lab05

February 16, 2024

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab05.ipynb")
```

1 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.

Getting help on lab: Whenever you feel stuck or need some further clarification, find a GSI or tutor, and they'll be happy to help!

Here are the policies for getting full credit:

1. For students enrolled in in-person **Regular Labs**, you will receive full lab credit by **attending** lab discussion, making significant progress on the lab notebook, and submitting it to Gradescope by **5pm on the Friday the same week it was released**.
2. For students enrolled in **Self Service**, you will receive full lab credit by **completing** the notebook, passing the test cases, and **submitting** it to Gradescope by **5pm on the Friday the same week it was released**.

Submission: Once you're finished, run all cells besides the last one, select File > Save Notebook, and then execute the final cell. The result will contain a zip file that you can use to submit on Gradescope.

Let's begin by setting 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')
```

1.1 1. Nachos and Conditionals

In Python, the boolean is a data type with only two possible 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.

[ ]:
```
3 > (1 + 1)
```

[ ]:
```
True
```

We can even assign the result of a comparison operation to a variable. Note that `==` and `=` are **not** the same!

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

[ ]:
```
True
```

Just like arithmetic operators can be applied on every item of an array, comparison operators can also be used on arrays to compare an entire array with some value. The output of this comparison 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(...)`.

1.1.1 `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. *Tip:* To keep running a cell multiple times you can use the keyboard shortcut `ctrl` + `return`.

[ ]:
```
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. Sampling **with replacement** means that after an element is drawn, it is replaced back to where you are sampling from and can be drawn again in the future. If we sample `n` times with replacement, each time, every element has an equal chance of being selected.

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

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

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

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

1.1.2 `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 1.1 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 manually enter the final 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') #SOLUTION
number_cheese
```

[ ]:
```
3
```

[ ]:
```
grader.check("q11")
```

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. `elif` can only be used if an `if` clause precedes it. 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 1.2 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`. Use the if statement to do this (do not directly reassign the variable `say_please`).

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

[ ]:
```
""" # 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
```

[ ]:
```
'More please'
```

[ ]:
```
grader.check("q12")
```

Question 1.3 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.

Nacho Type	Reaction
cheese	Cheesy!
salsa	Spicy!
both	Wow!
neither	Meh.

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

[ ]:
```
""" # 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
```

[ ]:
```
'Spicy!'
```

[ ]:
```
grader.check("q13")
```

Question 1.4 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: Consider using the `apply` method, which returns an array.

[ ]:
```
ten_nachos_tbl = Table().with_column('Nachos', ten_nachos)
ten_nachos_reactions = ten_nachos_tbl.with_column('Reactions', ten_nachos_tbl.
 ↪apply(nacho_reaction, 'Nachos')) #SOLUTION
ten_nachos_reactions
```

[ ]:
```
Nachos   | Reactions
neither | Meh.
cheese   | Cheesy!
both     | Wow!
both     | Wow!
cheese   | Cheesy!
salsa    | Spicy!
both     | Wow!
neither | Meh.
cheese   | Cheesy!
both     | Wow!
```

[ ]:
```
grader.check("q14")
```

Question 1.5 Using code, find the number of 'Wow!' reactions for the nachos in `ten_nachos_reactions`.

[ ]:
```
number_wow_reactions = np.count_nonzero(ten_nachos_reactions.
  ↪column('Reactions') == 'Wow!') #SOLUTION
number_wow_reactions
```

[ ]:
```
4
```

[ ]:
```
grader.check("q15")
```

1.2 2. Simulations and For Loops

Using a `for` statement, we can perform a task multiple times. This is known as iteration. The general structure of a for loop is:

`for <placeholder> in <array>:` followed by indented lines of code that are repeated for each element of the `array` being iterated over. You can read more about for loops here.

NOTE: We often use `i` as the `placeholder` in our class examples, but you could name it anything! Some examples can be found below.

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. Whichever name we pick, we need to use it consistently 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 2.1 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.

Hint: How can you use `longer_than_five` to keep track of the number of words that are more than five letters long?

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

""" # BEGIN PROMPT
longer_than_five = ...

for ... in ...:
    ...
"""; # 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
```

[ ]:
```
35453
```

[ ]:
```
grader.check("q21")
```

Another way we can use `for` loops is to repeat lines of code many times. Recall the structure of a `for` loop:

`for <placeholder> in <array>:` followed by indented lines of code that are repeated for each element of the array being iterated over.

Sometimes, we don't care about what the value of the placeholder is. We instead take advantage of the fact that the `for` loop will repeat as many times as the length of our array. In the following cell, we iterate through an array of length 5 and print out "Hello, world!" in each iteration, but we don't need to use the placeholder `i` in the body of our `for` loop.

[ ]:
```
for i in np.arange(5):
    print("Hello, world!")
```

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

Question 2.2 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 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
```

[ ]:
```
8582
```

[ ]:
```
grader.check("q22")
```

1.3 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.

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

[ ]:
```
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)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

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, we may have to come up with these estimates based only on a smaller sample. The numerical property of the population is known as a **parameter**, and the estimate is known as a **statistic** (e.g. the mean or median). 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.

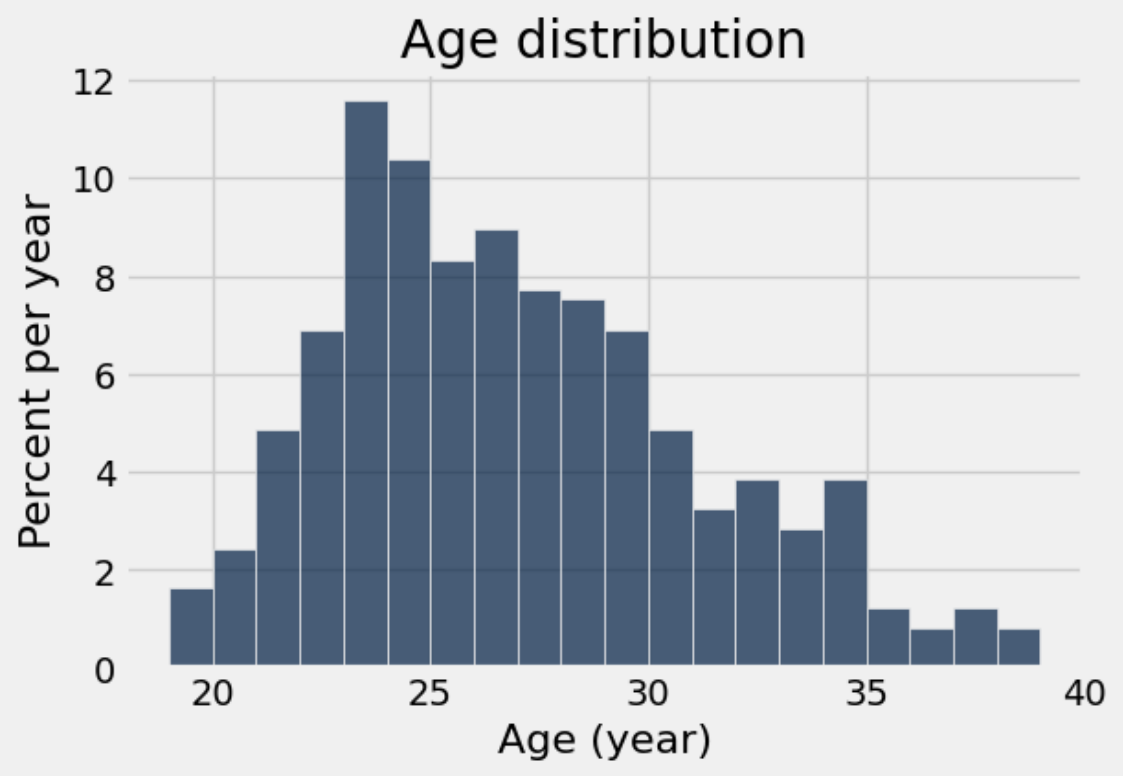
We've defined the `histograms` function below, which takes a table with columns `Age` and `Salary` and draws a histogram for each one. It uses bin widths of 1 year for `Age` and $1,000,000 for `Salary`.

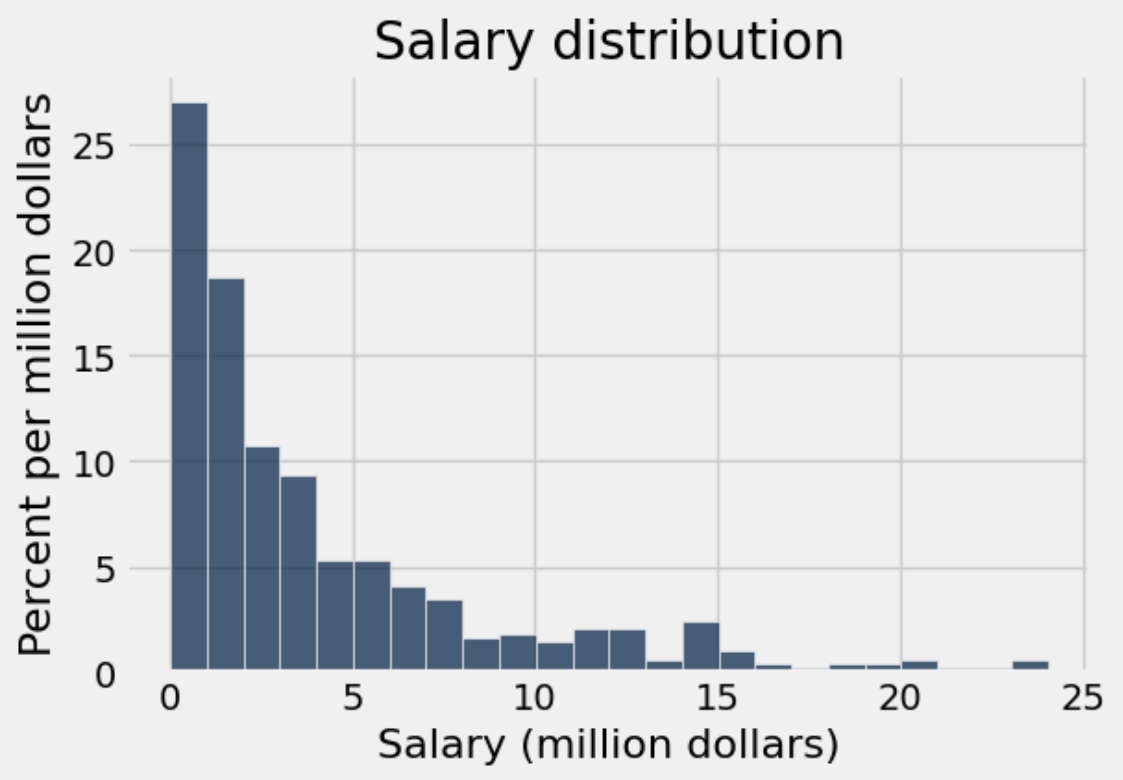
[ ]:
```
def histograms(t):
    ages = t.column('Age')
    salaries = t.column('Salary')/1000000
    t1 = t.drop('Salary').with_column('Salary', salaries)
    age_bins = np.arange(min(ages), max(ages) + 2, 1)
    salary_bins = np.arange(min(salaries), max(salaries) + 1, 1)
    t1.hist('Age', bins=age_bins, 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
```





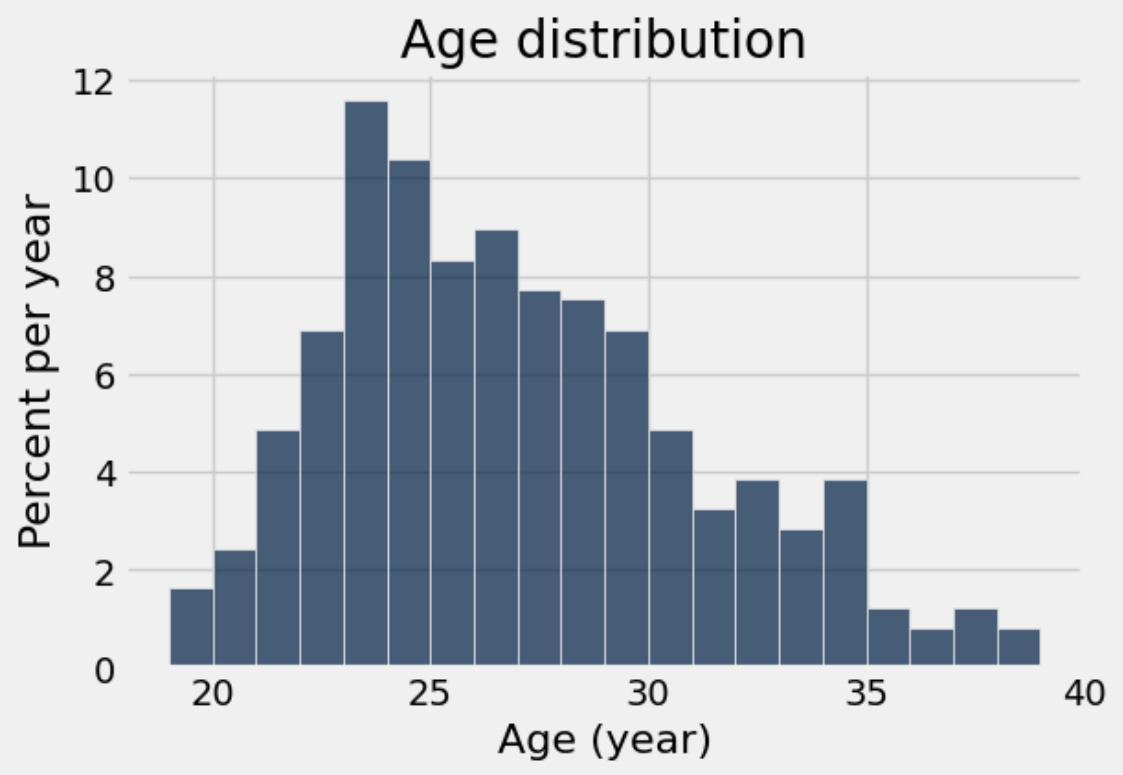
Question 3.1. Create a function called `compute_statistics` that takes a table containing an "Age" column and a "Salary" column and: - Draws a histogram of ages - Draws a histogram of salaries - Returns a two-element array containing the average age and average salary (in that order)

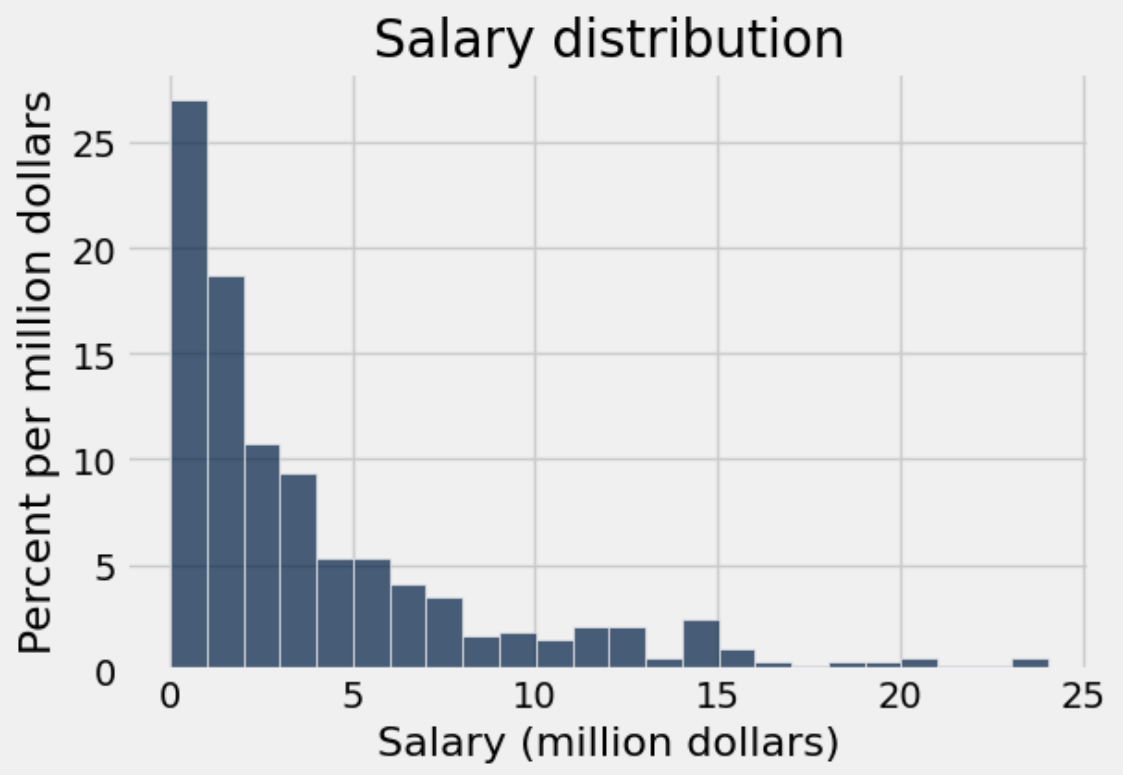
You can call the `histograms` function to draw the histograms!

[ ]:
```
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
```

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


Age distribution
Percent per year
0
2
4
6
8
10
12
20
25
30
35
40
Age (year)


Salary distribution
Percent per million dollars
0
5
10
15
20
25
0
5
10
15
20
25
Salary (million dollars)

[ ]:
```
grader.check("q31")
```

1.3.1 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.

1.3.2 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.

1.3.3 `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. Sampling with replacement means for any row selected randomly, there is a chance it can be selected again if we sample multiple times. `Sample` takes in the sample size as its argument and returns a **table** with only the rows that were selected. This differs from `np.random.choice`, which takes an array and outputs a random value from the array.

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)
```

[ ]:
```
PlayerName         | Salary
Gerald Green       | 3500000
Larry Drew         | 507336
Derrick Rose       | 18862876
Amar'e Stoudemire  | 23410988
Mike Scott         | 3333333
```

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
Tony Parker        | 12500000
```

```
Archie Goodwin     | 1112280
Leandro Barbosa    | 1448490
Charlie Villanueva | 1316809
Marcus Thornton    | 8575000
```

Question 3.2 Produce a simple random sample **without** replacement of size **44** from `full_data`. Then, run your analysis on it again by using the `compute_statistics` function you defined above. Run the cell a few times to see how the histograms and statistics change across different samples.

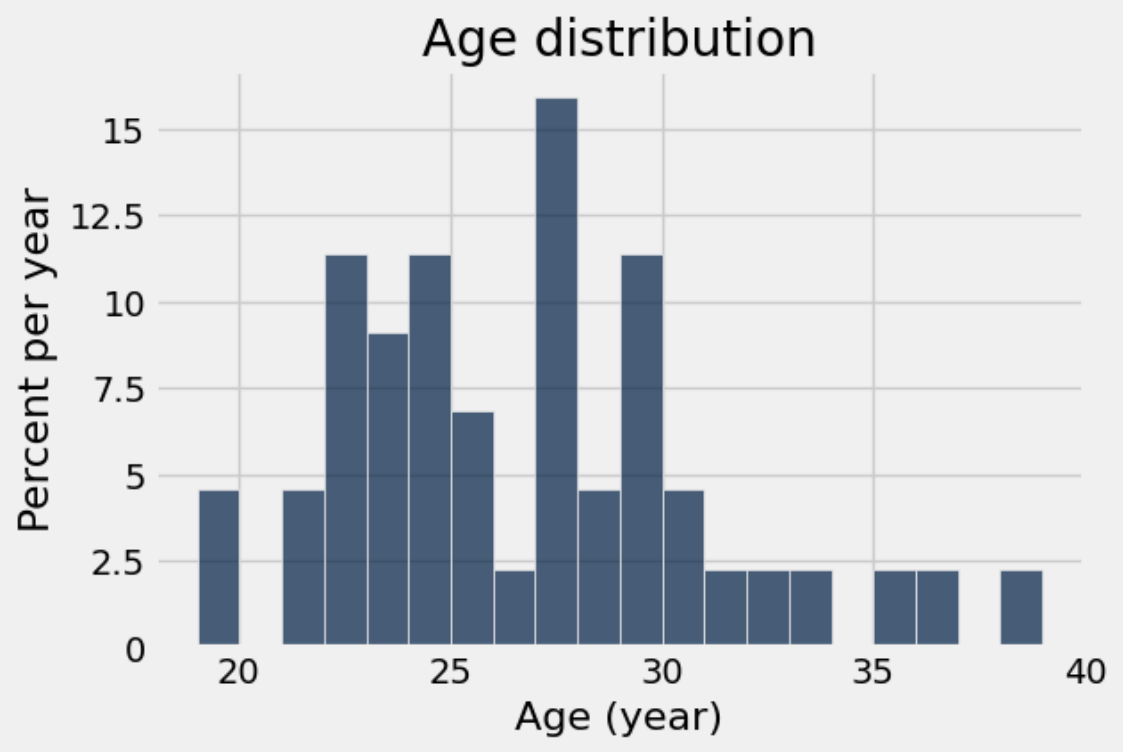
- How much does the average age change across samples?
- What about average salary?

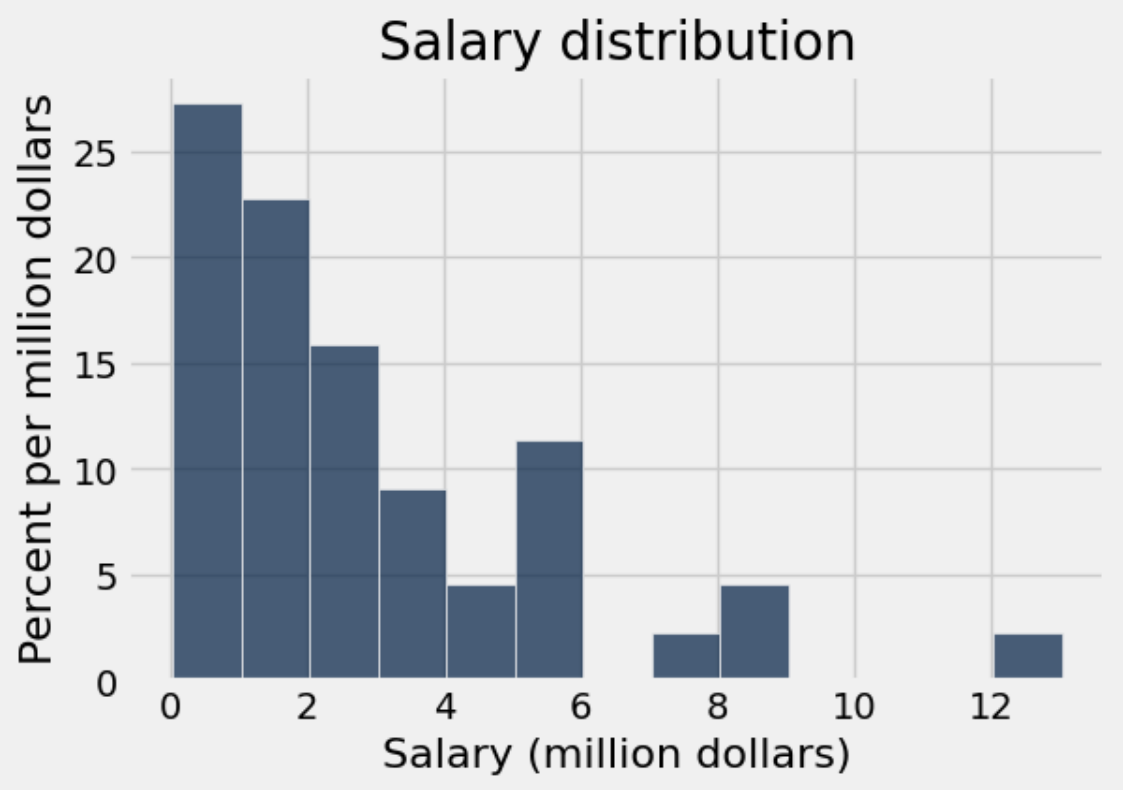
(FYI: srs = simple random sample, wor = without replacement)

Type your answer here, replacing this text.

[ ]:
```
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
```

[ ]:
```
array([  2.63181818e+01,   2.81983275e+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.

1.4 4. More Random Sampling Practice

More practice for random sampling using `np.random.choice`.

1.4.1 Simulations and For Loops (cont.)

Question 4.1 We can use `np.random.choice` to simulate multiple trials.

Stephanie decides to play a game rolling a standard six-sided die, where her score on each roll is determined by the face that is rolled. She wants to know what her total score would be if she rolled the die 1000 times. Write code that simulates her total score after 1000 rolls.

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

[ ]:
```
""" # BEGIN PROMPT
possible_point_values = ...
num_tosses = 1000
simulated_tosses = ...
total_score = ...
```

```
"""; # END PROMPT
# BEGIN SOLUTION NO PROMPT
possible_point_values = np.arange(1, 7)
num_rolls = 1000
simulated_rolls = np.random.choice(possible_point_values, num_rolls)
total_score = sum(simulated_rolls)
# END SOLUTION
total_score
```

[ ]:
```
3581
```

[ ]:
```
grader.check("q41")
```

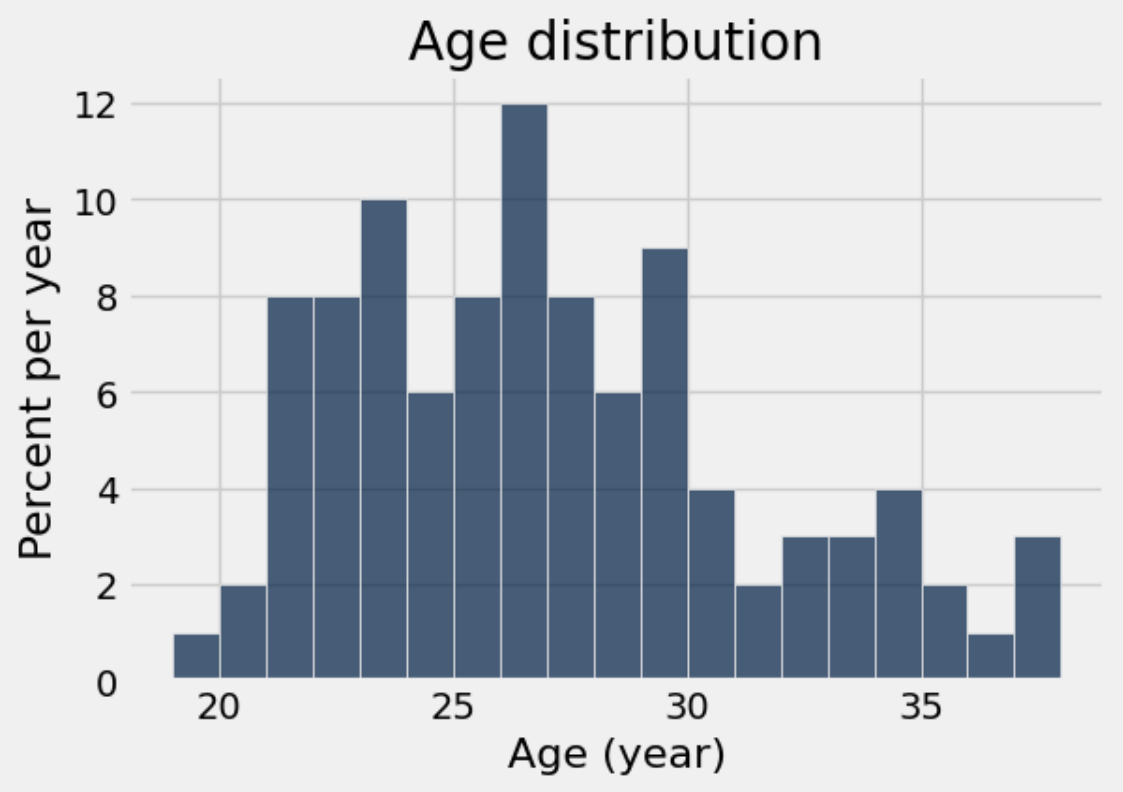
1.4.2 Simple random sampling (cont.)

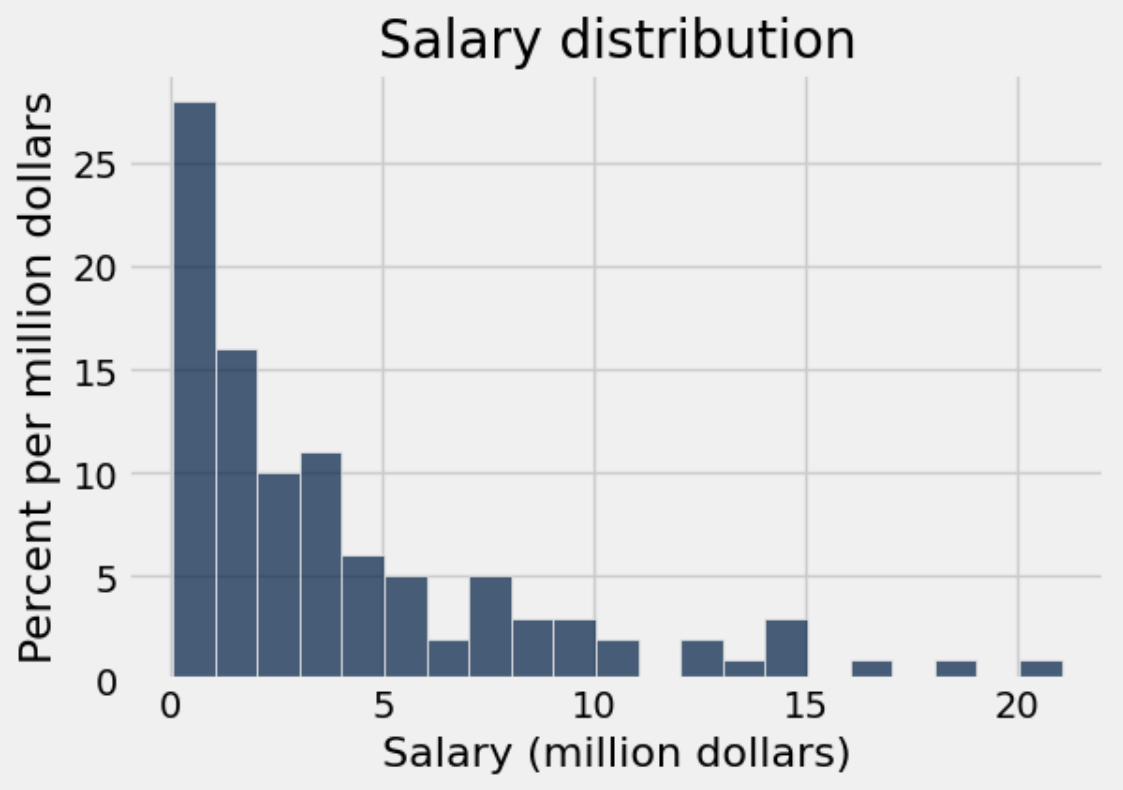
Question 4.2 As in the previous question, analyze several simple random samples of size 100 from `full_data` by using the `compute_statistics` function.
- Do the histogram shapes seem to change more or less across samples of 100 than across samples of size 44?
- Are the sample averages and histograms closer to their true values/shape for age or for salary? What did you expect to see?

Type your answer here, replacing this text.

[ ]:
```
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
```

[ ]:
```
array([  2.66400000e+01,   4.22556384e+06])
```


Age distribution
Percent per year
0
2
4
6
8
10
12
20
25
30
35
Age (year)


Salary distribution
Percent per million dollars
0
5
10
15
20
25
0
5
10
15
20
Salary (million dollars)

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.

Lincoln is very happy that you finished the lab!

You're done with lab!

Important submission information: - **Run all the tests** and verify that they all pass - **Save** from the **File** menu - **Run the final cell to generate the zip file** - **Click the link to download the zip file** - Then, go to Gradescope and submit the zip file to the corresponding assignment. The name of this assignment is "Lab XX Autograder", where XX is the lab number – 01, 02, 03, etc.

- If you finish early in Regular Lab, **ask one of the staff members to check you off**.

It is your responsibility to make sure your work is saved before running the last cell.

1.5 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False, run_tests=True)
```