Querying a DataFrame

Before we talk about how to query data frames, we need to talk about Boolean masking.
Boolean masking is the heart of fast and efficient querying in NumPy.
It's analogous a bit to masking used in other computational areas.

A Boolean mask is an array which can be of one dimension like a series, or two dimensions like a data frame, where each of the values in the array are either true or false.
This array is essentially overlaid on top of the data structure that we're querying.
And any cell aligned with the true value will be admitted into our final result, and any sign aligned with a false value will not.

Boolean masking is powerful conceptually and is the cornerstone of efficient NumPy and pandas querying.
This technique is well used in other areas of computer science, for instance, in graphics.
But it doesn't really have an analogue in other traditional relational databases, so
I think it's worth pointing out here.
Boolean masks are created by applying operators directly to the pandas series or
DataFrame objects.

df

	Animals	Owners
0	Dog	Chris
1	Bear	Kevyn
2	Tiger	Bob
3	Moose	Vinod
4	Giraffe	Daniel
5	Hippo	Fil
6	Mouse	Stephanie

+

Boolean mask

True	True
True	True
False	False
True	True
False	False
False	False
False	False

=

result

	Animals	Owners
0	Dog	Chris
1	Bear	Kevyn
3	Moose	Vinod

For instance, in our Olympics data set, you might be interested in seeing only those countries who have achieved a gold medal at the summer Olympics.
To build a Boolean mask for this query, we project the gold column using the indexing operator and apply the greater than operator with a comparison value of zero.

This is essentially broadcasting a comparison operator, greater than, with the results being returned as a Boolean series.
The resultant series is indexed where the value of each cell is either true or false depending on whether a country has won at least one gold medal, and the index is the country name.

```
df['Gold'] >0
```

```
Afghanistan (AFG)                                 False
Algeria (ALG)                                      True
Argentina (ARG)                                    True
Armenia (ARM)                                      True
Australasia (ANZ) [ANZ]                            True
                                                   ...
Independent Olympic Participants (IOP) [IOP]      False
Zambia (ZAM) [ZAM]                                False
Zimbabwe (ZIM) [ZIM]                               True
Mixed team (ZZX) [ZZX]                             True
Totals                                             True
Name: Gold, Length: 147, dtype: bool
```

So this builds us the Boolean mask, which is half the battle. What we want to do next is overlay that mask on the data frame.
We can do this using the where function.
The where function takes a Boolean mask as a condition, applies it to the data frame or series, and returns a new data frame or series of the same shape.
Let's apply this Boolean mask to our Olympics data and create a data frame of only those countries who have won a gold at a summer games.

```
only_gold = df.where(df['Gold'] > 0)
only_gold.head()
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Afghanistan (AFG)	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Algeria (ALG)	12.0	5.0	2.0	8.0	15.0	3.0	0.0	0.0	0.0	0.0	15.0	5.0	2.0	8.0	15.0
Argentina (ARG)	23.0	18.0	24.0	28.0	70.0	18.0	0.0	0.0	0.0	0.0	41.0	18.0	24.0	28.0	70.0
Armenia (ARM)	5.0	1.0	2.0	9.0	12.0	6.0	0.0	0.0	0.0	0.0	11.0	1.0	2.0	9.0	12.0
Australasia (ANZ) [ANZ]	2.0	3.0	4.0	5.0	12.0	0.0	0.0	0.0	0.0	0.0	2.0	3.0	4.0	5.0	12.0

We see that the resulting data frame keeps the original indexed values, and only data from countries that met the condition are retained.
All of the countries which did not meet the condition have NaN data instead. This is okay.
Most statistical functions built into the data frame object ignore values of NaN.

For instance, if we call the df.count on the only gold data frame, we see that there are 100 countries which have had gold medals awarded at the summer games, while if we call count on the original data frame, we see that there are 147 countries total.

```
: only_gold['Gold'].count()
: 100

: df['Gold'].count()
: 147
```

Often we want to drop those rows which have no data.
To do this, we can use the **drop NA function**.

```
only_gold = only_gold.dropna()
only_gold.head()
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Algeria (ALG)	12.0	5.0	2.0	8.0	15.0	3.0	0.0	0.0	0.0	0.0	15.0	5.0	2.0	8.0	15.0
Argentina (ARG)	23.0	18.0	24.0	28.0	70.0	18.0	0.0	0.0	0.0	0.0	41.0	18.0	24.0	28.0	70.0
Armenia (ARM)	5.0	1.0	2.0	9.0	12.0	6.0	0.0	0.0	0.0	0.0	11.0	1.0	2.0	9.0	12.0
Australasia (ANZ) [ANZ]	2.0	3.0	4.0	5.0	12.0	0.0	0.0	0.0	0.0	0.0	2.0	3.0	4.0	5.0	12.0
Australia (AUS) [AUS] [Z]	25.0	139.0	152.0	177.0	468.0	18.0	5.0	3.0	4.0	12.0	43.0	144.0	155.0	181.0	480.0

You can optionally provide drop NA the axes it should be considering.
Remember that the axes is just an indicator for the columns or rows and that the default is zero, which means rows.

When you find yourself talking about pandas and saying phrases like, often I want to, it's quite likely the developers have included a shortcut for this common operation.
For instance, in this example, we don't actually have to use the where function explicitly.
The pandas developers allow the indexing operator to take a Boolean mask as a value instead of just a list of column names.
The syntax might look a little messy, especially if you're not used to programming languages with overloaded operators, but the result is that you're able to filter and reduce data frames relatively quickly.

Here's a more concise example of how we could query this data frame.
You'll notice that there are no NaNs when you query the data frame in this manner.
pandas automatically filters out the rows with now values.

```
only_gold = df[df['Gold'] > 0]
only_gold.head()
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Algeria (ALG)	12	5	2	8	15	3	0	0	0	0	15	5	2	8	15
Argentina (ARG)	23	18	24	28	70	18	0	0	0	0	41	18	24	28	70
Armenia (ARM)	5	1	2	9	12	6	0	0	0	0	11	1	2	9	12
Australasia (ANZ) [ANZ]	2	3	4	5	12	0	0	0	0	0	2	3	4	5	12
Australia (AUS) [AUS] [Z]	25	139	152	177	468	18	5	3	4	12	43	144	155	181	480

One more thing to keep in mind if you're not used to Boolean or bit masking for data reduction. The output of two Boolean masks being compared with logical operators is another Boolean mask.
This means that you can chain together a bunch of and/or statements in order to create more complex queries and the result is a single Boolean mask.

For instance, we could create a mask for all of those countries who have received a gold in the summer Olympics and logically order that with all of those countries who have received a gold in the winter Olympics.
If we apply this to the data frame and use the length function to see how many rows there are, we see that there are 101 countries which have won a gold metal at some time.

Another example for fun.
Have there been any countries who have only won a gold in the winter Olympics and never in the summer Olympics?
Here's one way to answer that.

```
len(df[(df['Gold'] > 0) | (df['Gold.1'] > 0)])
```

```
101
```

```
df[(df['Gold.1'] > 0) & (df['Gold'] == 0)]
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Liechtenstein (LIE)	16	0	0	0	0	18	2	2	5	9	34	2	2	5	9

Extremely important, and often an issue for new users, is to remember that each Boolean mask needs to be encased in parenthesis because of the order of operations.

In this lecture, we took a look at Boolean masking.
We didn't have much code to write, but applying masks to data frames is really a core panda's work flow and is worth practicing.