Recap some of the Dict tricks

```
dat = {'CIE': 1, 'APH': 2, 'JSPS': 3 }
                      # Assign a dict key/value pairs
dat = {}              # Empty dict
dat['CIE']            # Access a value of the given key
dat.keys()            # Get all keys
dat['JAM'] = 4        # update value if the given key exists
                      # otherwise, add a new key/value pair
del dat['CIE']        # Remove an item of the given key
dir(dat)              # See all attributes and methods
```

!Caution! similar to a list assignment, a dict assignment is a reference assignment, not a new copy, i.e., `d1 = dat` is different from `d2 = dat.copy()`.

***** Note: The order of a dictionary may seem strange. Do not worry about it. Dictionary emphasizes access using keys, rather than an order.**

Don't worry if you don't get all the tricks, just have fun playing with them.

==========

Use `if __name__ == '__main__':` for better code organization and allowing smooth autograding.

P1. Write a function named `make_dict` taking no argument, creating a dict having key-value pairs of 'H': 1, 'He': 2, 'C': 6, 'N': 7, 'O': 8, and returning the created dict.

Example

When invoked by

```
atomic = make_dict()
```

```
print(atomic)
```

it results

```
======================================================================
{'He': 2, 'H': 1, 'N': 7, 'C': 6, 'O': 8}
======================================================================
```

P2. Write a function named `get_dict_val`. The function takes 3 arguments: a dict variable and two keys. Then, it gets values of those keys and returns a tuple of both values.

Hint: tuple looks like `(a, b)`.

Example

When invoked by

```
visits = {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}
visited = get_dict_val(visits, 'Arequipa', 'Denali NP')
print(visited)
```

it results

```
======================================================================
(2006, 2007)
======================================================================
```

P3. Write a function named `have_visited` taking a dict variable and a key as arguments, checking if the given key is in the dict, and returning a boolean result (`True` / `False`) of the checking.

Hint: operator `in` *may be handy. Try* `'Taktsang' in {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}` *vs* `'Samakand' in {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}`

Examples

When invoked by

```
visits = {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}
visited = have_visited(visits, 'Taktsang')
print(visited)
```

it results

```
========================================================================
True
========================================================================
```

When invoked by

```
visits = {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}
visited = have_visited(visits, 'Serengeti NP')
print(visited)
```

it results

```
========================================================================
False
========================================================================
```

Note: boolean is another data type. Check if data type is boolean by function `type`, e.g., `type(visited)`.

P4. Write a function named `add_place` taking a dict variable, a key, and a value as arguments, add key-value pair into the dict (or update the dict if the pair has been there), and returning an updated dict.

Examples

When invoked by

```
visits = {'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2015}
visits = add_place(visits, 'Taktsang', 2014)
print(visits)

visits = add_place(visits, 'Provins', 2013)
print(visits)
```

it results

```
======================================================================
{'Denali NP': 2007, 'Arequipa': 2006, 'Taktsang': 2014}
{'Denali NP': 2007, 'Provins': 2013, 'Arequipa': 2006, 'Taktsang':
2014}
======================================================================
```

P5. Dict is natural for counting. Write a function named `word_freq` to take a string as its argument, count occurrences of each word, and return the count as a dict. Discard any word of length 1 or less, e.g., *"Our class is a programming class."* should be counted to: `{'Our': 1, 'class': 2, 'is': 1, 'programming': 1}`. There will be no count for `'a'` (too short).

Use the P5 template. (`WordCount_Template.py`; note: template is only to encourage intended learning skills and allow smooth auto-grading.)

Example

When invoked by

```
txt = "Evil is done by oneself; " + \
        "by oneself is one defiled. \n " + \
        "Evil is left undone by oneself; " + \
        "by oneself is one cleansed. \n " + \
        "Purity and impurity are one's own doing. \n" + \
        "No one purifies another. \n" + \
        "No other purifies one."
        # excerpt from Attavagga: Self, www.accesstoinsight.org
print(txt)
wf = word_freq(txt)
print('\nCount:')
print(wf)
```

it results

```
======================================================================
```

```
Evil is done by oneself; by oneself is one defiled.

 Evil is left undone by oneself; by oneself is one cleansed.

 Purity and impurity are one's own doing.

No one purifies another.

No other purifies one.


Count:

{'doing': 1, 'another': 1, 'purifies': 2, 'own': 1, 'undone': 1, 'is':
4, 'other': 1, 'impurity': 1, 'left': 1, 'by': 4, 'oneself': 4, 'No':
2, 'Purity': 1, 'one': 4, 'cleansed': 1, 'and': 1, 'are': 1, 'done':
1, "one's": 1, 'Evil': 2, 'defiled': 1}

======================================================================
```

Note: punctuation has been cleaned off the keys, i.e., no key has a period, comma, or semicolon. Upper case, plurality, and tense stay.

P6. (difficulty 3*) Estimated probability using counting. Given a count as a dictionary data type (as from P5), write a function named `est_prob` to calculate estimate probabilities of each word, and return another dictionary of the estimate probabilities.

Example

When invoked by

```
wcount = {'culinary': 3, 'history': 5, 'dynasty': 1, 'silk': 2,
'Buddhist': 2, 'caves': 2, 'wall': 3, 'history': 4}

ps = est_prob(wcount)

print('Text:', wcount)

print('Count:', ps)
```

it results

```
======================================================================

Text: {'wall': 3, 'Buddhist': 2, 'silk': 2, 'dynasty': 1, 'culinary':
3, 'caves': 2, 'history': 4}
```

```
Count: {'history': 0.23529411764705882, 'caves': 0.11764705882352941,
'silk': 0.11764705882352941, 'Buddhist': 0.11764705882352941,
'culinary': 0.17647058823529413, 'wall': 0.17647058823529413,
'dynasty': 0.058823529411764705}

======================================================================
```

Note: The order of a dictionary may seem strange, but don't worry about it. Dictionary emphasizes access using keys, rather than order.

P7. (difficulty 4*) Baleen whales, e.g., gray whales, humpback whales, blue whales, feed on krill, plankton, and tiny marine organisms. A blue whale can eat as much as 40 million krill a day, approximately 3600 kg of krill. Krill are tiny shrimps found in oceans.

Given a dictionary of baleen whales and their average daily diet on krill during feeding season and a dictionary of whale counts observed during a field trip, write a function named `krill_consumption` to estimate the total amount of krill consumed daily by the observed whales during feeding season.

Examples

When invoked by

```
feeding = {'Humpback whale': 2000, 'Gray whale': 1500, 'Bowhead
whale': 2500, 'Blue whale': 3600}

whales = {'Humpback whale': 8, 'Gray whale': 3, 'Bowhead whale': 1,
'Blue whale': 12}


total_consum = krill_consumption(feeding, whales)

print('Estimate daily consumption: %d kg of krill'%total_consum)
```

it results

```
======================================================================

Estimate daily consumption: 66200 kg of krill

======================================================================
```

When invoked by

```
feeding = {'Humpback whale': 2000, 'Gray whale': 1500, 'Bowhead
whale': 2500, 'Blue whale': 3600}

whales = {'Humpback whale': 8, 'Gray whale': 3}


total_consum = krill_consumption(feeding, whales)

print('Estimate daily consumption: %d kg of krill'%total_consum)
```

it results

```
======================================================================

Estimate daily consumption: 20500 kg of krill

======================================================================
```

When invoked by

```
feeding = {'Humpback whale': 2000, 'Gray whale': 1500, 'Bowhead
whale': 2500, 'Blue whale': 3600}

whales = {'Bowhead whale': 80000}


total_consum = krill_consumption(feeding, whales)

print('Estimate daily consumption: %d kg of krill'%total_consum)
```

it results

```
======================================================================

Estimate daily consumption: 200000000 kg of krill

======================================================================
```

Note: given a shorter dictionary of whale counts, the function still works fine.

P8. (difficulty 4*) A kinetic power of flowing water can be estimated from

$$P = \frac{1}{2A^2}\,\rho Q^3,$$

where P is a kinetic power (in w), A is an estimate cross-section area (in m^2), ρ is a water density (in kg/m^3), and Q is a flow rate (in m^3/s).

Write a function named `mighty_river` to take in an argument: a dictionary of river information (having a key being a river name and a value being a list of a cross-section area, water density, and a flow rate), calculate the estimated power, and return the result in another dictionary using a river name as a key and calculated power as a value.

Use the P8 template. (`River_Template.py`; note: template is only to encourage intended learning skills and allow smooth auto-grading.)

Example

When invoked by

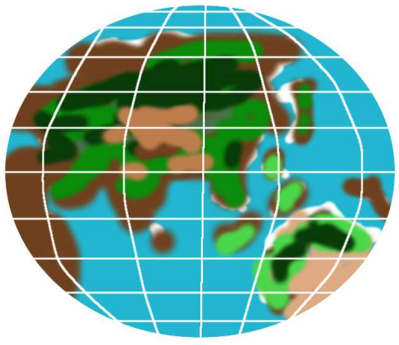
```
mighties = {'Amazon': [1.2e6, 1100, 210000],
                'Congo':  [2e6, 1150, 41200],
                'Yangtze': [800e3, 1200, 30000]}


river_power = mighty_river(mighties)
print(river_power)
```

it results

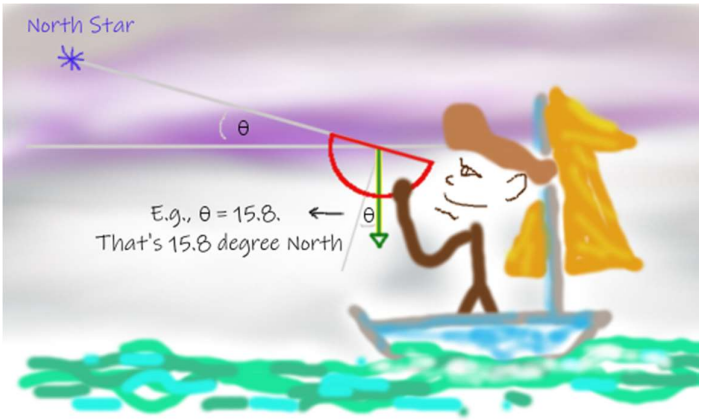
```
======================================================================
{'Amazon': 3537187.5, 'Congo': 10053.0884, 'Yangtze': 25312.5}
======================================================================
```

P9. Latitude and longitude are units as coordinates to pinpoint a location on earth surface. Latitude is described by imaginary horizontal lines. The 0-degree line is at the equator. The other lines lie parallel to the equator. The North Pole is at 90 degree north. The South Pole is at 90 degree south. Longitude is described by imaginary vertical lines running from the North Pole to the South



Pole. The 0-degree line passes through Greenwich, England. East/West is to specify whether the line located east or west of Greenwich.

Before the advent of GPS technology, sailors must learn skills to locate their whereabouts. For example, latitude can be deduced from the angle from the horizontal level to the Polaris when they are in the northern hemisphere, or to the South Star (*Sigma Octantis*) when they are in the southern hemisphere.



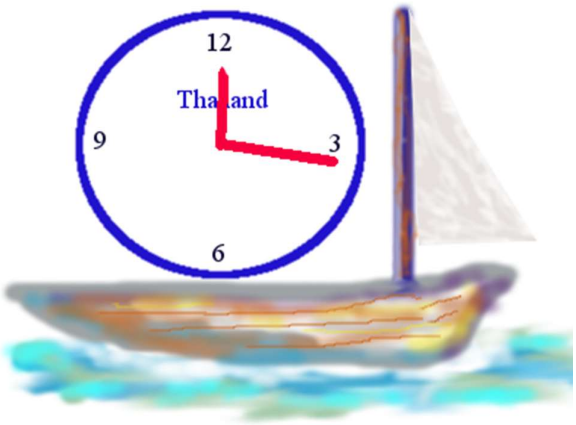
One way to deduce longitude is based on time at solar noon. The solar noon is the time when the shadow appears the shortest. Time at solar noon will be compared to 12:00pm GMT, which is the time at solar noon in Greenwich. That is, 4 minutes difference correspond to 1 degree longitude. If the GMT at the location is before 12:00pm GMT, the location lies east of Greenwich. Otherwise, the location lies west.

Write a function named `sailor`. The function takes a GMT at solar noon at the location in a dictionary of format like `{'H': 5, 'M': 40, 'am/pm': 'am'}`, figure out the longitude, and returns the longitude in tuple of longitude degree and direction (East/West), e.g., `(101.0, 'East')`. Note midnight is specified as `{'H': 00, 'M': 00, 'am/pm': 'am'}`.

Hint: (1) it is easier in minutes; (2) pm is just 12-hour ahead of am; (3) work it step by step.

360 degree (around the world) = 24 hours.
Therefore, 1 degree = (24*60)/360 = 4 min.

If 0 degree (at Greenwich) sees solar noon at 12:00pm GMT, then 1 degree East sees its solar noon at 11:56am GMT (4 min before 0 degree).



Solar noon (time at shadow casted shortest) is observed at 12:16pm (Thailand time). Thailand is at GMT+7. Therefore, 12:16pm (Thailand) is 5:16am GMT.

Since 5:16am is 6 hours 44 minutes before 12:00pm, the boat locates at 404/4 = 101 degree East.

Examples:

When it is invoked by

```
==========================================================================
r = sailor({'H': 5, 'M': 16, 'am/pm': 'am'})
print(r)
==========================================================================
```

we will see

```
==========================================================================
(101.0, 'East')
==========================================================================
```

When it is invoked by

```
==========================================================================
r = sailor({'H': 11, 'M': 0, 'am/pm': 'am'})
print(r)

r = sailor({'H': 12, 'M': 8, 'am/pm': 'pm'})
```

```
print(r)

r = sailor({'H': 7, 'M': 0, 'am/pm': 'pm'})
print(r)

r = sailor({'H': 0, 'M': 0, 'am/pm': 'am'})
print(r)

r = sailor({'H': 0, 'M': 4, 'am/pm': 'am'})
print(r)
```

==========

we will see

==========

```
(15.0, 'East')
(2.0, 'West')
(105.0, 'West')
(180.0, 'East')
(179.0, 'East')
```

==========

P10. Game theory is a study of social interaction among rational decision makers in such a way that a decision of any decision maker affects the others. For example (Prisoner's Dilemma), Lobha and Raga were arrested for theft. The police tries to get them to confess so each of them is offered a deal. If both do not confess, both will get 3-year sentence. However, if Lobha confesses and Raga does not, Lobha will get 1-year sentence and Raga will get 10-year sentence. Vice versa, if Lobha does not confess, but Raga does, Lobha will serve 10 years in prison while Raga will only have to do 1 year. If both confess, both will get 5-year sentence. Table below summarizes the 4 possible scenarios

Lobha \ Raga	Not confess	Confess
Not confess	3/3	10/1
Confess	1/10	5/5

It is clearly that the mutual interest is maximized when both do not confess. However, Lobha and Raga cannot trust one another. The best choice of Lobha regardless of Raga decision is to confess and so is Raga's. These rational decisions of both players can be settled and this state is called Nash equilibrium.

Write a function named `rational_decision`. The function takes 2 arguments: (1) information table of the 4 scenarios as a dictionary and (2) a name of a person the function finding the decision for. The function returns a rational decision of the given person regardless of the other's decision or `None` if there is no such an option. The information is represented as a dictionary with format:

`{'Lobha': [[3, 10], [1, 5]], 'Raga': [[3, 10], [1, 5]]}` each dict value is a list of 4 entries each of a number of years serving in prison for the corresponding scenario, i.e, in order, a person does not confess and the other does not confess, a person not confess but the other does, a person confesses but the other does not, and a person confesses and the other does too.

Examples:

When invoked by

```
=====================================================================
choices = ['not confess', 'confess']


s = {'Lobha': [[3, 10], [1, 5]], 'Raga': [[3, 10], [1, 5]]}


p = 'Lobha'
r = rational_decision(s, p)
print(p, ':', choices[r])
=====================================================================
```

we will see

```
======================================================================
Lobha : confess
======================================================================
```