Que - 1

In [1]:
```
# 1. Create a list words = ['is', 'it', 'good', '?']. a) Use a series of assignment statements (e.g. words
```

Part 1.a

In [2]:
```
# a) Using tmp varible

# Transform to ['it', 'is', 'good', '?']

words = ['is', 'it', 'good', '?']
tmp = words[0]
words[0] = words[1]
words[1] = tmp
words[3] = '!'
print(words)
```

Part 1.b

In [3]:
```
# b) Using tuples

# Transform to ['it', 'is', 'good', '!']

words = ('is', 'it', 'good', '?')
words_tuple = (words[1], words[0], words[2], '!')
list(words_tuple)
```

Que - 2

In [4]:
```
# 2.Write code that removes whitespace at the beginning and end of a string ('   this   is a sample  sentei
# a) do this task using split() and join()
# b) do this task using regular expression substitutions
```

Part 2.a

In [5]:
```
# a) do this task using split() and join()

sent = '   this   is a sample  sentence '
' '.join(sent.split())
```

Part 2.b

In [6]:
```
# b) do this task using regular expression substitutions

import re
re.sub(r'\s+', ' ', re.sub(r'^\s+|\s+$', '', sent))
```

Que - 3

In [7]:
```
# 3. sent1=['The', 'dog', 'gave', 'John', 'the', 'newspaper']. Now assign sent2=sent1. Modify sent1[1]='mo
# a) verify that sent2 has changed
# b) Now try the same exercise but instead assign sent2=sent1[:]. Modify sent1[1]='monkey' and see what ha
# c) Now define text1=[['The', 'dog', 'gave', 'John', 'the', 'newspaper'], ['John', 'is', 'happy']]. Now a
# d) Extract successive overlapping 4-grams from ['The', 'dog', 'gave', 'John', 'the', 'newspaper'].
```

Part 3.a

In [8]:
```
# a)

sent1=['The', 'dog', 'gave', 'John', 'the', 'newspaper']
sent2=sent1
sent2
```

In [9]:
```
sent1[1]='monkey'
```

In [10]:
```
sent1
```

In [11]:
```
sent2
```

In [12]:
```
print(sent1==sent2)

# sent2 has changed
```

We can see that sent 2 has changed. The second element in the list has changed from dog to monkey as we can see above. The change is true for the original list sent1 as well. This is like creating aliases.

Part 3.b

In [13]:
```
# b)

sent1=['The', 'dog', 'gave', 'John', 'the', 'newspaper']
sent2=sent1[:]
sent2
```

In [14]:
```
sent1[1]='monkey'
```

In [15]:
```
sent1
```

Out[15]:
```
['The', 'monkey', 'gave', 'John', 'the', 'newspaper']
```

In [16]:
```
sent2
```

Out[16]:
```
['The', 'dog', 'gave', 'John', 'the', 'newspaper']
```

In [17]:
```
print(sent1==sent2)

# sent2 has not changed
```

```
False
```

sent1[:] creates a shallow copy of the original list into sent 2. But it does not refer to the same list object. Hence we don't risk changing the original list by changing the copy created by sent[:].

NOTE: When reading, list is a reference to the original list, and list[:] shallow-copies the list.

Explanation: The sent2 keeps a reference of the original items and any changes to the sent1 now is not refelected on sent2. That's exactly what we see above. This is a shallow copy but since the list obejct here is NOT compunded it also acts as if it is a deep copy and thus sent2 is no longer same as sent1.

Part 3.c

In [18]:
```
# c)

text1=[['The', 'dog', 'gave', 'John', 'the', 'newspaper'], ['John', 'is', 'happy']]
text2=text1[:]
text1[0][1]='monkey'
```

In [19]:
```
text1
```

Out[19]:
```
[['The', 'monkey', 'gave', 'John', 'the', 'newspaper'],
 ['John', 'is', 'happy']]
```

In [20]:
```
text2
```

Out[20]:
```
[['The', 'monkey', 'gave', 'John', 'the', 'newspaper'],
 ['John', 'is', 'happy']]
```

In [21]:
```
print(text1==text2)

# text2 has changed
```

```
True
```

Explanation: In this case it is a compound object (lists of lists) and this is exactly where the difference between shallow copy and deep copy comes into picture. In case of a compunded object like the one we have over here, it will create a copy of the inside list and thus any change will be reflected on both.

Part 3.d

In [22]:
```
# d)

# Method 1:

sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
n = 4
[sent[i:i+n] for i in range(len(sent) - n + 1)]
```

Out[22]:
```
[['The', 'dog', 'gave', 'John'],
 ['dog', 'gave', 'John', 'the'],
 ['gave', 'John', 'the', 'newspaper']]
```

In [23]:
```
# Method 2:

import nltk
list(nltk.ngrams(sent,4))
```

Out[23]:
```
[('The', 'dog', 'gave', 'John'),
 ('dog', 'gave', 'John', 'the'),
 ('gave', 'John', 'the', 'newspaper')]
```

Que - 4

In [24]:
```
# 4.Write a function that prints any word that appeared in the last 20% of a text that had not been encoun
```

In [25]:
```
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

In [26]:
```
# I thought I can run the set function to identify the unique values from one list and from that
# just subtract the ones from another list against which we want to compare.

result = []
x = ['I','like','NLP','class']
y = ['Which','class','do','you','like']
for item in set(x) - set(y):
    result.append(item)

print(result)

# Here we can see it returns -- 'I and NLP' as words from set x because 'like and class' were present in s
# So I need to just include this inside a fucntion and replace
# x with last 20% of words from text1 and y with first 80% of words from text1.
```

```
['I', 'NLP']
```

In [27]:
```
result = []
def last20(text):
    limit = int(0.8*len(text))
    for word in set(text[limit:]) - set(text[:limit]):
        result.append(word)
    return result
```

In [28]:
```
print(last20(text1))
```

```
'sourceless', 'jubilations', 'candid', 'Bachelor', 'muster', 'expertness', 'jettest', 'defiance', 'Am
ericas', 'determining', 'marsh', 'Penetrating', 'STEP', 'Dead', 'preventer', 'economic', 'SLOWLY', 'o
val', 'affirm', 'wincing', 'aslope', 'Common', 'POKE', 'gambol', 'pitchers', 'moles', 'dartingly', 't
reading', 'urgent', 'inhabitable', 'Light', 'characterized', 'unaccounted', 'unbegotten', 'Boats', 'R
OLL', 'writhed', 'reforming', 'lasts', 'bewildering', '112', 'affinities', 'Ifs', 'clearness', 'geolo
gist', 'Europa', 'drama', 'Whose', 'attentive', 'omnitooled', 'belaying', 'Tunnel', 'bejuggled', '12
3', 'blunted', 'digger', 'unmanned', 'reelingly', 'unprepared', ';--"', 'blanched', 'cloudless', 'wor
kers', 'soliloquizer', 'bunting', 'moodily', 'contrastingly', 'crumb', 'trained', 'singed', 'sweetnes
s', 'chill', 'mastery', 'favouring', 'numbness', 'colic', 'catcher', 'unearthed', 'REPEATED', 'uncata
strophied', 'insolent', 'fallacious', 'pedlar', 'contributory', 'determine', '126', 'ditches', 'suppl
ication', 'undue', 'comforts', 'ravines', 'mixes', 'nutmeg', 'sweeps', 'shears', 'Miles', 'woeful',
'smoothed', 'metres', 'Walks', 'afire', 'LANTERNS', 'canals', 'whaleships', 'unbegun', 'Japans', 'blu
er', 'foster', 'judges', 'surveying', 'animate', 'wrapping', 'birch', 'feathering', 'LASHINGS', 'thin
kest', 'Albicore', 'Monsieurs', 'Tahitian', 'launch', 'queenly', 'bomb', 'abhorring', 'Inferable', 'k
indling', 'shortest', 'continents', 'stolidity', 'HEIGHT', 'interflow', 'loosening', 'THESE', 'Typhoo
n', 'hawk', 'domineered', 'fuller', 'Unmindful', 'forbearing', 'floes', 'weasel', 'repairing', 'Swa
y', 'incommoding', 'negroes', 'unfold', 'jobs', 'foreknew', 'hooped', 'ether', 'strivest', 'cymballin
g', 'display', 'unmoor', 'homes', 'jetting', 'inuendoes', 'crucible', 'lieutenant', 'suffered', 'bris
tling', 'blistered', 'prescient', 'Despatch', 'Lad', 'centrepiece', 'Instances', 'razors', 'mocked',
'Chase', 'slavery', 'sulphurous', 'initiate', 'Mate', 'expert', 'leapest', 'Dying', 'Hither', 'coasti
```

Que - 5

In [29]:
```
# 5.Write a program that takes the sentence ("we have seen two kinds of sequence object: strings and lists
```

In [30]:
```
import nltk
from nltk import word_tokenize, sent_tokenize
```

In [31]:
```
text = "we have seen two kinds of sequence object: strings and lists."
```

In [32]:
```
def process_sent(text):
    fdist = FreqDist(w.lower() for w in nltk.word_tokenize(text))
    for dict_key in sorted(fdist.keys()):
        print("Frequency of the token: {0:<8} is {1}".format(dict_key, fdist[dict_key]))
```

In [33]:
```
process_sent(text)

# It is arranged in alphabetical order of dict_key as asked in the question.
```

```
Frequency of the token: .        is 1
Frequency of the token: :        is 1
Frequency of the token: and      is 1
Frequency of the token: have     is 1
Frequency of the token: kinds    is 1
Frequency of the token: lists    is 1
Frequency of the token: object   is 1
Frequency of the token: of       is 1
Frequency of the token: seen     is 1
Frequency of the token: sequence is 1
Frequency of the token: strings  is 1
Frequency of the token: two      is 1
Frequency of the token: we       is 1
```

Que - 6

In [34]:
```
ood Today good'), omitting the n most frequently occurring words of the text. You should use w.lower() to n
```

In [35]:
```
text = 'big big world today tomorrow good Today good'
```

In [36]:
```
def shorten(raw, n):
    words = nltk.word_tokenize(raw.lower())
    fdist = FreqDist(w for w in words if w.isalpha())
    tmp = {w for w, freq in fdist.most_common(n)}
    revised_text = ' '.join(w for w in words if w not in tmp)
    return revised_text
```

In [37]:
```
shorten(text,1)
```

Out[37]:
```
'world today tomorrow good today good'
```

In [38]:
```
shorten(text,2)
```

Out[38]:
```
'world tomorrow good good'
```

In [39]:
```
shorten(text,3)
```

Out[39]:
```
'world tomorrow'
```

In [40]:
```
shorten(text,4)
```

Out[40]:
```
'tomorrow'
```

In [41]:
```
shorten(text,5)

# Any number of n > 5 will gives us a blank.
# This is important as it keeps removing the blank and replaces it with a new blank.
# There would thus be no error for shorten(text,6), shorten(text,7) and so on.
```

Out[41]: ''

Explanation: I created a function to read the frequency distribution of words which are alphabetic and then created a tmp variable to store the most common words. They are stored in the order of the input, so we can call it for n=1 then even if there is a tie it will omit the top most priority in the given sequence as available in text. This tmp variable acts as a pseudo index and when we run the shorten () function with some value for n, it will remove all the values until that count and return the remianing ones.

Que - 7

In [42]:
```
# 7. Write a list comprehension that sorts a list of WordNet synsets for proximity to a given synset. For
```

In [43]:
```
from nltk.corpus import wordnet as wn
```

In [44]:
```
wn_synsets = [wn.synset('lesser_rorqual.n.01'), wn.synset('killer_whale.n.01'), wn.synset('novel.n.01'), w
wn_synsets
```

Out[44]:
```
[Synset('lesser_rorqual.n.01'),
 Synset('killer_whale.n.01'),
 Synset('novel.n.01'),
 Synset('tortoise.n.01')]
```

In [45]:
```
wn_synsets[0]
```

Out[45]:
```
Synset('lesser_rorqual.n.01')
```

In [46]:
```
distance = [synset.shortest_path_distance(wn.synset('right_whale.n.01')) for synset in wn_synsets]

for i in range (len(distance)):
    # print("Shortest path distance of {0:<8} from right_whale.n.01 is {1}".format(wn_synsets[i], distance
    print("Shortest Path Distance of {0} from Synset('right_whale.n.01') is {1}".format(wn_synsets[i], dis
```

```
Shortest Path Distance of Synset('lesser_rorqual.n.01') from Synset('right_whale.n.01') is 3
Shortest Path Distance of Synset('killer_whale.n.01') from Synset('right_whale.n.01') is 5
Shortest Path Distance of Synset('novel.n.01') from Synset('right_whale.n.01') is 22
Shortest Path Distance of Synset('tortoise.n.01') from Synset('right_whale.n.01') is 12
```

In [47]:
```
sorted(wn_synsets, key=lambda x: x.shortest_path_distance(wn.synset('right_whale.n.01')))
```

Out[47]:
```
[Synset('lesser_rorqual.n.01'),
 Synset('killer_whale.n.01'),
 Synset('tortoise.n.01'),
 Synset('novel.n.01')]
```

Explanation: When we run the sorted function with lambda I can see that the sequence is refreshed based on the shortest distance from right_whale.n.01 and now tortoise comes before novel.

Que - 8

In [48]:
```
es of the word chair, then table would appear before chair in the output list.You should use lambda in the
```

In [49]:
```
# Given example:

words=['table','chair','desk','table','table','chair']
words
```

Out[49]:
```
['table', 'chair', 'desk', 'table', 'table', 'chair']
```

In [50]:
```
# Method 1: Using lambda in the sorted () as asked in the question

def remove_dup(words):
    fdist = FreqDist(words)
    return sorted(set(words), key=lambda x:fdist[x], reverse=True)
remove_dup(words)
```

Out[50]:
```
['table', 'chair', 'desk']
```

In [51]:
```
# Method 2: Using simple list comprehension with most_common method

def remove_dup_2(words):
    fdist = FreqDist(words)
    return [w for w, fd in fdist.most_common()]
remove_dup_2(words)
```

Out[51]:
```
['table', 'chair', 'desk']
```

In [52]:
```
# Method 3: Let user input the elements and then based on frequency of the elements in the list run the rei

print("Insert elements of your choice. Please ensure each element is seperated by a space")
x = input("Your input --> ")
word_list = x.split(' ')
result = list(word_list)
result.sort()
print(result)
```

```
Insert elements of your choice. Please ensure each element is seperated by a space
Your input --> table table table chair chair desk book book book book book
['book', 'book', 'book', 'book', 'book', 'chair', 'chair', 'desk', 'table', 'table', 'table']
```

In [53]:
```
remove_dup(word_list)
```

Out[53]:
```
['book', 'table', 'chair', 'desk']
```

Que - 9

In [54]:
```
function that takes a text (e.g. text3 from nltk.book) and a vocabulary (e.g. nltk.corpus.words.words()) as
```

In [55]:
```
def text_vs_vocabulary(text, vocabulary):
    return set(text).difference(vocabulary)
```

In [56]:
```
# Let's verify this for a small set:

text_vs_vocabulary(['I','like','NLP','class'], ['Which','class','do','you','like'])
```

Out[56]:
```
{'I', 'NLP'}
```

In [57]:
```
# We saw it is working for small set above. Let's run it on text3 from nltk.book over all corpus words fro

text_vs_vocabulary(text3, nltk.corpus.words.words())
```

```
 'Asenath',
 'heard',
 'Almodad',

 'Oh',
 'rebuked',
 'Timna',
 'Calah',
 'They',
 'preserved',
 'despised',
 'servants',
 'Is',
 'Say',
 'Timnah',
 'reproved',
 'Naamah',
 'Shalem',
 'Malchiel',
 'poured',
```

Que - 10

In [58]:
```
# 10. Import the itemgetter() function from the operator module in Python's standard library (i.e. from op
```

In [59]:
```
from operator import itemgetter

# Given example:

words= ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
print(sorted(words, key=itemgetter(1)))
print(sorted(words, key=itemgetter(-1)))
```

```
['gave', 'newspaper', 'The', 'the', 'dog', 'John']
['The', 'gave', 'the', 'dog', 'John', 'newspaper']
```

In [60]:
```
# Another example:

new_words= ['Public', 'schools', 'in', 'New York', 'are', 'returning', 'to', 'the', 'classroom', 'this', '
print(sorted(new_words, key=itemgetter(1)))
print(sorted(new_words, key=itemgetter(-1)))
```

```
['schools', 'New York', 'returning', 'week', 'the', 'this', 'classroom', 'in', 'to', 'are', 'Public']
['Public', 'are', 'the', 'returning', 'New York', 'week', 'classroom', 'in', 'to', 'schools', 'this']
```

In [61]:
```
help(itemgetter)
```

```
Help on class itemgetter in module operator:

class itemgetter(builtins.object)
 |  itemgetter(item, ...) --> itemgetter object
 |
 |  Return a callable object that fetches the given item(s) from its operand.
 |  After f = itemgetter(2), the call f(r) returns r[2].
 |  After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])
 |
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |      Return state information for pickling
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```

Explanation: Operator is a built-in module providing a set of convenient operators -- operator.itemgetter(n) constructs a callable that assumes an iterable object (e.g. list, tuple, set) as input, and fetches the n-th element out of it. I have called the help manual above to read the functional details from the developer of the package.

Therefore in the given example itemgetter(1) reads value in alphabetical order of the item in 1st index while itemgetter(-1) does it based on the last index.