6.009 TUTORIAL 2 TEACHING NOTES
for 9/12/2018

Blackboard: 6.009 Agenda
- Tutorial
  - Sets
  - Dictionaries
  - Wordplay

↳ after tutorial, add LA signup:

Expect 6.009 Pool (sign-in)

- office hours: help & checkoffs

① Start w/ sets.ipynb
a. Motivation: lists w/ linear time lookup, vs. constant time w/ sets
b. basic ops & environment diagrams
c. immutable / hashable
   - important to know
   - basic idea of hash so intuition of why constant time
d. EXAMPLE: is number met before

② Next: Dictionaries
a. CONCEPT: just like a set, except members are now keys, and (any) value associated with each key

b. Through dict mechanisms
c. EXAMPLES: remove_2nd_instance versions

↳ Note: as a side benefit we're also reminding students about other python tid bits:
- → d.get(val, 0)
- → list pop
- → enumerate
- → list slicing
- → list copy via [:] (shallow)

③ Wordplay ... mostly follow along, mutate in place

Sets

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference. Sets can also have important efficiency benefits.

One Motivation -- Lists can be slooooooow....

One motivation for using sets is that several important operations (adding an element, determining whether an element is in the set) take *constant time* regardless of the size of the set, rather than linear time in the size of the list.

In [1]:
```
big_num = 10000000 # ten million
big_num_list = list(range(big_num))
big_num_set = set(big_num_list)
```

In [2]:
```
small_num = 100
small_num_list = list(range(big_num - small_num, big_num))

# how many of small_num_list elements are in big_num_list?
import time
start = time.time()
count = 0
print("counting...")
for i in small_num_list:
    count = count + (1 if i in big_num_list else 0)
end = time.time()
print("count using list:", count, "time:", end-start, "sec")
```

```
counting...
count using list: 100 time: 11.38155746459961 sec
```

run 1st as-is; then uncomment this & see that it still runs very fast...

In [3]:
```
# how many of small_num_list elements are in big_num_set?
count = 0
##small_num_list = big_num_list
print("counting...")
start = time.time()
for i in small_num_list:
    count = count + 1 if i in big_num_set else 0
end = time.time()
print("count using set:", count, "time:", end-start, "sec")

small_num_set = set(small_num_list)
start = time.time()
count_intersection = len(big_num_set.intersection(small_num_set))
end2 = time.time()
print("count using set intersection:", count_intersection, "time:", end2-end, "sec")
```

```
counting...
count using set: 100 time: 0.0 sec
count using set intersection: 100 time: 0.0009970664978027344 sec
```

Another Motivation -- Conceptual clarity with set operations

In [4]:
```
# Lists can have duplicate elements, and lists are ordered
basket = ['apple', 'orange', 'apple', 'pear', 'orange']

# Creating a set from a list results in a set without duplicate elements
fruit1 = set(basket)
print(fruit1)
```

```
{'apple', 'orange', 'pear'}
```

In [5]:
```
# Adding the same element again to a set doesn't change the set
fruit1.add('apple')
print(fruit1)
```

```
{'apple', 'orange', 'pear'}
```

In [6]:
```
# But adding a different element does change (mutate) the set...
fruit1.add('banana')
print(fruit1)
```

```
{'banana', 'apple', 'orange', 'pear'}
```

In [7]:
```
# Can discard/remove elements
fruit1.discard('grape')  #no exception if element not in set
fruit1.remove('apple') #exception if element not in set
print(fruit1)
```

```
{'banana', 'orange', 'pear'}
```

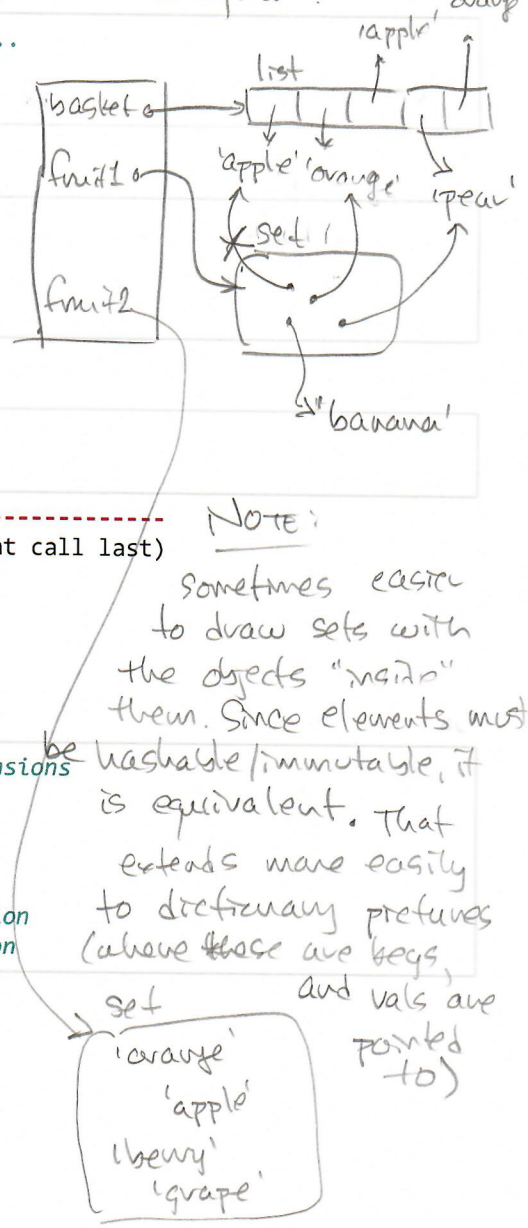
In [8]:
```
# Sets are unordered: cannot index or slice into a set
fruit1[0:]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-573d82862ca7> in <module>()
      1 # Sets are unordered: cannot index or slice into a set
----> 2 fruit1[0:]

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

In [9]:
```
# Can iterate over the elements in a set, in loops or comprehensions
for elt in fruit1:
    if 'n' in elt:
        print(elt)

print([elt for elt in fruit1 if 'n' in elt])  #List comprehension
print({elt for elt in fruit1 if 'n' in elt})  #set comprehension
```

```
banana
orange
['banana', 'orange']
{'orange', 'banana'}
```

Can either have class do these, or run through them quickly. I tend to do w/ an environment diagram.



NOTE: Sometimes easier to draw sets with the objects "inside" them. Since elements must be hashable/immutable, it is equivalent. That extends more easily to dictionary pictures (where keys are keys, and vals are pointed to)

Basic set operations

In [10]:
```
fruit2 = {'orange', 'apple', 'berry', 'grape', 'orange'}
print("fruit1 =", fruit1)
print("fruit2 =", fruit2)
```

```
fruit1 = {'banana', 'orange', 'pear'}
fruit2 = {'berry', 'grape', 'apple', 'orange'}
```

In [11]:
```
#Intersection
print("Intersection:", fruit1 & fruit2)

#Union
print("Union:", fruit1 | fruit2)

#Difference
print("Difference, fruit1 - fruit2:", fruit1 - fruit2)
print("Difference, fruit2 - fruit1:", fruit2 - fruit1)

#Symmetric Difference
print("Symmetric Difference:", fruit1 ^ fruit2) #elements in union but NOT in intersection
```

```
Intersection: {'orange'}
Union: {'pear', 'apple', 'orange', 'grape', 'berry', 'banana'}
Difference, fruit1 - fruit2: {'pear', 'banana'}
Difference, fruit2 - fruit1: {'grape', 'apple', 'berry'}
Symmetric Difference: {'pear', 'apple', 'grape', 'berry', 'banana'}
```

Some set relations

In [12]:
```
fruit3 = set()  #Create an empty set with set() NOT with {}
fruit3.add('banana')
fruit3.add('pear')

print("fruit1 =", fruit1)
print("fruit2 =", fruit2)
print("fruit3 =", fruit3)
```

```
fruit1 = {'banana', 'orange', 'pear'}
fruit2 = {'berry', 'grape', 'apple', 'orange'}
fruit3 = {'pear', 'banana'}
```

In [13]:
```
#Subset
fruit3.issubset(fruit1)
```

Out[13]: True

In [14]:
```
#Disjoint
fruit3.isdisjoint(fruit2)
```

Out[14]: True

In [15]:
```
#Superset
fruit1.issuperset(fruit3)
```

Out[15]: True

What kind of objects can be in a set?

The elements of sets must be immutable hashable objects. Thus numbers, strings, tuples (as long as all elements of the tuple are also immutable/hashable objects) can be members of sets, but lists cannot be members of sets. And sets cannot be members of sets! (See frozensets if you're interested in an immutable/hashable variant of sets, that *can* be elements of a set.) The hashable restriction is what makes it possible to determine whether an element is in a set using constant time with respect to the size of the set; i.e., one does not need to iterate over all elements of a set to determine whether that element is in the set. (See 6.006 for more details on how this hashing works.)

Example: Is the number met before

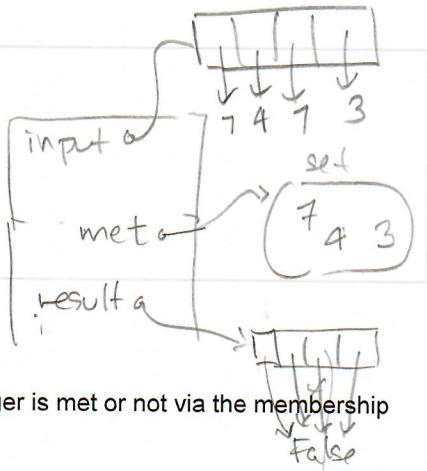
This section takes advantage of sets to solve a simple problem. Here we input a list of integers, your job is to return a list of Booleans which gives True if the number is met earlier in the input list and False if not.

First we generate a random list of integers, ranging from 0 to max_num. You could customize these two parameters to make you own inputs.

input = [7, 4, 7, 3]
# desired result = [False, False, True, False]

In [16]:
```
import random
max_num = 10
size = 20

#Randomly generate an integer list
input = [random.randint(0,max_num) for i in range(size)]
print(input)
```

```
[9, 5, 2, 4, 8, 5, 0, 5, 6, 5, 0, 10, 5, 5, 7, 5, 1, 6, 7, 2]
```



Appraaah:

Then, we use a set to store the numbers that we met before, and determine whether the next integer is met or not via the membership testing of set which takes constant time.

In [17]:
```
met = set() #Create an empty set
result = [False]*len(input) #Initialize a list with the same length of input
for index, number in enumerate(input):
    if number in met:
        result[index] = True #Indicate that the number has been met before
    else:
        met.add(number) #If not, add it to the set of previously met elements
print("input:", input)
print("result:", result)
```

```
input: [9, 5, 2, 4, 8, 5, 0, 5, 6, 5, 0, 10, 5, 5, 7, 5, 1, 6, 7, 2]
result: [False, False, False, False, False, True, False, True, False, True, True, False, True, T
rue, False, True, False, True, True, True]
```

Note: different implementations are possible, and might have different efficiencies. For example, we could create the result list one item at a time, but at the cost of repeated appends:

In [18]:
```
met = set()
result = []
for index, number in enumerate(input):
    if number in met:
        result.append(True)
    else:
        met.add(number) #If not, add it to the set of previously met elements
        result.append(False)
print("input:", input)
print("result:", result)
```

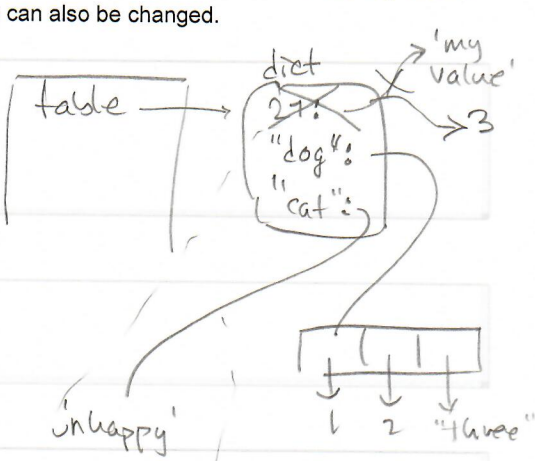
```
input: [9, 5, 2, 4, 8, 5, 0, 5, 6, 5, 0, 10, 5, 5, 7, 5, 1, 6, 7, 2]
result: [False, False, False, False, False, True, False, True, False, True, True, False, True, T
rue, False, True, False, True, True, True]
```

In [19]:
```python
#Even more (probably TOO) pythonic; many frown upon mutating inside a comprehension
met = set()
result = [True if val in met else (False, met.add(val))[0] for val in input]
print("input:", input)
print("result:", result)
```

```
input: [9, 5, 2, 4, 8, 5, 0, 5, 6, 5, 0, 10, 5, 5, 7, 5, 1, 6, 7, 2]
result: [False, False, False, False, False, True, False, True, False, True, True, False, True, T
rue, False, True, False, True, True, True]
```

Dictionaries

Dictionaries are like sets, except that the "elements" of the dictionary are treated as keys, and a value is associated with that key. As in sets, the keys to dictionaries must be immutable and hashable objects. However, the values associated with the key can be anything, and can be mutable. In addition, the association between key and value can also be changed.



In [1]:
```
table = {} #Create empty dictionary. dict() also works
table[27] = 'my value'
table["dog"] = [1, 2, "three"]
print(table)
```

```
{27: 'my value', 'dog': [1, 2, 'three']}
```

In [2]:
```
table[27] = 3
print("table:", table)
```

```
table: {27: 3, 'dog': [1, 2, 'three']}
```

In [3]:
```
#Is key in a dictionary? Can use 'in' syntax.
if 'dog' in table:
    table['cat'] = 'unhappy'
print("table:", table)
```

```
table: {27: 3, 'dog': [1, 2, 'three'], 'cat': 'unhappy'}
```

In [4]:
```
# Iterate over keys in a dictionary. Any order can occur!
for key in table:
    print("key:", key, "val:", table[key])
```

```
key: 27 val: 3
key: dog val: [1, 2, 'three']
key: cat val: unhappy
```

In [5]:
```
# Also, can iterate over key, val items in a dictionary. Any order can occur!
for key, val in table.items():
    print("key:", key, "val:", val)
```

```
key: 27 val: 3
key: dog val: [1, 2, 'three']
key: cat val: unhappy
```

In [6]:
```
# Remove element from a dictionary
del table[27]  #Exception if key is not in dictionary
print("table:", table)
```

```
table: {'dog': [1, 2, 'three'], 'cat': 'unhappy'}
```

In [7]:
```
# Dictionary comprehensions also possible
{n: n**3 for n in range(8)}
```

Out[7]:
```
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343}
```

Some fun with sets and dictionaries...

Problem:

Return a new list with the 2nd instance of the first element that is repeated in the input list removed. The rest of the list should remain unchanged (be the same as the input).

Goal:

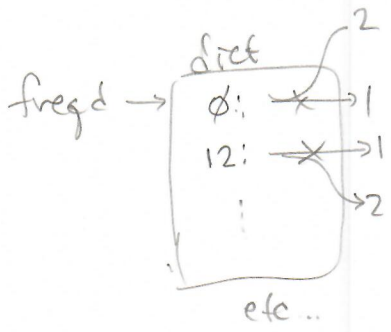
Our goal is to use this problem to build some familiarity with **sets and dictionaries** -- so the code versions below are written to uses sets and/or dictionaries. This is definitely not the only way to solve the problem, but it happens to also be the most efficient away. (At the very end, we'll also show a more "direct", but less efficient, solution which does not use sets or dictionaries.)

In [8]:
```
inp = [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]

inp_str = ['zero', 'twelve', 'twelve', 'zero', 'twelve', 'twelve',
           'thirty four', 'fifty six', 'twenty three', 'eleven',
           'forty five', 'two', 'three', 'four', 'eleven', 'ten', 'twelve']
```

Version 1

In [9]:
```
def remove_2nd_instance_v1(input):
    print("input:", input)

    #Create a dictionary with frequencies
    freqd = {}
    for val in input:
        freqd[val] = freqd.get(val, 0) + 1
        #The get() method returns the value of the key in the dictionary
        #if it exists, and otherwise returns the default value (2nd param)

    #Look for first element in list, that is repeated somewhere later
    for val in input:
        if freqd[val] >= 2:
            repeated = val
            break
    print("repeated:", repeated)

    #Look through the list to find where the 2nd instance of the repeat is
    index = len(input)
    count = 0
    for i in range(len(input)):
        if input[i] == repeated:
            count += 1
        if count == 2:
            index = i
            break
    print("index:", index)

    output = input[:]        #make copy of input
    #Remove this 2nd instance if it exists from a copy of the input
    if index < len(output):
        output.pop(index)
    return output

remove_2nd_instance_v1(inp)
```

edit [replaced val w/ x to avoid concept clash with dictionary "values"]



repeated → 0

```
input: [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
repeated: 0
index: 3
```

Out[9]:
```
[0, 12, 12, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

In [10]:
```
output = remove_2nd_instance_v1(inp_str)  #works also for lists of other kinds of elements
print(output)
```

```
input: ['zero', 'twelve', 'twelve', 'zero', 'twelve', 'twelve', 'thirty four', 'fifty six', 'twenty thr
ee', 'eleven', 'forty five', 'two', 'three', 'four', 'eleven', 'ten', 'twelve']
repeated: zero
index: 3
['zero', 'twelve', 'twelve', 'twelve', 'twelve', 'thirty four', 'fifty six', 'twenty three', 'eleven',
'forty five', 'two', 'three', 'four', 'eleven', 'ten', 'twelve']
```

Revised Spec

p. 1

Our procedure removed `0` (`'zero'`) from the list at index 3, since that is the first repeat of the "earliest" element that has a later

repeat. Is that what we want? Our specification is perhaps **ambiguous**. Arguably, we might want to remove 12 at index 2 since 12 appeared twice before 0 appeared twice. Let's clarify our spec: by 'first repeated element' we mean the element that appears twice **first**. Thus

```
inp = [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

should give

```
expect = [0, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12] # remove second 12
```

Version 2 -- remove the second appearance of the first element that makes a second appearance

In [11]:
```
def remove_2nd_instance_v2(input):
    print("input:", input)

    # Creat_e a dictionary whose values are frequencies AND indices of elements,
    # i.e., [freq_of_val, index_of_first_appearance_or_of_second_appearance_if_repeated].
    # We store either the index of the first element, or for repeated elements
    # we store the index of the 2nd instance (first repeat). Ignore additional repeats.
    freqd = {}

    for i in range(len(input)):
        val = input[i]
        if val in freqd:
            freqd[val][0] += 1
            if freqd[val][0] == 2:
                freqd[val][1] = i
        else:
            freqd[val] = [1, i]
    print("freqd =", freqd)

    #Get the index of the 2nd instance of the earliest appearing repeated element.
    index = len(input)
    for val in input:
        entry = freqd[val]
        if entry[0] >= 2:
            index = min(index, entry[1])

    #Remove this 2nd instance
    output = input[:]        #make copy of input
    if index < len(output):
        output.pop(index) #list pop -- mutates list
    return output

remove_2nd_instance_v2(inp)
```

```
input: [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
freqd = {0: [2, 3], 12: [5, 2], 34: [1, 6], 56: [1, 7], 23: [1, 8], 11: [2, 14], 45: [1, 10], 2: [1, 1
1], 3: [1, 12], 4: [1, 13], 10: [1, 15]}
```

Out[11]:
```
[0, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

Hurray! We correctly detect the second 12 as the first repeated element, and remove that.

But can we do better in terms of writing prettier and tighter code -- perhaps something using **sets** rather than **dictionaries**?

Version 3 -- simplified using a set rather than a dictionary, to detect first repeat

In [12]:
```
def remove_2nd_instance_v3(input):
    print("input:", input)

    index = len(input)
    repeated = set()
    for i, val in enumerate(input):
        if val in repeated:
            index = i
            break
        repeated.add(val)
    print("freq_set:", repeated)
    print("index:", index)

    #Remove this 2nd instance
    output = input[:index] + input[index+1:]  # list slicing rather than list pop
    return output

remove_2nd_instance_v3(inp)
```

```
input: [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
freq_set: {0, 12}
index: 2
```

Out[12]:
```
[0, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

Version 4 -- more direct but less efficient

Here's a very short solution to the original problem (with the first interpretation of the spec). It's interesting to think about how this works, and its virtues and/or failings compared to the earlier versions. This operates directly on the input list, without making any auxiliary sets and dictionaries, so is a more direct solution of the earlier problem. But it doesn't use sets and dictionaries, and as a result, it is slow for large inputs (taking quadratic time instead of linear). Think about why!

In [13]:
```
def remove_2nd_instance_v4(input):
    print("input:", input)
    for i in range(len(input)):
        ii = input.index(input[i],i+1)
        if ii != -1:
            return input[:ii] + input[ii+1:]
    return input[:]

remove_2nd_instance_v4(inp)
```

```
input: [0, 12, 12, 0, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

Out[13]:
```
[0, 12, 12, 12, 12, 34, 56, 23, 11, 45, 2, 3, 4, 11, 10, 12]
```

Consider it a challenge to you to revise this to go with our revised spec, i.e., to remove the first repeated element (12)!

Wordplay

Extended from code at http://programminghistorian.org/lessons/counting-frequencies (http://programminghistorian.org/lessons/counting-frequencies)

Suppose we have two books -- maybe we'd like to see if they were written by the same author, or are otherwise similar. One approach to this is to evaluate the word use frequency in both texts, and then compute a "similarity" or "distance" measure between those two word frequencies. A related approach is to evaluate the frequency of one word being followed by another word (a "word pair"), and see the similarity in use of word pairs by the two texts. Or maybe we're interested in seeing the set of all words that come after a given word in that text.

Here we'll get some practice using **dictionaries** and **sets**, with such wordplay as our motivating example.

Some data to play with...

In [1]:
```
word_string1 = 'it was the best of times it was the worst of times '
word_string2 = 'it was the age of wisdom it was the age of foolishness'
word_string = word_string1 + word_string2

words1 = word_string1.split()  #converts string with spaces to list of words
words2 = word_string2.split()
words = words1 + words2
print("words:", words)
```

```
words: ['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times', 'it',
'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of', 'foolishness']
```

Create some interesting dictionaries from these...

In [2]:
```
# Word frequencies
word_freq = {}
for w in words:
    word_freq[w] = word_freq.get(w,0) + 1
print("Word frequencies:", word_freq)
```

```
Word frequencies: {'it': 4, 'was': 4, 'the': 4, 'best': 1, 'of': 4, 'times': 2, 'worst': 1, 'age': 2,
'wisdom': 1, 'foolishness': 1}
```

NOTE: The above version using get is a useful method for dictionary objects:

```
d.get(key, default)
```

is equivalent to

```
if key in d:
    d[key]
else:
    default
```

already saw

In [3]:
```
# Word pair frequencies
word_pairs = {}
prev = words[0]
for w in words[1:]:
    pair = (prev, w)
    word_pairs[pair] = word_pairs.get(pair,0) + 1
    prev = w
print("Pair frequencies:", word_pairs)
```

```
Pair frequencies: {('it', 'was'): 4, ('was', 'the'): 4, ('the', 'best'): 1, ('best', 'of'): 1, ('of',
'times'): 2, ('times', 'it'): 2, ('the', 'worst'): 1, ('worst', 'of'): 1, ('the', 'age'): 2, ('age',
'of'): 2, ('of', 'wisdom'): 1, ('wisdom', 'it'): 1, ('of', 'foolishness'): 1}
```

In [4]:
```
# What words follow each word?
word_after = {}
prev = words[0]
for w in words[1:]:
    if prev not in word_after:
        word_after[prev] = {w}
    else:
        word_after[prev].add(w)
    prev = w
print("Words followed by\n" + str(word_after) + "\n")
```

```
Words followed by
{'it': {'was'}, 'was': {'the'}, 'the': {'age', 'worst', 'best'}, 'best': {'of'}, 'of': {'wisdom', 'fo
olishness', 'times'}, 'times': {'it'}, 'worst': {'of'}, 'age': {'of'}, 'wisdom': {'it'}}
```

In [5]:
```
# More Pythonic:
# zip is very handy for jointly processing the i_th element from multiple lists.
# Note that in python3 zip is a generator, so is very efficient.
word_after = {}
for w1, w2 in zip(words, words[1:]):
    word_after.setdefault(w1, set()).add(w2)
print("Words followed by\n" + str(word_after) + "\n")
```

```
Words followed by
{'it': {'was'}, 'was': {'the'}, 'the': {'age', 'worst', 'best'}, 'best': {'of'}, 'of': {'wisdom', 'fo
olishness', 'times'}, 'times': {'it'}, 'worst': {'of'}, 'age': {'of'}, 'wisdom': {'it'}}
```

Here `setdefault` is another useful method for dictionaries that saves a lot of initialization code:

```
d.setdefault(key, val)
```

is equivalent to

```
if key not in d:
    d[key] = val
d[key]
```

Rewriting as procedures so we can use them later:

In [6]:
```
def get_freq(words):
    word_freq = {}
    for w in words:
        word_freq[w] = word_freq.get(w,0) + 1
    return word_freq

def get_pair_freq(words):
    word_pairs = {}
    prev = words[0]
    for w in words[1:]:
        pair = (prev, w)
        word_pairs[pair] = word_pairs.get(pair,0) + 1
        prev = w
    return word_pairs
```

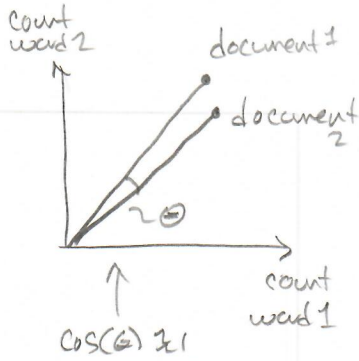
Suppose we want to identify the high frequency words (i.e., sort word frequency from high to low).

In [7]:
```
def sort_freq_dict(freq):
    aux = [(freq[key], key) for key in freq] #list comprehension
    aux.sort()    #sort in place
    aux.reverse() #reverse in place
    return aux    #return an (ordered) list, not a dictionary

#More pythonic:
def sort_freq_dict(freq):
    return sorted([(freq[key], key) for key in freq], reverse=True)

words_by_frequency = sort_freq_dict(word_freq)
print(words_by_frequency)
```

```
[(4, 'was'), (4, 'the'), (4, 'of'), (4, 'it'), (2, 'times'), (2, 'age'), (1, 'worst'), (1, 'wisdom'),
(1, 'foolishness'), (1, 'best')]
```

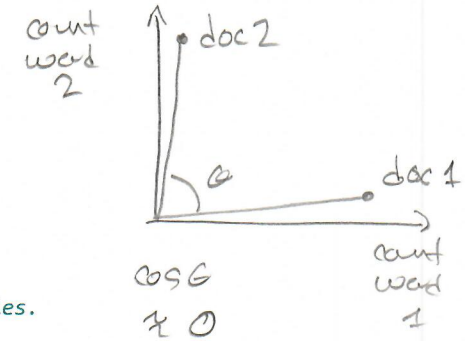
Next, we can build a similarity measure between two word frequencies. We'll use a typical "geometric" notion of distance or similarity referred to as "cosine similarity (https://en.wikipedia.org/wiki/Cosine_similarity)." We build this from vector measures of word frequency including the "norm" and the "dot product", and then calculate a (normalized) cosine distance.

The mathematical details are not crucial here -- but we do want to note how we use dictionary and set operations



In [8]:
```
def freq_norm(freq):
    """ Norm of frequencies in freq, as root-mean-square of freqs """
    sum_square = 0
    for w, num in freq.items():    #iterates over key, val in dictionary
        sum_square += num**2
    return sum_square**0.5

#More pythonic version:
def freq_norm(freq):
    return sum(num*num for num in freq.values())**0.5    #comprehension on just dict values

def freq_dot(freq1, freq2):
    """ Dot product of two word freqs, as sum of products of freqs for words"""
    words_in_both = set(freq1) & set(freq2)
    total = 0
    for w in words_in_both:
        total += freq1[w] * freq2[w]
    return total

#More pythonic version (actually easier to understand!)
def freq_dot(freq1, freq2):
    return sum(freq1[w] * freq2[w] for w in set(freq1) & set(freq2))

import math
#Returns similarity between two word (or pair) frequencies dictionaries.
# 1 indicates identical word (or pair) frequencies;
# 0 indicates completely different words (pairs)
def freq_similarity(freq1, freq2):
    d = freq_dot(freq1, freq2)/(freq_norm(freq1)*freq_norm(freq2))
    ang = math.acos(min(1.0, d))
    return 1 - ang/(math.pi/2)
```

⇒ Very similar



⇒ NOT similar

In [9]:
```
# Some quick tests/examples
x = {'a':40, 'b':2}
y = {'c':3, 'a':30}
print("Combined words:", set(x) | set(y))
print("freq_norm of", x, ":", freq_norm(x))
print("freq_norm of", y, ":", freq_norm(y))
print("freq_dot of", x, "and", y, ":", freq_dot(x,y))
print("freq_similarity:", freq_similarity(x,y))
```

```
Combined words: {'c', 'b', 'a'}
freq_norm of {'a': 40, 'b': 2} : 40.049968789001575
freq_norm of {'c': 3, 'a': 30} : 30.14962686336267
freq_dot of {'a': 40, 'b': 2} and {'c': 3, 'a': 30} : 1200
freq_similarity: 0.9290478472408689
```

In [10]:
```
# Try it out with our short phrases
words3 = "this is a random sentence good enough for any random day".split()
print("words1:", words1, "\nwords2:", words2, "\nwords3:", words3, "\n")

# build word and pair frequency dictionaries, and calculate some similarities
freq1 = get_freq(words1)
freq2 = get_freq(words2)
freq3 = get_freq(words3)
print("words1 vs. words2 -- word use similarity: ", freq_similarity(freq1, freq2))
print("words1 vs. words3 -- word use similarity: ", freq_similarity(freq1, freq3))
print("words3 vs. words3 -- word use similarity: ", freq_similarity(freq3, freq3))
```

```
words1: ['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times']
words2: ['it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of', 'foolishness']
words3: ['this', 'is', 'a', 'random', 'sentence', 'good', 'enough', 'for', 'any', 'random', 'day']

words1 vs. words2 -- word use similarity:  0.5184249085864179
words1 vs. words3 -- word use similarity:  0.0
words3 vs. words3 -- word use similarity:  1.0
```

In [11]:
```
# try that for similarity of WORD PAIR use...
pair1 = get_pair_freq(words1)
pair2 = get_pair_freq(words2)
pair3 = get_pair_freq(words3)
print("words1 vs. words2 -- pair use similarity: ", freq_similarity(pair1, pair2))
print("words1 vs. words3 -- pair use similarity: ", freq_similarity(pair1, pair3))
```

```
words1 vs. words2 -- pair use similarity:  0.29368642735616235
words1 vs. words3 -- pair use similarity:  0.0
```

Now let's do it with some actual books!

In [12]:
```
with open('hamlet.txt', 'r') as f:
    hamlet = f.read().replace('\n', '').lower()
hamlet_words = hamlet.split()

with open('macbeth.txt', 'r') as f:
    macbeth = f.read().replace('\n', '').lower()
macbeth_words = macbeth.split()

with open('alice_in_wonderland.txt', 'r') as f:
    alice = f.read().replace('\n', '').lower()
alice_words = alice.split()

print(len(hamlet_words), len(macbeth_words), len(alice_words))
```

```
28218 18154 15189
```

With the text from those books in hand, let's look at similarities...

In [13]:
```
hamlet_freq = get_freq(hamlet_words)
macbeth_freq = get_freq(macbeth_words)
alice_freq = get_freq(alice_words)
print("similarity of word freq between hamlet & macbeth:", freq_similarity(hamlet_freq, macbeth_freq))
print("similarity of word freq between alice & macbeth:", freq_similarity(alice_freq, macbeth_freq))

hamlet_pair = get_pair_freq(hamlet_words)
macbeth_pair = get_pair_freq(macbeth_words)
alice_pair = get_pair_freq(alice_words)
print("\nsimilarity of word pairs between hamlet & macbeth:", \
      freq_similarity(hamlet_pair, macbeth_pair))
print("similarity of word pairs between alice & macbeth:", \
      freq_similarity(alice_pair, macbeth_pair))
```

```
similarity of word freq between hamlet & macbeth: 0.8234901643970373
similarity of word freq between alice & macbeth: 0.7242123439984074

similarity of word pairs between hamlet & macbeth: 0.3195599532068242
similarity of word pairs between alice & macbeth: 0.23412902997911367
```

So we've confirmed that **Macbeth** is more similar to **Hamlet** than to **Alice in Wonderland**, both in word use and in word pair usage. Good to know!