Lab 2: Table Operations and Arrays

Welcome to Lab 2! In this lab, we'll learn how to import a module and practice table operations! we'll also see how to work with *arrays* of data, such as all the numbers between 0 and 100 or all the words in the chapter of a book. Lastly, we'll create tables and practice analyzing them with our knowledge of table operations.

First, set up the imports by running the cell below.

In [2]:
```
# Just run this cell

import numpy as np
from datascience import *
```

1. Review: The building blocks of Python code

The two building blocks of Python code are *expressions* and *statements*. An **expression** is a piece of code that

- is self-contained, meaning it would make sense to write it on a line by itself, and
- usually evaluates to a value.

Here are two expressions that both evaluate to 3:

```
3
5 - 2
```

One important type of expression is the **call expression**. A call expression begins with the name of a function and is followed by the argument(s) of that function in parentheses. The function returns some value, based on its arguments. Some important mathematical functions are listed below.

Function	Description
`abs`	Returns the absolute value of its argument
`max`	Returns the maximum of all its arguments
`min`	Returns the minimum of all its arguments
`pow`	Raises its first argument to the power of its second argument
`round`	Rounds its argument to the nearest integer

Here are two call expressions that both evaluate to 3:

```
abs(2 - 5)
max(round(2.8), min(pow(2, 10), -1 * pow(2, 10)))
```

The expression `2 - 5` and the two call expressions given above are examples of **compound expressions**, meaning that they are actually combinations of several smaller expressions. `2 - 5` combines the expressions `2` and `5` by subtraction. In this case, `2` and `5` are called **subexpressions** because they're expressions that are part of a larger expression.

A **statement** is a whole line of code. Some statements are just expressions. The expressions listed above are examples.

Other statements *make something happen* rather than *having a value*. For example, an **assignment statement** assigns a value to a name.

A good way to think about this is that we're **evaluating the right-hand side** of the equals sign and **assigning it to the left-hand side**. Here are some assignment statements:

```
height = 1.3
the_number_five = abs(-5)
absolute_height_difference = abs(height - 1.688)
```

An important idea in programming is that large, interesting things can be built by combining many simple, uninteresting things. The key to understanding a complicated piece of code is breaking it down into its simple components.

For example, a lot is going on in the last statement above, but it's really just a combination of a few things. This picture describes what's going on.

Question 1.1. In the next cell, assign the name `new_year` to the larger number among the following two numbers:

1. the **absolute value** of $2^5 - 2^{11} - 2^1 + 1$, and
2. $5 \times 13 \times 31 + 5$.

Try to use just one statement (one line of code).

In [1]:
```
new_year = 5*13*31+5
new_year
```

Out[1]:
```
2020
```

In [2]:
```
# TEST
new_year == 2020
```

Out[2]: True

We've asked you to use one line of code in the question above because it only involves mathematical operations. However, more complicated programming questions will more require more steps. It isn't always a good idea to jam these steps into a single line because it can make the code harder to read and harder to debug.

Good programming practice involves splitting up your code into smaller steps and using appropriate names. You'll have plenty of practice in the rest of this course!

2. Importing code

source

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to **import modules**. A module is a file with Python code that has defined variables and functions. By importing a module, we are able to use its code in our own notebook.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with a radius of 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

In [36]:
```
import math
radius = 5
area_of_circle = radius**2 * math.pi
area_of_circle
```

Out[36]:
```
78.53981633974483
```

In the code above, the line `import math` imports the math module. This statement creates a module and then assigns the name `math` to that module. We are now able to access any variables or functions defined within `math` by typing the name of the module followed by a dot, then followed by the name of the variable or function we want.

```
<module name>.<name>
```

Question 2.1. The module `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

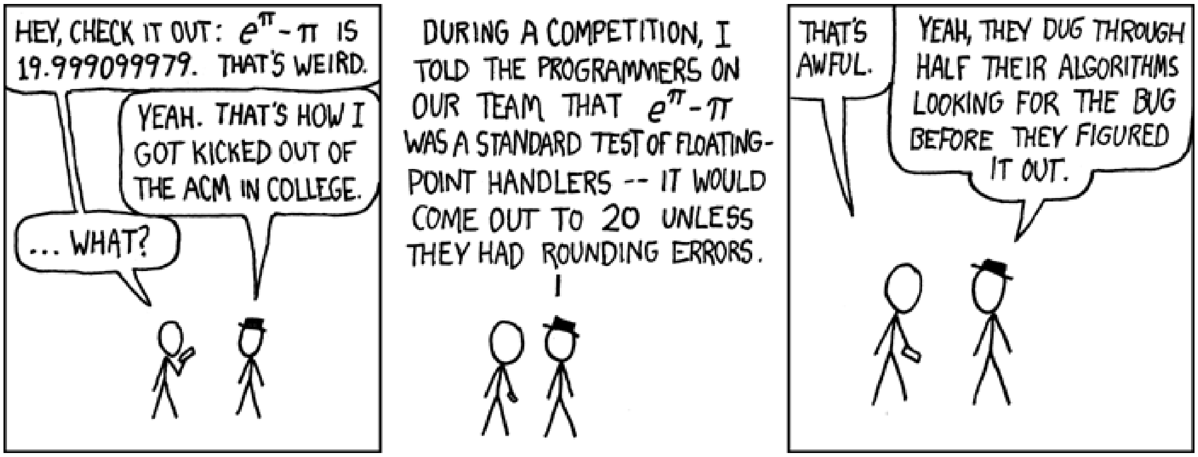
Remember: You can access `pi` from the `math` module as well!

In [6]:
```
near_twenty = math.e ** math.pi - math.pi
near_twenty
```

Out[6]:
```
19.99909997918947
```

In [7]:
```
# TEST
round(near_twenty, 8) == 19.99909998
```

Out[7]:
```
True
```



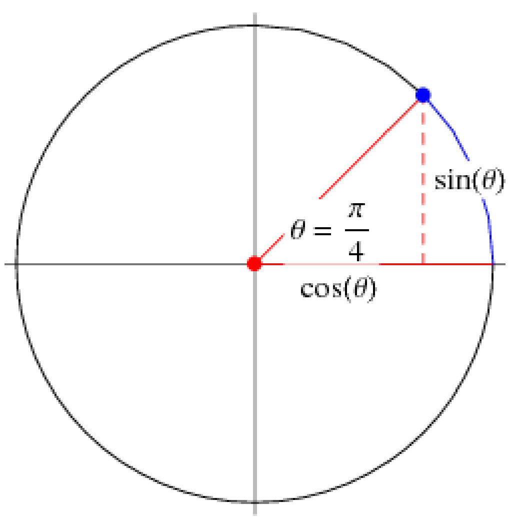
Source Explaination

2.1. Accessing functions

In the question above, you accessed variables within the `math` module.

Modules also define **functions**. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3. (Note that this sine function considers its argument to be in radians, not degrees. 180 degrees are equivalent to π radians.)

Question 2.1.1. A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse (the radius of the circle in the picture) is 1, then the height is $\sin(\frac{\pi}{4})$. Compute that value using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.



Source

In [8]:
```
sine_of_pi_over_four = math.sin(math.pi/4)
sine_of_pi_over_four
```

Out[8]:
```
0.7071067811865475
```

In [9]:
```
# TEST
# Hint: You can write the sine of 1.5*pi as:
#   math.sin(1.5 * math.pi)
import math
round(sine_of_pi_over_four, 8) == 0.70710678
```

Out[9]:
```
True
```

There are various ways to import and access code from outside sources. The method we used above — `import <module_name>` — imports the entire module and requires that we use `<module_name>.<name>` to access its code.

We can also import a specific constant or function instead of the entire module. Notice that you don't have to use the module name beforehand to reference that particular value. However, you do have to be careful about reassigning the names of the constants or functions to other values!

In [10]:
```
# Importing just cos and pi from math.
# We don't have to use `math.` in front of cos or pi
from math import cos, pi
print(cos(pi))

# We do have to use it in front of other functions from math, though
math.log(pi)
```

```
-1.0
```

Out[10]:
```
1.1447298858494002
```

Or we can import every function and value from the entire module.

In [11]:
```
# Lastly, we can import everything from math using the *
# Once again, we don't have to use 'math.' beforehand
from math import *
log(pi)
```

Out[11]:
```
1.1447298858494002
```

Don't worry too much about which type of import to use. It's often a coding style choice left up to each programmer. In this course, you'll always import the necessary modules when you run the setup cell (like the first code cell in this lab).

Let's move on to practicing some of the table operations you've learned in lecture!

3. Table operations

The table `farmers_markets.csv` contains data on farmers' markets in the United States (data collected by the USDA). Each row represents one such market.

Run the next cell to load the `farmers_markets` table.

In [3]:
```
# Just run this cell

farmers_markets = Table.read_table('farmers_markets.csv')
```

Let's examine our table to see what data it contains.

Question 3.1. Use the method `show` to display the first 5 rows of `farmers_markets`.

Note: The terms "method" and "function" are technically not the same thing, but for the purposes of this course, we will use them interchangeably.

Hint: `tbl.show(3)` will show the first 3 rows of `tbl`. Additionally, make sure not to call `.show()` without an argument, as this will crash your kernel!

In [4]:
```
farmers_markets.show(5)
```

FMID	MarketName	street	city	County	State	zip	x	y	
1012063	Caledonia Farmers Market Association - Danville	nan	Danville	Caledonia	Vermont	05828	-72.1403	44.411	ht
1011871	Stearns Homestead Farmers' Market	6975 Ridge Road	Parma	Cuyahoga	Ohio	44130	-81.7286	41.3751	
1011878	100 Mile Market	507 Harrison St	Kalamazoo	Kalamazoo	Michigan	49007	-85.5749	42.296	
1009364	106 S. Main Street Farmers Market	106 S. Main Street	Six Mile	nan	South Carolina	29682	-82.8187	34.8042	
1010691	10th Steet Community Farmers Market	10th Street and Poplar	Lamar	Barton	Missouri	64759	-94.2746	37.4956	

... (8541 rows omitted)

Notice that some of the values in this table are missing, as denoted by "nan." This means either that the value is not available (e.g. if we don't know the market's street address) or not applicable (e.g. if the market doesn't have a street address). You'll also notice that the table has a large number of columns in it!

`num_columns`

The table property `num_columns` returns the number of columns in a table. (A "property" is just a method that doesn't need to be called by adding parentheses.)

Example call: `<tbl>.num_columns`

Question 3.2. Use `num_columns` to find the number of columns in our farmers' markets dataset.

Assign the number of columns to `num_farmers_markets_columns`.

In [5]:
```
num_farmers_markets_columns = farmers_markets.num_columns
print("The table has", num_farmers_markets_columns, "columns in it!")
```

```
The table has 59 columns in it!
```

In [6]:
```
# TEST
num_farmers_markets_columns == 59
```

Out[6]:
```
True
```

`num_rows`

Similarly, the property `num_rows` tells you how many rows are in a table.

In [7]:
```
# Just run this cell

num_farmers_markets_rows = farmers_markets.num_rows
print("The table has", num_farmers_markets_rows, "rows in it!")
```

```
The table has 8546 rows in it!
```

`select`

Most of the columns are about particular products -- whether the market sells tofu, pet food, etc. If we're not interested in that information, it just makes the table difficult to read. This comes up more than you might think, because people who collect and publish data may not know ahead of time what people will want to do with it.

In such situations, we can use the table method `select` to choose only the columns that we want in a particular table. It takes any number of arguments. Each should be the name of a column in the table. It returns a new table with only those columns in it. The columns are in the order *in which they were listed as arguments*.

For example, the value of `farmers_markets.select("MarketName", "State")` is a table with only the name and the state of each farmers' market in `farmers_markets`.

Question 3.3. Use `select` to create a table with only the name, city, state, latitude (`y`), and longitude (`x`) of each market. Call that new table `farmers_markets_locations`.

Hint: Make sure to be exact when using column names with `select`; double-check capitalization!

In [8]:
```
farmers_markets_locations = farmers_markets.select('MarketName', 'city', 'State
farmers_markets_locations
```

Out[8]:

MarketName	city	State	x	y
Caledonia Farmers Market Association - Danville	Danville	Vermont	-72.1403	44.411
Stearns Homestead Farmers' Market	Parma	Ohio	-81.7286	41.3751
100 Mile Market	Kalamazoo	Michigan	-85.5749	42.296
106 S. Main Street Farmers Market	Six Mile	South Carolina	-82.8187	34.8042
10th Steet Community Farmers Market	Lamar	Missouri	-94.2746	37.4956
112st Madison Avenue	New York	New York	-73.9493	40.7939
12 South Farmers Market	Nashville	Tennessee	-86.7907	36.1184
125th Street Fresh Connect Farmers' Market	New York	New York	-73.9482	40.809
12th & Brandywine Urban Farm Market	Wilmington	Delaware	-75.5345	39.7421
14&U Farmers' Market	Washington	District of Columbia	-77.0321	38.917

... (8536 rows omitted)

In [9]:
```
# TEST
sorted(farmers_markets_locations.labels) == ['MarketName', 'State', 'city', 'x'
```

Out[9]:
```
True
```

In [10]:
```
# TEST
farmers_markets_locations.num_rows == 8546
```

Out[10]:
```
True
```

`drop`

`drop` serves the same purpose as `select`, but it takes away the columns that you provide rather than the ones that you don't provide. Like `select`, `drop` returns a new table.

Question 3.4. Suppose you just didn't want the `FMID` and `updateTime` columns in `farmers_markets`. Create a table that's a copy of `farmers_markets` but doesn't include those columns. Call that table `farmers_markets_without_fmid`.

In [11]:
```
farmers_markets_without_fmid = farmers_markets.drop('FMID', 'updateTime')
farmers_markets_without_fmid
```

Out[11]:

MarketName	street	city	County	State	zip	x	y	
Caledonia Farmers Market Association - Danville	nan	Danville	Caledonia	Vermont	05828	-72.1403	44.411	https://:
Stearns Homestead Farmers' Market	6975 Ridge Road	Parma	Cuyahoga	Ohio	44130	-81.7286	41.3751	
100 Mile Market	507 Harrison St	Kalamazoo	Kalamazoo	Michigan	49007	-85.5749	42.296	
106 S. Main Street Farmers Market	106 S. Main Street	Six Mile	nan	South Carolina	29682	-82.8187	34.8042	
10th Steet Community Farmers Market	10th Street and Poplar	Lamar	Barton	Missouri	64759	-94.2746	37.4956	
112st Madison Avenue	112th Madison Avenue	New York	New York	New York	10029	-73.9493	40.7939	
12 South Farmers Market	3000 Granny White Pike	Nashville	Davidson	Tennessee	37204	-86.7907	36.1184	
125th Street Fresh Connect Farmers' Market	163 West 125th Street and Adam Clayton Powell, Jr. Blvd.	New York	New York	New York	10027	-73.9482	40.809	
12th & Brandywine Urban Farm Market	12th & Brandywine Streets	Wilmington	New Castle	Delaware	19801	-75.5345	39.7421	
14&U Farmers' Market	1400 U Street NW	Washington	District of Columbia	District of Columbia	20009	-77.0321	38.917	

... (8536 rows omitted)

In [12]:

```
# TEST
farmers_markets_without_fmid.num_columns == 57
```

Out[12]:

```
True
```

In [13]:

```
# TEST
```

```
print(sorted(farmers_markets_without_fmid.labels))
```

```
['Bakedgoods', 'Beans', 'Cheese', 'Coffee', 'County', 'Crafts', 'Credit', 'Egg
s', 'Facebook', 'Flowers', 'Fruits', 'Grains', 'Herbs', 'Honey', 'Jams', 'Juic
es', 'Location', 'Maple', 'MarketName', 'Meat', 'Mushrooms', 'Nursery', 'Nut
s', 'Organic', 'OtherMedia', 'PetFood', 'Plants', 'Poultry', 'Prepared', 'SFMN
P', 'SNAP', 'Seafood', 'Season1Date', 'Season1Time', 'Season2Date', 'Season2Ti
me', 'Season3Date', 'Season3Time', 'Season4Date', 'Season4Time', 'Soap', 'Stat
e', 'Tofu', 'Trees', 'Twitter', 'Vegetables', 'WIC', 'WICcash', 'Website', 'Wi
ldHarvested', 'Wine', 'Youtube', 'city', 'street', 'x', 'y', 'zip']
```

Now, suppose we want to answer some questions about farmers' markets in the US. For example, which market(s) have the largest longitude (given by the `x` column)?

To answer this, we'll sort `farmers_markets_locations` by longitude.

In [14]:
```
farmers_markets_locations.sort('x')
```

Out[14]:

MarketName	city	State	x	y
Trapper Creek Farmers Market	Trapper Creek	Alaska	-166.54	53.8748
Kekaha Neighborhood Center (Sunshine Markets)	Kekaha	Hawaii	-159.718	21.9704
Hanapepe Park (Sunshine Markets)	Hanapepe	Hawaii	-159.588	21.9101
Kalaheo Neighborhood Center (Sunshine Markets)	Kalaheo	Hawaii	-159.527	21.9251
Hawaiian Farmers of Hanalei	Hanalei	Hawaii	-159.514	22.2033
Hanalei Saturday Farmers Market	Hanalei	Hawaii	-159.492	22.2042
Kauai Culinary Market	Koloa	Hawaii	-159.469	21.9067
Koloa Ball Park (Knudsen) (Sunshine Markets)	Koloa	Hawaii	-159.465	21.9081
West Kauai Agricultural Association	Poipu	Hawaii	-159.435	21.8815
Kilauea Neighborhood Center (Sunshine Markets)	Kilauea	Hawaii	-159.406	22.2112

... (8536 rows omitted)

Oops, that didn't answer our question because we sorted from smallest to largest longitude. To look at the largest longitudes, we'll have to sort in reverse order.

In [15]:
```
farmers_markets_locations.sort('x', descending=True)
```

Out[15]:

MarketName	city	State	x	y
Christian "Shan" Hendricks Vegetable Market	Saint Croix	Virgin Islands	-64.7043	17.7449
La Reine Farmers Market	Saint Croix	Virgin Islands	-64.7789	17.7322
Anne Heyliger Vegetable Market	Saint Croix	Virgin Islands	-64.8799	17.7099
Rothschild Francis Vegetable Market	St. Thomas	Virgin Islands	-64.9326	18.3428
Feria Agrícola de Luquillo	Luquillo	Puerto Rico	-65.7207	18.3782
El Mercado Familiar	San Lorenzo	Puerto Rico	-65.9674	18.1871
El Mercado Familiar	Gurabo	Puerto Rico	-65.9786	18.2526
El Mercado Familiar	Patillas	Puerto Rico	-66.0135	18.0069
El Mercado Familiar	Caguas zona urbana	Puerto Rico	-66.039	18.2324
El Maercado Familiar	Arroyo zona urbana	Puerto Rico	-66.0617	17.9686

... (8536 rows omitted)

(The `descending=True` bit is called an *optional argument*. It has a default value of `False`, so when you explicitly tell the function `descending=True`, then the function will sort in descending order.)

`sort`

Some details about sort:

1. The first argument to `sort` is the name of a column to sort by.
2. If the column has text in it, `sort` will sort alphabetically; if the column has numbers, it will sort numerically.
3. The value of `farmers_markets_locations.sort("x")` is a *copy* of `farmers_markets_locations`; the `farmers_markets_locations` table doesn't get modified. For example, if we called `farmers_markets_locations.sort("x")`, then running `farmers_markets_locations` by itself would still return the unsorted table.
4. Rows always stick together when a table is sorted. It wouldn't make sense to sort just one column and leave the other columns alone. For example, in this case, if we sorted just the `x` column, the farmers' markets would all end up with the wrong longitudes.

Question 3.5 Create a version of `farmers_markets_locations` that's sorted by **latitude (`y`)**, with the largest latitudes first. Call it `farmers_markets_locations_by_latitude`.

In [19]:
```
farmers_markets_locations_by_latitude = farmers_markets.sort("y", descending=Tr
farmers_markets_locations_by_latitude
```

Out[19]:

FMID	MarketName	street	city	County	State	zip	x	y
1004890	Tanana Valley Farmers Market	2600 College Road	Fairbanks	Fairbanks North Star	Alaska	99709	-147.781	64.862
1000189	Ester Community Market	Ester Community Park, Old Nenana Highway	Ester	Fairbanks North Star	Alaska	99725	-148.01	64.845
1000199	Fairbanks Downtown Market	1st Avenue	Fairbanks	Fairbanks North Star	Alaska	99701	-147.72	64.844
1006131	Nenana Open Air Market	Corner of Parks Highway & Main Street	Nenana	nan	Alaska	99704	-149.096	64.556
1005848	Highway's End Farmers' Market	Corner of Alaska Highway and Richardson Highway	Delta Junction	Fairbanks North Star	Alaska	99737	-145.733	64.038
1010822	MountainTraders	13440 E Main Street	Talkeetna	Matanuska-Susitna	Alaska	99676	-150.118	62.323
1009661	Talkeetna Farmers Market	13440 E Main Street	Talkeetna	Matanuska-Susitna	Alaska	99676	-150.118	62.322
1001904	Denali Farmers Market	Intersection of the Parks Highway and Susitna River Road	Anchorage	nan	Alaska	nan	-150.234	62.316
1006528	Kenny Lake Harvest II	Corner of Glenn and Richardson Highways	Valdez	nan	Alaska	99686	-145.476	62.107
1000982	Copper Valley Community Market	MM 101 Richardson Hwy Loop Road	Copper Valley	nan	Alaska	99737	-145.444	62.087

... (8536 rows omitted)

In [20]:

```
# TEST
type(farmers_markets_locations_by_latitude) == tables.Table
```

Out[20]:

```
True
```

In [21]:

```
# TEST
list(farmers_markets_locations_by_latitude.column('y').take(range(3))) == [64.8
```

Out[21]: True

Now let's say we want a table of all farmers' markets in California. Sorting won't help us much here because California is closer to the middle of the dataset.

Instead, we use the table method `where`.

In [27]:
```
california_farmers_markets = farmers_markets_locations.where('State', are.equal
california_farmers_markets
```

Out[27]:

MarketName	city	State	x	y
Adelanto Stadium Farmers Market	Victorville	California	-117.405	34.5593
Alameda Farmers' Market	Alameda	California	-122.277	37.7742
Alisal Certified Farmers' Market	Salinas	California	-121.634	36.6733
Altadena Farmers' Market	Altadena	California	-118.158	34.2004
Alum Rock Village Farmers' Market	San Jose	California	-121.833	37.3678
Amador Farmers' Market-- Jackson	Jackson	California	-120.774	38.3488
Amador Farmers' Market-- Pine Grove	Pine Grove	California	-120.774	38.3488
Amador Farmers' Market-- Sutter Creek	Sutter Creek	California	-120.774	38.3488
Anderson Happy Valley Farmers Market	Anderson	California	-122.408	40.4487
Angels Camp Farmers Market-Fresh Fridays	Angels Camp	California	-120.543	38.0722

... (745 rows omitted)

Ignore the syntax for the moment. Instead, try to read that line like this:

> Assign the name **`california_farmers_markets`** to a table whose rows are the rows in the **`farmers_markets_locations`** table **`where`** the **`'State'`** s **`are`** **`equal`** **`to`** **`California`**.

`where`

Now let's dive into the details a bit more. `where` takes 2 arguments:

1. The name of a column. `where` finds rows where that column's values meet some criterion.
2. A predicate that describes the criterion that the column needs to meet.

The predicate in the example above called the function `are.equal_to` with the value we wanted, 'California'. We'll see other predicates soon.

`where` returns a table that's a copy of the original table, but **with only the rows that meet the given predicate**.

Question 3.6. Use `california_farmers_markets` to create a table called `berkeley_markets` containing farmers' markets in Berkeley, California.

In [31]:
```
berkeley_markets = california_farmers_markets.where('city', are.equal_to('Berke
berkeley_markets
```

Out[31]:

MarketName	city	State	x	y
Downtown Berkeley Farmers' Market	Berkeley	California	-122.273	37.8697
North Berkeley Farmers' Market	Berkeley	California	-122.269	37.8802
South Berkeley Farmers' Market	Berkeley	California	-122.272	37.8478

In [32]:
```
# TEST
berkeley_markets.num_rows == 3
```

Out[32]:
```
True
```

In [33]:
```
# TEST
list(berkeley_markets.column('city')) == ['Berkeley', 'Berkeley', 'Berkeley']
```

Out[33]:
```
True
```

So far we've only been using `where` with the predicate that requires finding the values in a column to be *exactly* equal to a certain value. However, there are many other predicates. Here are a few:

Predicate	Example	Result
`are.equal_to`	`are.equal_to(50)`	Find rows with values equal to 50
`are.not_equal_to`	`are.not_equal_to(50)`	Find rows with values not equal to 50
`are.above`	`are.above(50)`	Find rows with values above (and not equal to) 50
`are.above_or_equal_to`	`are.above_or_equal_to(50)`	Find rows with values above 50 or equal to 50
`are.below`	`are.below(50)`	Find rows with values below 50
`are.between`	`are.between(2, 10)`	Find rows with values above or equal to 2 and below 10

4. Arrays

Computers are most useful when you can use a small amount of code to *do the same action* to *many different things*.

For example, in the time it takes you to calculate the 18% tip on a restaurant bill, a laptop can calculate 18% tips for every restaurant bill paid by every human on Earth that day. (That's if you're pretty fast at doing arithmetic in your head!)

Arrays are how we put many values in one place so that we can operate on them as a group. For example, if `billions_of_numbers` is an array of numbers, the expression

```
.18 * billions_of_numbers
```

gives a new array of numbers that contains the result of multiplying **each number** in `billions_of_numbers` by .18. Arrays are not limited to numbers; we can also put all the words in a book into an array of strings.

Concretely, an array is a **collection of values of the same type**.

4.1. Making arrays

First, let's learn how to manually input values into an array. This typically isn't how programs work. Normally, we create arrays by loading them from an external source, like a data file.

To create an array by hand, call the function `make_array`. Each argument you pass to `make_array` will be in the array it returns. Run this cell to see an example:

In [35]:
```
make_array(0.125, 4.75, -1.3)
```

Out[35]:
```
array([ 0.125,  4.75 , -1.3  ])
```

Each value in an array (in the above case, the numbers 0.125, 4.75, and -1.3) is called an *element* of that array.

Arrays themselves are also values, just like numbers and strings. That means you can assign them to names or use them as arguments to functions. For example, `len(<some_array>)` returns the number of elements in `some_array`.

Question 4.1.1. Make an array containing the numbers 0, 1, -1, π, and e, in that order. Name it `interesting_numbers`.

Hint: How did you get the values π and e in lab 2? You can refer to them in exactly the same way here.

In [37]:
```
interesting_numbers = make_array(0,1,-1, math.pi, math.e)
interesting_numbers
```

Out[37]:
```
array([ 0.        ,  1.        , -1.        ,  3.14159265,  2.71828183])
```

In [38]:
```
# TEST
type(interesting_numbers) == np.ndarray
```

Out[38]:
```
True
```

In [39]:
```
# TEST
len(interesting_numbers) == 5
```

Out[39]: True

In [40]:
```python
# TEST
all(interesting_numbers == np.array([0, 1, -1, math.pi, math.e]))
```

Out[40]: True

Question 4.1.2. Make an array containing the five strings `"Hello"`, `","`, `" "`, `"world"`, and `"!"`. (The third one is a single space inside quotes.) Name it `hello_world_components`.

Note: If you evaluate `hello_world_components`, you'll notice some extra information in addition to its contents: `dtype='<U5'`. That's just NumPy's extremely cryptic way of saying that the data types in the array are strings.

In [41]:
```python
hello_world_components = make_array('Hello', ',', ' ', 'world', '!')
hello_world_components
```

Out[41]: array(['Hello', ',', ' ', 'world', '!'], dtype='<U5')

In [42]:
```python
# TEST
type(hello_world_components) == np.ndarray
```

Out[42]: True

In [43]:
```python
# TEST
len(interesting_numbers) == 5
```

Out[43]: True

In [44]:
```python
# TEST
all(hello_world_components == np.array(["Hello", ",", " ", "world", "!"]))
```

Out[44]: True

`np.arange`

Arrays are provided by a package called NumPy (pronounced "NUM-pie"). The package is called `numpy`, but it's standard to rename it `np` for brevity. You can do that with:

```
import numpy as np
```

Very often in data science, we want to work with many numbers that are evenly spaced within some range. NumPy provides a special function for this called `arange`. The line of code `np.arange(start, stop, step)` evaluates to an array with all the numbers starting at `start` and counting up by `step`, stopping **before** `stop` is reached.

Run the following cells to see some examples!

In [45]:
```
# This array starts at 1 and counts up by 2
# and then stops before 6
np.arange(1, 6, 2)
```
Out[45]:
```
array([1, 3, 5])
```

In [46]:
```
# This array doesn't contain 9
# because np.arange stops *before* the stop value is reached
np.arange(4, 9, 1)
```
Out[46]:
```
array([4, 5, 6, 7, 8])
```

Question 4.1.3. Import `numpy` as `np` and then use `np.arange` to create an array with the multiples of 99 from 0 up to (**and including**) 9999. (So its elements are 0, 99, 198, 297, etc.)

In [54]:
```
import numpy as np
multiples_of_99 = np.arange(0, 10000, 99)
multiples_of_99
```
Out[54]:
```
array([   0,   99,  198,  297,  396,  495,  594,  693,  792,  891,  990,
       1089, 1188, 1287, 1386, 1485, 1584, 1683, 1782, 1881, 1980, 2079,
       2178, 2277, 2376, 2475, 2574, 2673, 2772, 2871, 2970, 3069, 3168,
       3267, 3366, 3465, 3564, 3663, 3762, 3861, 3960, 4059, 4158, 4257,
       4356, 4455, 4554, 4653, 4752, 4851, 4950, 5049, 5148, 5247, 5346,
       5445, 5544, 5643, 5742, 5841, 5940, 6039, 6138, 6237, 6336, 6435,
       6534, 6633, 6732, 6831, 6930, 7029, 7128, 7227, 7326, 7425, 7524,
       7623, 7722, 7821, 7920, 8019, 8118, 8217, 8316, 8415, 8514, 8613,
       8712, 8811, 8910, 9009, 9108, 9207, 9306, 9405, 9504, 9603, 9702,
       9801, 9900, 9999])
```

In [55]:
```
# TEST
type(multiples_of_99) == np.ndarray
```
Out[55]:
```
True
```

In [56]:
```
# TEST
len(multiples_of_99) == 102
```
Out[56]:
```
True
```

In [57]:
```
# TEST
all(multiples_of_99 == np.arange(0, 9999+99, 99))
```
Out[57]:
```
True
```

4.2. Working with single elements of arrays ("indexing")

Let's work with a more interesting dataset. The next cell creates an array called `population_amounts` that includes estimated world populations in every year from **1950** to roughly the present. (The estimates come from the US Census Bureau website.)

Rather than type in the data manually, we've loaded them from a file on your computer called `world_population.csv`.

In [58]:
```
population_amounts = Table.read_table("world_population.csv").column("Populatic
population_amounts
```

Out[58]:
```
array([2557628654, 2594939877, 2636772306, 2682053389, 2730228104,
       2782098943, 2835299673, 2891349717, 2948137248, 3000716593,
       3043001508, 3083966929, 3140093217, 3209827882, 3281201306,
       3350425793, 3420677923, 3490333715, 3562313822, 3637159050,
       3712697742, 3790326948, 3866568653, 3942096442, 4016608813,
       4089083233, 4160185010, 4232084578, 4304105753, 4379013942,
       4451362735, 4534410125, 4614566561, 4695736743, 4774569391,
       4856462699, 4940571232, 5027200492, 5114557167, 5201440110,
       5288955934, 5371585922, 5456136278, 5538268316, 5618682132,
       5699202985, 5779440593, 5857972543, 5935213248, 6012074922,
       6088571383, 6165219247, 6242016348, 6318590956, 6395699509,
       6473044732, 6551263534, 6629913759, 6709049780, 6788214394,
       6866332358, 6944055583, 7022349283, 7101027895, 7178722893,
       7256490011])
```

Here's how we get the first element of `population_amounts`, which is the world population in the first year in the dataset, 1950.

In [59]:
```
population_amounts.item(0)
```

Out[59]:
```
2557628654
```

The value of that expression is the number 2557628654 (around 2.5 billion), because that's the first thing in the array `population_amounts`.

Notice that we wrote `.item(0)`, not `.item(1)`, to get the first element. This is a weird convention in computer science. 0 is called the *index* of the first item. It's the number of elements that appear *before* that item. So 3 is the index of the 4th item.

Here are some more examples. In the examples, we've given names to the things we get out of `population_amounts`. Read and run each cell.

In [60]:
```
# The 13th element in the array is the population
# in 1962 (which is 1950 + 12).
population_1962 = population_amounts.item(12)
population_1962
```

Out[60]:
```
3140093217
```

In [61]:
```
# The 66th element is the population in 2015.
population_2015 = population_amounts.item(65)
population_2015
```

Out[61]:
```
7256490011
```

In [62]:
```
# The array has only 66 elements, so this doesn't work.
# (There's no element with 66 other elements before it.)
```

```
population_2016 = population_amounts.item(66)
population_2016
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [62], in <cell line: 3>()
      1 # The array has only 66 elements, so this doesn't work.
      2 # (There's no element with 66 other elements before it.)
----> 3 population_2016 = population_amounts.item(66)
      4 population_2016

IndexError: index 66 is out of bounds for axis 0 with size 66
```

Since `make_array` returns an array, we can call `.item(3)` on its output to get its 4th element, just like we "chained" together calls to the method `replace` earlier.

In [63]:
```
make_array(-1, -3, 4, -2).item(3)
```

Out[63]:
```
-2
```

Question 4.2.1. Set `population_1973` to the world population in 1973, by getting the appropriate element from `population_amounts` using `item`.

In [64]:
```
population_1973 = population_amounts.item(23)
population_1973
```

Out[64]:
```
3942096442
```

In [65]:
```
# TEST
population_1973 == 3942096442
```

Out[65]:
```
True
```

4.3. Doing something to every element of an array

Arrays are primarily useful for doing the same operation many times, so we don't often have to use `.item` and work with single elements.

Logarithms

Here is one simple question we might ask about world population:

> How big was the population in *orders of magnitude* in each year?

Orders of magnitude quantify how big a number is by representing it as the power of another number (for example, representing 104 as $10^{2.017033}$). One way to do this is by using the logarithm function. The logarithm (base 10) of a number increases by 1 every time we multiply the number by 10. It's like a measure of how many decimal digits the number has, or how big it is in orders of magnitude.

We could try to answer our question like this, using the `log10` function from the `math` module and the `item` method you just saw:

In [66]:
```
population_1950_magnitude = math.log10(population_amounts.item(0))
population_1951_magnitude = math.log10(population_amounts.item(1))
population_1952_magnitude = math.log10(population_amounts.item(2))
population_1953_magnitude = math.log10(population_amounts.item(3))
...
```

Out[66]:
```
Ellipsis
```

But this is tedious and doesn't really take advantage of the fact that we are using a computer.

Instead, NumPy provides its own version of `log10` that takes the logarithm of each element of an array. It takes a single array of numbers as its argument. It returns an array of the same length, where the first element of the result is the logarithm of the first element of the argument, and so on.

Question 4.3.1. Use `np.log10` to compute the logarithms of the world population in every year. Give the result (an array of 66 numbers) the name `population_magnitudes`. Your code should be very short.

In [70]:
```
population_magnitudes = np.log10(population_amounts)
population_magnitudes
```

Out[70]:
```
array([9.40783749, 9.4141273 , 9.42107263, 9.42846742, 9.43619893,
       9.44437257, 9.45259897, 9.46110062, 9.4695477 , 9.47722498,
       9.48330217, 9.48910971, 9.49694254, 9.50648175, 9.51603288,
       9.5251    , 9.53411218, 9.54286695, 9.55173218, 9.56076229,
       9.56968959, 9.57867667, 9.58732573, 9.59572724, 9.60385954,
       9.61162595, 9.61911264, 9.62655434, 9.63388293, 9.64137633,
       9.64849299, 9.6565208 , 9.66413091, 9.67170374, 9.67893421,
       9.68632006, 9.69377717, 9.70132621, 9.70880804, 9.7161236 ,
       9.72336995, 9.73010253, 9.73688521, 9.74337399, 9.74963446,
       9.75581413, 9.7618858 , 9.76774733, 9.77343633, 9.77902438,
       9.7845154 , 9.78994853, 9.7953249 , 9.80062024, 9.80588805,
       9.81110861, 9.81632507, 9.82150788, 9.82666101, 9.83175555,
       9.83672482, 9.84161319, 9.84648243, 9.85132122, 9.85604719,
       9.8607266 ])
```

In [71]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
type(population_magnitudes) == np.ndarray
```

Out[71]:
```
True
```

In [72]:
```
# TEST
# You made an array, but it doesn't have the right numbers in it.
sum(abs(population_magnitudes - np.log10(population_amounts))) < 1e-6
```

Out[72]:
```
True
```

What you just did is called **elementwise** application of `np.log10`, since `np.log10` operates separately on each element of the array that it's called on. Here's a picture of what's going on:

The textbook's section on arrays has a useful list of NumPy functions that are designed to work elementwise, like `np.log10`.

Arithmetic

Arithmetic also works elementwise on arrays, meaning that if you perform an arithmetic operation (like subtraction, division, etc) on an array, Python will do the operation to every element of the array individually and return an array of all of the results. For example, you can divide all the population numbers by 1 billion to get numbers in billions:

In [73]:
```
population_in_billions = population_amounts / 1000000000
population_in_billions
```

Out[73]:
```
array([2.55762865, 2.59493988, 2.63677231, 2.68205339, 2.7302281 ,
       2.78209894, 2.83529967, 2.89134972, 2.94813725, 3.00071659,
       3.04300151, 3.08396693, 3.14009322, 3.20982788, 3.28120131,
       3.35042579, 3.42067792, 3.49033371, 3.56231382, 3.63715905,
       3.71269774, 3.79032695, 3.86656865, 3.94209644, 4.01660881,
       4.08908323, 4.16018501, 4.23208458, 4.30410575, 4.37901394,
       4.45136274, 4.53441012, 4.61456656, 4.69573674, 4.77456939,
       4.8564627 , 4.94057123, 5.02720049, 5.11455717, 5.20144011,
       5.28895593, 5.37158592, 5.45613628, 5.53826832, 5.61868213,
       5.69920299, 5.77944059, 5.85797254, 5.93521325, 6.01207492,
       6.08857138, 6.16521925, 6.24201635, 6.31859096, 6.39569951,
       6.47304473, 6.55126353, 6.62991376, 6.70904978, 6.78821439,
       6.86633236, 6.94405558, 7.02234928, 7.10102789, 7.17872289,
       7.25649001])
```

You can do the same with addition, subtraction, multiplication, and exponentiation (`**`). For example, you can calculate a tip on several restaurant bills at once (in this case just 3):

In [74]:
```
restaurant_bills = make_array(20.12, 39.90, 31.01)
print("Restaurant bills:\t", restaurant_bills)

# Array multiplication
tips = .2 * restaurant_bills
print("Tips:\t\t\t", tips)
```

```
Restaurant bills:        [20.12 39.9  31.01]
Tips:                    [4.024 7.98  6.202]
```

Question 4.3.2. Suppose the total charge at a restaurant is the original bill plus the tip. If the tip is 20%, that means we can multiply the original bill by 1.2 to get the total charge. Compute the total charge for each bill in `restaurant_bills`, and assign the resulting array to `total_charges`.

In [75]:
```
total_charges = restaurant_bills * 1.2
total_charges
```

Out[75]:
```
array([24.144, 47.88 , 37.212])
```

In [76]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
type(total_charges) == np.ndarray
```

Out[76]:
```
True
```

Question 4.3.3. The array `more_restaurant_bills` contains 100,000 bills! Compute the total charge for each one. How is your code different?

In [77]:
```
more_restaurant_bills = Table.read_table("more_restaurant_bills.csv").column("E
more_total_charges = more_restaurant_bills * 1.2
more_total_charges
```

Out[77]:
```
array([20.244, 20.892, 12.216, ..., 19.308, 18.336, 35.664])
```

In [78]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
type(more_total_charges) == np.ndarray
```

Out[78]:
```
True
```

In [79]:
```
# TEST
# You made an array, but it doesn't have the right numbers in it.
sum(abs(more_total_charges - 1.2 * more_restaurant_bills)) < 1e-6
```

Out[79]:
```
True
```

The function `sum` takes a single array of numbers as its argument. It returns the sum of all the numbers in that array (so it returns a single number, not an array).

Question 4.3.4. What was the sum of all the bills in `more_restaurant_bills`, *including tips*?

In [81]:
```
sum_of_bills = sum(more_restaurant_bills * 1.2)
sum_of_bills
```

Out[81]:
```
1795730.0640000193
```

In [82]:
```
# TEST
round(sum_of_bills, 2) == 1795730.06
```

Out[82]:
```
True
```

Question 4.3.5. The powers of 2 ($2^0 = 1$, $2^1 = 2$, $2^2 = 4$, etc) arise frequently in computer science. (For example, you may have noticed that storage on smartphones or USBs come in powers of 2, like 16 GB, 32 GB, or 64 GB.) Use `np.arange` and the exponentiation operator `**` to compute the first 30 powers of 2, starting from `2^0`.

Hint 1: `np.arange(1, 2**30, 1)` creates an array with 2^{30} elements and **will crash your kernel**.

Hint 2: Part of your solution will involve `np.arange`, but your array shouldn't have more than 30 elements.

In [84]:
```
powers_of_2 = 2 ** np.arange(30)
powers_of_2
```

Out[84]:
```
array([        1,         2,         4,         8,        16,        32,
              64,       128,       256,       512,      1024,      2048,
            4096,      8192,     16384,     32768,     65536,    131072,
          262144,    524288,   1048576,   2097152,   4194304,   8388608,
        16777216,  33554432,  67108864, 134217728, 268435456, 536870912])
```

In [85]:
```
# TEST
all(powers_of_2 == 2 ** np.arange(30))
```

Out[85]:
```
True
```

5. Creating Tables

An array is useful for describing a single attribute of each element in a collection. For example, let's say our collection is all US States. Then an array could describe the land area of each state.

Tables extend this idea by containing multiple arrays, each one describing a different attribute for every element of a collection. In this way, tables allow us to not only store data about many entities but to also contain several kinds of data about each entity.

For example, in the cell below we have two arrays. The first one, `population_amounts`, was defined above in section 4.2 and contains the world population in each year (estimated by the US Census Bureau). The second array, `years`, contains the years themselves. These elements are in order, so the year and the world population for that year have the same index in their corresponding arrays.

In [86]:
```
# Just run this cell

years = np.arange(1950, 2015+1)
print("Population column:", population_amounts)
print("Years column:", years)
```

```
Population column: [2557628654 2594939877 2636772306 2682053389 2730228104 278
2098943
 2835299673 2891349717 2948137248 3000716593 3043001508 3083966929
 3140093217 3209827882 3281201306 3350425793 3420677923 3490333715
 3562313822 3637159050 3712697742 3790326948 3866568653 3942096442
 4016608813 4089083233 4160185010 4232084578 4304105753 4379013942
 4451362735 4534410125 4614566561 4695736743 4774569391 4856462699
 4940571232 5027200492 5114557167 5201440110 5288955934 5371585922
 5456136278 5538268316 5618682132 5699202985 5779440593 5857972543
 5935213248 6012074922 6088571383 6165219247 6242016348 6318590956
 6395699509 6473044732 6551263534 6629913759 6709049780 6788214394
 6866332358 6944055583 7022349283 7101027895 7178722893 7256490011]
Years column: [1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 196
2 1963
 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977
 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991
 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015]
```

Suppose we want to answer this question:

> In which year did the world's population cross 6 billion?

You could technically answer this question just from staring at the arrays, but it's a bit convoluted, since you would have to count the position where the population first crossed 6 billion, then find the corresponding element in the years array. In cases like these, it might be easier to put the data into a *`Table`*, a 2-dimensional type of dataset.

The expression below:

- creates an empty table using the expression `Table()`,
- adds two columns by calling `with_columns` with four arguments,
- assigns the result to the name `population`, and finally
- evaluates `population` so that we can see the table.

The strings `"Year"` and `"Population"` are column labels that we have chosen. The names `population_amounts` and `years` were assigned above to two arrays of the **same length**. The function `with_columns` (you can find the documentation here) takes in alternating strings (to represent column labels) and arrays (representing the data in those columns). The strings and arrays are separated by commas.

In [87]:
```
population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

Out[87]:

Population	Year
2557628654	1950
2594939877	1951
2636772306	1952
2682053389	1953
2730228104	1954
2782098943	1955
2835299673	1956
2891349717	1957
2948137248	1958
3000716593	1959

... (56 rows omitted)

Now the data is combined into a single table! It's much easier to parse this data. If you need to know what the population was in 1959, for example, you can tell from a single glance.

Question 5.1. In the cell below, we've created 2 arrays. Using the steps above, assign `top_10_movies` to a table that has two columns called "Rating" and "Name", which hold `top_10_movie_ratings` and `top_10_movie_names` respectively.

In [88]:
```
top_10_movie_ratings = make_array(9.2, 9.2, 9., 8.9, 8.9, 8.9, 8.9, 8.9, 8.9, 8
top_10_movie_names = make_array(
        'The Shawshank Redemption (1994)',
        'The Godfather (1972)',
        'The Godfather: Part II (1974)',
        'Pulp Fiction (1994)',
        "Schindler's List (1993)",
        'The Lord of the Rings: The Return of the King (2003)',
        '12 Angry Men (1957)',
        'The Dark Knight (2008)',
        'Il buono, il brutto, il cattivo (1966)',
        'The Lord of the Rings: The Fellowship of the Ring (2001)')

top_10_movies = Table().with_columns(
    "Rating", top_10_movie_ratings,
    "Name", top_10_movie_names
)

# We've put this next line here
# so your table will get printed out
# when you run this cell.
top_10_movies
```

Out[88]:

Rating	Name
9.2	The Shawshank Redemption (1994)
9.2	The Godfather (1972)
9	The Godfather: Part II (1974)
8.9	Pulp Fiction (1994)
8.9	Schindler's List (1993)
8.9	The Lord of the Rings: The Return of the King (2003)
8.9	12 Angry Men (1957)
8.9	The Dark Knight (2008)
8.9	Il buono, il brutto, il cattivo (1966)
8.8	The Lord of the Rings: The Fellowship of the Ring (2001)

In [89]:

```
# TEST
type(top_10_movies) == tables.Table
```

Out[89]:

```
True
```

In [90]:

```
# TEST
top_10_movies.select('Rating', 'Name').sort('Name')
```

Out[90]:

Rating	Name
8.9	12 Angry Men (1957)
8.9	Il buono, il brutto, il cattivo (1966)
8.9	Pulp Fiction (1994)
8.9	Schindler's List (1993)
8.9	The Dark Knight (2008)
9.2	The Godfather (1972)
9	The Godfather: Part II (1974)
8.8	The Lord of the Rings: The Fellowship of the Ring (2001)
8.9	The Lord of the Rings: The Return of the King (2003)
9.2	The Shawshank Redemption (1994)

Loading a table from a file

In most cases, we aren't going to go through the trouble of typing in all the data manually. Instead, we load them in from an external source, like a data file. There are many formats for data files, but CSV ("comma-separated values") is the most common.

`Table.read_table(...)` takes one argument (a path to a data file in **string** format) and returns a table.

Question 5.2. `imdb.csv` contains a table of information about the 250 highest-rated movies on IMDb. Load it as a table called `imdb`.

In [91]:
```
imdb = Table.read_table('imdb.csv')
imdb
```

Out[91]:

Votes	Rating	Title	Year	Decade
88355	8.4	M	1931	1930
132823	8.3	Singin' in the Rain	1952	1950
74178	8.3	All About Eve	1950	1950
635139	8.6	Léon	1994	1990
145514	8.2	The Elephant Man	1980	1980
425461	8.3	Full Metal Jacket	1987	1980
441174	8.1	Gone Girl	2014	2010
850601	8.3	Batman Begins	2005	2000
37664	8.2	Judgment at Nuremberg	1961	1960
46987	8	Relatos salvajes	2014	2010

... (240 rows omitted)

In [92]:
```
# TEST
type(imdb) == tables.Table
```

Out[92]:
```
True
```

In [93]:
```
# TEST
imdb.num_rows == 250
```

Out[93]:
```
True
```

In [94]:
```
# TEST
imdb.select('Votes', 'Rating', 'Title', 'Year', 'Decade').sort(0).take(range(2,
```

Out[94]:

Votes	Rating	Title	Year	Decade
31003	8.1	Le salaire de la peur	1953	1950
32385	8	La battaglia di Algeri	1966	1960
35983	8.1	The Best Years of Our Lives	1946	1940

6. More Table Operations!

Now that you've worked with arrays, let's add a few more methods to the list of table operations.

`column`

`column` takes the column name of a table (in string format) as its argument and returns the values in that column as an **array**.

In [95]:
```
# Returns an array of movie names
top_10_movies.column('Name')
```

Out[95]:
```
array(['The Shawshank Redemption (1994)', 'The Godfather (1972)',
       'The Godfather: Part II (1974)', 'Pulp Fiction (1994)',
       "Schindler's List (1993)",
       'The Lord of the Rings: The Return of the King (2003)',
       '12 Angry Men (1957)', 'The Dark Knight (2008)',
       'Il buono, il brutto, il cattivo (1966)',
       'The Lord of the Rings: The Fellowship of the Ring (2001)'],
      dtype='<U56')
```

`take`

The table method `take` takes as its argument an array of numbers. Each number should be the index of a row in the table. It returns a **new table** with only those rows.

You'll usually want to use `take` in conjunction with `np.arange` to take the first few rows of a table.

In [96]:
```
# Take first 5 movies of top_10_movies
top_10_movies.take(np.arange(0, 5, 1))
```

Out[96]:

Rating	Name
9.2	The Shawshank Redemption (1994)
9.2	The Godfather (1972)
9	The Godfather: Part II (1974)
8.9	Pulp Fiction (1994)
8.9	Schindler's List (1993)

You can find more table operations in the documentation for datascience.tables.

You're done with Lab 2! Don't forget to choose **print** to save it as PDF as well. Submit both the notebook and the PDF to Canvas.