
----------
      pclass  survived     sex   age
0          1         1  female  29.0
2          1         0  female   2.0
4          1         0  female  25.0
6          1         1  female  63.0
8          1         1  female  53.0
...      ...       ...     ...   ...
1286       3         1  female  38.0
1290       3         1  female  47.0
1300       3         1  female  15.0
1304       3         0  female  14.5
1305       3         0  female   NaN

[466 rows x 4 columns]
male
----------
      pclass  survived   sex      age
1          1         1  male   0.9167
3          1         0  male  30.0000
5          1         1  male  48.0000
7          1         0  male  39.0000
9          1         0  male  71.0000
...      ...       ...   ...      ...
1302       3         0  male      NaN
1303       3         0  male      NaN
1306       3         0  male  26.5000
1307       3         0  male  27.0000
1308       3         0  male  29.0000

[843 rows x 4 columns]
```

Pandas Groupby & Aggregates

In [1]:
```
import pandas as pd
```

In [2]:
```
carstocks = pd.read_csv("C:/Users/ashuv/Deskt... taAnaly...  stocks.csv")
```

In [4]:
```
carstocks["Close"].mean()
```

Out[4]:
```
79.83897420512822
```

In [5]:
```
carstocks[carstocks["Symbol"] == "RIVN"]["Clos ].mean()
```

Out[5]:
```
127.52307653846154
```

In [6]:
```
carstocks[carstocks["Symbol"] == "GM"]["Close"].mean()
```

Out[6]:
```
62.16461546153845
```

In [7]:
```
carstocks[carstocks["Symbol"] == "LCID"]["Close ].mean()
```

Out[7]:
```
49.82923061538462
```

Groupby Basics ¶

In [8]:
```
carstocks.groupby("Symbol")["Close"].mean()
```

Out[8]:
```
Symbol
GM        62.164615
LCID      49.829231
RIVN     127.523077
Name: Close, dtype: float64
```

In [9]:
```
titanic = pd.read_csv("C:/Users/ashuv/Desktop/D... Analysis...  ...csv")
titanic['age'] = titanic["age"].replace(['?'], [None]).astype(    )
```

In [10]:
```
df = titanic[["pclass", "survived", "sex", "age"]]
```

In [11]:
```
gbo = df.groupby(by="sex")
```

In [12]:
```
gbo
```

Out[12]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000... ...ACDFDC0>
```

In [13]:
```
gbo.ngroups
```

Out[13]:
```
2
```

In [20]: gbo["age"]

In [14]: gbo.groups

Out[14]:
```
{'female': [0, 2, 4, 6, 8, 11, 12, 13, 17, 18, 21, 23, 24, 27, 28, 32, 33, 35, 3
6, 41, 42, 43, 44, 48, 50, 55, 57, 59, 61, 63, 65, 66, 67, 69, 72, 73, 76, 78, 7
9, 82, 83, 85, 88, 90, 92, 95, 97, 98, 99, 102, 103, 104, 105, 107, 108, 111, 11
2, 113, 116, 117, 121, 122, 124, 127, 129, 130, 131, 134, 137, 139, 141, 144, 14
6, 149, 153, 155, 159, 160, 161, 163, 167, 168, 169, 176, 178, 180, 181, 182, 18
6, 187, 188, 190, 192, 193, 195, 198, 199, 204, 207, 208, ...], 'male': [1, 3, 5,
7, 9, 10, 14, 15, 16, 19, 20, 22, 25, 26, 29, 30, 31, 34, 37, 38, 39, 40, 45, 46,
47, 49, 51, 52, 53, 54, 56, 58, 60, 62, 64, 68, 70, 71, 74, 75, 77, 80, 81, 84, 8
6, 87, 89, 91, 93, 94, 96, 100, 101, 106, 109, 110, 114, 115, [illegible], 119, 120, 123,
125, 126, 128, 132, 133, 135, 136, 138, 140, 142, 143, 145, 147, 148, 150, 151, 1
52, 154, 156, 157, 158, 162, 164, 165, 166, 170, 171, 172, 173, 174, 175, 177, 17
9, 183, 184, 185, 189, 191, ...]}
```

In [15]: df

Out[15]:

	pclass	survived	sex	age
0	1	1	female	29.0000
1	1	1	male	0.9167
2	1	0	female	2.0000
3	1	0	male	30.0000
4	1	0	female	25.0000
...	...	...	...	...
1304	3	0	female	14.5000
1305	3	0	female	NaN
1306	3	0	male	26.5000
1307	3	0	male	27.0000
1308	3	0	male	29.0000

1309 rows × 4 columns

```
In [16]: df.groupby("age").groups

Out[16]: {0.1667: [763], 0.3333: [747], 0.4167: [1240], 0.6667: [427], [illegible] [657, 658, 11
11], 0.8333: [359, 548, 611], 0.9167: [1, 590], 1.0: [339, 478, [illegible], 762, 826, 89
5, 937, 1048, 1101, 1187], 2.0: [2, 514, 540, 587, 624, 866, [illegible], 1103, 1144, 11
56, 1209, 1230], 3.0: [479, 515, 549, 641, 734, 1098, 1112], 4.0: [94, 340, 588,
622, 894, 916, 934, 1142, 1189, 1206], 5.0: [591, 639, 643, [illegible], [illegible]], 6.0: [273,
430, 623, 678, 1025, 1097], 7.0: [434, 1102, 1143, 1256], 8.0: [illegible], 385, 398, 54
1, 1099, 1145], 9.0: [627, 640, 679, 733, 807, 820, 825, 1082, [illegible], 1257], 10.0:
[828, 1141, 1207, 1265], 11.0: [54, 628, 827, 855], 11.5: [[illegible]], 12.0: [341, 58
2, 1056], 13.0: [249, 501, 601, 642, 653], 14.0: [55, 513, [illegible], [illegible], 1057, 1105,
1236, 1279], 14.5: [1171, 1304], 15.0: [193, 350, 792, 1007, [illegible], 1300], 16.0:
[159, 187, 195, 416, 510, 602, 604, 709, 761, 787, 810, 818, [illegible], 1093, 1104, 116
1, 1232, 1244, 1275], 17.0: [53, 92, 229, 295, 390, 458, 482, [illegible], 650, 700, 701,
738, 740, 755, 772, 791, 841, 885, 910, 1133], 18.0: [11, 198, [illegible], 250, 270, 28
9, 326, 331, 386, 394, 395, 405, 408, 445, 558, 607, 612, 619, [illegible], 661, 665, 67
6, 695, 698, 717, 719, 786, 799, 809, 859, 938, 1045, 1060, [illegible], 1157, 1205, 126
0, 1273, 1288], 18.5: [568, 692, 919], 19.0: [27, 114, 137, [illegible], [illegible], 337, 344, 3
64, 503, 518, 530, 534, 552, 621, 669, 694, 731, 737, 744, 771, [illegible], 839, 898, 10
11, 1050, 1108, 1127, 1217, 1226], 20.0: [353, 446, 520, 559, [illegible], 615, 633, 654,
664, 673, 687, 718, 836, 846, 883, 907, 970, 1049, 1089, 1091, [illegible], 1191, 1278],
20.5: [977], 21.0: [190, 251, 307, 315, 317, 383, 404, 419, [illegible], [illegible], 453, 533, 5
53, 648, 675, 685, 693, 696, 702, 703, 704, 713, 754, 806, 850, [illegible], 908, 911, 91
3, 1017, 1020, 1062, 1065, 1107, 1117, 1140, 1182, 1204, 1224, [illegible], 1295], 22.0:
[36, 73, 122, 130, 220, 227, 236, 361, 380, 463, 468, 481, [illegible], [illegible], 671, 686, 68
9, 690, 725, 743, 753, 769, 785, 817, 862, 867, 890, 915, 932, [illegible], 952, 953, 98
6, 1046, 1067, 1079, 1119, 1147, 1201, 1227, 1277, 1280, 1281], [illegible]: [741], 23.
0: [102, 113, 140, 214, 225, 272, 332, 345, 403, 447, 465, [illegible], [illegible], 571, 579, 64
5, 649, 652, 780, 784, 861, 904, 980, 1075, 1090, 1223], 23.5: [[illegible]], 24.0: [12,
16, 111, 132, 153, 199, 255, 268, 271, 349, 376, 392, 421, [illegible], [illegible], 437, 438, 44
2, 460, 462, 467, 486, 494, 550, 599, 616, 637, 660, 708, 712, [illegible], 722, 752, 77
9, 783, 840, 845, 965, 985, 1010, 1019, 1040, 1125, 1134, [illegible], [illegible], 1237], 24.
5: [1192], 25.0: [4, 25, 26, 143, 144, 327, 354, 356, 370, [illegible], [illegible], 555, 557, 56
5, 567, 605, 617, 635, 751, 766, 814, 853, 878, 941, 966, 1024, 1118, 1120, 1129,
1165, 1190, 1234, 1238, 1254], 26.0: [13, 22, 72, 346, 360, [illegible], [illegible], 517, 554, 5
98, 609, 613, 614, 631, 634, 663, 670, 677, 716, 764, 803, 849, [illegible], 889, 933, 93
6, 949, 975, 1061, 1113], 26.5: [1306], 27.0: [64, 71, 87, 90, [illegible], 151, 313, 348,
401, 507, 539, 556, 573, 575, 585, 630, 667, 730, 750, 857, 870, 877, 899, 906, 9
78, 1026, 1229, 1296, 1299, 1307], 28.0: [29, 52, 112, 203, [illegible], [illegible], 334, 338, 3
73, 375, 388, 431, 519, 527, 546, 572, 707, 711, 749, 838, 844, [illegible], 864, 869, 97
2, 1021, 1059, 1083, 1087, 1126, 1270, 1271], 28.5: [222, 1066, [illegible]], 29.0: [0,
24, 189, 226, 369, 372, 374, 391, 407, 452, 521, 526, 574, 580, [illegible], 688, 715, 74
6, 880, 893, 935, 950, 951, 990, 1058, 1100, 1196, 1231, 1258, [illegible]], 30.0: [3, 3
2, 67, 110, 117, 182, 191, 194, 209, 230, 258, 323, 325, 381, [illegible], 424, 426, 433,
476, 496, 499, 538, 545, 560, 562, 578, 608, 651, 697, 726, 732, [illegible], 760, 778, 8
75, 912, 969, 974, 1218, 1267], 30.5: [992, 1251], 31.0: [89, [illegible], 213, 239, 298,
309, 319, 378, 379, 474, 493, 577, 580, 596, 723, 724, 823, [illegible], 1036, 1094, 122
8, 1274, 1276], 32.0: [18, 278, 336, 389, 443, 464, 498, 536, [illegible], 655, 674, 684,
714, 776, 834, 905, 909, 959, 981, 1088, 1110, 1131, 1220, 1248], 32.5: [173, 51
2, 584, 1285], 33.0: [51, 65, 88, 207, 242, 245, 248, 457, 542, [illegible], 656, 765, 78
1, 821, 891, 897, 914, 996, 1051, 1222, 1269], 34.0: [259, 328, [illegible], 396, 400, 41
4, 415, 423, 466, 484, 537, 544, 748, 888, 1031, 1239], 34.5: [[illegible], 960], 35.0:
[28, 127, 129, 149, 163, 164, 167, 183, 257, 261, 302, 362, 417, [illegible], 563, 603, 6
18, 638, 691, 729, 995, 1008, 1148], 36.0: [19, 49, 56, 57, [illegible], [illegible], 82, 105, 109,
202, 244, 322, 329, 333, 342, 355, 409, 448, 485, 543, 592, [illegible], [illegible], 759, 770, 9
39, 963, 968, 1259, 1266, 1298], 36.5: [516, 758], 37.0: [20, [illegible], 126, 208, 212,
368, 710, 837, 943], 38.0: [85, 103, 138, 165, 168, 234, 411, [illegible], 626, 646, 699,
824, 1139, 1286], 38.5: [1169], 39.0: [7, 76, 84, 101, 180, [illegible], [illegible], 291, 296, 5
04, 509, 522, 629, 632, 790, 917, 964, 1064, 1146, 1183], 40.0[illegible], 150, 260, 27
5, 299, 352, 406, 497, 564, 576, 583, 610, 644, 662, 683, 831, [illegible], 1210], 40.5:
[796, 797, 1264], 41.0: [38, 44, 175, 502, 532, 566, 822, 848, [illegible], 1106, 1158],
42.0: [23, 34, 47, 156, 162, 177, 185, 347, 357, 358, 399, 454, [illegible]
3, 873, 1084], 43.0: [120, 238, 281, 435, 535, 728, [illegible], 489, [illegible]
```

TextParser is also equipped with a get_chunk method that enables you to read pieces of an arbitrary size.

Writing Data to Text Format

Data can also be exported to a delimited format. Let's consider one of the CSV files read before:

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Using DataFrame's to_csv method, we can write the data out to a comma-separated file:

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to sys.stdout so it prints the text result to the console):

```
In [45]: import sys

In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:


```
In [48]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a to_csv method:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

```
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

Working with Delimited Formats

It's possible to load most forms of tabular data from disk using funct
das.read_table. In some cases, however, some manual processing may
It's not uncommon to receive a file with one or more malformed line
read_table. To illustrate the basic tools, consider a small CSV file:

```
In [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

For any file with a single-character delimiter, you can use Python's built
ule. To use it, pass any open file or file-like object to csv.reader:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

Iterating through the reader like a file yields tuples of values with any
ters removed:


```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

From there, it's up to you to do the wrangling necessary to put the dat
that you need it. Let's take this step by step. First, we read the file into a li

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

Then, we split the lines into the header line and the data lines:

```
In [58]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a   co
and the expression zip(*values), which transposes rows   s:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}
```

Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

Before we look at a large file, we make the pandas display settings more compact:

```
In [33]: pd.options.display.max_rows = 10
```

Now we have:

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
```



```
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

If you want to only read a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

To read a file in pieces, specify a `chunksize` as a number of rows:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

The `TextParser` object returned by `read_csv` allows you to iterate over the parts of the file according to the `chunksize`. For example, we can iterate over `ex6.csv`, aggregating the value counts in the `'key'` column like so:

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

We have then:

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

```
5,6,7,8,world
9,10,11,12,foo
```

Here I used the Unix `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of `cat` to achieve the same effect.

Since this is comma-delimited, we can use `read_csv` to read it into a DataFrame:

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

We could also have used `read_table` and specified the delimiter:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

A file will not always have a header row. Consider this file:

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```


Suppose you wanted the `message` column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named `'message'` using the `index_col` argument:

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

While you could do some munging by hand, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for `read_table`. This can be expressed by the regular expression `\s+`, so we have then:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Because there was one fewer column name than the number of data rows, `read_table` infers that the first column should be the DataFrame's index in this special case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see a partial listing in Table 6-2). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```


Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* value. By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')
```



```
In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

The `na_values` option can take either a list or set of strings to consider missing values:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Different NA sentinels can be specified for each column in a dict

Suppose you wanted the message column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named 'message' using the index_col argument:

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
    ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. Consider a text file that looks like this:

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
```



```
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

While you could do some munging by hand, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for read_table. This can be expressed by the regular expression \s+, so we have then:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
```