Name:______________________________ **Partner:** _________________________

Python Activity 38: Hash Function

Learning Objectives
Students will be able to:
Content:
- Describe how a **Hash Table algorithm** works.
- Define a **Hash Function's** role in dictionaries and sets.
- List the main requirements of a good Hash Function.

Process:
- Write a Hash Function for user-defined types..

Prior Knowledge
- Activities 1-28, Dictionaries, Sets

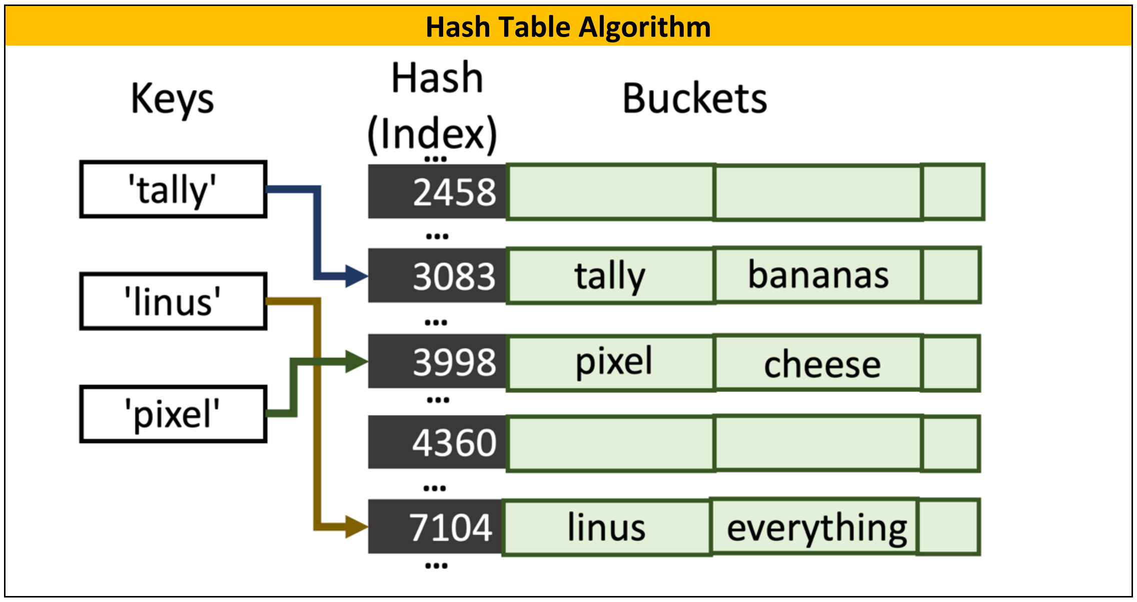
If you encounter any issues/typos, let Iris know! Questions? Ask Iris or the POGILing forum

Concept Model:

FYI: A python Dictionary is an implementation of a ***Hash Table algorithm.*** Hash Tables are an array of items which calculate an index from the item's key and use this index to place the data into the array. A ***Hash Function*** is the function that calculates this index (i.e., a ***Hash***) from the item's key.

The following diagram of a Hash Table represents the following python dictionary (a mapping of dog names as keys to their favorite food as values):

```
dogFood = {     'tally':'bananas',   'linus':'everything',
                'pixel':'cheese'     }
```



1. Where are the Hash Table keys located in this diagram?

____

____

2. If we wanted to look for the value mapped to `'tally'` what Hash/index would we look for?

____

FYI: For a given value, its Hash will always be the same, although it may change between successive python sessions.

3. What is stored in the bucket located at index 7104?

____

4. What appears to be stored in the Buckets in the Hash Table diagram?

____

____

Let's add another key-value mapping to our dictionary:

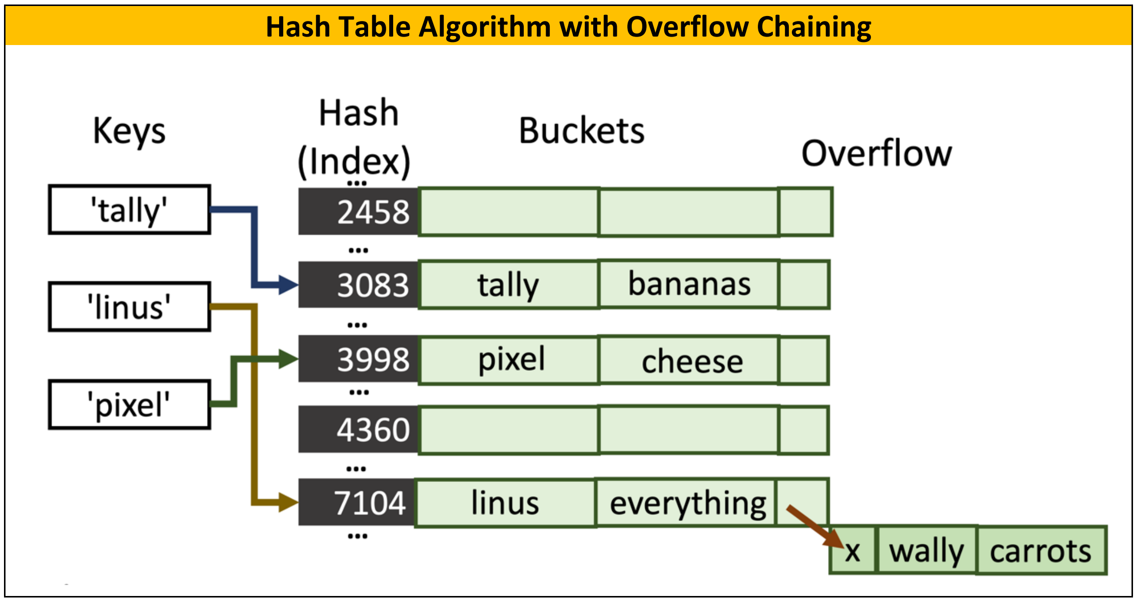
```
dogFood['wally'] = 'carrots'
```

5. Let's assume the Hash for `'wally'` is 7104. Why might this be a problem?

____

____

One option to resolve this issue is to link the new data with the overlapping hash to the previous data at that same Hash index, like in the diagram below:



6. In this example, how many key-value pairs are stored at Hash 7104? ____________

FYI: Different values may produce the same Hash. In the context of a Hash Table algorithm, we call this a ***collision***.

7. If we want to `print('dogFood['wally'])` python has to first retrieve the Hash for `'wally'`, then look at that index in the Hash Table, then iterate through all the items in the buckets located at that hash until it finds the `'wally'` key. How many items does python have to look at before it find the `'wally'` key?
8. If we add `dogFood['annie']='hamburger'` to our dictionary, and the Hash for `'annie'` is 3998, what are the steps necessary to `print('dogFood['annie'])` ?

Critical Thinking Questions:

FYI: In python, **`hash(obj)`** is a built-in function that produces an integer value that represents the Hash for `obj`.

1. Examine the sample code below from interactive python:

Interactive Python

```
0 >>> s = 'hello world'
1 >>> s2 = 'hello world'
2 >>> s is s2
3 False
4 >>> hash(s)
5 4963799451833479185
6 >>> hash(s2)
7 4963799451833479185
```

a. What would `>>> s == s2` output?

b. Are `s` and `s2` the same object instance, how do you know?

c. What does the value outputted on line 5 represent?

d. What does the value outputted on line 7 represent?

e. What would `>>> hash(s) == hash(s2)` output?

f. What does this tell us about objects that share a value, but are different instances?

g. What might `>>> hash('hello world')` output?

2. Examine the sample code below from interactive python:

Interactive Python

```
>>> s = 'hello world'
>>> t = s + '!'
>>> hash(s)
4963799451833479185
hash(t)
-8774050965770600213
```

a. How does `t` differ from `s`? How many characters differ?

b. How does `hash(t)` differ from `hash(s)`? How many digits differ?

c. What does this tell us about the Hashes generated for objects that are very similar?

3. Examine the sample code below from the Terminal:

Terminal

```
0 -> python3
1 >>> s = 'hello world'
2 >>> hash(s)
3 4963799451833479185
4 >>> exit() # exits interactive python
5 -> python3 # starts a new session of interactive python
6 >>> s = 'hello world'
7 >>> hash(s)
8 4686556288558268365
```

a. How does `s` on line 1 differ from `s` on line 6?

b. How does the value from `hash(s)` on line 2 differ from the value from `hash(s)` on line 7? ____________________

c. What happens between lines 2 and 7 that might explain this?

d. What does your answer in 3c tell us about the Hashes generated for values across different python sessions?

4. Examine the sample code below from interactive python:

Interactive Python

```
0 >>> hash(1)
1 1
2 >>> hash(2)
3 2
4 >>> hash(1000000000000000000)
5 1000000000000000000
6 >>> hash(10000000000000000000)
7 7766279631452241196
```

a. How do the arguments and values returned on lines 0 & 1, and 2 & 3 differ?

b. What might be the output for `hash(3)`?

c. What does this say about Hashes for integers?

d. How do the arguments and values returned on lines 4 & 5 differ?

e. How do the arguments and values returned on lines 6 and 7 differ?

f. What does this say about Hashes for large integers?

5. Examine the sample code below from interactive python:

Interactive Python

```
0 >>> class Flower:
1 ...       slots = ['sepals', 'petals']
2 ...       def __hash__(self):
3 ...            return self.petals + self.sepals
4 >>> rose = Flower()
5 >>> rose.petals = 10
6 >>> rose.sepals = 5
7 >>> hash(rose)
8 15
```

a. If we were to call `rose.__hash__()` (*underscore guilt!*), what would the output be?

b. How does the output from `rose.__hash__()` differ from the output of `hash(rose)` on line 7?

c. What special method is likely being called on line 7?

d. Write a few lines of python to generate another `Flower` with the same Hash as `rose` using different values for `petals` and `sepals`:

____

____

____

e. How common might the Hash value for `rose` be?

____

f. If it's super common for different flowers to have the same sum of `petals` & `sepals`, how might we modify `Flower`'s hash function to have fewer collisions?

____

g. Write a few lines of python to make a better `__hash__` method for `Flower`:

____

____

____

____

FYI: A good Hash Function produces equivalently sized integers for values evenly distributed to decrease collisions. However, if there are no collisions with a large dataset, this may be inefficient.

Application Questions: Use the Python Interpreter to check your work

1. Write a `__hash__(self)` method for the `Course` class which represents a college course with a department prefix (`_dept`), course number (`_num`) and target GPA average (`_gpa`) as slots:

```
class Course:
    __slots__ = ['_dept', '_num', '_gpa']
def __hash__(self):
```

____

____

____

____

____

____

____

Will your Hash Function produce integers?

____

Will your Hash Function produce the same value for objects with the same value?

Will your Hash Function produce evenly distributed values for `Course` objects?

2. Write a `__hash__(self)` method for the `Automobile` class which represents a car with a `make` (i.e., brand like Tesla) and a `model` (like model S):

```
class Automobile:
     __slots__ = ['_make', '_model']
def __hash__(self):
```

Will your Hash Function produce integers?

Will your Hash Function produce the same value for objects with the same value?

Will your Hash Function produce evenly distributed values for `Automobile` objects?