Dictionaries

Think of an on-line dictionary. You type in a word and the dictionary returns one or more meanings of the word you entered.

We can think of the dictionary as a "list" that is indexed by the "word" whose definition you seek and the value that is returned is the set of meaning associated with that word.

Or …

Think of an on-line phone book. You time in the name of the person whose phone number you want and the phone book app returns the associated number.

We can model the above in Python using the **dictionary**.

```
>>>
>>> pb=dict()
>>> pb['Bob']='212-444-5678'
>>> pb['Joan']='718-767-3223'
>>> pb['George]='212-998-6756'

SyntaxError: invalid syntax
>>> pb['George']='212-998-6756'
>>>
>>> pb['Bob']
'212-444-5678'
>>> pb['Bob']='617-788-3479'
>>> pb['Bob']
'617-788-3479'
>>>
>>> pb['Chuck']
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    pb['Chuck']
KeyError: 'Chuck'
>>>
>>> 'Bob' in pb
True
>>> 'Chuc' in pb
False
>>> 'Chuck' in pb
False
>>>
>>>
>>>
>>> for i in pb:
        print(i)


Bob
Joan
George
>>>
>>> for i in pb:
        print(i, pb[i])


Bob 617-788-3479
Joan 718-767-3223
George 212-998-6756
>>>
```

Here are some of the dictionary methods:

`S= dict()` also s={}

Create a new dictionary.

`len(d)`

Return the number of items in the dictionary *d*.

`d[key]`

Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in the map.

`d[key] = value`

Set `d[key]` to *value*.

`del d[key]`

Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.

`key in d`

Return `True` if *d* has a key *key*, else `False`.

`key not in d`

Equivalent to `not key in d`.

`clear()`

Remove all items from the dictionary.

`copy()`

Return a shallow copy of the dictionary.

Create a new dictionary with keys from *seq* and values set to *value*.

`items()`

Return a new view of the dictionary's items (`(key, value)` pairs).

`keys()`

Return a new view of the dictionary's keys.

`pop`(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

`values`()
Return a new view of the dictionary's values

There are additional methods. Check out the Python on-line documentation.

Dictionaries in some more detail

A dictionary is a mapping between keys and values. You create a dictionary by enclosing the key-value pairs (key:value) each separated by a colon), in curly braces ({ }), each pair separated by a comma like this:

```
s = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
   }
```

To access members of a dictionary, use the indexing operator as follows:

```
name = s['name']
cost = s['shares'] * s['price']
```

Inserting or modifying objects works like this:

```
s['shares'] = 75
s['date'] = '2007-06-07'
```

A dictionary is a useful way to define an object that consists of named fields. However, dictionaries are also commonly used as a mapping for performing fast lookups on unordered data.

For example, here's a dictionary of stock prices:

```
prices = {
   'GOOG' : 490.1,
   'AAPL' : 123.5,
   'IBM' : 91.5,
   'MSFT' : 52.13
}
```

Given such a dictionary, you can look up a price:

```
p = prices['IBM']
```

Dictionary membership is tested with the in operator:

```
if 'IBM' in prices:
   p = prices['IBM']
else:
   p = 0.0
```

This particular sequence of steps can also be performed more compactly using the **get**() method:

```
p = prices.get('IBM', 0.0)    # prices['IBM'] if it exists, else 0.0 # or any other default
```

Use the del statement to remove an element of a dictionary:

```
del prices['GOOG']
```

```
>>> prices = {
    'GOOG' : 490.1,
    'AAPL' : 123.5,
    'IBM' : 91.5,
    'MSFT' : 52.13
}

>>> del prices['TYU']
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    del prices['TYU']
KeyError: 'TYU'
>>> prices.pop('TYU',-1)
-1
>>>
```

Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples.

For example, tuples are often used to construct composite or multipart keys:

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42     # Parens omitted
```

Any kind of object can be placed into a dictionary, including other dictionaries.

However, mutable data structures such as lists, sets, and dictionaries **cannot be used as keys.**

Dictionaries are often used as building blocks for various algorithms and data-handling problems.

One such problem is tabulation.

For example, here's how you could count the total number of shares for each stock name in earlier data:

```
portfolio = [
  ('ACME', 50, 92.34),
  ('IBM', 75, 102.25),
  ('PHP', 40, 74.50),
  ('IBM', 50, 124.75)
]

total_shares = { s[0]: 0 for s in portfolio }  # dictionary comprehension
for name, shares, _ in portfolio:
   total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

In this example, { s[0]: 0 for s in portfolio } is an example of a dictionary comprehension.

It creates a dictionary of key-value pairs from another collection of data. In this case, it's making an initial dictionary mapping stock names to 0. The for loop that follows iterates over the dictionary and adds up all of the held shares for each stock symbol.

Another way to do the above tabulation:

Aside: Many common data processing tasks such as this one have already been implemented by **library modules.**

For example, the collections module has a Counter object that can be used for this task:

```
from collections import Counter # 1

total_shares = Counter() #2
for name, shares, _ in portfolio:# 3
   total_shares[name] += shares

# total_shares = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

How does this work?

Its just like the example above, but since this operation of using dicts to tabulate is quite common, a separate class was added for it. The counter is an iterable where each element consists of a key and a count.

1. The Counter is a subclass of dict specialized for counting hashable objects. It's a collection where elements are stored as dictionary keys, and their counts are stored as dictionary values.
2. Initializing total_shares as a Counter. This will be used to keep track of the total number of shares for each stock.

3. This loop iterates over portfolio, which is presumably a list of tuples. Each tuple in portfolio represents a stock, with its elements being the stock's name, the number of shares, and a third unspecified value (ignored with _).

- name, shares, _: This is tuple unpacking. name gets the name of the stock, shares gets the number of shares, and _ is a placeholder for the third element in the tuple (which is not used in this code).
- total_shares[name] += shares: For each stock, this line adds the number of shares to the count of that stock in total_shares. If the stock name (name) isn't already in total_shares, it's added with a count of 0, and then the shares are added to it.

Let’s expand a bit on the Counter class.

Python's Specialized Counting Dictionary

Counting occurrences of items—such as words in a text, elements in a dataset, or inventory items—is a very common programming task. While Python dictionaries can certainly handle these jobs, there's an even better tool in Python's standard library designed exactly for this purpose: **`collections.Counter`**.

What exactly is a `Counter`?
Formally, it's a specialized dictionary that maps unique elements to integer counts, which makes it Python's built-in implementation of what's called a **multiset** (also known as a bag). Unlike regular sets, which only store distinct elements once, a multiset allows elements to appear multiple times and keeps track of how many times each appears.

Creating Counters

To use `Counter`, you first import it from the built-in `collections` module:

```
from collections import Counter
```

You can create a `Counter` in several intuitive ways:

```
# From a string: counts each character
c1 = Counter("banana")
# Output: Counter({'a': 3, 'n': 2, 'b': 1})

# From a list of items
c2 = Counter(['red', 'blue', 'red', 'green', 'blue', 'red'])
# Output: Counter({'red': 3, 'blue': 2, 'green': 1})

# Directly using keyword arguments
c3 = Counter(apples=5, oranges=2)
# Output: Counter({'apples': 5, 'oranges': 2})
```

Accessing Counts

Accessing counts with a `Counter` is easy and intuitive. Importantly, missing keys default gracefully to zero:

```
c = Counter("apple")
print(c['p'])  # Outputs: 2
print(c['z'])  # Outputs: 0 (no KeyError!)
```

Updating Counts

Counters make incrementing counts straightforward, even for new elements:

```
c = Counter()
c['red'] += 1   # Adds 'red' with count 1
c.update(['red', 'blue', 'red'])  # Adds counts from iterable
# Result: Counter({'red': 3, 'blue': 1})
```

You can subtract counts similarly:

```
c.subtract(['red', 'green'])
# Result: Counter({'red': 2, 'blue': 1, 'green': -1})
```

Negative counts indicate items were removed more times than they appeared. Often, negative or zero counts aren't needed; quickly remove them using:

```
c += Counter()  # or c = +c
# Result: Counter({'red': 2, 'blue': 1})
```

Counter Arithmetic: Combining and Comparing

Counters support intuitive arithmetic operations, allowing you to easily merge or compare multiple counters:

- **Addition (+)**: combines counts by adding corresponding elements.
- **Subtraction (-)**: subtracts counts, discarding any negative results.
- **Intersection (&)**: takes minimum counts—useful for common occurrences.
- **Union (|)**: takes maximum counts—useful for merged tallies.

```
c1 = Counter(a=4, b=2)
c2 = Counter(a=1, b=3, c=1)

print(c1 + c2)  # Counter({'a': 5, 'b': 5, 'c': 1})
print(c1 - c2)  # Counter({'a': 3})
print(c1 & c2)  # Counter({'a': 1, 'b': 2})
print(c1 | c2)  # Counter({'a': 4, 'b': 3, 'c': 1})
```

Helpful Methods for Data Analysis

Counters offer several convenient methods for data analysis:

- **`most_common(n)`**: quickly retrieve the `n` most frequent elements.
- **`elements()`**: expand counts back into an iterable of repeated elements.
- **`total()`** (Python 3.10+): sum all counts for a quick multiset size calculation.

Example usage:

```
text = "to be or not to be that is the question"
word_counts = Counter(text.split())

print(word_counts.most_common(3))
# [('to', 2), ('be', 2), ('or', 1)]

print(word_counts.total())
# 10 (total number of words)

# Iterate through elements based on their frequency
for word in word_counts.elements():
    print(word, end=' ')
# "to to be be or not that is the question"
```

When to Reach for a Counter—and When Not To

- **Use a Counter when:**
  - Counting elements (words, letters, items, votes, etc.) in collections.
  - Constructing frequency distributions or histograms quickly.
  - Handling sparse numeric datasets with integer tallies.
  - Quickly identifying most or least common items.
- **Avoid using Counter if:**

- You need guaranteed ordering based on insertion (instead, consider `OrderedDict`).
- Your keys must hold mutable/unhashable objects (since keys must be hashable).
- You require floating-point or non-integer arithmetic directly within the collection.

Performance Considerations

Counters perform extremely well in typical scenarios. Construction from an iterable is linear (`O(N)`), and most other operations are fast dictionary lookups (`O(1)` on average per element). Memory overhead is minimal, similar to standard dictionaries.

Edge Cases and Gotchas

- **Negative counts:**
  Counters allow negative values after subtraction. If undesirable, quickly discard them:
- `c = Counter(a=1)`
- `c.subtract(['a', 'a'])  # Counter({'a': -1})`
- `c = +c  # now empty Counter`
- **Nonexistent keys:**
  Unlike normal dicts, accessing nonexistent keys returns 0, not a KeyError.

Summary (TL;DR)

The `collections.Counter` class provides a clean, intuitive, and powerful way to tally and manipulate item frequencies in Python. With built-in support for common operations, arithmetic, and convenience methods, it should be the default go-to structure whenever your Python code needs to keep track of "how many times" something happens or appears.

Creating an empty dictionary

As we saw earlier,there are 2two ways.

```
prices = {}        # An empty dict
prices = dict()    # An empty dict
```

1. It is more idiomatic to **use {} for an empty dictionary**—although caution is required since it might look like you are trying to create an empty set (use set() instead).

2. **dict**() is commonly **used to create dictionaries from key-value values.**

For example:

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict(pairs)
```

How to obtain a list of dictionary keys:

Two ways:

1. **convert a dictionary to a list:**

```
syms = list(prices)     # syms = ['AAPL', 'MSFT', 'IBM', 'GOOG']
```

2. **Alternatively,** you can obtain the keys using **dict**.keys():

```
syms = prices.keys()
```

The difference between these two methods is that

keys() returns a special "keys **view**" a list of the keys that is attached to the dictionary and actively reflects changes made to the dictionary.

list(prices) returns a list of the keys that is "frozen" at the time that the lisyt constructor was called.

For example:

```
>>> d = { 'x': 2, 'y':3 }
>>> d.keys()
dict_keys(['x', 'y'])
>>> k=list(d)
>>> k
['x', 'y']
>>> d['z']=4
>>> d.keys()
dict_keys(['x', 'y', 'z'])
>>> k
['x', 'y']
>>>
```

The “keys()” function is an example of a **view**.

In Python, a "view" refers to a special object that provides a dynamic view on the entries of a dictionary-like object (such as a standard dict or a collections.Counter). It is continuously updated so that when the keys() function is called it doesn’t recalculate it rather reflects the current view as opposed to:

```
>>> k=list(d)
>>> k
['x', 'y', 'z']
>>>
```

which will recreate the list of keys each time it is called.

These views are **dynamic** in the sense that they reflect changes made to the underlying dictionary.

There are three primary types of views available in Python dictionaries:

Keys View:

Obtained using the .keys() method on a dictionary.

It provides a dynamic view of all the keys in the dictionary. If the dictionary changes, the keys view changes accordingly.
Example: dict_keys(['a', 'b', 'c'])

Values View:

Obtained using the .values() method on a dictionary.
This view shows all the values in the dictionary. Like the keys view, it updates in real-time as the dictionary's values change.
Example: dict_values([1, 2, 3])

Items View:

Obtained using the .items() method on a dictionary.
It provides a view of all the key-value pairs in the dictionary as tuples. Any change in the dictionary is immediately reflected in the items view.

Example: dict_items([('a', 1), ('b', 2), ('c', 3)])

These views are particularly useful because they allow you to observe the changes in the dictionary without needing to create a new list of its contents every time it updates. This makes your code more efficient and elegant, especially when working with large datasets or in scenarios where the dictionary is frequently updated.

Some examples:

Iterating over the entire contents of a dictionary as key-value pairs:

```
for sym, price in prices.items():
    print(f'{sym} = {price}')
```

Here are some more examples from the Python documentation.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
```

```
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

What is the order of elements in the dictionary?

The keys always appear in the same order as the items were initially inserted into the dictionary. The list conversion above will preserve this order.

This can be useful when dicts are used to represent key-value data read from files and other data sources. The dictionary will preserve the input order. This might help readability and debugging. It's also nice if you want to write the data back to a file.

Note: Prior to Python 3.6, however, this ordering was not guaranteed, so you cannot rely upon it if compatibility with older versions of Python is required. Order is also not guaranteed if multiple deletions and insertions have taken place. Its possible (though not very likely) that the ordering might once again be unpredictable as it was before Python 3.6.

Therefore, if you absolutely need the key value pairs you can use **"collections.OrderedDict":**

collections.OrderedDict is a subclass of the built-in dict class that is guaranteed to maintain the order of the keys in which they were inserted.

For example:

```
import collections

# Create an empty ordered dictionary
od = collections.OrderedDict()

# Add items to the dictionary in a specific order
od['apple'] = 1
od['banana'] = 2
od['orange'] = 3
od['grape'] = 4

# Print the items in the order they were added
for key, value in od.items():
    print(key, value)
```

The output is:

```
apple 1
banana 2
```

orange 3
grape 4

Dictionary comprehensions

A compact way to process all or part of the elements in an iterable and return a dictionary with the results.

`results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`.

Problem

Using a dictionary comprehension create a dictionary with n associated to the nth prime number for the first k primes. K is provided by the user and is not part of the comprehension.

Answer

Copying dictionaries

To copy a dictionary in Python, you can use either the dict() constructor or the dictionary method copy().

Here's an example of how to use the copy() method to create a shallow copy of a dictionary:

original_dict = {'a': 1, 'b': 2, 'c': 3}
copy_dict = original_dict.**copy()**

print(original_dict) # {'a': 1, 'b': 2, 'c': 3}
print(copy_dict) # {'a': 1, 'b': 2, 'c': 3}

In this example, we create an original_dict with three key-value pairs. We then create a **shallow copy** of the dictionary using the copy() method, and assign it to the copy_dict variable.

When we print both dictionaries, we see that they contain the same key-value pairs, indicating that the copy() method created a copy of the original dictionary.

Note that both of these methods create a shallow copy of the dictionary, which means that any mutable objects (such as lists or other dictionaries) contained in the original dictionary will still be referenced by both the original and copied dictionaries. I

If you want to create a deep copy of a dictionary that also copies any mutable objects contained within it, you can use the copy module's deepcopy() function instead.

Deep copy

In Python, you can create a deep copy of a dictionary using the **copy.deepcopy()** method from the built-in **copy** module. A deep copy is a new dictionary that is completely independent of the original dictionary. Any changes made to the new dictionary will not affect the original dictionary.

For example:

```
import copy

# Define a dictionary
my_dict = {'a': [1, 2], 'b': [3, 4]}

# Create a deep copy of the dictionary
my_dict_copy = copy.deepcopy(my_dict)

# Modify the copy
my_dict_copy['a'][0] = 5

# Print both dictionaries
print(my_dict)      # Output: {'a': [1, 2], 'b': [3, 4]}
print(my_dict_copy) # Output: {'a': [5, 2], 'b': [3, 4]}
```

In this example, we start by defining a dictionary named **my_dict**. We then create a deep copy of **my_dict** using the **copy.deepcopy()** method and assign it to a variable named **my_dict_copy**.

Next, we modify the value associated with the key 'a' in **my_dict_copy** to **[5, 2]**.

Finally, we print both dictionaries to show that the original dictionary **my_dict** has not been modified, while the copied dictionary **my_dict_copy** has been modified.

Important: Note that **copy.deepcopy()** recursively copies all nested data structures in the dictionary, so if your dictionary contains lists, dictionaries, or other mutable objects, a deep copy is necessary to ensure that these objects are also copied and not simply referenced.

For example:

```
import copy

original_dict = {
    'a': 1,
    'b': {
        'c': 2,
        'd': {
            'e': 3,
            'f': {
                'g': 4
            }
        }
    }
}
```

```
# Create a deep copy of the original dictionary
copy_dict = copy.deepcopy(original_dict)

# Modify the copy to demonstrate that it is independent of the original
copy_dict['b']['d']['f']['g'] = 5

# Print the original and copied dictionaries
print(original_dict)
print(copy_dict)
```

The result is:

{'a': 1, 'b': {'c': 2, 'd': {'e': 3, 'f': {'g': **4**}}}}
{'a': 1, 'b': {'c': 2, 'd': {'e': 3, 'f': {'g': **5**}}}}

So, the original dictionary has 'g': **4** but the copy has 'g': **5**.

Dictionaries and **kwargs in Python functions

Recall that **kwargs is a syntax used in Python to pass a dictionary of keyword arguments to a function.

The syntax **kwargs in a function parameter list allows the function to accept an arbitrary number of keyword arguments as a dictionary.

For example:

```
def my_func(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Now, you can call this function with any number of keyword arguments:

```
my_func(apple=3, banana=5, orange=2)
```

and the output will be:

```
apple: 3
banana: 5
orange: 2
```

In this example, the ****kwargs** syntax allows the function to accept any number of keyword arguments and store them in a dictionary named **kwargs**. Inside the function, we can access the keyword arguments as key-value pairs in the **kwargs** dictionary.

So, the relation between ****kwargs** and dictionaries is that ****kwargs** allows you to pass a dictionary of keyword arguments to a function, and the function can then access the keyword arguments as a dictionary.

This can be a convenient way to pass a variable number of arguments to a function, especially when you don't know in advance how many arguments will be passed.

Problem: Trace this program and check it with program run below.

```
def z(a,*b,**c): # recall the order
    print(type(a),type(b),type(c))
    print(a,b[1],c,sep='\n')
    print(len(c),list(c.items()))
    g=list(c.items())
    c[g[0]]=156
    print(c.items())
    print (type(c.items()))
    d=[i for i in c.keys()]
    print(c[d[0]])
    print()
    z=list(c)
    print(c[list(c)[0]])
    print()
    for i in iter(c):
        print(i)


  z(12,3,4,5,k=87,l=19)
```

The output:

```
<class 'int'> <class 'tuple'> <class 'dict'>
12
4
{'k': 87, 'l': 19}
2 [('k', 87), ('l', 19)]
dict_items([('k', 87), ('l', 19), (('k', 87), 156)])
<class 'dict_items'>
87

87

k
l
('k', 87)
>>>
```

The program defines a function `z` and then calls it with a set of arguments. Here is the execution step-by-step:

Function Definition

def z(a, *b, **c):

- `a` captures the first positional argument.
- `*b` captures additional positional arguments in a tuple.
- `**c` captures keyword arguments in a dictionary.

Function Call

z(12, 3, 4, 5, k=87, l=19)

- `a` will be `12`.
- `b` will be a tuple `(3, 4, 5)`.
- `c` will be a dictionary `{'k': 87, 'l': 19}`.

Execution Steps Inside the Function

1. `print(type(a), type(b), type(c))`:
   - Prints the types of `a`, `b`, `c`: `<class 'int'>`, `<class 'tuple'>`, `<class 'dict'>`.

2. `print(a, b[1], c, sep='\n')`:
   - Prints `12`, `4` (second element of `b`), and `{'k': 87, 'l': 19}`.

3. `print(len(c), list(c.items()))`:
   - Prints `2` (number of key-value pairs in `c`) and the list of key-value pairs `[('k', 87), ('l', 19)]`.

4. `g = list(c.items())`:
   - `g` becomes `[('k', 87), ('l', 19)]`.

5. `c[g[0]] = 156`:
   - Adds a new key-value pair to `c` where the key is the tuple `('k', 87)` and the value is `156`. Now `c` is `{'k': 87, 'l': 19, ('k', 87): 156}`.

6. `print(c.items())`:
   - Prints the items of `c`: `[('k', 87), ('l', 19), (('k', 87), 156)]`.

7. `print(type(c.items()))`:
   - Prints the type of `c.items()`: `<class 'dict_items'>`.

8. `d = [i for i in c.keys()]`:
   - `d` becomes a list of keys of `c`: `['k', 'l', ('k', 87)]`.

9. `print(c[d[0]])`:
   - Prints the value associated with the first key in `d` (which is `'k'`), so `87`.

10. `z = list(c)`:
    - `z` becomes a list of keys of `c`: `['k', 'l', ('k', 87)]`.

11. `print(c[list(c)[0]])`:
    - Prints the value associated with the first key in `c` (which is `'k'`), so `87`.

12. The `for` loop `for i in iter(c): print(i)`:
    - Iterates over each key in `c` and prints it. The output will be `'k'`, `'l'`, `('k', 87)`.

The program demonstrates the use of variable positional arguments (`*b`), keyword arguments (`**c`), and various dictionary operations. It shows how to access and modify the dictionary, including adding a tuple as a key.

The technique of using the **kwargs dictionary will allow us to create multiple constructors for Python classes.

Let's look at some larger examples of dictionaries. The first will be will be a **scrabble dictionary** and the second will be to create an **inverted index for a file**.
But first, some preliminaries.

Problem:

Write a function, **signature**(n) that and returns a string with same letters in lexicographic order.

For example, **signature**(stop) ➔ opst

Problem:

Scrabble Descrambler – Very Lite

You are in the middle of an intense game of Scrabble, and you find that your tiles have the letters "ACENRT". What can you possibly do with that??

You whip out your smart-phone and run your handy Scrabble Descrambler! You simply enter the letters on the tiles, and PRESTO!!!, all legal Scrabble words with those letters magically appear on your screen. How cool is that?

Your assignment: write the Scrabble Descrambler.

How?

For this problem, we will only deal with six letter words.

1. From main page of the lecture web site copy the contents at the link "Legal Six Letter Scrabble Words" and store it in a file in your Python directory called wordlist.txt.

2. Create a Python dictionary where for each entry, the key is the signature and the associated value is a list of all words with the same signature.

3. Finally, in a loop, ask the user for the six letters that they want to look up, and your program will return all valid six-letter Scrabble words matching the request.

Pickle

The pickle function is a powerful tool that can be used to save and restore Python objects. It can be used to save objects to a file, send objects over a network, or store objects in a database.

pickle is used to **serialize** and **deserialize** Python objects. Serialization is the process of converting an object into a sequence of bytes, while deserialization is the process of converting a sequence of bytes back into an object.

Think of a complex data structure, say a dictionary where each value is a dictionary each of whose values is a binary tree. How could you store such a structure in a file so that it could be reconstructed later? This is what pickle does.

The pickle function uses a binary format to store objects. This format is efficient and can be used to store any type of Python object. You can unpickle a pickle file to reconstruct the original data structure. The pickle function is also secure, as it can be used to store objects that contain sensitive data.

Here is an example of how to use the pickle function to save a list of numbers to a file:

```
import pickle

numbers = [1, 2, 3, 4, 5]

with open("numbers.pkl", "wb") as f:
    pickle.dump(numbers, f)
```

To restore the list of numbers from the file, you can use the following code:

```
with open("numbers.pkl", "rb") as f:
    restored_numbers = pickle.load(f)

print(restored_numbers)
```

We create our Scrabble dictionary and then pickle it for later use.

```
def signature(n):
   return ''.join(sorted(n))

 #create or load?
 mode=input("Create or Load C or L: ")
 print()
 if mode.upper()=='C':
 # create a "Scrabble Dictionary"
    d={}
    print('Creating dictionary ... please wait.')
    f = open('C:/python32/six letter words.txt', 'r')
    print()
    sl = f.read()

    z=sl.split(' ')
    print()
    for w in z:
       sig=signature(w)
```

```
        if sig not in d:
            d[sig]=[]
            d[sig].append(w)
        else:
            d[sig].append(w)
    f.close()
else:
    print('Unpickling dictionary ... please wait.')
    f=open('slwords','rb')
    d=pickle.load(f)  # this “unpickles”

word=input("Please enter word: ")
print()

while word!='done':
    if len(word)!=6:
        print("word not 6 chars")
    else:
        word=word.upper()
        word=signature(word)

        if word in d:
            print(d[word])
        else:
            print(word,' not found.')

    word=input("Please enter word: ")
    print()

f=open('slwords','wb') print('Pickling ... please wait.')
pickle.dump(d,f) # this pickles
f.close()
```

Another dictionary problem: **Inverted Index**

Problem: Read a text file and create an inverted index for the file.

What is an inverted index of a text file?

An inverted index is a dictionary whose **key** is a "word" in the file and whose **value** is a **list** of tuples **(line number, number of times "word" appears on the line).**

Part 1.

Write a program that reads a text file and creates a simplified version of in inverted index.

Your program will create a dictionary, invertedDict, whose key is the word in the file and whose value is a **list** of integers representing the line numbers in the file where word is found.

What if the word is found k times on some line?

Then the line number will appear k times in the list.

For example, say the word "cat" appears 3 times on line 4, 2 times on line 6, and 4 time on line 8, then the dictionary entry for cat will be:

invertedDict["cat"] ->[4,4,4,6,6,8,8,8,8].

Part 2.

Write a function squish(x) where x is a list of integers so that squish**(-**[4,4,4,6,6,8,8,8,8]) returns -[(4,3),(6,2),(8,4)].

Part 3.

Modify part 1so that

invertedDict["cat"] ->[(4,3),(6,2),(8,4)].

Test your program on the text of the Gettysburg Address.

First do following:

Problem

Write a function squish(x) where x is a list of integers so that squish**(-**[4,4,4,6,6,8,8,8,8]) returns -[(4,3),(6,2),(8,4)].

```
def squish(x):
    if not x:  # Check if the list is empty
        return []

    result = []
    count = 1
    current = x[0]

    for i in range(1, len(x)):
        if x[i] == current:
            count += 1
        else:
            result.append((current, count))
            current = x[i]
            count = 1

    result.append((current, count))  # Add the last element
    return result

# Example usage
print(squish([4, 4, 4, 6, 6, 8, 8, 8, 8]))  # Outputs: [(4, 3), (6, 2), (8, 4)]
```

Another way using library functions:

```
from itertools import groupby

def squish(x):
    return [(key, len(list(group))) for key, group in groupby(x)]

So, for example:
print(squish([4, 4, 4, 6, 6, 8, 8, 8, 8]))  # Outputs: [(4, 3), (6, 2), (8, 4)]
```

groupby is a very useful function

The **groupby** function from Python's **itertools** module is used to group consecutive elements in an iterable that have the same value. Let's break down the structure of the groups created by **groupby** and examine their types with an example.

When **groupby** is applied to an iterable, it returns an iterator that produces pairs of values. Each pair consists of:

1. **The Group Key**: This is the value on which the grouping is being performed. It's the common value shared by all elements in the current group.
2. **The Group Itself**: This is an iterator that yields all items in the current group. The items in each group are those that have the same key value and are consecutive in the original iterable.

Example

from itertools import groupby

data = [1, 1, 2, 2, 2, 3, 3, 3, 3]

When **groupby** is applied to **data**:

groups = groupby(data)

groups will be an iterator where each element is a tuple: **(key, group)**.

Let's iterate over **groups** and print out the structure:

for key, group in groups: print(f"Key: {key}, Group: {list(group)}")

This will output something like:

Key: 1, Group: [1, 1]

Key: 2, Group: [2, 2, 2]

Key: 3, Group: [3, 3, 3, 3]

What is the type of a Group

- The type of each **group** in the pairs returned by **groupby** is an iterator. It's not a list or any other collection type by default. If you need a list, you have to explicitly convert it using **list(group)**.

Important Note

- **groupby** only groups **consecutive** items. If the same value appears in non-consecutive positions in the iterable, it will be treated as belonging to different groups.
- The iterable should be **sorted** on the same key function if you expect to group all identical items together, regardless of their original order.

This understanding of **groupby** is essential when working with data aggregations or transformations in Python, as it provides a powerful tool for grouping data efficiently.

Inverted index

```
def squish(x):
    result=[]
    count=0

    a=x[0]
    for i in range(len(x)):

        if x[i]==a:
            count+=1
        else:
            result.append((count,a)) count=1
            a=x[i] result.append((count,a))

    return(result)

def clean(x):

        s = {'.', ',', '-'}

        return ''.join(i for i in x if i not in s)
d={}
ln=0
for line in open('GB.txt'):ln+=1
    line=clean(line)
    l=line.split() for
    word in l:
        if word not in d: d[word]=[]
            d[word].append(1)
            d[word].append([ln])
        else:
            d[word][0]+=1 d[word][1].append(ln)

print('list of d')
ld=list(d)
ld.sort()
for k in ld:
    print(k,d[k])
```

A refactored, more “Pythonic” version: which do you think is “better”?

```
from collections import defaultdict

def clean(text):
    """Remove punctuation from the text."""
    exclude = {'.', ',', '-'}
    return ''.join(char for char in text if char not in exclude)

def squish(x):
    """Count consecutive occurrences of each line number."""
    if not x:
        return []

    result = []
    current = x[0]
    count = 1

    for number in x[1:] + [None]:  # Append None to handle the last element
        if number == current:
            count += 1
        else:
            result.append((current, count))
            current = number
            count = 1

    return result

def build_inverted_index(file_path):
    """Build an inverted index from the given file, with squished line numbers."""
    inverted_index = defaultdict(lambda: {'count': 0, 'lines': []})

    with open(file_path, 'r') as file:
        for ln, line in enumerate(file, 1):
            cleaned_line = clean(line)
            words = cleaned_line.split()
            for word in words:
                entry = inverted_index[word]
                entry['count'] += 1
                entry['lines'].append(ln)

    # Apply squish to line numbers for each word
    for word, data in inverted_index.items():
        data['lines'] = squish(sorted(data['lines']))

    return inverted_index

def print_inverted_index(inverted_index):
    """Print the sorted inverted index with squished line numbers."""
    print('List of words:')
    for word in sorted(inverted_index):
        data = inverted_index[word]
        print(f"{word}: Count={data['count']}, Line Numbers (squished)={data['lines']}")
```

```
# Usage example
inverted_index = build_inverted_index('GB.txt')
print_inverted_index(inverted_index)
```

Key improvements:

1. **Use of `defaultdict`**:
   - The `defaultdict` from the `collections` module simplifies the management of the inverted index dictionary. It automatically initializes dictionary entries to a default value (`{'count': 0, 'lines': []}`) if they don't exist. This avoids the need for explicit checks and initializations when adding new words to the index.

2. **Readability and Maintainability**:
   - The code is structured into distinct functions with clear responsibilities, making it easy to read, understand, and maintain. Each function is well-documented with docstrings, explaining its purpose and usage.

3. **Pythonic Iteration and List Comprehensions**:
   - The use of list comprehensions and Pythonic iteration techniques, such as enumerating over file lines and iterating with a sentinel value (`None`), showcases idiomatic Python practices that contribute to the code's conciseness and clarity.

The use of `None` as a sentinel value in iterating sequences is a clever Pythonic technique to handle edge cases, particularly when processing the last element of a list. In the provided code snippet, `None` is appended to the end of the list of line numbers (`x[1:] + [None]`) during the iteration within the `squish` function. This approach ensures that the loop has an additional iteration beyond the actual data, which is crucial for including the last group of elements in the result. The sentinel `None` guarantees a change in value compared to any legitimate list element, thus triggering the conditional logic that appends the final group of elements to the `result` list.

For example, consider a list of line numbers `x = [2, 2, 3, 4, 4, 4]` where a word appears in a document. Without the sentinel value, the iteration would stop after processing the last `4`, potentially leaving the group `[4, 4, 4]` unaccounted for because there's no subsequent element to compare and trigger the group appending logic. By adding `None` (`x[1:] + [None]` becomes `[2, 3, 4, 4, 4, None]`), the iteration extends by one step, where `None` doesn't equal the last number `4`, thus ensuring the group `[4, 4, 4]` is correctly appended to `result` as `(4, 3)`.

This technique is particularly elegant in Python due to the language's dynamic typing and truthiness evaluation, where `None` serves as a universal "different" value that can be easily distinguished from any numerical or textual data in a list. It simplifies the logic, especially in cases where the task involves grouping or counting consecutive elements, by providing a clear and concise way to ensure the processing of terminal elements in a sequence. The use of `None` in this manner enhances code readability and maintainability, demonstrating an idiomatic approach to solving common iteration challenges in Python.

Problem

Create a sample CSV file with last name, first name and exam score. The exam score should come from a normal distribution with a mean of 75 and a standard deviation of 15.

```
import os
import csv
import random
import numpy as np
from faker import Faker

# Initialize Faker to generate random names
fake = Faker()

# Function to generate random data
def generate_data_normal_dist(num_rows, mean=75, std_dev=15):
    data = []
    for _ in range(num_rows):
        last_name = fake.last_name()
        first_name = fake.first_name()
        score = np.random.normal(mean, std_dev)
        # Ensure score is within 0-100 range
        score = min(max(score, 0), 100)
        data.append([last_name, first_name, score])
    return data

# Generating data for 100 rows as an example
data = generate_data_normal_dist(100)

# File path for the CSV
# Get the path to the Documents directory
csv_file_path = os.path.join('C:/Users/waxman/Documents', 'random_names_scores.csv')

# Writing data to a CSV file
with open(csv_file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['LastName', 'FirstName', 'Score'])
    writer.writerows(data)
```

The **Faker** module is a Python library that allows you to generate fake data, such as names, addresses, and much more. It's often used for testing and filling databases with random data.

To install the **Faker** module, you can use **pip**, which is the package installer for Python.

pip install Faker

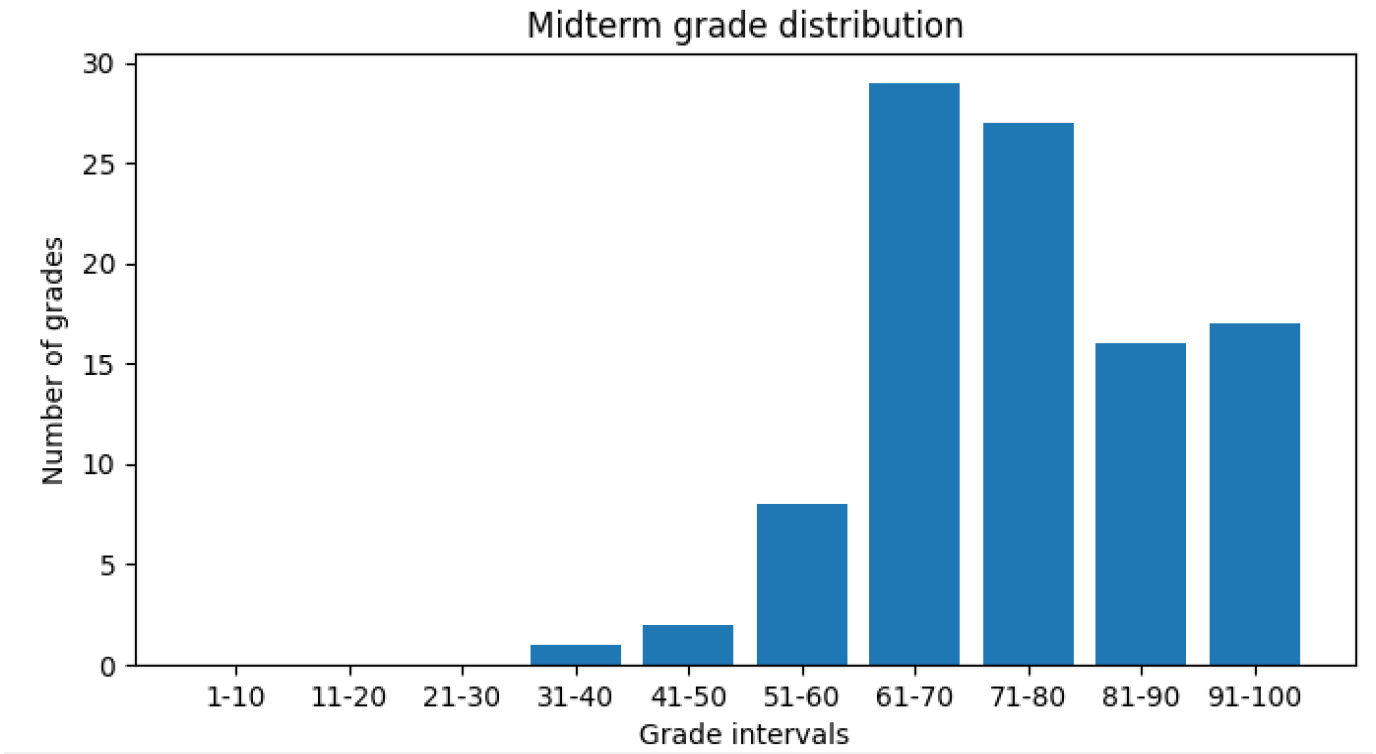
Problem

We have a CVS file like the one above with students names and exam scores. We would like to write a python program to show the breakdown of scores by interval: how many between 1-10, 1-20 etc.

We want to print a table like this:

```
Numbers in interval 1-10: 0
Numbers in interval 11-20: 0
Numbers in interval 21-30: 0
Numbers in interval 31-40: 1
Numbers in interval 41-50: 2
Numbers in interval 51-60: 8
Numbers in interval 61-70: 29
Numbers in interval 71-80: 27
Numbers in interval 81-90: 16
Numbers in interval 91-100: 17
```

And a bar chart like this:



Here is a program to do it.

```
import csv
import os

#File path for the CSV
csv_file_path = os.path.join('C:/Users/waxman/Documents', 'random_names_scores.csv')

# Check if the file exists before opening it
if os.path.exists(csv_file_path):
    # Open the CSV file for reading
    with open(csv_file_path, mode="r") as file:
        # Create a CSV reader
        csv_reader = csv.reader(file)

        # Skip the header row
        next(csv_reader)  # This advances the iterator to the next row
        grades=[]
        # Iterate through rows and process data
        for row in csv_reader:
            grades.append(row[2])# skip the names
else:
    print(f"The CSV file '{csv_file_name}' does not exist in the Documents directory.")

# Filtering and converting grades to floats, keeping only positive values - don't want grade of 0 or blank
positive_float_grades = [float(grade) for grade in grades if grade and float(grade) > 0]

def count_numbers_in_intervals(positive_float_grades):
    interval_counts = {}

    for interval in range(1, 101, 10):
        count = sum(1 for num in positive_float_grades if interval <= num < interval + 10)
        interval_counts[f"{interval}-{interval + 9}"] = count

    return interval_counts

interval_counts = count_numbers_in_intervals(positive_float_grades)

for interval, count in interval_counts.items():
    print(f"Numbers in interval {interval}: {count}")
```

#now plot the results

import matplotlib.pyplot as plt

```
def plot_bar_graph(data_dict):
    # Extracting keys and values from the dictionary
    keys = data_dict.keys()
    values = data_dict.values()

    # Creating the bar graph
    plt.figure(figsize=(8, 4))
    plt.bar(keys, values)

    # Adding labels and title (optional)
    plt.xlabel('Grade intervals')
    plt.ylabel('Number of grades')
    plt.title('Midterm grade distribution')

    # Displaying the bar graph
    plt.show()

plot_bar_graph(interval_counts)
```

Pyplot is a Matplotlib module that provides a MATLAB-like interface.

The line `plt.figure(figsize=(8, 4))` is a command from Matplotlib, a popular data visualization library in Python, and it's used to create a new figure with a specific size.

Here's a breakdown of what this command does:

plt.figure(): This function is used to create a new figure. A figure in Matplotlib is like a canvas on which you can draw plots and other visual elements. When you create a figure, you are essentially initializing a new area for plotting.

figsize=(8, 4): This parameter specifies the size of the figure in inches. The `figsize` argument is a tuple, where the first element is the width and the second is the height of the figure. In this case, `figsize=(8, 4)` means the figure is 8 inches wide and 4 inches tall.

The size is in inches, which can be a bit unintuitive if you're used to working in pixels or other units. However, inches are a standard in the publishing world, which is why Matplotlib uses them.
The size in inches is converted to pixels using the figure's DPI (dots per inch) setting, which can be adjusted but defaults to 100 in most Matplotlib backends. So, in this case, the actual size of the figure would be 800x400 pixels at the default DPI.

Purpose: Setting the figure size is important for a couple of reasons. First, it helps ensure that your plot has the right aspect ratio and size for your analysis or presentation needs. Second, if you're saving the plot to a file, it helps control the size and resolution of the output file.

The takeaway:

plt.figure(figsize=(8, 4)) in Matplotlib is used to initialize a new plotting area or figure with a width of 8 inches and a height of 4 inches. This allows for more control over the size and aspect ratio of the plots you create.

We can redo the interval count function using groupby from itertools.

```
def count_numbers_in_intervals(grades):
    # Initialize all intervals with zero count
    interval_counts = {f"{i}-{i + 9}": 0 for i in range(1, 101, 10)}

    # inner function to determine the interval for a grade
    def interval_key(grade):
        return f"{int((grade - 1) // 10) * 10 + 1}-{int((grade - 1) // 10) * 10 + 10}"

    for key, group in groupby(sorted(grades), interval_key):
        interval_counts[key] = len(list(group))

    return interval_counts
```

Here is how it works:

This function takes a list of grades, determines the interval each grade belongs to, and counts the number of grades in each of these intervals, returning a dictionary that maps intervals to their respective grade counts.

1. Initialization of `interval_counts`dictionary:

interval_counts = {f"{i}-{i + 9}": 0 for i in range(1, 101, 10)}

This line initializes a dictionary named `interval_counts` where each key is a string representing a 10-point interval (e.g., "1-10", "11-20", ..., "91-100"). Each value is initialized to 0. This setup ensures that all intervals are accounted for, even if no grades fall into some intervals.

2. Definition of `interval_key` function:

def interval_key(grade):

return f"{int((grade - 1) // 10) * 10 + 1}-{int((grade - 1) // 10) * 10 + 10}"

This is an **inner function** that takes a single grade and calculates the interval it belongs to. The calculation `int((grade - 1) // 10) * 10 + 1` finds the lower bound of the interval. For example, if `grade` is 35, `(35 -

1) // 10` equals 3, then `3 * 10 + 1` equals 31, so the interval starts at 31. Similarly, the upper bound is calculated to be `31-40`.

3. Grouping Grades by Intervals and Counting:

```
for key, group in groupby(sorted(grades), interval_key):
    interval_counts[key] = len(list(group))
```

groupby(sorted(grades), interval_key)`: This line sorts the grades and then groups them using the `groupby` function from the `itertools` module. As we saw above, the `groupby` function requires the data to be sorted by the same key that is used for grouping. Here, it groups the grades based on the intervals determined by the `interval_key` function.

`for key, group in ...`: This is a loop over the groups created by `groupby`. Each group is a tuple where `key` is the interval (e.g., "31-40"), and `group` is an iterator of grades that fall into that interval.

`interval_counts[key] = len(list(group))`: For each group, it converts the `group` iterator to a list to count how many grades fall into this interval, and then updates the corresponding count in the `interval_counts` dictionary.

4. Return Statement:

```
return interval_counts
```

Finally, the function returns the `interval_counts` dictionary containing the counts of grades in each interval.