Lecture 3-3

Week 3 Friday

Thank you Miles Chen, PhD

Adapted from Chapter 11 of Think Python by Allen B Downey
Additional content on Dictionaries adapted from "Whirlwind Tour of Python" by Jake VanderPlas

Dictionaries

- A dictionary is a mutable structure like a list.
- List items are indexed/accessed by their position but dictionary values are indexed/accessed by their key.
- Dictionaries (dicts) are mappings of **keys** to **values**.
- Common operations on a dictionary are storing a value with a key and extracting a value given a key.

Dictionary Creation

- We commonoly create dictionaries with the curly braces `{}` and colons `:` in the form `key : value`.
- If your keys are strings, you'll need to enclose them with quotes.
- The documentation: **https://docs.python.org/3/library/stdtypes.html#dict**

In [1]:

```
people = {'adam':25 , 'bob': 19, 'carl': 30}
```

In [2]:

```
people
```

Out[2]:

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

Dictionary Creation (cont'd)

If all of the keys are simple strings with no spaces, you can create the dictionary directly with `dict()`.

```
In [3]:
        people2 = dict(adam = 25, bob = 19, carl = 30)
```

```
In [4]:
        people2
```

```
Out[4]:

{'adam': 25, 'bob': 19, 'carl': 30}
```

More ways to create a Dictionary

Dictionaries can also be created in a few more ways:

Call `dict()` on a zip object

In [5]:

```
        # zip two lists together
zip(['adam','bob','carl'] , [25, 19, 30])
```

Out[5]:

```
<zip at 0x7fc718a764c0>
```

the zip function creates a zip object of tuples which we will put into the dict( ) function

In [6]:

```python
people3 = dict(zip(['adam','bob','carl'] , [25, 19, 30]))
```

In [7]:

```python
people3
```

Out[7]:

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

Use `dict()` on a list of key-value pairs

In [8]:

```
        p4 = [('adam', [25, 150]), ('bob', [19, 121]) , ('carl', [30, 214]) ]
people4 = dict(p4)
```

In [9]:

```
        people4
```

Out[9]:

```
{'adam': [25, 150], 'bob': [19, 121], 'carl': [30, 214]}
```

Accessing items in the dictionary

In [10]:

```
people['bob'] # use square brackets and the key
```

Out[10]:

```
19
```

In [11]:

```
people.get('bob') # can also be done with method get()
```

Out[11]:

```
19
```

```
In [12]:
         people['joe'] # if you ask for a key that doesn't exist you get an
error
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/3607908182.py in <module>
----> 1 people['joe'] # if you ask for a key that doesn't exist you get an error

KeyError: 'joe'
```

```
In [13]:
         print(people.get('joe') ) # if you use get() and it does not find,
returns None
```

```
None
```

In [14]:

```
people.get('joe', 0) # You can also specify a default value to return if the key is not found
```

Out[14]:

```
0
```

Dictionary keys and values

- Dictionary keys can be any **immutable** object: strings, numbers (integers recommended), tuples, even functions.
- Dictionary keys must be unique within a dictionary.
- Python uses complex algorithms to determine where the key:value pairs are stored in a dictionary for fast access, so their order can be unpredictable

There is no restriction on what can be a dictionary value.

In [15]:

```
d = {1:len, 3:{"first":"a"}, 4:(1, 2), 2:[20, 4, 5]}
```

In [16]:

```
d[1]
```

Out[16]:

```
<function len(obj, /)>
```

```
In [17]:
d[2]
```

```
Out[17]:
[20, 4, 5]
```

```
In [18]:
d[3]
```

```
Out[18]:
{'first': 'a'}
```

```
In [19]:
d[4]
```

```
Out[19]:
(1, 2)
```

Basic dictionary operations

To obtain the keys

```
In [20]:
list(people)
```

```
Out[20]:
['adam', 'bob', 'carl']
```

```
In [21]:
len(people)
```

```
Out[21]:
3
```

In [22]:

```
            shallow_people = people4.copy() # creates a shallow copy
people4, shallow_people
```

Out[22]:

```
({'adam': [25, 150], 'bob': [19, 121], 'carl': [30, 21
4]},
 {'adam': [25, 150], 'bob': [19, 121], 'carl': [30, 21
4]})
```

In [23]:

```
            people4['adam'].append("baseball")
people4, shallow_people
```

Out[23]:

```
({'adam': [25, 150, 'baseball'], 'bob': [19, 121], 'car
l': [30, 214]},
 {'adam': [25, 150, 'baseball'], 'bob': [19, 121], 'car
l': [30, 214]})
```

hashable keys

- While lists can be values in a dictionary, they cannot be keys.
- Only immutable objects are hashable and can serve as keys, so mutable objects like lists or other dictionaries are not allowed to be used as keys.

In [24]:

```
          l = [1, 2] # list
t = (1, 2) # tuple
```

In [25]:

```
          d = {} # empty dictionary
```

In [26]:

```
          d[l] = "won't work"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/151697439.py in <module>
----> 1 d[l] = "won't work"

TypeError: unhashable type: 'list'
```

In [27]:

```
          d[t] = "this is okay" # assign value to key tuple
d
```

Out[27]:

```
{(1, 2): 'this is okay'}
```

Tuples and other immutable objects are considered "hashable". Mutable objects like lists are considered to be unhashable.

what is hashable?

So one of my favorite websites says:

> *hashable is a feature of Python objects that tells if the object has a hash value or not. If the object has a hash value then it can be used as a key for a dictionary or as an element in a set.*

Perhaps the simple way to understand a hash value is it servess as a unique identifier like a student id number. When we create objects:

In [28]:

```
          t1 = (1, 5, 6)
t2 = (1, 5, 6)
hash(t1), hash(t2) # objects which compare equal have the same hash value
```

Out[28]:

```
(7957444921743822512, 7957444921743822512)
```

Duplicate Keys

Python will not produce an error if you create a dictionary with duplicated keys.
However only the last instance of the unique key will be stored.

```
In [29]:
d = {"a":1, "b":10, "a":2, "b":0, "a": 3}
```

```
In [30]:
d
```

```
Out[30]:
{'a': 3, 'b': 0}
```

In [31]:

```
d["b"]
```

Out[31]:

```
0
```

In [32]:

```
d["a"]
```

Out[32]:

```
3
```

Dictionaries are not indexed by position

Dictionaries are not indexed by position, so you cannot use numeric indexes. If you provide a number, that number needs be a key in the dictionary.

```
In [33]:
        print(people)
```

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

```
In [34]:
        people[0]
```

```
---------------------------------------------------------
--------------------
KeyError                                 Traceback (mos
t recent call last)
```

```
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykern
el_70654/2855141036.py in <module>
----> 1 people[0]

KeyError: 0
```

Dictionaries cannot be sliced

You cannot slice a dictionary the way you would with a list. You can only get one value back at a time.

```
In [35]:
        print(people)
```

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

```
In [36]:
        people[0:2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (mos
t recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykern
```

```
el_70654/1680036604.py in <module>
----> 1 people[0:2]

TypeError: unhashable type: 'slice'
```

In [37]:

```python
people["adam":"carl"]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/416686165.py in <module>
----> 1 people["adam":"carl"]

TypeError: unhashable type: 'slice'
```

Checking for an entry

The `in` operator applies to the keys. If you want to check the existence of a value, you'll have to use the `dict.values()` view object.

```
In [38]:
'adam' in people
```

```
Out[38]:
True
```

```
In [39]:
19 in people
```

```
Out[39]:
False
```

In [40]:

```
19 in people.values()
```

Out[40]:

```
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order. As the list gets longer, the search time gets longer in direct proportion.
Python dictionaries use a data structure called a hashtable that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items are in the dictionary.

What is a hashtable?

- An elegant way to use a dictionary (and they can be extremely large dictionaries, e.g., every social security number in the US and the persons associated)
- The `in` operator's average time to search for an element is O(1) ("order 1": a constant-time method - previous slide)
- And it's worst-case time is O(n) (linear with the size of dictionary, so twice as large means twice as long)
- You don't need to possess a deep understanding for Stats 21, but if you are interested in data theory, it's important to possess the ability to construct optimal hash functions
- All a hash function does is maps a huge value or string to a small integer that can be used as the index in the hash table. Check out:

https://algs4.cs.princeton.edu/34hash/

Adding and modifying dictionary entries

You can use key mapping to create new entries in the dictionary. You can also use it to modify the value associated with a key.

In [41]:

```
        people
```

Out[41]:

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

In [42]:

```
        people['derek'] = 33  # new entry
people['adam'] = 26    # modifies existing key-value pair
```

In [43]:

```
        people
```

Out[43]:

```
{'adam': 26, 'bob': 19, 'carl': 30, 'derek': 33}
```

Removing keys from a dictionary

To remove a key, use `del`

```
In [44]:
people
```

```
Out[44]:
{'adam': 26, 'bob': 19, 'carl': 30, 'derek': 33}
```

```
In [45]:
del people['carl']
```

```
In [46]:
people
```

```
Out[46]:
{'adam': 26, 'bob': 19, 'derek': 33}
```

Dictionary Methods

Use `dictionary_name.pop()` to remove an entry from the dictionary while getting the value associated with the key.

```
In [47]:
          people.pop()  # pop method requires a key that exists in the 
dictionary
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/2228188281.py in <module>
----> 1 people.pop()  # pop method requires a key that exists in the dictionary

TypeError: pop expected at least 1 argument, got 0
```

```
In [48]:
            people.pop('adam')
```

```
Out[48]:

26
```

```
In [49]:
            print(people)
```

```
{'bob': 19, 'derek': 33}
```

the `update()` method

`dict.update()` can be used to add more keys from another dictionary

```
In [50]:
        peopleA = {'adam':25 , 'bob': 19, 'carl': 30}
```

```
In [51]:
        peopleB = {'dave':35 , 'earl': 22, 'fred': 27}
```

```
In [52]:
        peopleA.update(peopleB)
```

```
In [53]:
        peopleA
```

```
Out[53]:

{'adam': 25, 'bob': 19, 'carl': 30, 'dave': 35, 'earl':
22, 'fred': 27}
```

If the dictionary used to update has keys that exist in the first dictionary, the keys will be overwritten with the updated keys.

In [54]:
```
peopleA
```

Out[54]:
```
{'adam': 25, 'bob': 19, 'carl': 30, 'dave': 35, 'earl':
22, 'fred': 27}
```

In [55]:
```
peopleC = {'fred':99 , 'gary': 18}
```

In [56]:
```
peopleA.update(peopleC)
```

In [57]:
```
print(peopleA)
```

```
{'adam': 25, 'bob': 19, 'carl': 30, 'dave': 35, 'earl':
22, 'fred': 99, 'gary': 18}
```

Dictionary view objects

Dictionaries support dynamic view objects. This means that the values in the view objects change when the dictionary changes.
the view objects are generated by the following methods:

- `dict.keys()`
- `dict.values()`
- `dict.items()`

In [58]:

```
people = {'adam':25 , 'bob': 19, 'carl': 30}
```

In [59]:

```
people
```

Out[59]:

```
{'adam': 25, 'bob': 19, 'carl': 30}
```

In [60]:

```
            names = people.keys()
ages = people.values()
```

In [61]:

```
            names
```

Out[61]:

```
dict_keys(['adam', 'bob', 'carl'])
```

In [62]:

```
        ages
```

Out[62]:

```
dict_values([25, 19, 30])
```

In [63]:

```
        # I create a new key-value pair in the dictionary
people['ed'] = 40
```

In [64]:

```
        # without redefining what names or ages are, the view object
updates
names
```

Out[64]:

```
dict_keys(['adam', 'bob', 'carl', 'ed'])
```

In [65]:

```
        ages
```

Out[65]:

```
dict_values([25, 19, 30, 40])
```

view objects support only a few functions: `len()` or `in`
If you need to do more, you can convert the view object to a list or other iterable type, but you'll lose the dynamic aspect of the view object

```
In [66]:
len(ages)
```

```
Out[66]:
4
```

```
In [67]:
35 in ages
```

```
Out[67]:
False
```

In [68]:

```
age_list = list(ages)
```

In [69]:

```
age_list
```

Out[69]:

```
[25, 19, 30, 40]
```

In [70]:

```
        # add a new key-value pair in the dictionary
people['frank'] = 29
```

In [71]:

```
        ages # the view object is dynamic
```

Out[71]:

```
dict_values([25, 19, 30, 40, 29])
```

In [72]:

```
        age_list # the list created earlier is not
```

Out[72]:

```
[25, 19, 30, 40]
```

In [73]:

```
ages[3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/3034873480.py in <module>
----> 1 ages[3]

TypeError: 'dict_values' object is not subscriptable
```

In [74]:

```
ages['bob']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykern
el_70654/3511970310.py in <module>
----> 1 ages['bob']

TypeError: 'dict_values' object is not subscriptable
```

`.items()` is a view object containing tuples of key-value pairs.

```
In [75]:
dic_items = people.items()
```

```
In [76]:
dic_items
```

```
Out[76]:
dict_items([('adam', 25), ('bob', 19), ('carl', 30), ('e
d', 40), ('frank', 29)])
```

```
In [77]:
list(people.items())
```

```
Out[77]:
[('adam', 25), ('bob', 19), ('carl', 30), ('ed', 40),
('frank', 29)]
```

Application: Using a dictionary as a collection of counters

You are given a string and you want to count how many times each letter appears.
There are a few ways we can do this.

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Let's use the dictionary approach:

In [78]:

```
        def histogram(string):
    d = {}
    for character in string:
        if character not in d:
            d[character] = 1
        else:
            d[character] += 1
    return d
```

In [79]:

```
        h = histogram("yellowwooddoor")
h
```

Out[79]:

```
{'y': 1, 'e': 1, 'l': 2, 'o': 5, 'w': 2, 'd': 2, 'r': 1}
```

Iterating over a dictionary

```
In [80]:
            for key in h:
    print(key, h[key])
```

```
y 1
e 1
l 2
o 5
w 2
d 2
r 1
```

It might appear like the letters are arranged in order of appearance, but this is not always the case. This is just a coincidence of the order of letters in the string. You cannot count on dictionary keys to be sorted in any meaningful way - recall Python optimizes for speed.

If you need them to appear in alphabetical order you can use `sorted()` on the dictionary

```
In [81]:
        for key in sorted(h):
    print(key, h[key])
```

```
d 2
e 1
l 2
o 5
r 1
w 2
y 1
```

Reverse Lookup Search

Dictionaries are designed to return values when you provide the key. If you need to find the key associated with a particular value, it's a bit harder and requires us to perform a search.

In [82]:

```
        def reverse_lookup(dictionary, value):
    for key in dictionary:
        if dictionary[key] == value:
            return key
    raise LookupError("Value does not appear in dictionary")
```

In [83]:

```
        h
```

Out[83]:

```
{'y': 1, 'e': 1, 'l': 2, 'o': 5, 'w': 2, 'd': 2, 'r': 1}
```

```
In [84]:
reverse_lookup(h, 2)
```

```
Out[84]:
'l'
```

```
In [85]:
reverse_lookup(h, 4)
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/1435244396.py in <module>
----> 1 reverse_lookup(h, 4)

/var/folders/2z/y8vhcz612_5bbs23g3cpndqm0000gn/T/ipykernel_70654/496591784.py in reverse_lookup(dictionary, value)
```

```
      3         if dictionary[key] == value:
      4             return key
----> 5     raise LookupError("Value does not appear in
 dictionary")

LookupError: Value does not appear in dictionary
```

The `raise` statement

- We will look more that this later, but for now
- The raise statement can be used to handle errors.
- In our code we tell Python to raise a Lookup Exception with a (helpful) message to the user.
- There are several types of exceptions that we can raise (check the documentation)

https://docs.python.org/3/library/exceptions.html

Dictionaries and lists

Lists can appear as values in a dictionary.
For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

In [86]:

```
     def invert_dict(d):
inverse = dict()
for key in d:
    val = d[key]
    if val not in inverse:
        inverse[val] = [key]
    else:
        inverse[val].append(key)
return inverse
```

In [87]:

```
h = histogram("beekeeper")
```

In [88]:

```
h
```

Out[88]:

```
{'b': 1, 'e': 5, 'k': 1, 'p': 1, 'r': 1}
```

In [89]:

```
inverse = invert_dict(h)
```

In [90]:

```
inverse
```

Out[90]:

```
{1: ['b', 'k', 'p', 'r'], 5: ['e']}
```