Lab 6

Welcome to Lab 6! In this lab we will learn about sampling strategies. More information about Sampling in the textbook can be found [here! (https://www.inferentialthinking.com/chapters/08/5/sampling.html)](https://www.inferentialthinking.com/chapters/08/5/sampling.html)

The data used in this lab will contain salary data and statistics for basketball players from the 2014-2015 NBA season. This data was collected from [basketball-reference (http://www.basketball-reference.com)](http://www.basketball-reference.com) and [spotrac (http://www.spotrac.com)](http://www.spotrac.com).

In [1]:

```python
# 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 plots
plots.style.use('fivethirtyeight')

# Don't change this cell; just run it.
from client.api.notebook import Notebook
ok = Notebook('lab06.ok')
_ = ok.auth(inline=True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_20804\1846456633.py in <module>
     12
     13 # Don't change this cell; just run it.
---> 14 from client.api.notebook import Notebook
     15 ok = Notebook('lab06.ok')
     16 _ = ok.auth(inline=True)

ModuleNotFoundError: No module named 'client'
```

1. Dungeons and Dragons and Sampling

In the game Dungeons & Dragons, each player plays the role of a fantasy character.

A player performs actions by rolling a 20-sided die, adding a "modifier" number to the roll, and comparing the total to a threshold for success. The modifier depends on her character's competence in performing the action.

For example, suppose Alice's character, a barbarian warrior named Roga, is trying to knock down a heavy door. She rolls a 20-sided die, adds a modifier of 11 to the result (because her character is good at knocking down doors), and succeeds if the total is greater than 15.

** Question 1.1 ** Write code that simulates that procedure. Compute three values: the result of Alice's roll (`roll_result`), the result of her roll plus Roga's modifier (`modified_result`), and a boolean value indicating whether the action succeeded (`action_succeeded`). **Do not fill in any of the results manually**; the entire simulation should happen in code.

Hint: A roll of a 20-sided die is a number chosen uniformly from the array `make_array(1, 2, 3, 4, ..., 20)` . So a roll of a 20-sided die *plus 11* is a number chosen uniformly from that array, plus 11.

In [3]:

```python
possible_rolls = np.arange(1, 20+1, 1)
roll_result = np.random.choice(possible_rolls)
modified_result = roll_result + 11
action_succeeded = modified_result > 15

# The next line just prints out your results in a nice way
# once you're done.  You can delete it if you want.
print("On a modified roll of {:d}, Alice's action {}.".format(modified_result, "succeeded" if action_succeeded else "failed")
```

```
On a modified roll of 13, Alice's action failed.
```

In [ ]:

```python
_ = ok.grade('q1_1')
```

** Question 1.2 ** Run your cell 7 times to manually estimate the chance that Alice succeeds at this action. (Don't use math or an extended simulation.). Your answer should be a fraction.

In [4]:

```python
rough_success_chance = 1/7
```

In [ ]:
```python
_ = ok.grade('q1_2')
```

Suppose we don't know that Roga has a modifier of 11 for this action. Instead, we observe the modified roll (that is, the die roll plus the modifier of 11) from each of 7 of her attempts to knock down doors. We would like to estimate her modifier from these 7 numbers.

** Question 1.3 ** Write a Python function called `simulate_observations`. It should take no arguments, and it should return an array of 7 numbers. Each of the numbers should be the modified roll from one simulation. **Then**, call your function once to compute an array of 7 simulated modified rolls. Name that array `observations`.

In [5]:
```python
modifier = 11
num_observations = 7

def simulate_observations():
    """Produces an array of 7 simulated modified die rolls"""
    return np.random.choice(possible_rolls, num_observations) + modifier

observations = simulate_observations()
observations
```

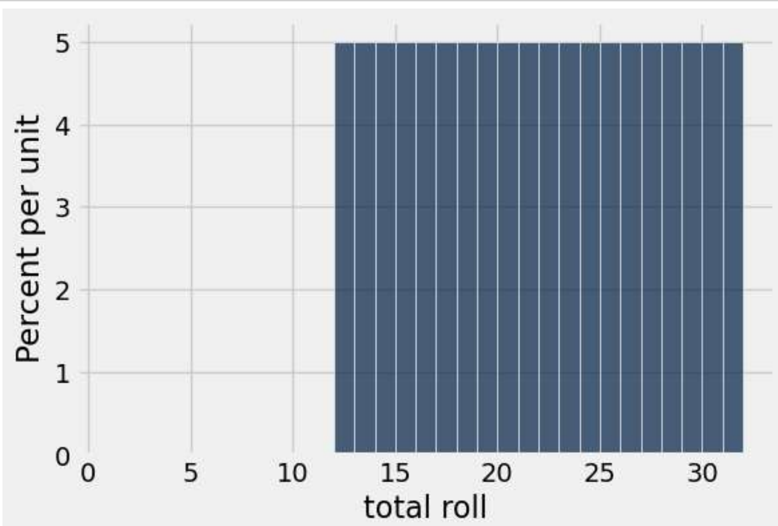
Out[5]:
```
array([24, 18, 13, 16, 14, 26, 26])
```

In [ ]:
```python
_ = ok.grade('q1_3')
```

* *Question 1.4 ** Draw a histogram to display the *probability distribution* of the modified rolls we might see. Check with a neighbor or a TA to make sure you have the right histogram.

In [6]:
```python
# We suggest using these bins.
roll_bins = np.arange(1, modifier+2+20, 1)

Table().with_column("total roll", np.arange(1+modifier, 20+modifier+1)).hist("total roll", bins=roll_bins)
```



Now let's imagine we don't know the modifier and try to estimate it from `observations`.

One straightforward (but clearly suboptimal) way to do that is to find the *smallest* total roll, since the smallest roll on a 20-sided die is 1, which is roughly 0.

** Question 1.5 ** Using that method, estimate `modifier` from `observations`. Name your estimate `min_estimate`.

In [7]:
```python
min_estimate = min(observations)
min_estimate
```

Out[7]:
```
13
```

In [ ]:
```python
_ = ok.grade('q1_5')
```

Another way to estimate the modifier involves the mean of `observations`.

** Question 1.6 ** Figure out a good estimate based on that quantity. **Then**, write a function named `mean_based_estimator` that computes your estimate. It should take an array of modified rolls (like the array `observations`) as its argument and return an estimate of `modifier` based on those numbers.

In [8]:
```
def mean_based_estimator(nums):
    """Estimate the roll modifier based on observed modified rolls in the array nums."""
    return np.mean(nums) - 10

# Here is an example call to your function.  It computes an estimate
# of the modifier from our 7 observations.
mean_based_estimate = mean_based_estimator(observations)
mean_based_estimate
```

Out[8]: 9.571428571428573

In [ ]:
```
_ = ok.grade('q1_6')
```

2. Sampling

Run the cell below to load the player and salary data.

In [9]:
```
player_data = Table().read_table("player_data.csv")
salary_data = Table().read_table("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	Points
A.J. Price	62552	28	TOT	26	32	46	7	0	14	133
Aaron Brooks	1145685	30	CHI	82	166	261	54	15	157	954
Aaron Gordon	3992040	19	ORL	47	169	33	21	22	38	243

... (489 rows omitted)

Rather than getting data on every player, 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. Instead, we often make *statistical inferences* about a large underlying population using a smaller sample.

A statistical inference is a statement about some statistic of the underlying population, such as "the average salary of NBA players in 2014 was $3". You may have heard the word "inference" used in other contexts. It's important to keep in mind that statistical inferences, unlike, say, logical inferences, can be wrong.

A general strategy for inference using samples is to estimate statistics of the population by computing the same statistics on a sample. This strategy sometimes works well and sometimes doesn't. The degree to which it gives us useful answers depends on several factors, and we'll touch lightly on a few of those today.

One very important factor in the utility of samples is how they were gathered. We have prepared some example sample datasets to simulate inference from different kinds of samples for the NBA player 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 loading and analysis code into two 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.

Question 2.1. Complete the `histograms` function, which takes a table with columns `Age` and `Salary` and draws a histogram for each one. Use the min and max functions to pick the bin boundaries so that all data appears for any table passed to your function. Use the same bin widths as before (1 year for `Age` and $1,000,000 for `Salary`).

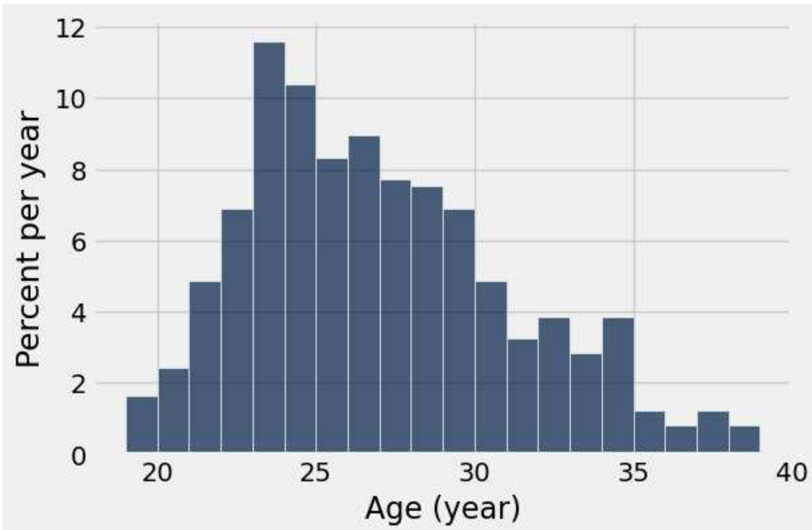
In [10]:

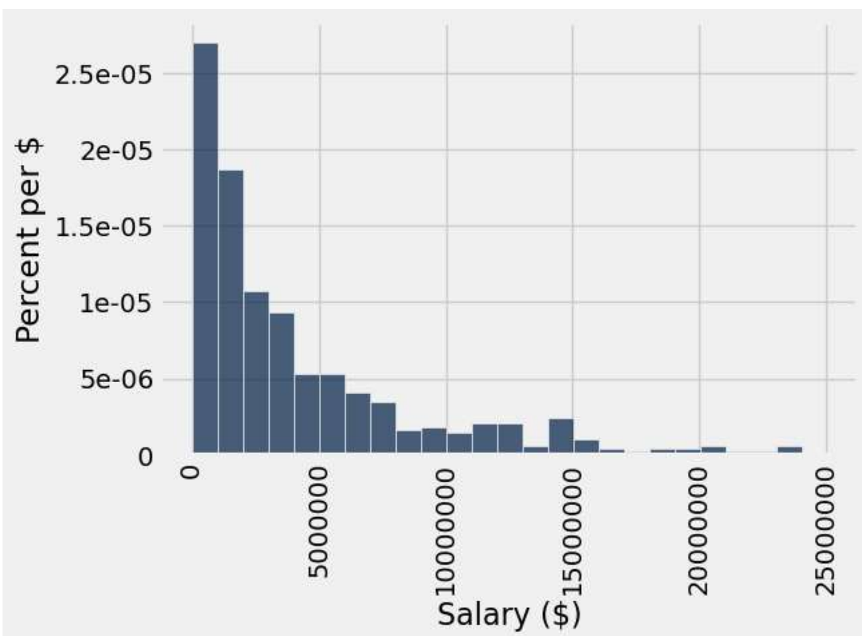
```
def histograms(t):
    ages = t.column('Age')
    salaries = t.column('Salary')
    age_bins = np.arange(min(ages), max(ages) + 2, 1)
    salary_bins = np.arange(min(salaries), max(salaries) + 2000000, 1000000)
    t.hist('Age', bins=age_bins, unit='year')
    t.hist('Salary', bins=salary_bins, unit='$')
    return age_bins # Keep this statement so that your work can be checked

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

```
Two histograms should be displayed below

C:\Users\ariel\anaconda3\lib\site-packages\datascience\tables.py:5865: UserWarning: FixedFormatter should only be used toget
her with FixedLocator
  axis.set_xticklabels(ticks, rotation='vertical')
```





In [ ]:

```
_ = ok.grade('q2_1') # Warning: Charts will be displayed while running this test
```

Question 2.2. Create a function called `compute_statistics` that takes a Table containing ages and salaries and:

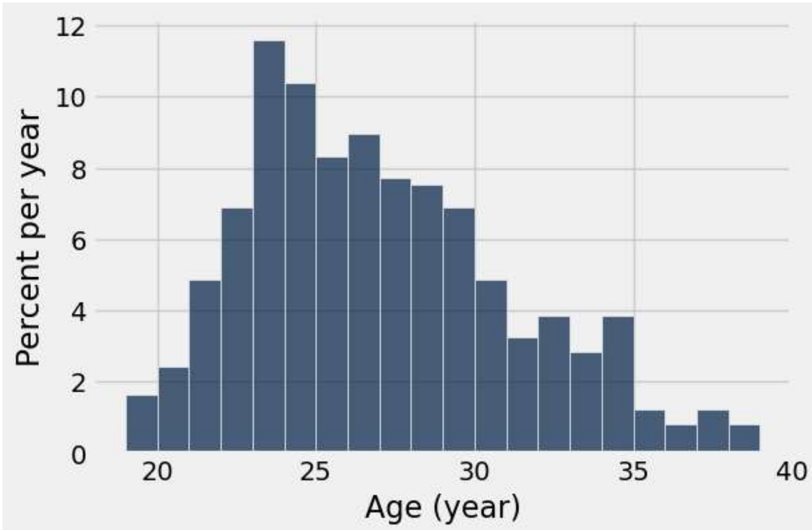
- Draws a histogram of ages
- Draws a histogram of salaries
- Return a two-element list containing the average age and average salary

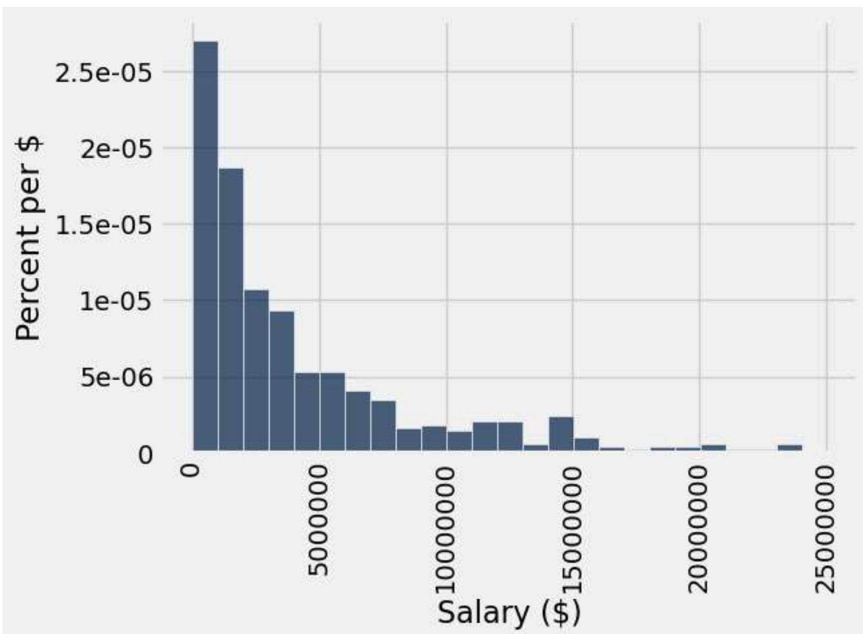
You can call your `histograms` function to draw the histograms!

In [11]:

```
def compute_statistics(age_and_salary_data):
    histograms(age_and_salary_data)
    age = age_and_salary_data.column("Age")
    salary = age_and_salary_data.column("Salary")
    return make_array(np.mean(age), np.mean(salary))


full_stats = compute_statistics(full_data)
```





In [ ]:

```
_ = ok.grade('q2_2') # Warning: Charts will be displayed while running this test
```

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 that's 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 2.3 Assign `convenience_sample_data` to a subset of `full_data` that contains only the rows for players under the age of 22.

In [12]:
```
convenience_sample = full_data.where("Age", are.below(22))
convenience_sample
```

Out[12]:

PlayerName	Salary	Age	Team	Games	Rebounds	Assists	Steals	Blocks	Turnovers	Points
Aaron Gordon	3992040	19	ORL	47	169	33	21	22	38	243
Alex Len	3649920	21	PHO	69	454	32	34	105	74	432
Andre Drummond	2568360	21	DET	82	1104	55	73	153	120	1130
Andrew Wiggins	5510640	19	MIN	82	374	170	86	50	177	1387
Anthony Bennett	5563920	21	MIN	57	216	48	27	16	36	298
Anthony Davis	5607240	21	NOP	68	696	149	100	200	95	1656
Archie Goodwin	1112280	20	PHO	41	74	44	18	9	48	231
Ben McLemore	3026280	21	SAC	82	241	140	77	19	138	996
Bradley Beal	4505280	21	WAS	63	241	194	76	18	123	962
Bruno Caboclo	1458360	19	TOR	8	2	0	0	1	4	10

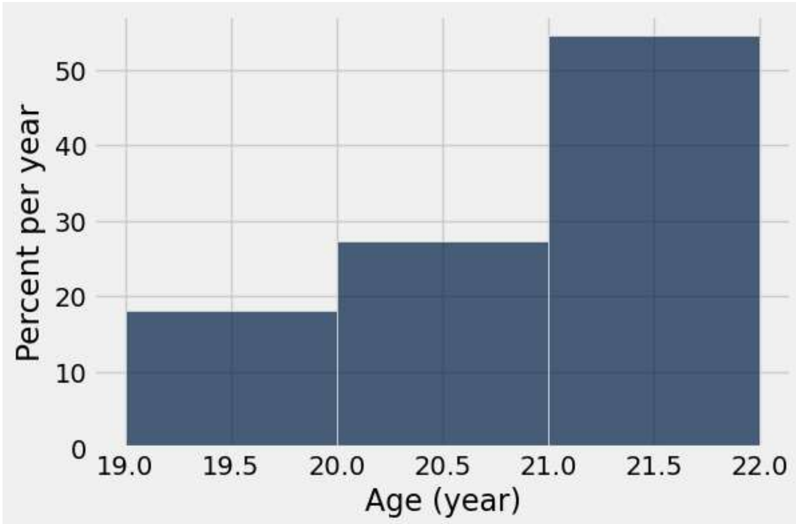
... (34 rows omitted)

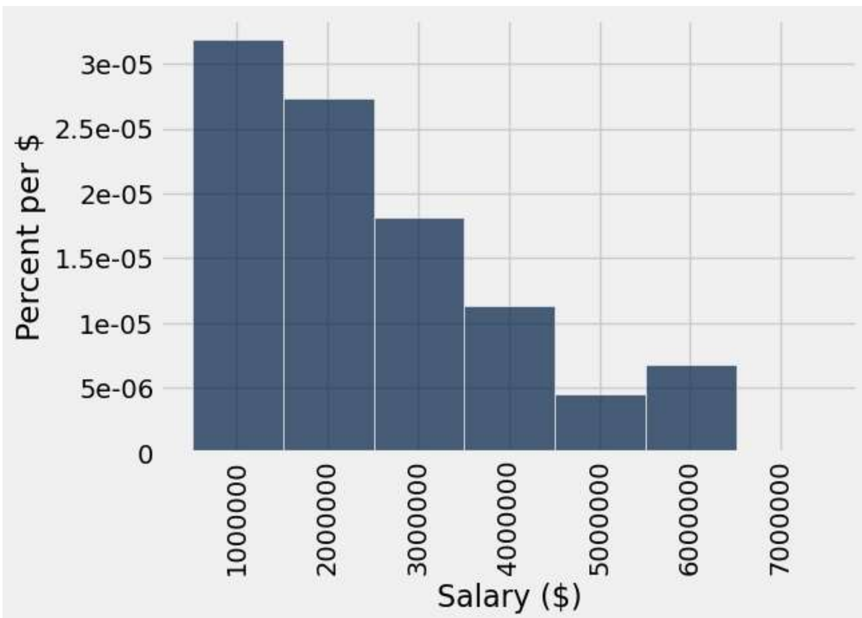
In [ ]:
```
_ = ok.grade('q2_3')
```

Question 2.4 Assign `convenience_stats` to a list 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*.

In [13]:
```
convenience_stats = compute_statistics(convenience_sample)
convenience_stats
```

Out[13]: array([2.03636364e+01,   2.38353382e+06])





In [ ]:
```
_ = ok.grade('q2_4')
```

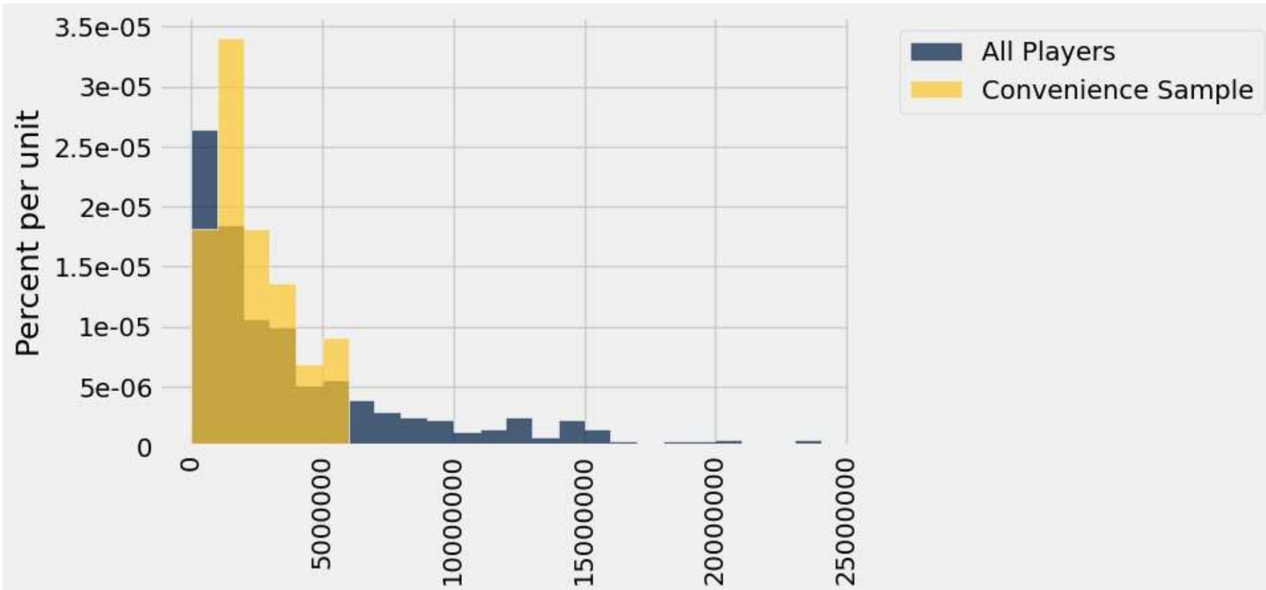
Next, we'll compare the convenience sample salaries with the full data salaries in a single histogram. To do that, we'll need to use the `counts` option of the `hist` method, which indicates that all columns are counts of the bins in a particular column. The following cell should not require any changes;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

In [14]:

```python
def compare_salaries(first, second, first_title, second_title):
    """Compare the salaries in two tables."""
    max_salary = max(np.append(first.column('Salary'), second.column('Salary')))
    bins = np.arange(0, max_salary+1e6+1, 1e6)
    first_binned = first.bin('Salary', bins=bins).relabeled(1, first_title)
    second_binned = second.bin('Salary', bins=bins).relabeled(1, second_title)
    first_binned.join('bin', second_binned).hist(counts='bin')

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

```
C:\Users\ariel\anaconda3\lib\site-packages\datascience\tables.py:5398: UserWarning: counts arg of hist is deprecated; use bi
n_column
  warnings.warn("counts arg of hist is deprecated; use bin_column")
```



Question 2.5 Does the convenience sample give us an accurate picture of the age and salary of the full population of NBA players in 2014-2015? 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, the sample does not provide an accurate picture of the age and salary of the full population of NBA players. This is geared towards players under the age of 22.

Simple random sampling

A more principled approach is to sample uniformly at random from the players. If we ensure that each player is selected at most once, this is a *simple random sample without replacement*, sometimes abbreviated to "simple random sample" or "SRSWOR". Imagine writing down each player's name on a card, putting the cards in an urn, and shuffling the urn. Then, pull out cards one by one and set them aside, stopping when the specified *sample size* is reached.

We've produced two samples of the `salary_data` table in this way: `small_srswor_salary.csv` and `large_srswor_salary.csv` contain, respectively, a sample of size 44 (the same as the convenience sample) and a larger sample of size 100.

The `load_data` function below loads a salary table and joins it with `player_data` .

In [19]:

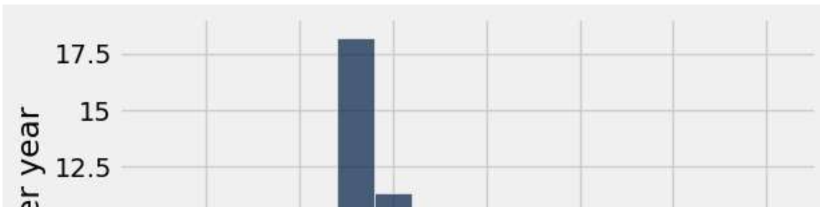
```python
def load_data(salary_file):
    return player_data.join('Name', Table.read_table(salary_file), 'PlayerName')
```

Question 2.6 Run the same analyses on the small and large samples that you previously ran on the full dataset and on the convenience sample. Compare the accuracy of the estimates of the population statistics that we get from the convenience sample, the small simple random sample, and the large simple random sample. (Just notice this for yourself -- the autograder will check your sample statistics but will not validate whatever you do to compare.)

In [20]:
```
# Original:
small_srswor_data = load_data("small_srswor_salary.csv")
small_stats = compute_statistics(small_srswor_data)
large_srswor_data = load_data("large_srswor_salary.csv")
large_stats = compute_statistics(large_srswor_data)
print('Full data stats:                ', full_stats)
print('Small simple random sample stats:', small_stats)
print('Large simple random sample stats:', large_stats)
```

```
C:\Users\ariel\anaconda3\lib\site-packages\datascience\tables.py:5865: UserWarning: FixedFormatter should only be used to
gether with FixedLocator
  axis.set_xticklabels(ticks, rotation='vertical')
C:\Users\ariel\anaconda3\lib\site-packages\datascience\tables.py:5865: UserWarning: FixedFormatter should only be used to
gether with FixedLocator
  axis.set_xticklabels(ticks, rotation='vertical')

Full data stats:                  [  2.65365854e+01   4.26977577e+06]
Small simple random sample stats: [  2.63181818e+01   4.28391089e+06]
Large simple random sample stats: [  2.64200000e+01   4.82132250e+06]
```



In [ ]:
```
_ = ok.grade('q2_6')
```

Producing simple random samples

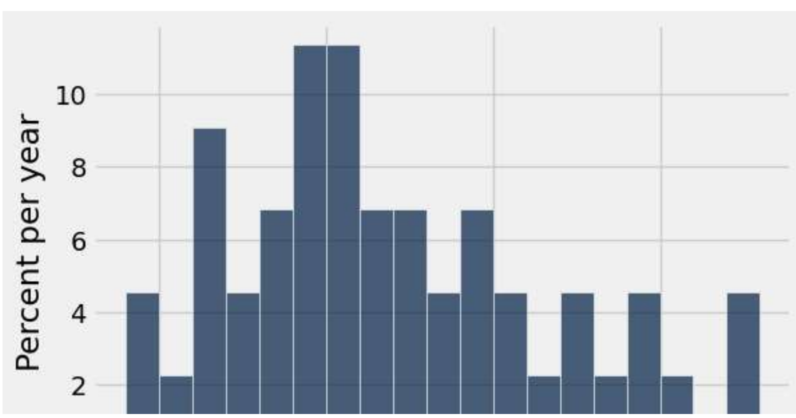
Often it's useful to take random samples even when we have a larger dataset available. The randomized response technique was one example we saw in lecture. Another is to help us understand how inaccurate other samples are.

Tables provide the method `sample()` for producing random samples. Note that its default is to sample with replacement. To see how to call `sample()` , search the documentation on `data8.org/datascience` , or enter `full_data.sample?` into a code cell and press Enter.

Question 2.7 Produce a simple random sample of size 44 from `full_data` . (You don't need to bother with a join this time -- just use `full_data.sample(...)` directly. That will have the same result as sampling from `salary_data` and joining with `player_data` .) Run your analysis on it again. Are your results similar to those in the small sample we provided you? Run your code several times to get new samples. How much do things change across samples?

In [23]:
```
my_small_srswor_data = full_data.sample(44, with_replacement = False)
my_small_stats = compute_statistics(my_small_srswor_data)
my_small_stats
```

Out[23]:
```
array([  2.65227273e+01,   4.63677957e+06])
```

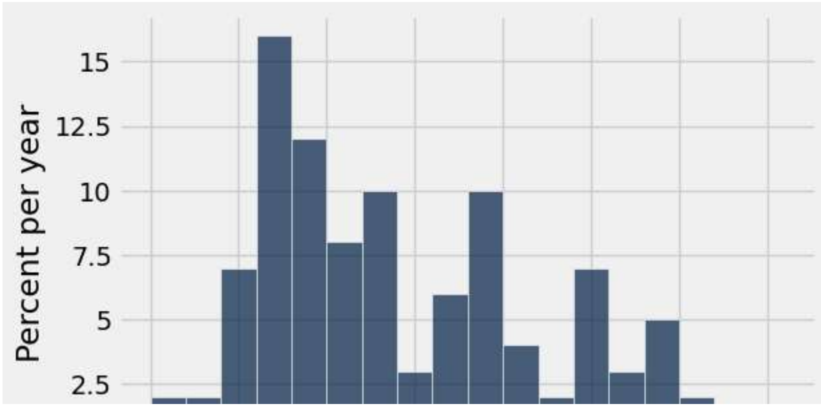


The results are similar but they are not the same. The averga age seems to remain the same, but the salary does change due to the larger variety of salary.

Question 2.8 As in the previous question, analyze several simple random samples of size 100 from `full_data` . Do the average and histogram statistics seem to change more or less across samples of this size than across samples of size 44? And are the sample averages and histograms closer to their true values for age or for salary? What did you expect to see?

In [25]:
```
my_large_srswor_data = full_data.sample(100, with_replacement = False)
compute_statistics(my_large_srswor_data)
```

Out[25]:
```
array([  2.67300000e+01,   4.37909152e+06])
```



The average and histogram statistics change less with the change of sample size. They are closer to their true valyes, which could be expected due to sampling a larger population size.

In [26]:
```
# For your convenience, you can run this cell to run all the tests at once!
import os
_ = [ok.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_20804\785757325.py in <module>
      1 # For your convenience, you can run this cell to run all the tests at once!
      2 import os
----> 3 _ = [ok.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]

~\AppData\Local\Temp\ipykernel_20804\785757325.py in <listcomp>(.0)
      1 # For your convenience, you can run this cell to run all the tests at once!
      2 import os
----> 3 _ = [ok.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]

NameError: name 'ok' is not defined
```

In [27]:
```
_ = ok.submit()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_20804\3901802801.py in <module>
----> 1 _ = ok.submit()

NameError: name 'ok' is not defined
```