Lab 5: Simulations

Welcome to Lab 5!

We will go over iteration and simulations, as well as introduce the concept of randomness.

The data used in this lab will contain salary data and other statistics for basketball players from the 2014-2015 NBA season. This data was collected from the following sports analytic sites: Basketball Reference and Spotrac.

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

```
# Run this cell, but please don't change it.

# These lines import the Numpy and Datascience modules.
import numpy as np
from datascience import *

# These lines do some fancy plotting magic
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

1. Nachos and Conditionals

In Python, the boolean data type contains only two unique values: `True` and `False`. Expressions containing comparison operators such as `<` (less than), `>` (greater than), and `==` (equal to) evaluate to Boolean values. A list of common comparison operators can be found below:

< , > less than, greater than

<= , >= less than or equal to, greater than or equal to

== equal

!= not equal

Run the cell below to see an example of a comparison operator in action.

```
3 > 1 + 1
```

```
True
```

We can even assign the result of a comparison operation to a variable.

```
result = 10 / 2 == 5
result
```

```
True
```

Arrays are compatible with comparison operators. The output is an array of boolean values.

```
make_array(1, 5, 7, 8, 3, -1) > 3
```

```
array([False,  True,  True,  True, False, False], dtype=bool)
```

One day, when you come home after a long week, you see a hot bowl of nachos waiting on the dining table! Let's say that whenever you take a nacho from the bowl, it will either have only **cheese**, only **salsa**, **both** cheese and salsa, or **neither** cheese nor salsa (a sad tortilla chip indeed).

Let's try and simulate taking nachos from the bowl at random using the function, `np.random.choice(...)`.

`np.random.choice`

`np.random.choice` picks one item at random from the given array. It is equally likely to pick any of the items. Run the cell below several times, and observe how the results change.

```
nachos = make_array('cheese', 'salsa', 'both', 'neither')
np.random.choice(nachos)
```

```
'both'
```

To repeat this process multiple times, pass in an int `n` as the second argument to return `n` different random choices. By default, `np.random.choice` samples **with replacement** and returns an *array* of items.

Run the next cell to see an example of sampling with replacement 10 times from the `nachos` array.

```
np.random.choice(nachos, 10)
```

```
array(['cheese', 'both', 'salsa', 'both', 'both', 'neither', 'neither',
       'both', 'neither', 'both'],
      dtype='<U7')
```

To count the number of times a certain type of nacho is randomly chosen, we can use `np.count_nonzero`

`np.count_nonzero`

`np.count_nonzero` counts the number of non-zero values that appear in an array. When an array of boolean values are passed through the function, it will count the number of `True` values (remember that in Python, `True` is coded as 1 and `False` is coded as 0.)

Run the next cell to see an example that uses `np.count_nonzero`.

```
np.count_nonzero(make_array(True, False, False, True, True))
```

```
3
```

Question 1. Assume we took ten nachos at random, and stored the results in an array called `ten_nachos` as done below. Find the number of nachos with only cheese using code (do not hardcode the answer).

Hint: Our solution involves a comparison operator (e.g. `==`, `<`, ...) and the `np.count_nonzero` method.

```
ten_nachos = make_array('neither', 'cheese', 'both', 'both', 'cheese', 'salsa', 'both', 'neither', 'cheese', 'both')
number_cheese = np.count_nonzero(ten_nachos == 'cheese')
number_cheese
```

```
3
```

Conditional Statements

A conditional statement is a multi-line statement that allows Python to choose among different alternatives based on the truth value of an expression.

Here is a basic example.

```
def sign(x):
    if x > 0:
        return 'Positive'
    else:
        return 'Negative'
```

If the input `x` is greater than `0`, we return the string `'Positive'`. Otherwise, we return `'Negative'`.

If we want to test multiple conditions at once, we use the following general format.

```
if <if expression>:
    <if body>
elif <elif expression 0>:
    <elif body 0>
elif <elif expression 1>:
    <elif body 1>
...
```

```
else:
    <else body>
```

Only the body for the first conditional expression that is true will be evaluated. Each `if` and `elif` expression is evaluated and considered in order, starting at the top. As soon as a true value is found, the corresponding body is executed, and the rest of the conditional statement is skipped. If none of the `if` or `elif` expressions are true, then the `else body` is executed.

For more examples and explanation, refer to the section on conditional statements here.

Question 2. Write a function called `nacho_reaction` that returns a reaction (as a string) based on the type of nacho passed in as an argument. Use the Nacho Types and Reactions below to match the nacho type to the appropriate reaction.

Nacho Type: cheese -- Reaction: Cheesy!

Nacho Type: salsa -- Reaction: Saucy!

Nacho Type: neither -- Reaction: Meh

Nacho Type: both -- Reaction: Wow!

```
def nacho_reaction(nacho):
    if nacho == "cheese":
        return "Cheesy!"
    if nacho == "salsa":
        return "Saucy!"
    if nacho == "neither":
        return "Meh"
    if nacho == "both":
        return "Wow!"

  #i know i didnt use elif- the return statements act as a guard clause and make it not matter!

spicy_nacho = nacho_reaction('neither')
spicy_nacho
```

```
'Meh'
```

Question 3. Create a table `ten_nachos_reactions` that consists of the nachos in `ten_nachos` as well as the reactions for each of those nachos. The columns should be called `Nachos` and `Reactions`.

Hint: Use the `apply` method.

```
ten_nachos_tbl = Table().with_column('Nachos', ten_nachos)
ten_nachos_reactions = ten_nachos_tbl.apply(nacho_reaction, "Nachos")
ten_nachos_reactions
```

```
array(['Meh', 'Cheesy!', 'Wow!', 'Wow!', 'Cheesy!', 'Saucy!', 'Wow!',
       'Meh', 'Cheesy!', 'Wow!'],
      dtype='<U7')
```

Question 4. Using code, find the number of 'Wow!' reactions for the nachos in `ten_nachos_reactions`.

```
number_wow_reactions = np.count_nonzero(ten_nachos_reactions == 'Wow!')
number_wow_reactions
```

```
4
```

2. Simulations and For Loops

Using a `for` statement, we can perform a task multiple times. This is known as iteration.

One use of iteration is to loop through a set of values. For instance, we can print out all of the colors of the rainbow.

```
rainbow = make_array("red", "orange", "yellow", "green", "blue", "indigo", "violet")

for color in rainbow:
    print(color)
```

```
red
orange
yellow
green
blue
indigo
violet
```

We can see that the indented part of the `for` loop, known as the body, is executed once for each item in `rainbow`. The name `color` is assigned to the next value in `rainbow` at the start of each iteration. Note that the name `color` is arbitrary; we could easily have named it something else. The important thing is we stay consistent throughout the `for` loop.

```
for another_name in rainbow:
    print(another_name)
```

```
red
orange
yellow
green
blue
indigo
violet
```

In general, however, we would like the variable name to be somewhat informative.

Question 1. In the following cell, we've loaded the text of *Pride and Prejudice* by Jane Austen, split it into individual words, and stored these words in an array `p_and_p_words`. Using a `for` loop, assign `longer_than_five` to the number of words in the novel that are more than 5 letters long.

Hint: You can find the number of letters in a word with the `len` function. You can make the "counter variable" longer_than_five start at 0, then make a for loop that adds 1 each time a word in `p_and_p_words` is longer than 5 letters long.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
austen_string = open('/content/drive/MyDrive/Austen_PrideAndPrejudice.txt', encoding='utf-8').read()
p_and_p_words = np.array(austen_string.split())

longer_than_five = 0

for x in p_and_p_words:
  if(len(x)>=5):
    longer_than_five += 1

longer_than_five
```

```
48032
```

Question 2. Using a simulation with 10,000 trials, assign num_different to the number of times, in 10,000 trials, that two words picked uniformly at random (with replacement) from Pride and Prejudice have different lengths.

Hint 1: What function did we use in section 1 to sample at random with replacement from an array?

Hint 2: Remember that `!=` checks for non-equality between two items.

```
trials = 10000
num_different = 0

for x in range(trials):
  if(len(np.random.choice(p_and_p_words))!=len(np.random.choice(p_and_p_words))):
    num_different += 1;
num_different
```

```
8656
```

We can also use `np.random.choice` to simulate multiple trials.

Question 3. Allie is playing darts. Her dartboard contains ten equal-sized zones with point values from 1 to 10. Write code that simulates her total score after 1000 dart tosses.

Hint: First decide the possible values you can take in the experiment (point values in this case). Then use `np.random.choice` to simulate Allie's tosses. Finally, sum up the scores to get Allie's total score.

```
possible_point_values = np.arange(0,11,1)
num_tosses = 1000
simulated_tosses = np.random.choice(possible_point_values)
total_score = 0
for x in range(num_tosses):
  total_score += np.random.choice(possible_point_values)
total_score
```

```
4930
```

3. Sampling Basketball Data

We will now introduce the topic of sampling, which we'll be discussing in more depth in this week's lectures. This code will be a gentle walkthrough, but if you wish to read more about different kinds of samples before attempting this question, you can check out section 10 of the textbook.

Run the cell below to load player and salary data that we will use for our sampling.

```
player_data = Table().read_table("/content/drive/MyDrive/player_data.csv")
salary_data = Table().read_table("/content/drive/MyDrive/salary_data.csv")
full_data = salary_data.join("PlayerName", player_data, "Name")

# The show method immediately displays the contents of a table.
# This way, we can display the top of two tables using a single cell.
player_data.show(3)
salary_data.show(3)
full_data.show(3)
```

Name	Age	Team	Games	Rebounds	Assists	Steals	Blocks	Turnovers	Points
James Harden	25	HOU	81	459	565	154	60	321	2217
Chris Paul	29	LAC	82	376	838	156	15	190	1564
Stephen Curry	26	GSW	80	341	619	163	16	249	1900

... (489 rows omitted)

PlayerName	Salary
Kobe Bryant	23500000
Amar'e Stoudemire	23410988
Joe Johnson	23180790

... (489 rows omitted)

PlayerName	Salary	Age	Team	Games	Rebounds	Assists	Steals	Blocks	Turnovers	Poi
A.J. Price	62552	28	TOT	26	32	46	7	0	14	
Aaron Brooks	1145685	30	CHI	82	166	261	54	15	157	
Aaron Gordon	3992040	19	ORL	47	169	33	21	22	38	

Rather than getting data on every player (as in the tables loaded above), imagine that we had gotten data on only a smaller subset of the players. For 492 players, it's not so unreasonable to expect to see all the data, but usually we aren't so lucky.

If we want to make estimates about a certain numerical property of the population (known as a statistic, e.g. the mean or median), we may have to come up with these estimates based only on a smaller sample. Whether these estimates are useful or not often depends on how the sample was gathered. We have prepared some example sample datasets to see how they compare to the full NBA dataset. Later we'll ask you to create your own samples to see how they behave.

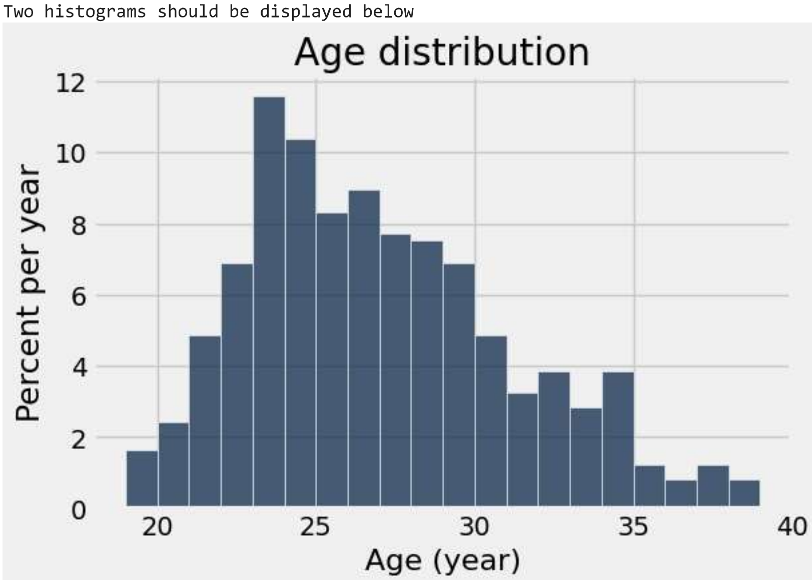
To save typing and increase the clarity of your code, we will package the analysis code into a few functions. This will be useful in the rest of the lab as we will repeatedly need to create histograms and collect summary statistics from that data.

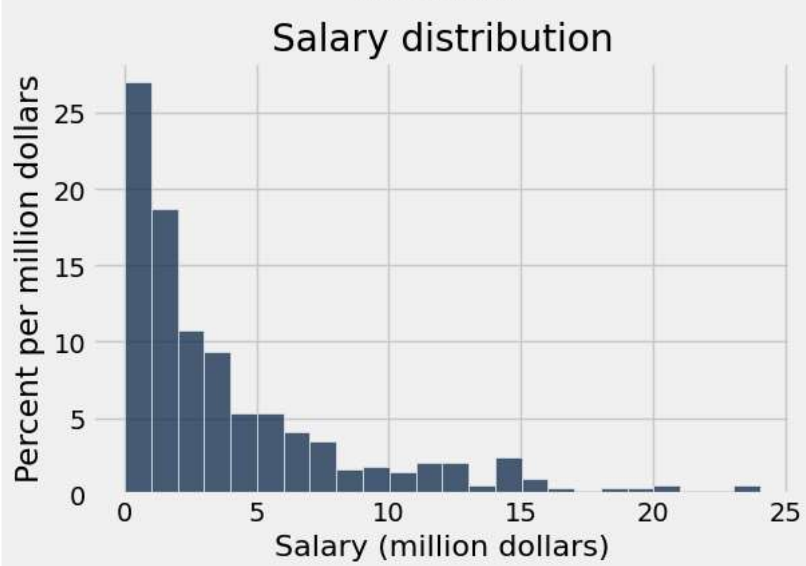
We've defined the `histograms` function below, which takes a table with columns `Age` and `Salary` and draws a histogram for each one. It uses bin widths of 1 year for `Age` and $1,000,000 for `Salary`.

```
def histograms(t):
    ages = t.column('Age')
    salaries = t.column('Salary')/1000000
    t1 = t.drop('Salary').with_column('Salary', salaries)
    age_bins = np.arange(min(ages), max(ages) + 2, 1)
    salary_bins = np.arange(min(salaries), max(salaries) + 1, 1)
    t1.hist('Age', bins=age_bins, unit='year')
    plt.title('Age distribution')
    t1.hist('Salary', bins=salary_bins, unit='million dollars')
    plt.title('Salary distribution')

histograms(full_data)
print('Two histograms should be displayed below')
```

```
Two histograms should be displayed below
```





Question 1. Create a function called `compute_statistics` that takes a table containing ages and salaries and:

- Draws a histogram of ages
- Draws a histogram of salaries
- Returns a two-element array containing the average age and average salary (in that order)

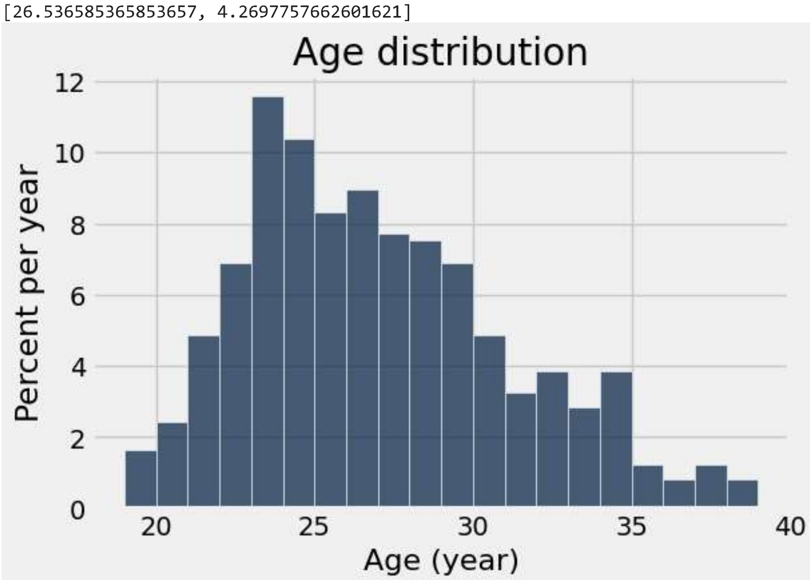
You can call the `histograms` function to draw the histograms!

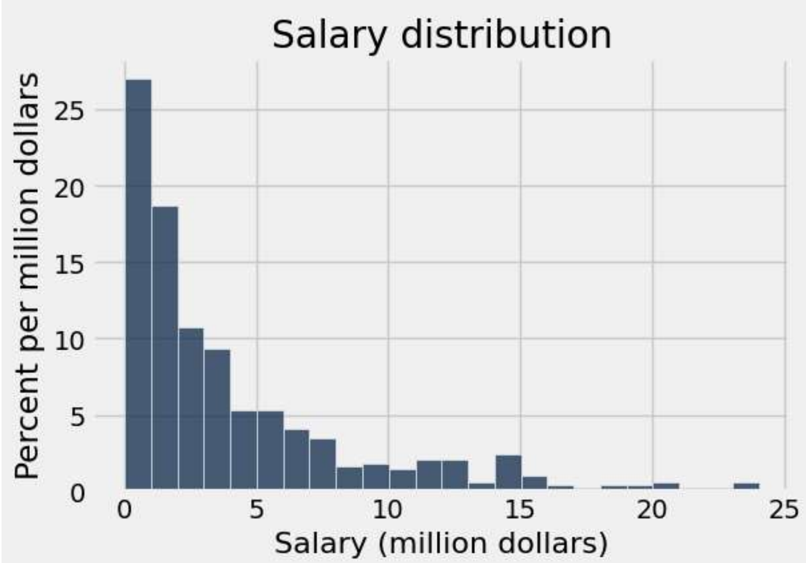
Note: More charts will be displayed when running the test cell. Please feel free to ignore the charts.

```
def compute_statistics(age_and_salary_data):
    age = np.average(age_and_salary_data.column('Age'))
    salary = np.average(age_and_salary_data.column('Salary')/1000000)
    histograms(age_and_salary_data)
    return [age,salary]
```

```
full_stats = compute_statistics(full_data)
full_stats
```

```
[26.536585365853657, 4.2697757662601621]
```





Convenience sampling

One sampling methodology, which is **generally a bad idea**, is to choose players who are somehow convenient to sample. For example, you might choose players from one team who are near your house, since it's easier to survey them. This is called, somewhat pejoratively, *convenience sampling*.

Suppose you survey only *relatively new* players with ages less than 22. (The more experienced players didn't bother to answer your surveys about their salaries.)

Question 2. Assign `convenience_sample` to a subset of `full_data` that contains only the rows for players under the age of 22.

```
convenience_sample = full_data.where("Age", are.below_or_equal_to(22))
convenience_sample.show()
```

Clarkson	507336	22	LAL	59	191	200	31	12	90
Julius Randle	2997360	20	LAL	1	0	0	0	0	1
Jusuf Nurkic	1762680	20	DEN	62	382	50	52	68	86
K.J. McDaniels	507336	21	TOT	62	200	72	44	70	105
Kentavious Caldwell-Pope	2772480	21	DET	82	255	109	93	18	94
Kyle Anderson	1093680	21	SAS	33	72	28	15	7	10
Kyrie Irving	7070730	22	CLE	75	237	389	114	20	186
Lucas Nogueira	1762680	22	TOR	6	11	1	2	0	2
Marcus Smart	3283320	20	BOS	67	222	208	99	18	90
Maurice Harkless	1887840	21	ORL	45	106	25	32	9	27
Meyers Leonard	2317920	22	POR	55	250	32	10	14	39
Michael Kidd-Gilchrist	5016960	21	CHO	55	416	77	30	38	63
Mitch McGary	1400040	22	OKC	32	165	14	16	16	31
Nerlens Noel	3315120	20	PHI	75	611	128	133	142	146
Nick Johnson	507336	22	HOU	28	39	11	7	3	19
Nik Stauskas	2745840	21	SAC	73	88	67	20	17	40
Noah Vonleh	2524200	19	CHO	25	86	4	4	9	11
Otto Porter	4470480	21	WAS	74	221	65	44	30	52
P.J. Hairston	1149720	22	CHO	45	92	21	21	13	22
Quincy Miller	183049	22	TOT	10	20	8	7	5	5
Ricky Ledo	816482	22	TOT	17	36	19	6	1	26
Rodney Hood	1290360	22	UTA	50	117	83	30	12	45
Rudy Gobert	1127400	22	UTA	82	775	109	64	189	111
Sergey Karasev	1533840	21	BRK	33	66	46	23	1	24
Shabazz Muhammad	1971960	22	MIN	38	154	44	18	7	35
Shane Larkin	1606080	22	NYK	76	176	226	93	9	83
Sim Bhullar	29843	22	SAC	3	1	1	0	1	0
Spencer Dinwiddie	700000	21	DET	34	48	104	19	6	33

Question 3. Assign `convenience_stats` to an array of the average age and average salary of your convenience sample, using the `compute_statistics` function. Since they're computed on a sample, these are called *sample averages*.

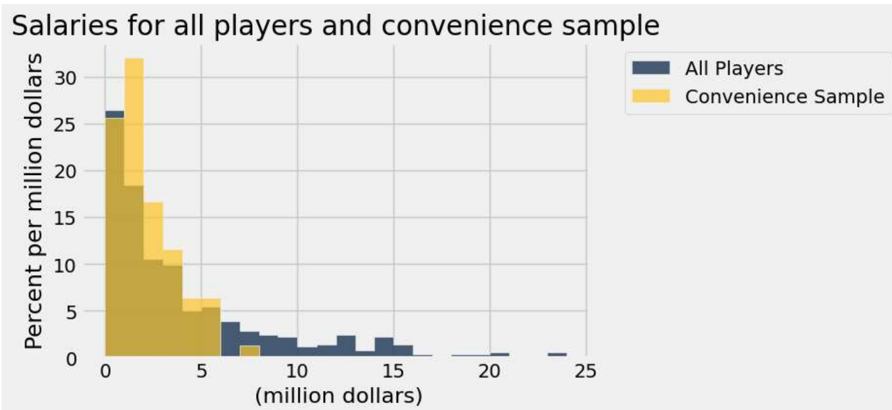
```
convenience_stats = [np.average(convenience_sample.column('Age')),np.average(convenience_sample.column('Salary')/1000000)]
convenience_stats
```

```
[21.076923076923077, 2.1527852435897437]
```

Next, we'll compare the convenience sample salaries with the full data salaries in a single histogram. To do that, we'll need to use the `bin_column` option of the `hist` method, which indicates that all columns are counts of the bins in a particular column. The following cell does not require any changes; **just run it**.

```
def compare_salaries(first, second, first_title, second_title):
    """Compare the salaries in two tables."""
    first_salary_in_millions = first.column('Salary')/1000000
    second_salary_in_millions = second.column('Salary')/1000000
    first_tbl_millions = first.drop('Salary').with_column('Salary', first_salary_in_millions)
    second_tbl_millions = second.drop('Salary').with_column('Salary', second_salary_in_millions)
    max_salary = max(np.append(first_tbl_millions.column('Salary'), second_tbl_millions.column('Salary')))
    bins = np.arange(0, max_salary+1, 1)
    first_binned = first_tbl_millions.bin('Salary', bins=bins).relabeled(1, first_title)
    second_binned = second_tbl_millions.bin('Salary', bins=bins).relabeled(1, second_title)
    first_binned.join('bin', second_binned).hist(bin_column='bin', unit='million dollars')
    plt.title('Salaries for all players and convenience sample')

compare_salaries(full_data, convenience_sample, 'All Players', 'Convenience Sample')
```



Question 4. Does the convenience sample give us an accurate picture of the salary of the full population? Would you expect it to, in general? Before you move on, write a short answer in English below. You can refer to the statistics calculated above or perform your own analysis.

No it doesn't give us an accurate picture- and I wouldn't expect it to- filtering the table to only contain people younger than 22 will obviously leave out lots of important data- most players salaries increase with time in the league.

Simple random sampling

A more justifiable approach is to sample uniformly at random from the players. In a **simple random sample (SRS) without replacement**, we ensure that each player is selected at most once. Imagine writing down each player's name on a card, putting the cards in an box, and shuffling the box. Then, pull out cards one by one and set them aside, stopping when the specified sample size is reached.

Producing simple random samples

Sometimes, it's useful to take random samples even when we have the data for the whole population. It helps us understand sampling accuracy.

`sample`

The table method `sample` produces a random sample from the table. By default, it draws at random **with replacement** from the rows of a table. It takes in the sample size as its argument and returns a **table** with only the rows that were selected.

Run the cell below to see an example call to `sample()` with a sample size of 5, with replacement.

```
# Just run this cell

salary_data.sample(5)
```

The optional argument `with_replacement=False` can be passed through `sample()` to specify that the sample should be drawn without replacement.

Run the cell below to see an example call to `sample()` with a sample size of 5, without replacement.

```
# Just run this cell

salary_data.sample(5, with_replacement=False)
```

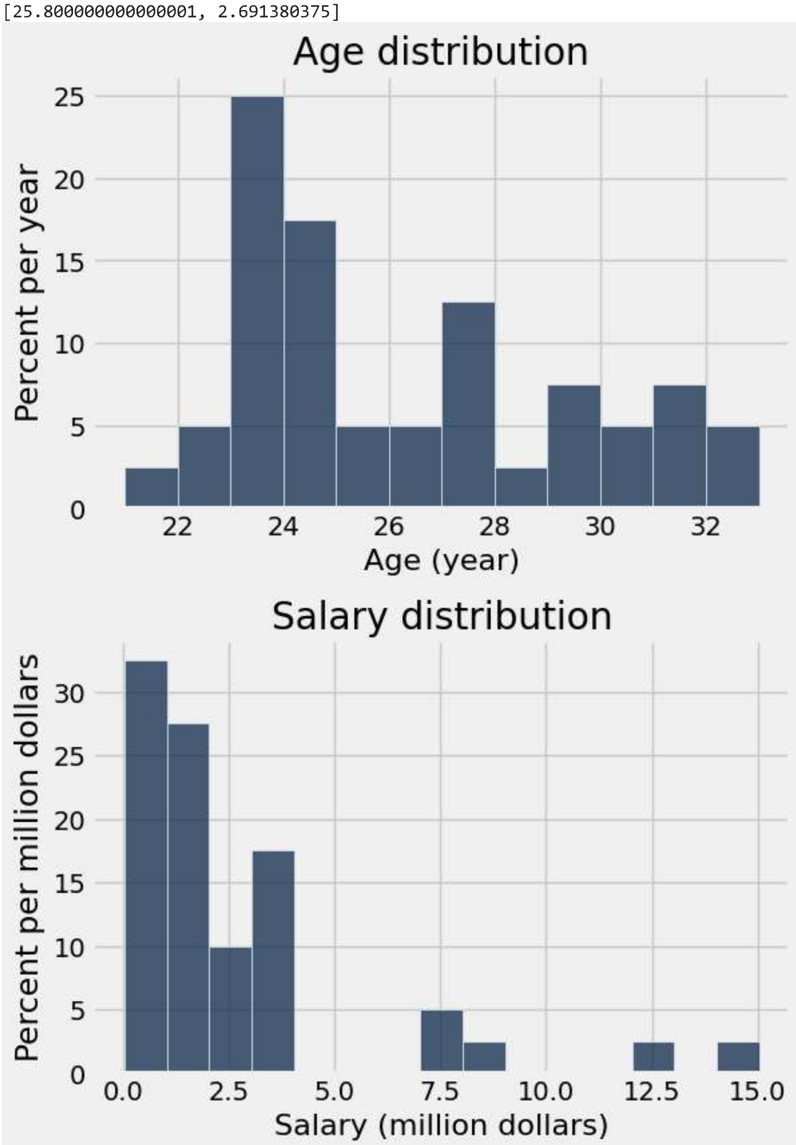
PlayerName	Salary
David Wear	29843
Andrea Bargnani	11500000
Elton Brand	2000000
Dirk Nowitzki	7974482
Julius Randle	2997360

Question 5. Produce a simple random sample of size 44 from `full_data`. Run your analysis on it again. Run the cell a few times to see how the histograms and statistics change across different samples.

- How much does the average age change across samples?
- What about average salary?

```
my_small_srswor_data = full_data.sample(40)
my_small_stats = [np.average(my_small_srswor_data.column('Age')),np.average(my_small_srswor_data.column('Salary')/1000000)]
histograms(my_small_srswor_data)
my_small_stats
```

```
[25.800000000000001, 2.691380375]
```



Write your answer here, replacing this text.

Question 6. As in the previous question, analyze several simple random samples of size 100 from `full_data`.

- Do the histogram shapes seem to change more or less across samples of 100 than across samples of size 44?
- Are the sample averages and histograms closer to their true values/shape for age or for salary? What did you expect to see?

```
my_large_srswor_data = full_data.sample(100)
my_large_stats = [np.average(my_large_srswor_data.column('Age')),np.average(my_large_srswor_data.column('Salary')/1000000)]
histograms(my_large_srswor_data)
my_large_stats
```