CS1302 - Lecture 9 Dictionaries and Sets (Associative_Containers.ipynb)
(Dr Helena WONG)

======================================

Start-up / Discussion activities:

======================================

Understand the following? Try!

[A] tuple, list, comprehension, generator, unpacking

```
a1 = tuple(range(5))
a2 = list(range(5))
a3 = [*range(5)]
a4 = (*range(5),)
# a4 = (*range(5)) # What is this? Try!

a5 = [x for x in range(5)]
a6 = (x for x in range(5))    # What is this? Try!
a7 = (*(x for x in range(5)),) # What is this? Try!
```

Ref:
https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Generators_and_Comprehensions.html
Generator:
- Allows to generate arbitrarily-many items in a series,
  without having to store them all in memory at once.
- A generator can be iterated over once, after which it is exhausted
  and must be re-defined in order to be iterated over again.

https://www.geeksforgeeks.org/python-list-comprehensions-vs-generator-expressions/
*The **generator** yields one item at a time and generates item only when in demand.*
*Whereas, in a list **comprehension**, Python reserves memory for the whole list.*
Thus we can say that the generator expressions are memory efficient than the lists.

https://djangostars.com/blog/list-comprehensions-and-generator-expressions/

More to try:

```
g = (x for x in range(5))
items_1 = (*g,)
items_2 = (*g,)

g = (x for x in range(5))
k = (*g,) # or: k = (*(x for x in range(5)),)
items_3 = k
items_4 = k
```

[B] Rolling a dice

```
import random

# change the seed (default system time) to a fixed value
random.seed(1234)

x = (random.randint(1,6),
     random.randint(1,6),
     random.randint(1,6)
)
print(*x) # Print out __ items
print(x) # Print out one item
```

[C] The function to get both multiple positional (non-keyword) and keyword arguments

```
def myFun(*args, **kwargs):
    print(args)
    print(kwargs)

myFun(1,2,3,4,'add oil!', mary="A", peter="B")
```

The keyword arguments form a **dictionary** of 2 entries: {'mary': 'A', 'peter': 'B'}

[D] Changing collection size during a for-loop

Tuple

```
data = tuple()
data += (1,)    # or data = data + (1,)
data += (2,)
data += (3,)
for x in data:
    data+=(100,)
print(data)    # Output {1,2,3,100} or {1,2,3,100,100,100} or problem?
```

List

```
data = list()
data += [1]    # or use .append()
data += [2]
data += [3]
for x in data:
    data += [100]
print(data)    # Output what?
```

Set

```
data = set()
data |= {1}    # or use .add()
data |= {2}
data |= {3}
for x in data:
    data |= {100}
print(data)    # Output what?

# You cannot change the size of a set during iteration because it is built
# on a hash table data structure, which makes it impossible to guarantee
# a stable iteration order if its size changes.
```

Dict

```
data = dict()
data[1] = 'a'
data[2] = 'b'
data[3] = 'c'

for x in data:
    data[x+1] = 'z'
print(data)    # Output what?
```

=====================================

[Helena's supplements]

1. Motivation for Dictionaries and Sets
2. Constructing dictionary/set
    Hashability
3. Accessing keys/values
4. Other operators and method

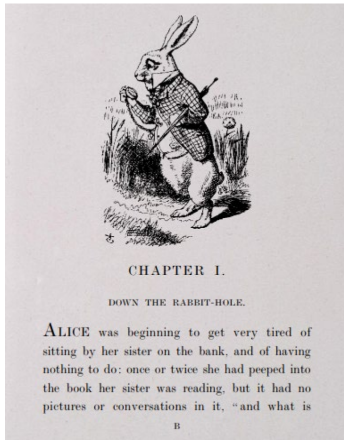
=====================================

Distributions: count, fractional

```
import random
random.seed(1234)

x = (random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6),
    random.randint(1,6)
)
print(x)

print("Distribution: ")
print(x.count(1))
print(x.count(2))
print(x.count(3))
print(x.count(4))
print(x.count(5))
print(x.count(6))

print("Distribution: ")
print(x.count(1)/len(x))
print(x.count(2)/len(x))
print(x.count(3)/len(x))
print(x.count(4)/len(x))
print(x.count(5)/len(x))
print(x.count(6)/len(x))

# We could have written:
random.seed(1234)
x = [random.randint(1,6) for i in range(10)]
distribution = [x.count(i) / len(x) for i in range(1,7)]
print(x)
print(distribution)

# Let' only count those with non-zero frequency
# distinct_x = [i for i in [1,2,3,4,5,6] if x.count(i)>0]
distinct_x = [i for i in range(1,7) if x.count(i)>0]
distribution = [x.count(dx) / len(x) for dx in distinct_x]
print(distinct_x)
print(distribution)

# The one in the lecture notebook is a bit more complicated:
distinct_x = [i for i in range(1, 7) if x.count(i) > 0]
distribution = [x.count(distinct_x[i]) / len(x) for i in range(len(distinct_x))]
print(distinct_x)
print(distribution)
```

====================

What if many letters (words) in an article



CHAPTER I.

DOWN THE RABBIT-HOLE.

ALICE was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is

B

https://www.adobe.com/be_en/active-use/pdf/Alice_in_Wonderland.pdf

Remarks: Unicode contains 1,114,112 code points

====================
Basics of set and dictionary

```
s1 = {1, 1, 2, 2, 3, 3, 3}
s1.remove(2)
s1.add(10)

list1 = [1, 1, 2, 2, 3, 3, 3]
s2 = set(list1)

dict1 = {"Alice": 170, "Beth": 175, "Cecil": 168}

print(len(s1))
print(len(dict1))

dict1 = {"Alice": 170, "Beth": 175, "Cecil": 168}
value = dict1.pop("Cecil")
dict1["Ceci"] = value
# dict1["Ceci"] = dict1.pop("Cecil") # "Change the key"
```

*An associative container in Python is a data structure used to store and retrieve data based on a specific **key** rather than an integer index or positional order.*
=> Dictionary and Set

====================
2. Constructing dictionary/set
====================

creating dictionary by (1) enclosure

```
empty_dictionary = {}
a = {'a': 0, 'b': 1}
b = {**a, 'c': 0, 'd': 1}
```

====================
unpacking: * and **

```
a = {'a': 0, 'b': 1}
c1 = {*a} #c1 is a set which contains only the keys
c2 = {**a}
```

====================

empty dictionary and empty set

```
a_dict = {'a': 0, 'b': 1}
a_set = {1,2,'Apple'}

empty_dict = {}
empty_set = {*()}

empty_dict2 = dict()
empty_set2 = set()
```

=====================

creating set by constructors

```
s1 = set()    # set() will create an empty set
s2 = set('abc')
s3 = set(range(2))
s4 = set(['abc',range(2)])
s5 = set(s4)
```

=====================

creating dictionary by constructors

about enumerate()

```
l1 = ["eat","sleep","repeat"]

for element in enumerate(l1):
    print(element)

for element in enumerate(l1,100):
    print(element)

for index, element in enumerate(l1,100):
    print(index, element)

l1 = ["eat","sleep","repeat"]
print(enumerate(l1))
print(*enumerate(l1))
```

we get tuples from an enumerate object:

```
l1 = ["eat","sleep","repeat"]
x1 = [x for x in enumerate(l1)]
x2 = [x for x in enumerate(l1,100)]
x3 = ["{}:{}".format(x,y) for (x,y) in enumerate(l1,100)]
```

=====================

creating dictionary by constructors (enum, zip, keyword arguments)

```
d1 = dict() #this will create an empty dictionary

d2 = dict(enumerate('abc'))

d3 = dict(zip('abc','123'))
d4 = dict(one=1,two=2)
d5 = dict(d4)
# d6 = d4 # alising
```

=====================

Try:

```
d4 = dict(one=1, two=2)
d4b = {"one":1, "two":2}
```

==================

fromkeys

```
keys = {'a', 'e', 'i', 'o', 'u' }
vowels_1 = dict.fromkeys(keys)
vowels_2 = dict.fromkeys(keys, 1)
```

===================

comprehension

```
my_string = "abcad"
my_set = set(my_string)

# comprehension for list and set
counts1 = [my_string.count(k) for k in my_set]
counts2 = {my_string.count(k) for k in my_string} # warning! ^_^

===================
# dictionary comprehension:    {key:value for loop}
def distribute(string):
    return {k : string.count(k)/len(string) for k in set(string)}

my_dict=distribute("abcad")
```

=====================

Views

```
a = {'a': 0, 'b': 1}
x1 = a.keys()       # A View object
x2 = a.values()     # A View object
x3 = a.items()      # A View object
'''
They provide a dynamic view
on the dictionary's entries,
which means that when the
dictionary changes, the
view reflects these changes.
'''
a['a']=10
print(*x3)
```

===================

Dictionary Views: keys(), values(), items()

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x=car.keys()
y=car.values()
z=car.items()
print('before change:')
print(x)
print(y)
print(z)

car['year']=1989
print('after change:')
print(x)
print(y) #note both y and z are updated
print(z)
```

Digest it:

Read the dictionary:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

a). Complete the blanks:

The keys are: ____________________

The values are: ____________________

b). Which loop(s) below can list each entry to show **both the key and the value**?

```
for x in car:
    print(x)

for x in car.items():
    print(x)

for x,y in car.items():
    print(x,y)

for x in car:
    print(x, car[x])

for x in car.keys():
    print(x, car[x])
```

===================

Dictionary: .pop()

Dictionary: .popitem()

Dictionary: .clear()

Set: .pop() and .clear()

Dictionary:

```
words1 = {'a':'apple', 'b':'boy', 'c':'cat'}
word1 = words1.pop('a')

# ===================
words2 = {'a':'apple', 'b':'boy', 'c':'cat'}
word2 = words2.popitem()

# ===================
    # Another example
    person = {}

    person['name'] = 'Phill'
    person['age']=22
    person['salary']=3500.0

    print(person)

    # ('salary', 3500.0) is inserted at the last, so it is removed.
    result = person.popitem()

# ===================
words = {'a':'apple', 'b':'boy', 'c':'cat'}
words.clear()


# ===================
```

set:

```
words = {'apple', 'boy', 'cat'} # ordering could be different
print(words)
w1 = words.pop() # work like Dictionary's popitem()
w2 = words.pop()
words.clear()
```

==================

ordering

```
#this example shows set and dictionary are unordered
set1={1,2,3}
set2={3,2,1}
dict1={'a':1,'b':2,'c':3}
dict2={'c':3,'b':2,'a':1}

print(set1==set2)
print(dict1==dict2)


#but tuple and list are ordered
list1=[1,2,3]
list2=[3,2,1]
tuple1=(1,2,3)
tuple2=(3,2,1)

print(list1==list2)
print(tuple1==tuple2)
```

when keys are identical

```
# For set and dict,    identical keys are merged to the same entry
a = {0: "a", 0.0: "b", 2: "b"}
b1= {0, False}
b2= {False, 0}
b3= {0j, 0, 0.0, False, ""}
```

==================

Hashability

```
# int / float / str / tuple can be keys
dictionary1={1:"apple",2:"banana"}
dictionary2={1.1:"apple",2.1:"banana"}
dictionary3={"1":"apple","2":"banana"}
dictionary4={(1,2):"apple",(2,3):"banana"}

# list/dictionary/set cannot be keys because they may not be hashable
# dictionary5={[1,2]:"apple",[2,3]:"banana"}
# dictionary6={dictionary1:"apple",dictionary2:"banana"}
# dictionary7={{1,2}:"apple",{2,3}:"banana"}

"Hashing"
```

Suppose we have a list of students and scores:

50170223	Amy	90
58790127	Bob	89
51298288	Carl	55
52356272	Daisy	36
58179975	Emily	74
57853637	Frank	60
..		

How to store the data
so that later we can look up very fast?

Hash Tables: https://www.youtube.com/embed/LPzN8jgbnvA (in Associative_Containers.ipynb)

Hashing:

1) Suppose we map student ID to slot number by %20 => hash key
Here f(id): id%20 is the hash function

2) Later if we need to look for id 58179967,
we calculate the hash key: 7 => check on 58790127 => check on 58179967 (found!)
we don't need to compare student ID numbers through the student list!

Hash table of 20 entries:

Slot number (Hash key)						
0						
1						
2						
3	50170223	Amy	90			
4						
5						
6						
7	58790127	Bob	89	58179967	Emily	74
8	51298288	Carl	55			
9						
10						
11						
12	52356272	Daisy	36			
13						
14						
15						
16						
17	57853637	Frank	60			
18						
19						
20						

Ref https://stackoverflow.com/questions/41102610/load-factor-in-hashmap-with-linkedlist

hash()

```
x1 = hash((1,2,3))
x2 = hash("apple")
```

The hasable(obj) function:

```
def hashable(obj):
    try:
        hash(obj)
    except Exception:
        return False
    return True


for i in 0, 0.0, 0j, "", False, (), [], {}, set(), frozenset(), ([],):
    if hashable(i):
        print("{} may be hashable. E.g., hash({!r}) == {}".format(type(i), i, hash(i)))
    else:
        print("{} may not be hashable.".format(type(i)))
```

==============================

More about Hashing

"Hashing" is quite a feature related to the topic and is important in computer science.

==============================

Ref 1: https://betterprogramming.pub/3-essential-questions-about-hashable-in-python-33e981042bcb

Hashable: A characteristic of a Python object to indicate whether the object has a hash value, which allows the object to serve as a key in a dictionary or an element in a set.
All immutable objects are hashable.

```
a1 = {'apple','boy',('cat','dog')}
a2 = {'apple','boy',['cat','dog']} # TypeError: unhashable type: 'list'
```

Hashable data types: int, float, str, tuple, and NoneType.
Unhashable data types: dict, list, and set.

Above 3 unhashable data types are all mutable, while 5 hashable are all immutable.

Ref 2: https://tutorial.eyehunts.com/python/what-are-mutable-and-immutable-data-types-in-python
Python **Mutable data types** are those whose values can be changed in place whereas **Immutable data types** are those that can never change their value in place.

Here are Mutable data types : Lists, Dictionaries, Sets
And Immutable data types: Integers, Floating-Point numbers, Booleans, Strings, Tuples

==============================

Pitfall:

Try

```
a = (1,)
b = ((),)
x3 = hash(((),))
x4 = hash(([],))
```

See Exercise 4 in Associative_Containers.ipynb

===================
3. Accessing keys/values
===================

```
# how we can access elements in set/dictionary
fruits = {"apple", "banana", "cherry"}
price = {"apple": 3.2, "banana": 4.5, "cherry": 6.8}
print("fruits =", fruits)
print("price =", price)

for element in fruits:
        print(element)
for key in price:
        print(key, price[key])

p1 = price["apple"]
p2 = price["banana"]
p3 = price["cherry"]
# p4 = price[0]     # KeyError
# p5 = fruits[0] # 'set' object is not subscriptable
```

set is not ordered.
dict is insertion-ordered since python 3.7.
dict is subscriptable. E.g., d[k] gives the value in dictionary d associated with key k.
set does not implement __getitem__ and is therefore not subscriptable.

===================

```
# we can insert a new key-value pair, then access it
price['watermelon'] = 1.2
print(price['watermelon'])
print(price)

# this is how we change the values
price['apple']=1
price['banana']=2
price['cherry']=3
price['watermelon']=4
print(price)
```

====================

```
# delete an entry by the key
del price['apple']
print(price)

# another example:
x = 5
d = {'a':x,'b':2}
del d['a']
# d['a']     # KeyError: 'a'
x    #x is still there, it's not deleted
```

===================
distribute (1)
code only:

```
def distribute(seq):
    dist = {}
    for i in seq:
        if i not in dist:
            dist[i] = 1
        else:
            dist[i] += 1
    return dist

d = distribute('What is the distribution of different characters?')
print(d)
```

with comments:

```
def distribute(seq): # 'What is ...'
    dist = {} # empty dictionary
    for i in seq: # 'W', 'h', 'a', 't', ...
        if i not in dist:
            dist[i] = 1 # add a new entry for i, count is 1
        else:
            dist[i] += 1
            # try changing above two 1's to 1/len(seq)
    return dist

d = distribute('What is the distribution of different characters?')
print(d) # print(*(k+": "+str(round(d[k],2)) for k in d))
```

the version in the lecture notebook:

```
def distribute(seq):
    dist = {}
    for i in seq:
        dist[i] = (dist[i] if i in dist else 0) + 1/len(seq)
    return dist
```

===================
get()

```
#This example shows how get() works
car = {
    "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x1 = car.get("model")
x2 = car.get("price", 15000) # specify a default value
x3 = car.get("price") # None will be returned
```

===================
distribute (2), using get()

```
def distribute(seq): # 'What is ...'
    dist = {} # empty dictionary
    for i in seq: # 'W', 'h', 'a', 't', ...
        dist[i] = dist.get(i,0) + 1/len(seq)
    return dist

d = distribute('What is the distribution of different characters?')
print(d)
```

==================

Sorting

```
a = {'a','c','d','b','e'}
b = {3:'d',0:'a',2:'c',4:'e',1:'b'}
sorted_elements = sorted(a)
sorted_keys = sorted(b)
```

==================

add, discard, remove

```
a = set('abc')
a.add('d')
a.discard('a')
a.remove('b')
a.clear()
a.discard('a')    # no error
a.remove('b')     # KeyError
```

====================================

Composite numbers

https://byjus.com/maths/is-1-a-prime-number/#:~:text=For%20a%20number%20to%20be,is%20not%20a%20prime%20number

The positive integers having more than two factors are composite numbers.
(i.e. 3 or more factors)

Among 1 to 20, which numbers are composite and which are not composite numbers?

composite numbers: 4,6,8,9,10, ___________

not composite numbers: 1,2,3,5,7, __________

Find composite numbers

```
Idea:
# first include all multiples of 2: 4, 6, 8, …
# then include from multiples of 3: 6, 9, 12, …

Improvement:
# first include such multiples of 2: 2**2, 2**2+2, 2**2+4,...
# then include these multiples of 3: 3**2, 3**2+3, 3**2+6...
```

```
def composite_set(stop): #stop: 100
    return {x
            for factor in range(2,stop)    # factor: 2,3,4,..99
                for x in range(factor**2,stop,factor)}
                                            # when factor is 2, x:4,6,8,10,..
                                            # when factor is 3, x:9,12,15,..
                                            # ..
print(composite_set(100))     #    {4, 6, 8, 9, 10, 12, 14, 15, 16, 18, ..}
```

====================================

4. Other operators and method

==================

operators *, + (Not for set/dictionary)

list1 = [1,2,3]
list2 = [4,5,6]
list3 = list1*3
list4 = list1+list2

==================

unpack a set
set1 = set('abc')
set2 = set('cde')
concatenated_set = {*set1,*set2}

==================

unpack a dictionary
dict1 = dict(enumerate('abc'))
dict2 = dict(enumerate('def',start=2))
concatenated_set = {*dict1,*dict2}
concatenated_dict = {**dict1,**dict2}

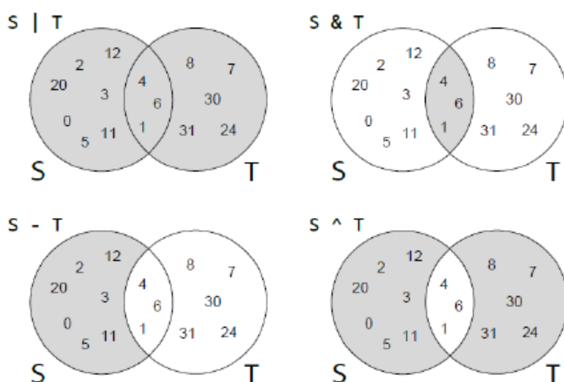
==================

operators Union, intersection, symmetric_difference
a, b = {1,2}, {2,3}
union = a | b
intersection = a & b
symmetric_difference = a ^ b

- more information can be found in Chapter 11.8 of the reference book

Operation	Mathematical Notation	Python Syntax	Result Type	Meaning
Union	$A \cup B$	`A | B`	set	Elements in A or B or both
Intersection	$A \cap B$	`A & B`	set	Elements common to both A and B
Set Difference	$A - B$	`A - B`	set	Elements in A but not in B
Symmetric Difference	$A \oplus B$	`A ^ B`	set	Elements in A or B, but not both
Set Membership	$x \in A$	`x in A`	bool	x is a member of A
Set Membership	$x \notin A$	`x not in A`	bool	x is not a member of A
Set Equality	$A = B$	`A == B`	bool	Sets A and B contain exactly the same elements
Subset	$A \subseteq B$	`A <= B`	bool	Every element in set A also is a member of set B
Proper Subset	$A \subset B$	`A < B`	bool	A is a subset B, but B contains at least one element not in A

```
S = {0, 1, 2, 3, 4, 5, 6, 11, 12, 20}
T = {1, 4, 6, 7, 8, 24, 31}
```



==================

set and dictionary's **object methods** .copy: return result
set and dictionary's **class methods** set.copy, dict.copy: return result

a = {0,1,2}
a1 = a.copy()   # object method
a2 = set.copy(a) # class method

d = {0:"none",1:"one",2:"two"}
d1 = d.copy()   # object method
d2 = dict.copy(d) # class method

===================
set's **object method** .intersection_update: mutate
set's **class method** set.intersection: return result
a = {0,1,2}
b = {1,2,3}

abc1 = a.copy()
abc1.intersection_update(b,{2,3,4}) # object method

abc2 = set.intersection(a,b,{2,3,4}) # class method

abc3 = a&b&{2,3,4} # operator

===================
set's **object method** .update: mutate
set's **class method** set.union: return result
a = {0,1,2}
b = {1,2,3}

ab1 = a.copy()
ab1.update(b) # object method

ab2 = set.union(a,b) # class method

ab3 = a|b # operator

===================
dictionary's **object method** .update: mutate, return None
dictionary's **class method** dict.update: **also mutate**, return None
a = {1: "one", 2: "two"}
b = {3: "three",4:"four"}

ab1 = a.copy()
ab1.update(b) # object method

ab2 = a.copy()
dict.update(ab2,b) # class method, return None

ab3 = a|b # operator

===================

*** Learn more in Associative_Containers.ipynb !!**

- Exercise 10: For dict, there is also a method called setdefault.

```
# help(dict.setdefault):
# Insert key with a value of default if key is not in the dictionary.
# Return the value for key if key is in the dictionary, else default
```

A simple example: *"A for **Ada** and **April**, P for **Peter, …**"*

```
data = {}

name = "Ada"
v = data.setdefault("A",[])
v += [name]
print(data) # {'A': ['Ada']}

name = "April"
v = data.setdefault("A",[])
v += [name]
print(data) # {'A': ['Ada', 'April']}


name = "Peter"
v = data.setdefault("P",[])
v += [name]
print(data) # {'A': ['Ada', 'April'], 'P': ['Peter']}


name = "Paul"
v = data.setdefault("P",[])
v += [name]
print(data) # {'A': ['Ada', 'April'], 'P': ['Peter', 'Paul']}
```

Exercise 10 solution:

define a function group_by_type that
takes a sequence seq of objects and
returns a dictionary d such that d[repr(t)] returns the list of objects in seq of type t

```
def group_by_type(seq):
    group = {}
    for i in seq:
        group.setdefault(repr(type(i)), []).append(i)
    return group

group_by_type(
    [
        *range(3),
        *"abc",
        *[i / 2 for i in range(3)],
        *[(i,) for i in range(3)],
        *[[i] for i in range(3)],
        *[{i} for i in range(3)],
        *[{i: i} for i in range(3)],
        print,
        hash,
        int,
        str,
        float,
        set,
        dict,
        (i for i in range(10)),
        enumerate("abc"),
        range(3),
        zip(),
        set.add,
        dict.copy,
    ]
)
```