Lab 5: Simulations

Welcome to Lab 5!

We will go over [iteration (https://www.inferentialthinking.com/chapters/09/2/Iteration.html)](https://www.inferentialthinking.com/chapters/09/2/Iteration.html) and [simulations (https://www.inferentialthinking.com/chapters/09/3/Simulation.html)](https://www.inferentialthinking.com/chapters/09/3/Simulation.html), as well as introduce the concept of [randomness (https://www.inferentialthinking.com/chapters/09/Randomness.html)](https://www.inferentialthinking.com/chapters/09/Randomness.html).

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 (http://www.basketball-reference.com)](http://www.basketball-reference.com) and [Spotrac (http://www.spotrac.com)](http://www.spotrac.com).

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

In [1]:
```
# 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')

# Don't change this cell; just run it.
from client.api.notebook import Notebook
ok = Notebook('lab05.ok')
```

```
=====================================================================
Assignment: Lab 5
OK, version v1.14.19
=====================================================================
```

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!

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

In [7]:
```
3 > 1 + 1
```

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

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

In [8]:
```
result = 10 / 2 == 5
result
```

Out[8]:
```
True
```

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

In [9]:
```
make_array(1, 5, 7, 8, 3, -1) > 3
```

Out[9]:
```
array([False,  True,  True,  True, False, False])
```

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.

In [10]:
```
nachos = make_array('cheese', 'salsa', 'both', 'neither')
np.random.choice(nachos)
```

Out[10]:
```
'cheese'
```

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.

In [11]:
```
np.random.choice(nachos, 10)
```

Out[11]:
```
array(['neither', 'neither', 'cheese', 'neither', 'both', 'both',
       'cheese', 'cheese', 'neither', 'cheese'], 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` .

In [12]:
```
np.count_nonzero(make_array(True, False, False, True, True))
```

Out[12]: 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.

```
BEGIN QUESTION
name: q11
```

In [14]:
```
ten_nachos = make_array('neither', 'cheese', 'both', 'both', 'cheese',
'salsa', 'both', 'neither', 'cheese', 'both')
number_cheese = np.count_nonzero(ten_nachos == 'cheese') #SOLUTION
number_cheese
```

Out[14]: 3

In [15]:
```
# TEST
number_cheese == 3
```

Out[15]: True

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 (https://www.inferentialthinking.com/chapters/09/1/conditional-statements.html)](https://www.inferentialthinking.com/chapters/09/1/conditional-statements.html).

Question 2. Complete the following conditional statement so that the string `'More please'` is assigned to the variable `say_please` if the number of nachos with cheese in `ten_nachos` is less than `5` .

Hint: You should be using `number_cheese` from Question 1.

```
BEGIN QUESTION
name: q12
```

In [16]:
```
""" # BEGIN PROMPT
say_please = '?'

if ...:
    say_please = 'More please'
"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
say_please = '?'

if number_cheese < 5:
    say_please = 'More please'
# END SOLUTION
say_please
```

Out[16]: 'More please'

In [17]:
```
# TEST
say_please == 'More please'
```

Out[17]: True

Question 3. 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 table below to match the nacho type to the appropriate reaction.

Hint: If you're failing the test, double check the spelling of your reactions.

```
BEGIN QUESTION
name: q13
```

In [18]:

```
""" # BEGIN PROMPT
def nacho_reaction(nacho):
    if nacho == "cheese":
        return ...
    ... :
        ...
    ... :
        ...
    ... :
        ...
"""; # END PROMPT

# BEGIN SOLUTION NO PROMPT
def nacho_reaction(nacho):
    if nacho == 'cheese':
        return 'Cheesy!'
    elif nacho == 'salsa':
        return 'Spicy!'
    elif nacho == 'both':
        return 'Wow!'
    else:
        return 'Meh.'
# END SOLUTION
spicy_nacho = nacho_reaction('salsa')
spicy_nacho
```

Out[18]: 'Spicy!'

In [19]:

```
# TEST
nacho_reaction('salsa')
```

Out[19]: 'Spicy!'

In [20]:

```
# TEST
nacho_reaction('cheese')
```

Out[20]: 'Cheesy!'

In [21]:

```
# TEST
nacho_reaction('both')
```

Out[21]: 'Wow!'

In [22]:

```
# TEST
nacho_reaction('neither')
```

Out[22]: 'Meh.'

Question 4. 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.

```
BEGIN QUESTION
name: q14
```

In [23]:
```
ten_nachos_tbl = Table().with_column('Nachos', ten_nachos)
""" # BEGIN PROMPT
...
"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
ten_nachos_reactions = ten_nachos_tbl.with_column('Reactions', ten_nacho
s_tbl.apply(nacho_reaction, 'Nachos'))
# END SOLUTION
ten_nachos_reactions
```

Out[23]:

Nachos	Reactions
neither	Meh.
cheese	Cheesy!
both	Wow!
both	Wow!
cheese	Cheesy!
salsa	Spicy!
both	Wow!
neither	Meh.
cheese	Cheesy!
both	Wow!

In [24]:
```
# TEST
# One or more of the reaction results could be incorrect
np.count_nonzero(ten_nachos_reactions.column('Reactions') == make_array(
'Meh.', 'Cheesy!', 'Wow!', 'Wow!', 'Cheesy!', 'Spicy!', 'Wow!', 'Meh.',
'Cheesy!', 'Wow!')) == 10
```

Out[24]: True

Question 5. Using code, find the number of 'Wow!' reactions for the nachos in `ten_nachos_reactions` .

```
BEGIN QUESTION
name: q15
```

In [25]:
```
number_wow_reactions = np.count_nonzero(ten_nachos_reactions.column('Rea
ctions') == 'Wow!') #SOLUTION
number_wow_reactions
```

Out[25]: 4

In [26]:
```
# TEST
2 < number_wow_reactions < 6
```

Out[26]: True

In [27]:
```
# TEST
# Incorrect value for number_wow_reactions
number_wow_reactions == 4
```

Out[27]: True

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.

In [28]:
```
rainbow = make_array("red", "orange", "yellow", "green", "blue", "indig
o", "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.

In [31]:
```
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.

```
BEGIN QUESTION
name: q21
```

In [32]:
```
austen_string = open('Austen_PrideAndPrejudice.txt', encoding='utf-8').r
ead()
p_and_p_words = np.array(austen_string.split())

""" # BEGIN PROMPT
longer_than_five = ...

# a for loop would be useful here

"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
longer_than_five = 0

for word in p_and_p_words:
    if len(word) > 5:
        longer_than_five = longer_than_five + 1
# END SOLUTION
longer_than_five
```

Out[32]: 35453

In [33]:
```
# TEST
longer_than_five == 35453
```

Out[33]: True

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.

```
BEGIN QUESTION
name: q22
```

In [34]:
```
""" # BEGIN PROMPT
trials = 10000
num_different = ...

for ... in ...:
    ...
"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
trials = 10000
num_different = 0

for i in np.arange(trials):
    words = np.random.choice(p_and_p_words, 2)
    if len(words.item(0)) != len(words.item(1)):
        num_different = num_different + 1
# END SOLUTION
num_different
```

Out[34]: 8580

In [35]:
```
# TEST
8100 <= num_different <= 9100
```

Out[35]: True

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.

```
BEGIN QUESTION
name: q23
```

In [36]:
```
""" # BEGIN PROMPT
possible_point_values = ...
num_tosses = 1000
simulated_tosses = ...
total_score = ...
"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
possible_point_values = np.arange(1, 11)
num_tosses = 1000
simulated_tosses = np.random.choice(possible_point_values, num_tosses)
total_score = sum(simulated_tosses)
# END SOLUTION
total_score
```

Out[36]:
```
5414
```

In [37]:
```
# TEST
1000 <= total_score <= 10000
```

Out[37]:
```
True
```

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. We'll guide you through this code, 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 (https://www.inferentialthinking.com/chapters/10/Sampling_and_Empirical_Distributions.html).

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

In [38]:
```
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 (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.

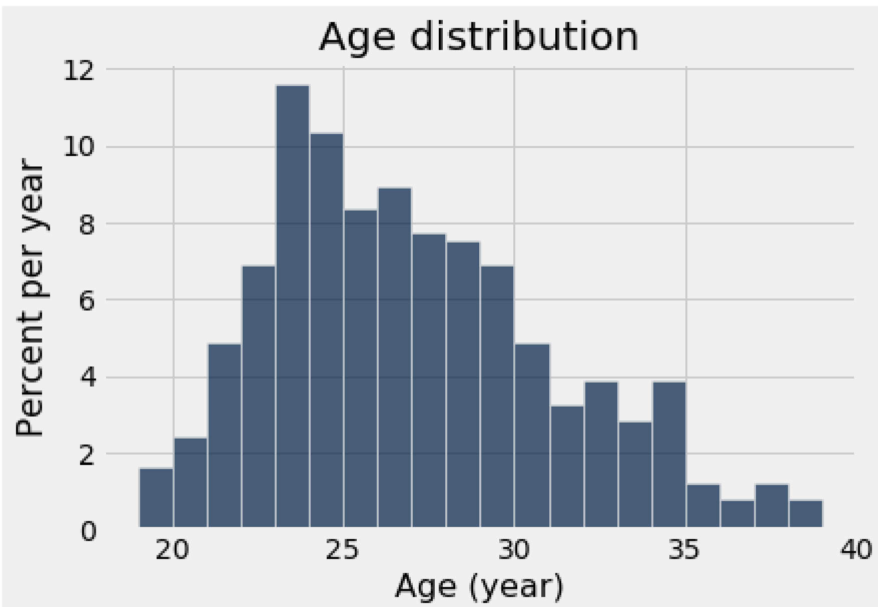
To save typing and increase the clarity of your code, we will package the analysis code into a few functions. This will be useful in the rest of the lab as we will repeatedly need to create histograms and collect summary statistics from that data.

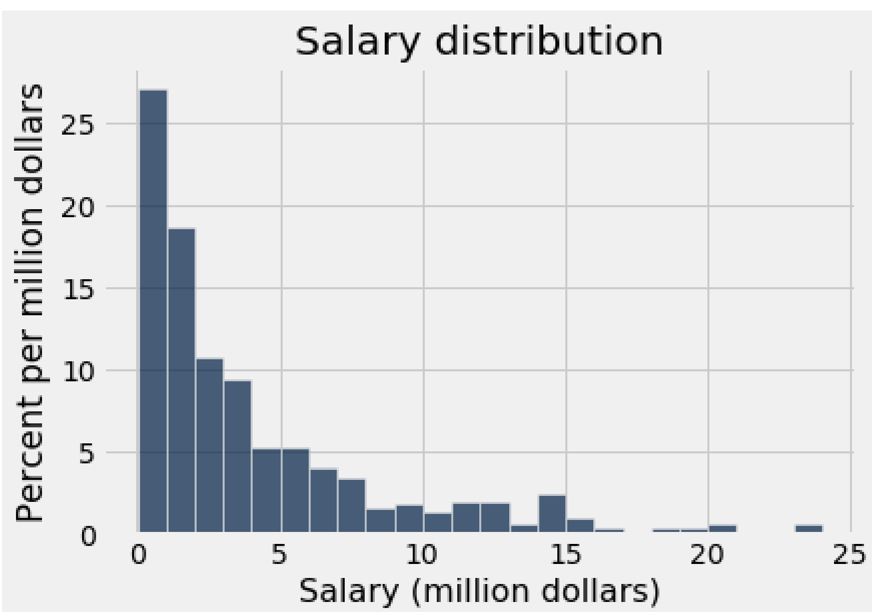
We've defined the `histograms` function below, which takes a table with columns `Age` and `Salary` and draws a histogram for each one. It uses bin widths of 1 year for `Age` and $1,000,000 for `Salary` .

In [39]:
```
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!

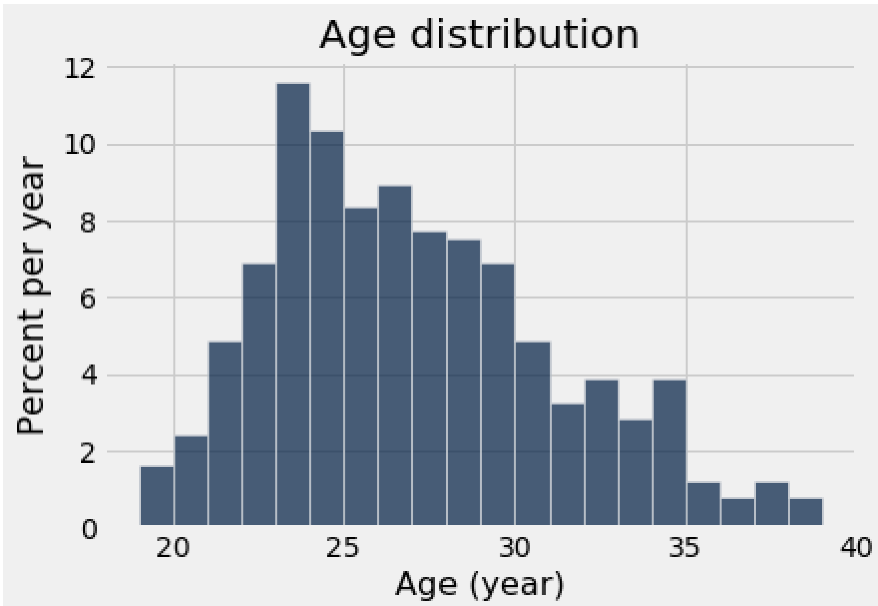
Note: More charts will be displayed when running the test cell. Please feel free to ignore the charts.

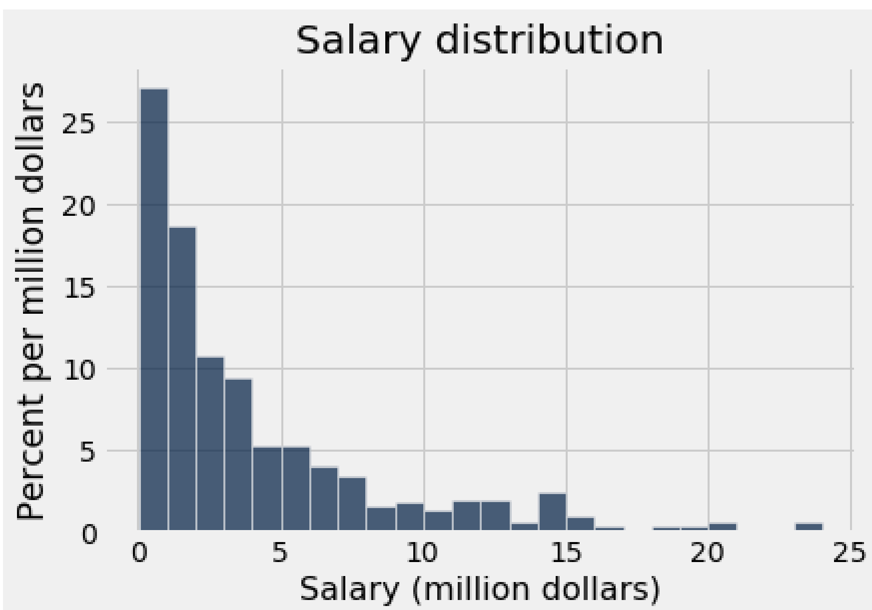
```
BEGIN QUESTION
name: q31
```

In [40]:
```
def compute_statistics(age_and_salary_data):
    histograms(age_and_salary_data) #SOLUTION
    age = age_and_salary_data.column("Age") #SOLUTION
    salary = age_and_salary_data.column("Salary") #SOLUTION
    return make_array(np.mean(age), np.mean(salary)) #SOLUTION


full_stats = compute_statistics(full_data)
full_stats
```

Out[40]:
```
array([2.65365854e+01, 4.26977577e+06])
```





In [41]:
```
# TEST
stats = compute_statistics(full_data)
plt.close()
plt.close()
round(float(stats[0]), 2) == 26.54
```

Out[41]: True

In [42]:
```
# TEST
stats = compute_statistics(full_data)
plt.close()
plt.close()
round(float(stats[1]), 2) == 4269775.77
```

Out[42]: True

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.

```
BEGIN QUESTION
name: q32
```

In [43]:
```
convenience_sample = full_data.where("Age", are.below(22)) #SOLUTION
convenience_sample
```

Out[43]:

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 [44]:
```
# TEST
convenience_sample.num_columns
```

Out[44]: 11

In [45]:
```
# TEST
convenience_sample.num_rows
```

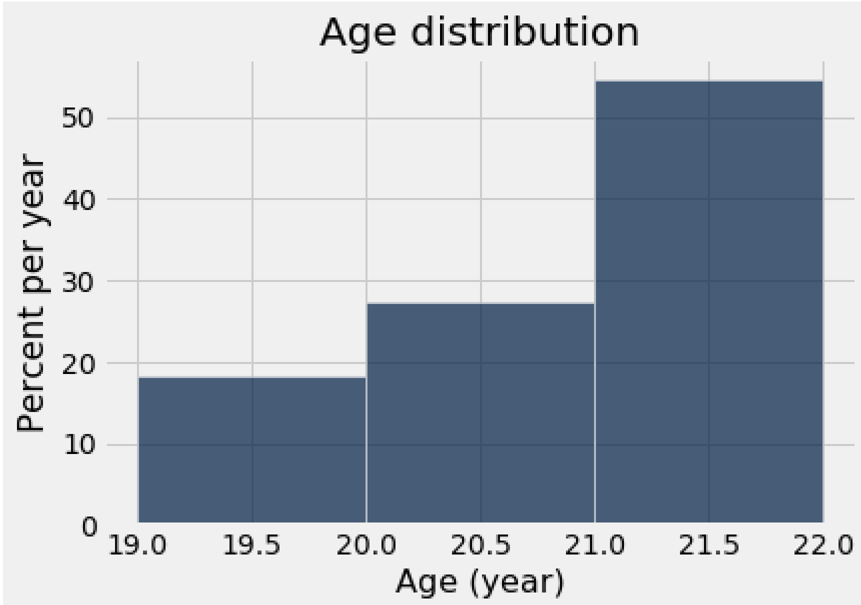
Out[45]: 44

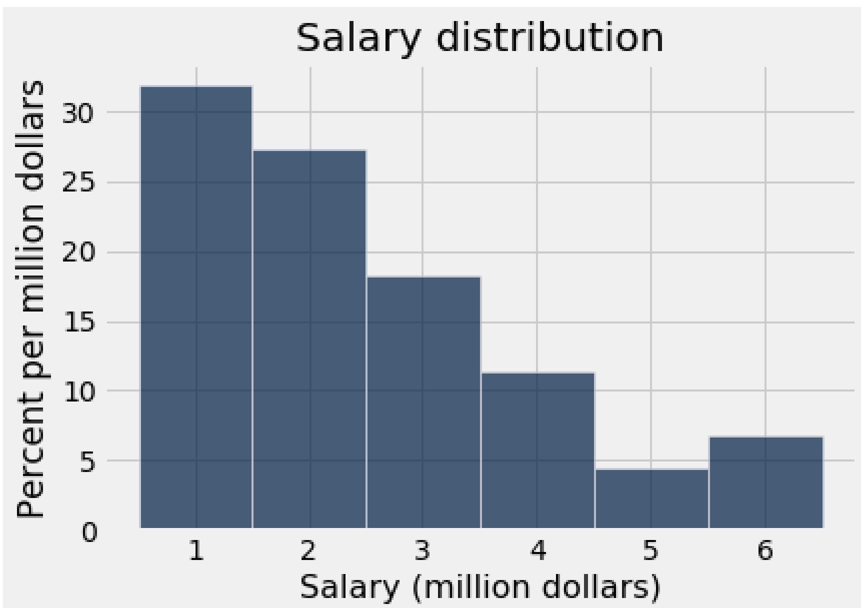
Question 3. Assign `convenience_stats` to an array of the average age and average salary of your convenience sample, using the `compute_statistics` function. Since they're computed on a sample, these are called *sample averages*.

```
BEGIN QUESTION
name: q33
```

In [46]:
```
convenience_stats = compute_statistics(convenience_sample) #SOLUTION
convenience_stats
```

Out[46]: array([2.03636364e+01, 2.38353382e+06])





In [47]:
```
# TEST
len(convenience_stats)
```

Out[47]: 2

In [48]:
```
# TEST
round(float(convenience_stats[0]), 2) == 20.36
```

Out[48]: True

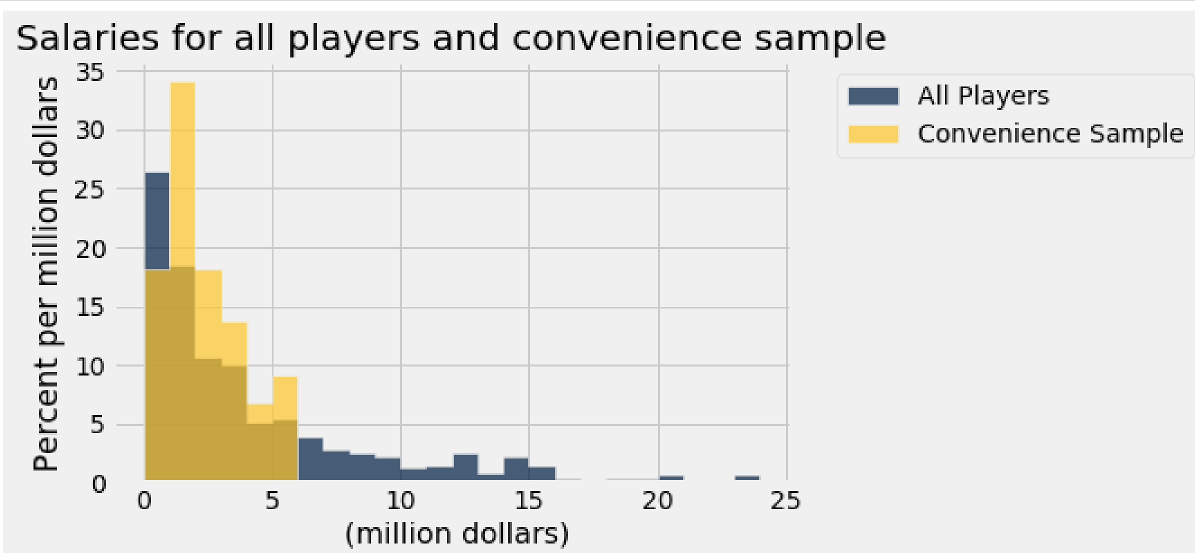
In [49]:
```
# TEST
round(float(convenience_stats[1]), 2) == 2383533.82
```

Out[49]:
```
True
```

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**.

In [50]:
```
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', firs
t_salary_in_millions)
    second_tbl_millions = second.drop('Salary').with_column('Salary', se
cond_salary_in_millions)
    max_salary = max(np.append(first_tbl_millions.column('Salary'), seco
nd_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).relabel
ed(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', 'Convenie
nce 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.

SOLUTION: No, the convenience sample does not give us an accurate picture of the salary of the full population of NBA players in 2014-2015. We would not expect it to, because it is biased towards players younger than 22.

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.

In [51]:
```
# Just run this cell

salary_data.sample(5)
```

Out[51]:

PlayerName	Salary
Pero Antic	1250000
Jason Terry	5850313
Luke Babbitt	981084
Ish Smith	981084
T.J. Warren	1953120

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.

In [52]:
```python
# Just run this cell

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

Out[52]:

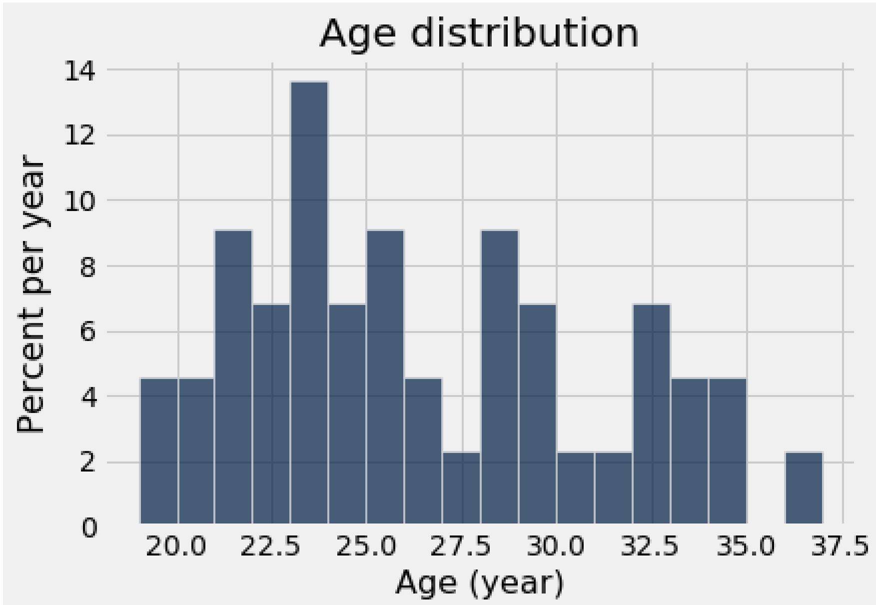
PlayerName	Salary
Patrick Beverley	915243
Jeff Ayres	1828750
Rudy Gay	19317326
Dwight Howard	21436271
P.J. Hairston	1149720

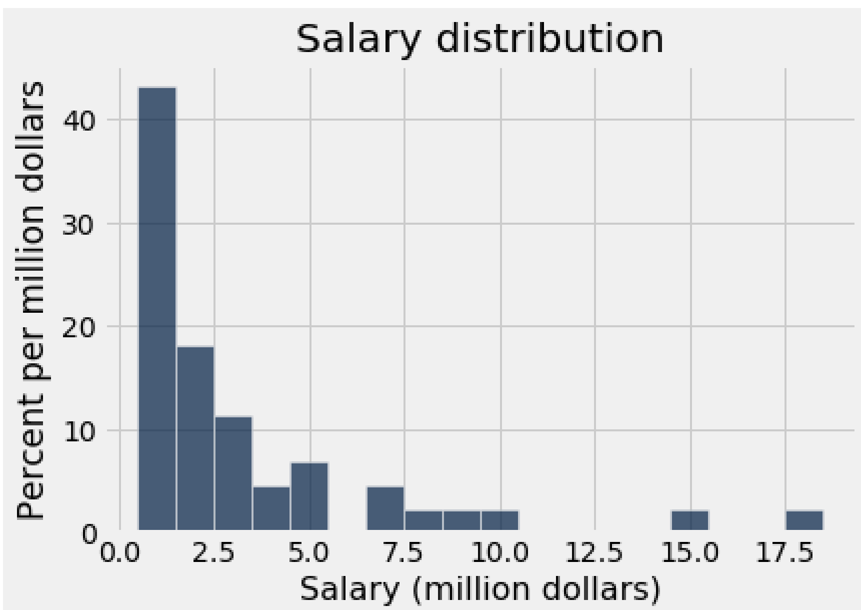
Question 5. Produce a simple random sample of size 44 from `full_data`. Run your analysis on it again. Run the cell a few times to see how the histograms and statistics change across different samples.

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

In [53]:
```
my_small_srswor_data = full_data.sample(44, with_replacement = False) #SOLUTION
my_small_stats = compute_statistics(my_small_srswor_data) #SOLUTION
my_small_stats
```

Out[53]:
```
array([2.59772727e+01, 3.31178255e+06])
```





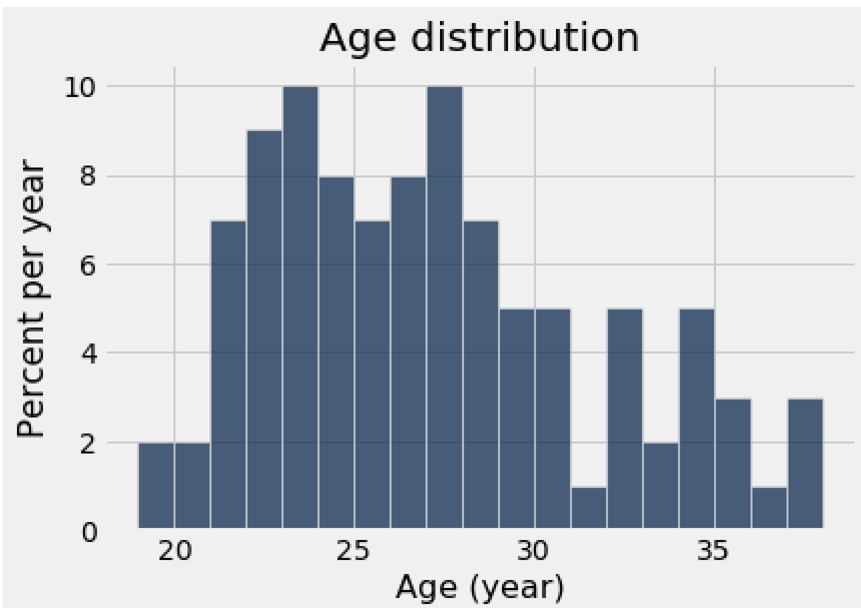
SOLUTION: The average age tends to stay around the same value as there is a limited range of ages for NBA players, but the salary changes by a sizeable factor due to larger variability in salary.

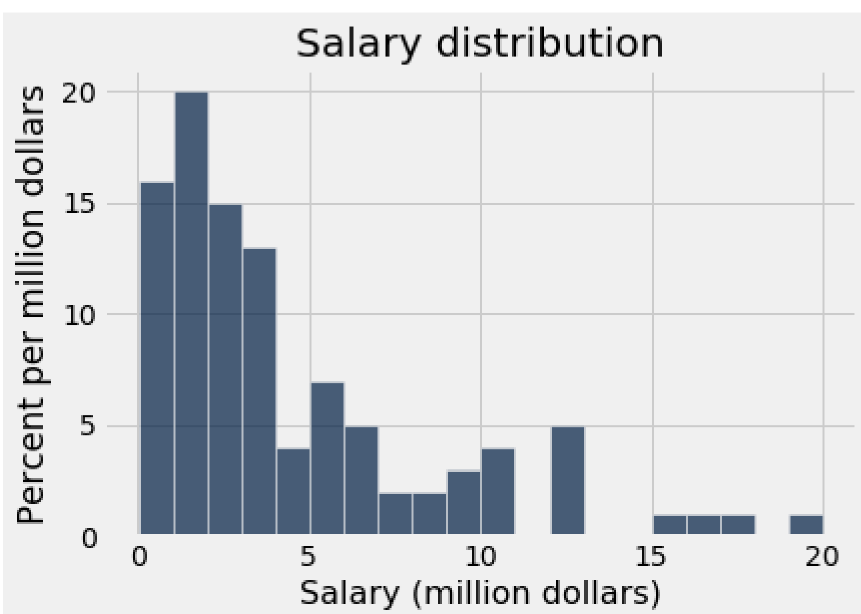
Question 6. As in the previous question, analyze several simple random samples of size 100 from `full_data`.

- 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?

In [54]:
```
my_large_srswor_data = full_data.sample(100, with_replacement = False) #
SOLUTION
my_large_stats = compute_statistics(my_large_srswor_data) #SOLUTION
my_large_stats
```

Out[54]:
```
array([2.66800000e+01, 4.47264256e+06])
```





SOLUTION: The average and histogram statistics seem to change less across samples of 100. They are closer to their true values, which is what we'd expect to see because we are sampling a larger subset of the population. The histogram and sample average for age seem closer to their true value or shape. We'd expect this because players' ages are less variable than their salaries.

Congratulations, you're done with Lab 5! Be sure to

- **Run all the tests** (the next cell has a shortcut for that).
- **Save and Checkpoint** from the `File` menu.
- **Run the cell at the bottom to submit your work**.
- And ask one of the staff members to check you off.

In [55]:
```
# For your convenience, you can run this cell to run all the tests at on
ce!
import os
_ = [ok.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]
```

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

In [ ]:
```
```